

This document is for the sole use of Giovanni Ferreira Aluno

Verilog Language and Application

Course Version 28.0

Lecture Manual

Revision 1.0

cadence®

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

© 1990–2023 Cadence Design Systems, Inc. All rights reserved.

Printed in the United States of America.

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

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

All other trademarks are the property of their respective holders.

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

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

The publication may not be modified in any way;

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

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

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

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

Open SystemC, Open SystemC Initiative, OSCI, SystemC, and SystemC Initiative are trademarks or registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries and are used with permission.

All other trademarks are the property of their respective holders.

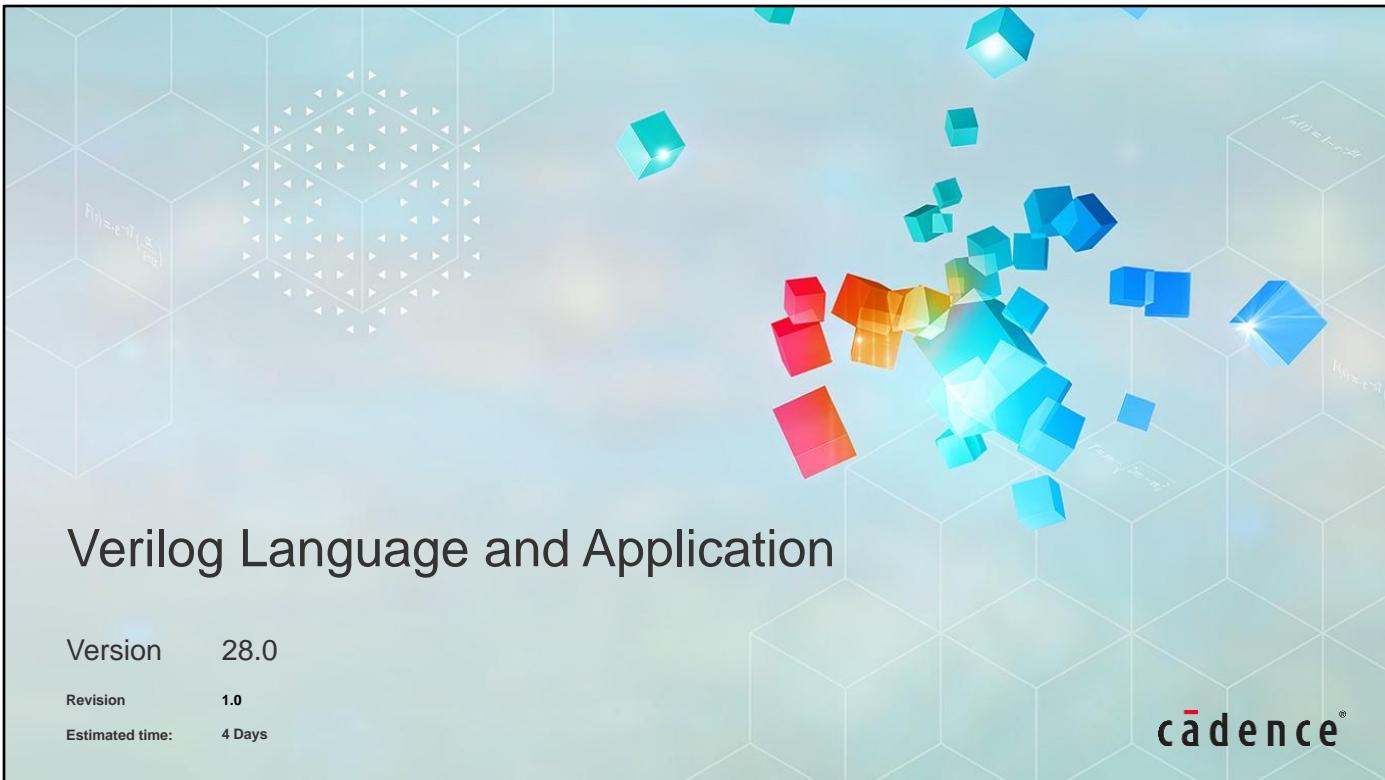
Table of Contents

Verilog Language and Application

Module 1	About This Course	2
There are no labs in this module		
Module 2	Describing Verilog Applications	9
Lab 2-1 Exploring the VeriRISC CPU Design		
Module 3	Verilog Introduction	26
Lab 3-1 Modeling an Address Multiplexor		
Module 4	Choosing Between Verilog Data Types.....	46
Lab 4-1 Modeling a Data Driver		
Module 5	Using Verilog Operators	76
Lab 5-1 Modeling the Arithmetic Logic Unit		
Module 6	Making Procedural Statements.....	98
Lab 6-1 Modeling a Controller		
Module 7	Using Blocking and Nonblocking Assignments.....	122
Lab 7-1 Modeling a Generic Register		
Module 8	Using Continuous and Procedural Assignments.....	138
Lab 8-1 Modeling a Single-Bidirectional-Port Memory		
Module 9	Understanding the Simulation Cycle.....	157
Lab 9-1 Modeling a Generic Counter		
Module 10	Using Functions and Tasks	180
Lab 10-1 Modeling the Counter Using Functions		
Lab 10-2 Modeling the Memory Test Block Using Tasks		

Module 11	Directing the Compiler	201
	Lab 11-1 Verifying the VeriRISC CPU Design	
Module 12	Introducing the Process of Synthesis	215
	Lab 12-1 Exploring the Synthesis Process	
Module 13	Coding RTL for Synthesis	232
	Lab 13-1 Using a Component Library	
Module 14	Designing Finite State Machines	264
	Lab 14-1 Coding State Machines in Multiple Styles	
Module 15	Avoiding Simulation Mismatches	283
	There are no labs in this module	
Module 16	Managing the RTL Coding Process	300
	There are no labs in this module	
Module 17	Managing the Logic Synthesis Process	315
	There are no labs in this module	
Module 18	Coding and Synthesizing an Example Verilog Design.....	348
	Lab 18-1 Coding a Serial-to-Parallel Interface Receiver	
Module 19	Using Verification Constructs	367
	Lab 19-1 Resolving a Deadlocked System	
Module 20	Coding Design Behavior Algorithmically	384
	There are no labs in this module	
Module 21	Using System Tasks and System Functions	398
	Lab 21-1 Adding System Tasks and System Functions to a Beverage Dispenser Design	
Module 22	Generating Test Stimulus.....	426
	Lab 22-1 Verifying a Serial Interface Receiver	

Module 23	Developing a Testbench	458
Lab 23-1 Testing a VeriRISC CPU Model		
Module 24	Example Verilog Testbench	492
Lab 24-1 Developing a Script-Driven Testbench Using Verilog 1995		
Lab 24-2 Developing a Script-Driven Testbench Using Verilog 2001		
Module 25	Course Conclusions	499
Module 26	Next Steps	502
Appendix A	Configurations	506
Lab A-1 Configuring a Simulation		
Appendix B	Modeling with Verilog Primitives and UDPs	517
Lab B-1 Using Built-In Verilog Primitives with a Macro Library		
Lab B-2 Using User-Defined Verilog Primitives with a Macro Library		
Appendix C	SDF Annotation Overview.....	555
Lab C-1 Annotating an SDF with Timing		
Appendix D	SystemVerilog Overview and UVM Methodology Introduction.....	578
There are no labs in this appendix		



Verilog Language and Application

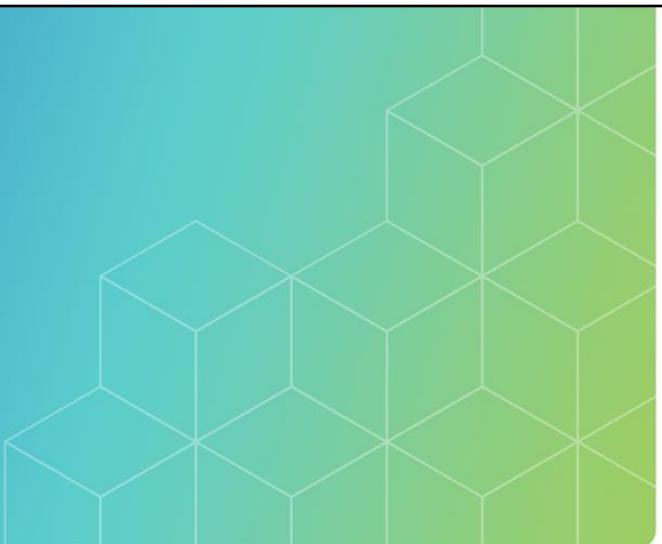
Version 28.0

Revision 1.0

Estimated time: 4 Days

cadence®

This page does not contain notes.



Module 1

About This Course

cadence®

This page does not contain notes.

Course Prerequisites

Before taking this course, you need to have:

- Basic computer literacy – you must know how to use a shell and editor of your choice and navigate the file system.
- A basic understanding of digital hardware design and verification.
- Knowledge of a procedural programming language will facilitate your learning experience.



This page does not contain notes.

Course Objectives

In this course, you:

- Use fundamental Verilog language constructs required for design, verification and logic synthesis.
- Analyze and ensure that Verilog designs meet the requirements for synthesis.
- Compare mismatches that can happen between pre-synthesis and post-synthesis simulation and the process of synthesis.
- Debug digital designs by developing Verilog test environments of significant capability and complexity.
- Apply system tasks and functions to verify digital designs using the Xcelium™ Simulator.



This page does not contain notes.

Course Agenda

Day 1

- Describing Verilog Applications
 - Lab: Exploring the VeriRISC CPU Design
- Verilog Introduction
 - Lab: Modeling an Address Multiplexor
- Choosing Between Verilog Data Types
 - Lab: Modeling a Data Driver
- Using Verilog Operators
 - Lab: Modeling the Arithmetic Logic Unit

Day 2

- Making Procedural Statements
 - Lab: Modeling a Controller
- Using Blocking and Nonblocking Assignments
 - Lab: Modeling a Generic Register
- Using Continuous and Procedural Assignments
 - Lab: Modeling a Single-Bidirectional-Port Memory
- Understanding the Simulation Cycle
 - Lab: Modeling a Generic Counter
- Using Functions and Tasks
 - Lab: Modeling the Counter Using Functions
 - Lab: Modeling the Memory Test Block Using Tasks
- Directing the Compiler
 - Lab: Verifying the VeriRISC CPU Design

Day 3

- Introducing the Process of Synthesis
 - Lab: Exploring the Synthesis Process
- Coding RTL for Synthesis
 - Lab: Using a Component Library
- Designing Finite State Machines
 - Lab: Coding State Machines in Multiple Styles
- Avoiding Simulation Mismatches
- Managing the RTL Coding Process
- Managing the Logic Synthesis Process
- Coding and Synthesizing an Example Verilog Design
 - Lab: Coding a Serial-to-Parallel Interface Receiver

Day 4

- Using Verification Constructs
 - Resolving a Deadlocked System
- Coding Design Behavioral Algorithmically
- Using System Tasks and System Functions
 - Adding System Tasks and System Functions to a Beverage Dispenser Design
- Generating Test Stimulus
 - Verifying a Serial Interface Receiver
- Developing a Testbench
 - Testing a VeriRISC CPU Model
- Example Verilog Testbench
 - Developing a Script-Driven Testbench Using Verilog 1995
 - Developing a Script-Driven Testbench Using Verilog 2001



This page does not contain notes.

Software and Licenses

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

https://www.cadence.com/en_US/home/training/all-courses/82110.html

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



This page does not contain notes.

Become Cadence Certified by Earning a Digital Badge



Digital badges indicate mastery in a certain technology or skill and give managers and potential employers a way to validate your expertise.

- Cadence Training Services offers digital badges for our popular training courses.
- Your digital badge can be added to your email signature or social media platforms like LinkedIn or Facebook.

Benefits of Cadence Certified Digital Badges

- Validate expertise
 - Expand career opportunities
- Professional credibility
 - Stand apart from your peers
- For more information, go to www.cadence.com/training or email es_digitalbadge@cadence.com.



How do I register to take the exam?

- Log in to our [Learning Management System](#), click on the course in your transcript, and go to the Content tab to locate the exam.

How long will it take to complete the exam?

- Most exams take 45 to 90 minutes to complete. You may retake the exam multiple times to pass the exam.

How do I access and use the digital badge?

- After you pass the exam, you get a digital badge and instructions on how to place it on social media sites.

How is the digital badge validated?

- [Credly](#) validates the digital badge as issued to you by Cadence and includes the details of the criteria you completed to earn the badge.



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

This page does not contain notes.

Icons Used in This Class



Best Practice



Language Syntax



Concept/
Glossary



Frequently Asked
Questions /
Quiz



Error Message



Problem & Solution



Quick Reference



Tool Command



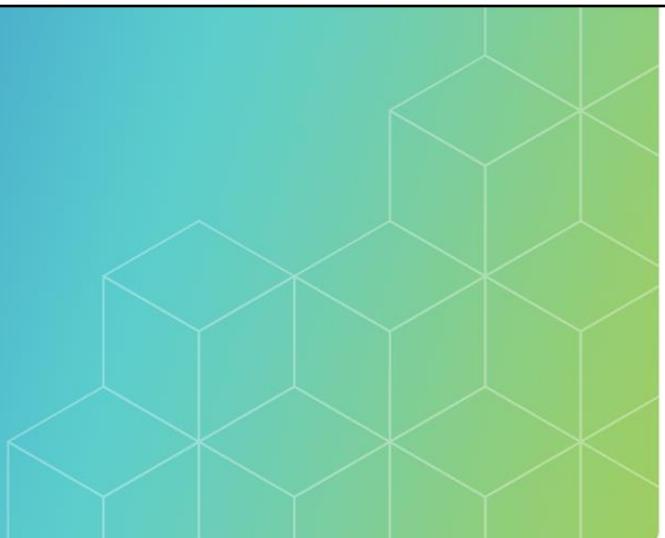
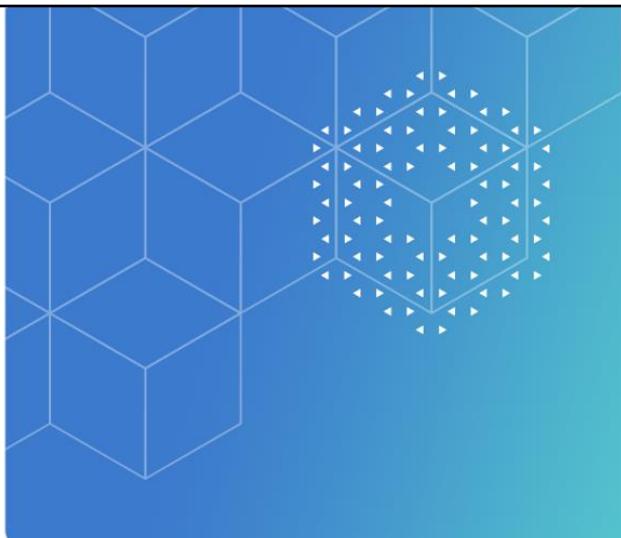
How To



Lab List

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

This page does not contain notes.



Module 2

Describing Verilog Applications

cadence®

This module introduces the definition and use of a hardware description language.

Module Objective

In this module, you learn:

- The nature and use of a Hardware Description Language (HDL)

Topics

- What is an HDL?
- Levels of abstraction
- Benefits of using HDLs
- The Verilog HDL



Your objective is to be able to explain the nature and use of a Hardware Description Language. To do that, you need to know what a HDL is, what they are used for, and who uses one.

Terms and Definitions

Hardware Description Language (HDL)	A programming-like language that describes the functionality of digital electronic hardware circuits
Simulator	Software that imitates the functionality of electronic hardware as described by the HDL
Level of Abstraction	The level of detail provided by the utilized modeling style, such as behavioral and gate level
Application Specific Integrated Circuit (ASIC)	Integrated circuit developed for a specific application
ASIC Vendor	The company manufacturing the chips. This company is responsible for developing and providing the library cells
Bottom-Up Design Flow	A design methodology in which you build the low-level components first and then connect them to make large systems
Top-Down Design Flow	A design methodology in which you define the system at a very high level of abstraction and then iteratively refine it
Register Transfer level (RTL)	A level of abstraction that defines storage elements and their clocks
Tool Command Language (Tcl)	A scripting language used for issuing commands to interactive programs



- A Hardware Description Language (HDL) is a programming-like language that describes the functionality of digital electronic hardware circuits.
- A Simulator is software that imitates the functionality of electronic hardware as described by the HDL.
- A Level of Abstraction is the level of detail provided by the utilized modeling style, such as behavioral and gate level.
- An Application Specific Integrated Circuit (ASIC) is an integrated circuit developed for a specific application.
- An ASIC vendor is a company that manufactures ASICs. The company provides the cell libraries for its technology.
- A Bottom-Up Design Flow is a design methodology in which you build the low-level components first and then connect them to make larger systems.
- A Top-Down Design Flow is a design methodology in which you define the system at a very high level of abstraction and then iteratively refine it.
- The Register Transfer level (RTL) is a level of abstraction that defines storage elements and their clocks.
- The Tool Command Language (Tcl) is a scripting language used for issuing commands to interactive programs.

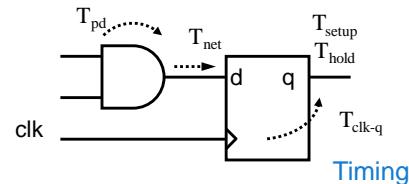
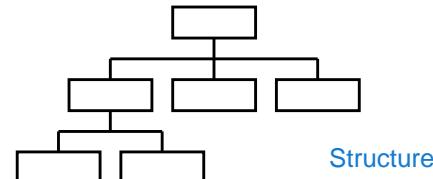
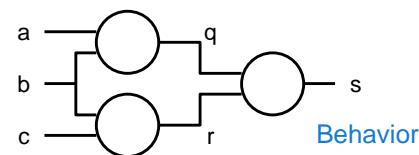
What Is a Hardware Description Language (HDL)?

An HDL is a programming language with special constructs for Modeling hardware concurrency and timing.

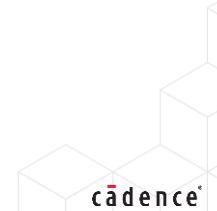
An HDL supports design at multiple levels of abstraction:

- Behavioral Modeling
 - Sequential behaviors
 - Parallel behaviors
- Structural Modeling
 - Hardware component hierarchy
 - Software subroutine hierarchy

An HDL typically supports simulation of estimated design timing.



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



An HDL is similar to a procedural programming language, but also contains constructs to describe digital electronic systems. An HDL contains features and constructs to support description of:

- Behavior – Both serial and parallel. In serial behavior, you pass the output of one functional block to the input of another, which is similar to the behavior of a conventional software language. However in parallel behavior, you can pass a block output to the inputs of a number of blocks acting in parallel where many separate events happen at the same moment in time.
- Structure – Both physical, such as hierarchical block diagrams and component netlists, and software, such as subroutines. This allows you to describe large, complex systems and manage their complexity.
- Time – Programming languages have no concept of time. An HDL has to model propagation delays, clock periods, and timing checks.

An HDL typically supports multiple abstraction levels. You can describe hardware behaviorally, without and with sufficient detail for logic synthesis, and as a structured netlist of predefined components that can themselves be as simple as a transistor and as complex as another behavioral design.

An HDL Supports Multiple Levels of Abstraction

Behavioral Level

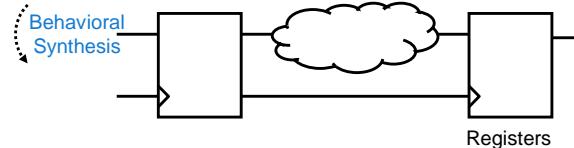
- Algorithms



Behavior

Register Transfer Level (RTL)

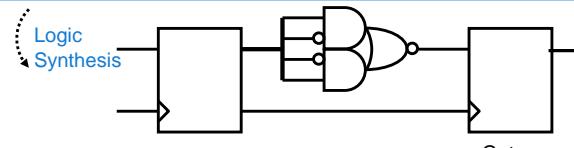
- Nets and registers



Registers

Gate Level

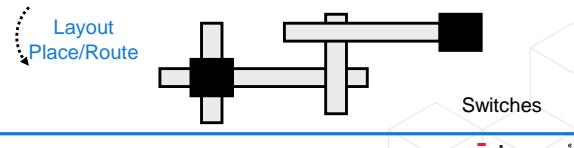
- Built-in and user-defined primitives



Gates

Switch Level

- Built-in switch primitives



Switches

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

cadence®

You can describe a system as a group of hierarchical models of varying amount of detail.

An HDL supports multiple levels of such detail. The three main levels of abstraction are:

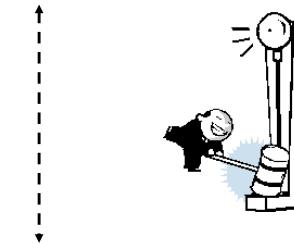
- The behavioral level:
 - You describe the system using mathematical equations; and
 - You can omit timing – the system may simulate in zero-time like a software program.
- The Register Transfer Level (RTL):
 - You partition the system into combinational and sequential logic, using constructs and coding styles supported by logic synthesis; and
 - You define timing in terms of cycles based on one or more defined clock(s).
- The structural level:
 - You instantiate and interconnect predefined components;
 - Which can include vendor-provided macrocells; and
 - Which can include logic primitives built into the language.

Abstraction Level Tradeoffs

Simulation effort is proportional to detail:

- Behavioral Level
 - Algorithms
- Register Transfer Level (RTL)
 - Nets and registers
- Gate Level
 - Built-in and user-defined primitives
- Switch Level
 - Built-in switch primitives

- Less detail
 - Faster design entry
 - Faster simulation



- More detail
 - Slower design entry
 - Slower simulation

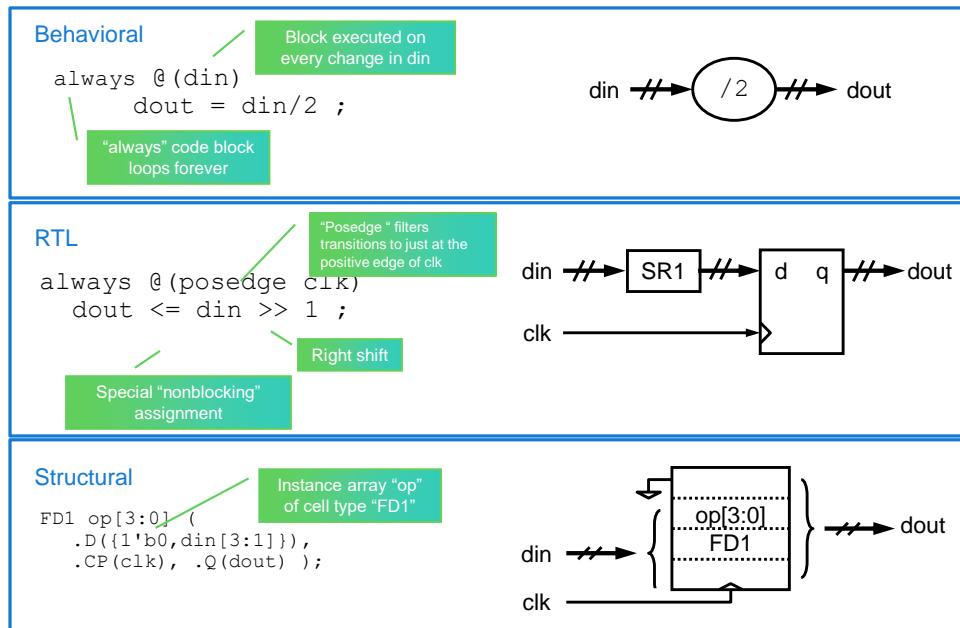
14 © Cadence Design Systems, Inc. All rights reserved.



Users of an HDL model at different levels of abstraction:

- Block architects model functionality at the behavioral level for maximum simulation speed and for the ease with which they can quickly modify the architecture as they explore alternatives;
- Block implementers refine the functional blocks to the RT level for a logic synthesis tool; and
- Library developers model cells at the gate level for simulation and the physical level for place and route tools.

Abstraction Level Example: Divide by 2



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

cadence®

These code fragments illustrate the three main levels of abstraction:

- The behavioral level of abstraction describes the design behavior with no hint about how the operation is implemented;
- The RT level of abstraction describes the design behavior with sufficient detail that logic synthesis can infer an implementation involving an edge-triggered storage device; and
- The structural level of abstraction instantiates and connects a predefined storage device from a macro library, not caring how the component is itself implemented.

Why Use a Hardware Description Language?

The benefits of using an HDL include the following:

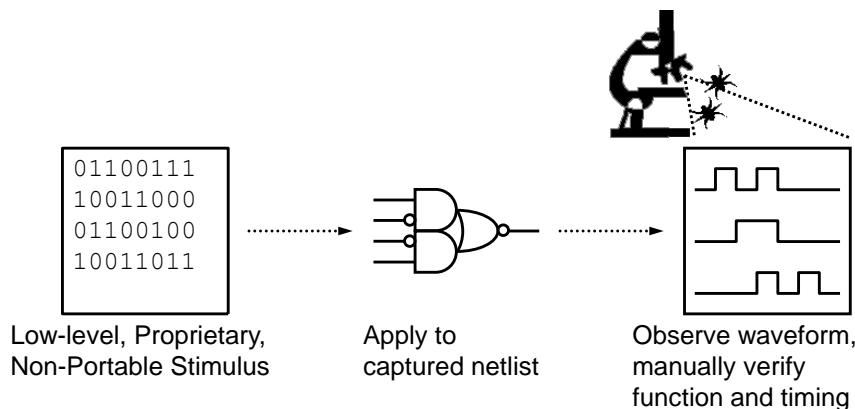
- You write HDL in simple ASCII text.
 - Capture the design quickly and easily manage modifications
- You can design at a higher abstraction level.
 - Easily explore design alternatives
 - Find problems earlier in the design cycle
- Use of a standard HDL enables design reuse.
 - Between design teams and partners, and through the design flow
 - No re-entry, reformatting, or translation
- The description is independent of the implementation.
 - You can delay the choice of implementation technology
 - You can more easily make architectural and functional changes
 - You can more easily adapt the design to future projects



Benefits of using an HDL include the following:

- You can capture and modify the design more quickly in an HDL than by schematic capture.
- Earlier design capture at the higher abstraction level means earlier simulation, which facilitates design alternative exploration to produce a more optimal architecture and partitioning, and allows a more thorough verification.
- Use of a standard HDL facilitates reuse of designs from previous projects or from commercial Intellectual Property (IP) providers. You can move or switch between different tools and vendors without re-entry, reformat or translation, of the design description.
- Your RT-level implementation is almost 100% independent of the implementation technology, so you can defer selection of the target ASIC or FPGA technology until the design is mostly entered, and you can switch technology or vendor with a minimum of redesign.

Legacy Schematic-Based Design



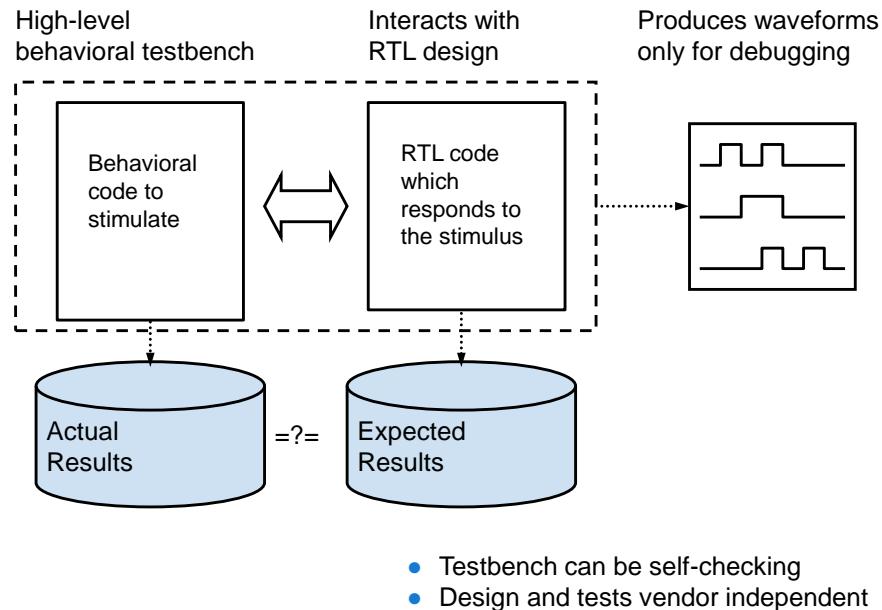
Let's compare to an HDL ...

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



Designers used to capture the design intent using a proprietary schematic capture tool, manually developed test stimulus in a proprietary tabular format, and simulated the design using a proprietary simulator. Those who had money to burn had graphical displays to examine the results, but most of us had plain old text terminals, so could examine the results only as the 0 and 1 characters. We did not use the Z and X characters because those values do not exist in hardware and we used simulation only to generate expected results for a device test machine.

HDL-Based Simulation



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



In more recent times, we capture the design intent and its test using the same standard HDL and simulate them together using a choice of tools running on a wide choice of third-party platforms and we have high-definition wide-screen displays that we do not have to stare at quite so much because our tests to a large extent pinpoint any problems with a high degree of accuracy.

Who Uses Hardware Description Languages?

The following personnel use an HDL:

- System architects doing high-level architectural exploration
- Verification personnel testing components and systems
- Hardware designers implementing RT-level code for synthesis
- Model developers describing system-level IP and ASIC/FPGA macrocells



- System architects create system-level HDL models for high level architectural exploration;
- Verification personnel create HDL testbenches for testing components and systems;
- Hardware designers implement the system-level models as Register Transfer-level HDL for synthesis; and
- Model developers describe system-level IP and ASIC or FPGA macrocells in an HDL.

Some HDL Adoption Issues

Issues in using HDLs include the following:

- Yet one more thing to learn!
 - A new programming language to understand
 - Simulation and synthesis EDA tools to learn
- Major change in design methodology
 - No standard “off the shelf” design methodology
 - Needs to be planned and implemented
- Primarily targets digital design
 - Analog extensions exist
- Design planning and partitioning required before coding
- Coding style influences project time-to-closure
- Other specific issues as we will see during the course



Issues in using an HDL include that:

- Adopting an HDL requires you to learn a new language, and if you previously used schematic capture, to change to a more software-like design methodology.
- The way in which you write your code can affect the simulation and synthesis efficiency and the area and performance of the product.

The Verilog HDL

The Verilog HDL is used primarily for design, verification, and implementation of digital logic.

- Developed in 1983 by Phil Moorby and Prabhu Goel at Automated Integrated Design Systems
 - Currently IEEE Std. 1364-2005
- Syntax: Similar to C programming language
- Paradigm: Procedural assignment of values to variables
- Types: Loosely typed Language
- Comprehensive language with no facility for user extensions



- Verilog was initially developed in 1983 by Phil Moorby and Prabhu Goel at Automated Integrated Design Systems. The Verilog syntax is similar to C, and like C, Verilog is loosely typed and case-sensitive.
- For variable types, Verilog includes only reg, integer, time, and real.
- Verilog has several net types. The net type determines how to resolve the value of a net having multiple drivers.
- Verilog has several transistor and gate-level primitives, a construct for user-defined primitives, and a Programmable Logic Array (PLA) modeling mechanism.
- For nets and primitives, you can optionally specify a propagation delay.
- For continuous assignments and primitives, you can optionally specify a drive strength.
- For modules, you can optional specify path delays, pulse filtering, and timing checks.

Verilog Language Versions

The IEEE Std 1364-2005 had relatively few major extensions over Verilog-2001.

uwire, `pragma, `begin_keywords, `end_keywords			Verilog-2005
ANSI C style ports generate localparam constant functions			Verilog-2001
modules parameters function/tasks always @ assign	\$finish \$fopen \$fclose \$display \$fwrite \$monitor 'define 'ifdef 'else 'include 'timescale	initial disable events wait # @ fork-join	standard file I/O \$value\$plusargs 'ifndef 'elsif 'line @* (* attributes *) configurations memory part selects variable part select wire, wand wor, supply0 supply1 tri, tri0, tri1, triand, trior trireq + = * / % >> <<
		reg integer real time packed arrays 2D memory	multi dimensional arrays signed types automatic ** (power operator) Verilog-1995



The IEEE Std 1364 standardized in 1995 what was previously the proprietary Verilog hardware description language.

The 2001 update to the Verilog standard added several convenience features, targeted at hardware design and verification. The IEEE Std 1364-2005 had relatively few major extensions over Verilog-2001, such as the uwire type, encryption, a `pragma directive, and a `begin_keywords compatibility directive.

Module Review

What is an HDL?

How is the HDL description independent of the product implementation and why is this an advantage?

What level of abstraction is used in:

- Testbenches?
- Synthesizable designs?
- Netlists?



This page does not contain notes.

Module Review Solutions

1. What is an HDL?

- An HDL is a programming language with special constructs for modeling hardware, such as abstraction, concurrent behavior, and timing.

2. How is the HDL description independent of the product implementation and why is this an advantage?

- The HDL is standard and the synthesizable subset is standard.
You specify the target technology upon synthesis.

3. What level of abstraction is used in:

- a. Testbenches?
- b. Synthesizable designs?
- c. Netlists?

- Testbenches – Behavioral
- Synthesizable designs – RTL
- Netlists – Structural

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



This page does not contain notes.



Lab

Lab 2-1 Exploring the VeriRISC CPU Design

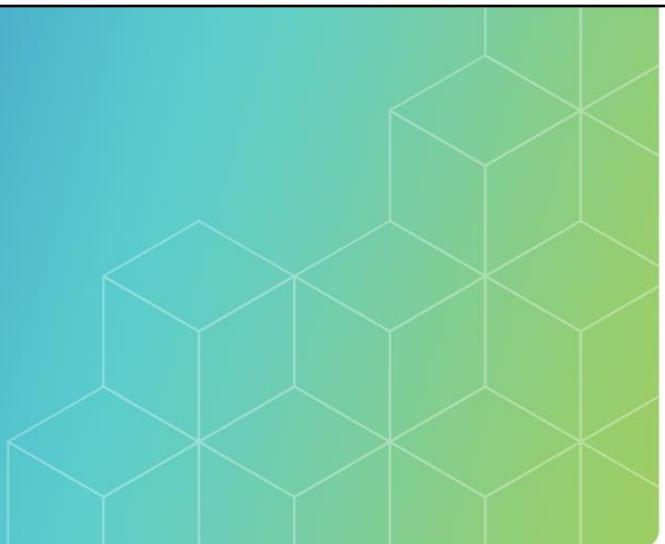
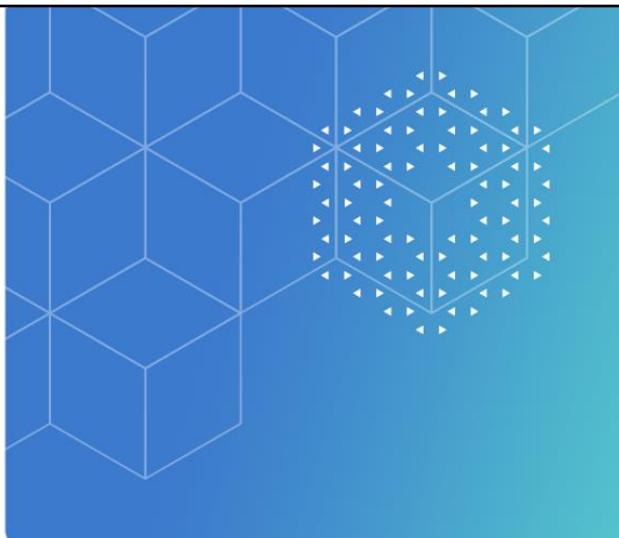
- To become familiar with the lab model, which is a RISC CPU design.

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



The objective of this lab is to become familiar with the lab model, which is a RISC CPU Design.

For this part, you examine the lab model description in the lab instructions.



Module 3

Verilog Introduction

cadence®

This module examines the fundamental language constructs and how you use them to describe a design.

Module Objective

In this module, you:

- Use basic Verilog constructs to describe a simple design

Topics

- Describing design modules
- Representing hierarchy
- Describing module behavior
- Synchronizing module behaviors
- Communicating between behaviors
- Rules for identifiers, comments, white space
- Configuring and compiling a design



Your objective is to get started using Verilog to describe the behavior of a digital design. To do that, you need to know some fundamental Verilog language constructs.

Describing Design Modules

- Start with the **module** keyword.
- Describe the module interface.

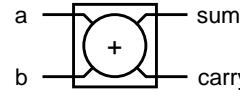
- Verilog-1995

- list of ports
 - (*a*, *b*);
 - followed by port declaration
 - input a, b;*

- Verilog-2001

- list of port declarations
 - (input a, b, ...);*

- Describe the module behavior.
- End with **endmodule** keyword.



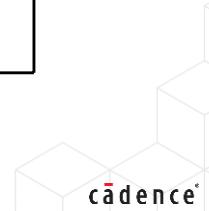
```
module halfadd (a, b, sum, carry);
  input a, b;
  output sum, carry;
  assign sum = a ^ b;
  assign carry = a & b;
endmodule
```

Verilog-1995

```
module halfadd (
  input a, b, output sum, carry
);
  assign sum = a ^ b;
  assign carry = a & b;
endmodule
```

Verilog-2001

 Verilog identifiers are case sensitive.
Verilog keywords are always lowercase.



The basic building block of design hierarchy is the module declaration. A module can represent the complete system, its major subblocks such as the CPU, further subblocks such as the ALU, and physical blocks such as an ASIC cell. Each module instance is its own scope in which it can instantiate primitives, nets, variables and other modules.

Each module declaration starts with the *module* keyword followed by a unique definition name for the module. Most module declarations then follow with a list of ports or a list of port declarations. A module declaration typically then declares additional module items, and concludes with the *endmodule* keyword.

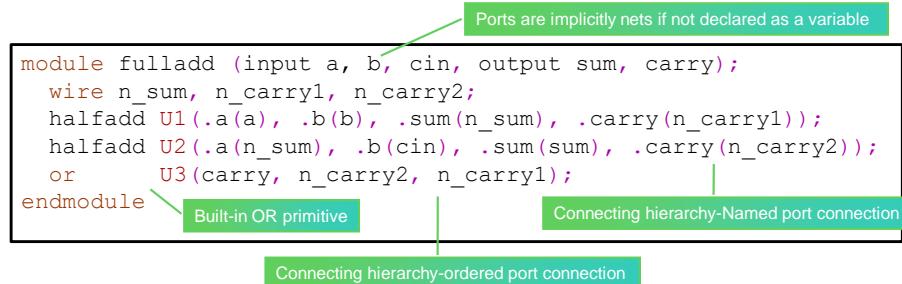
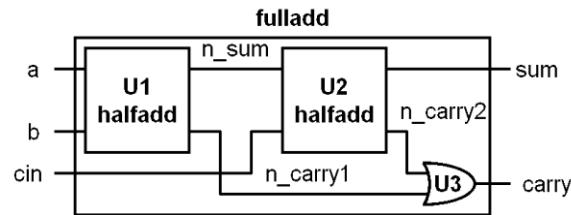
Here is a module describing a half-adder block. The module definition name is **halfadd**. The first example uses the Verilog-1995 list of ports syntax to list the ports ***a*, *b*, *sum*** and ***carry***. It also declares the ports later as module items. The second example uses the Verilog-2001 syntax to list the port declarations. You can use either syntax for any module, but you cannot mix the two syntaxes in a single module declaration.

When you declare a port, you are explicitly or implicitly declaring a net or variable of that name and permitting the scope instantiating the module to make a connection to that net or variable. If you do not explicitly declare a net or variable, then a single-bit net is implicitly declared for the port.

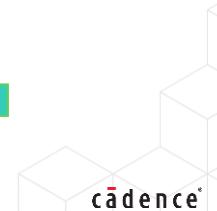
The module describes the half-adder functionality by making two continuous assignment statements to the module output port nets. The statements assign the results of Boolean expressions. Continuous assignments are processes, and they react to changes of their inputs. Upon any change of the ***a*** or ***b*** input port net values, the simulator automatically updates the expression values and assigns the new values to the output port nets.

Creating Hierarchy

- Declare local nets and variables.
- Instantiate module(s).
 - Connect instance ports to local nets and variables.
- A port of an instance of a module can be connected using:
 - Named port connection.
 - Ordered port connection.



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



You create hierarchy by declaring ports, nets, variables and module instances and connecting ports of the module instances to the locally declared ports, nets, and variables.

You describe a full adder as a module definition that declares two instances of the half adder and an instance of an “or” built-in primitive and makes the appropriate connections.

This module first declares the nets it will use to connect the instance ports. You generally need to declare items before you reference them. Verilog implicitly declares single-bit nets for connections you make to module ports, so for this example you can elect to omit the net declarations.

You give each module an instance name (U1 and U2) that is unique within the module. You can use the instance names to uniquely reference each instance in the design.

Verilog provides built-in primitives for basic logic functions. This example instantiates (U3) an “or” primitive. For this example you could have alternatively used a continuous assignment.

```

assign carry = n_carry2 | n_carry1;
-----

module_instantiation ::= 
  module_identifier [ parameter_value_assignment ] module_instance { , 
  module_instance } ;

module_instance ::= name_of_instance ([list_of_port_connections])

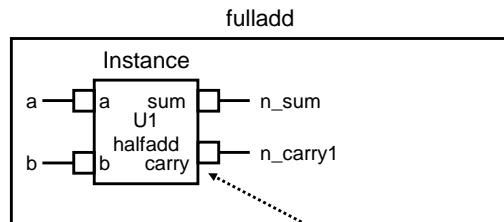
list_of_port_connections ::= 
  ordered_port_connection { , ordered_port_connection }
  | named_port_connection { , named_port_connection }
  
```

Connecting Hierarchy – Named Port Connection

Map local port, net or variable to instance port by name – explicitly specify the instance port name.



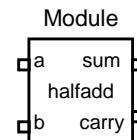
With this syntax you
are less likely to make
a mistake!



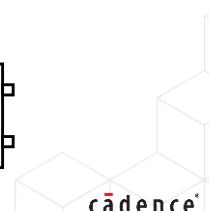
```
module fulladd (input a, b, cin, output sum, carry);
    wire n_sum, n_carry1, n_carry2;
    halfadd U1(.a(a), .b(b), .sum(n_sum), .carry(n_carry1));
    halfadd U2(.a(n_sum), .b(cin), .sum(sum), .carry(n_carry2));
    or     U3(carry, n_carry2, n_carry1);
endmodule
```

Wire `n_carry1` of module
`fulladd` mapped to output
carry of instance `U1` of
module `halfadd`

```
module halfadd (a, b, sum, carry);
    input a, b;
    output sum, carry;
    assign sum = a ^ b;
    assign carry = a & b;
endmodule
```



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



To connect a module instance's port to locally declared ports, nets and variables, you use either the ordered port connection syntax or the named port connection syntax.

This example uses the named port connection syntax, in which you explicitly name the port to which you connect the expression. The expression is often simply a locally declared port, net or variable, but it can also be a concatenation of bit and part selections, and for input ports it can include other operators. You can omit any port connection by simply not naming the port.

Leaving an output port unconnected is common, e.g., for a device that has an active-low version and an active-high version of the same output signal. Leaving an input port unconnected is usually an error. Leaving an input port unconnected feeds a high-impedance value into the module or primitive, which almost all modules and primitives handle as an unknown logic value.

The named port connection syntax explicitly specifies the instance port for the connection. This syntax is verbose, but quickly readable and less likely to cause connection errors.

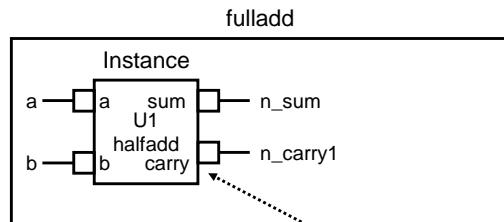
```
list_of_port_connections ::=  
    ordered_port_connection { , ordered_port_connection }  
  | named_port_connection { , named_port_connection }  
  
named_port_connection ::=  
    { attribute_instance } .port_identifier ( [ expression ] )
```

Connecting Hierarchy – Ordered Port Connection

Map local port, net or variable to instance port by position – in the order the ports are declared.



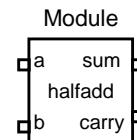
With this syntax you can very easily make a mistake!



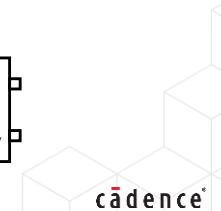
```
module fulladd (input a, b, cin, output sum, carry);
    wire n_sum, n_carry1, n_carry2;
    halfadd U1(a, b, n_sum, n_carry1);
    halfadd U2(n_sum, cin, sum, n_carry2);
    or     U3(carry, n_carry2, n_carry1);
endmodule
```

Wire `n_carry1` of module `fulladd` mapped to output `carry` of instance `U1` of module `halfadd`

```
module halfadd (a, b, sum, carry);
    input a, b;
    output sum, carry;
    assign sum = a ^ b;
    assign carry = a & b;
endmodule
```



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



To connect a port of an instance of a module to locally declared ports, nets and variables, you use either the ordered port connection syntax or the named port connection syntax.

This example uses the ordered port connection syntax, in which you simply list the expressions connected to the instance ports in the same order that the instantiated module or primitive declares the ports. The expression is often simply a locally declared port, net or variable, but it can also be a concatenation of bit and part selections, and for input ports can include other operators. You can omit any port connection expression, but need to retain the comma as a place holder to make any later port connections.

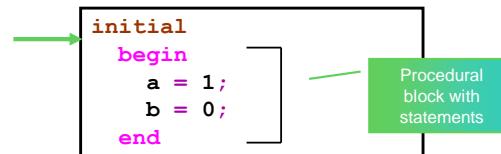
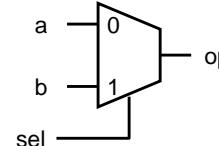
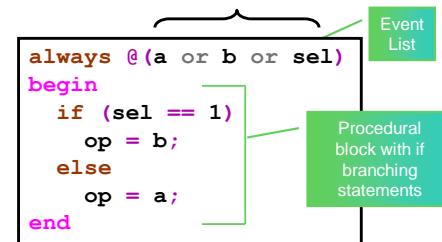
Leaving an output port unconnected is common, e.g., for a device that has an active-low version and an active-high version of the same output signal. Leaving an input port unconnected is usually an error. Leaving an input port unconnected feeds a high-impedance value into the module or primitive, which almost all modules and primitives handle as an unknown logic value.

You and your coworkers may only with difficulty trace a signal path through ordered port connections, as the port ordering is not immediately obvious. You are also very likely to create incorrect connections. The more verbose named port connection syntax solves both problems.

```
list_of_port_connections ::=  
    ordered_port_connection { , ordered_port_connection }  
  | named_port_connection { , named_port_connection }  
  
ordered_port_connection ::= { attribute_instance } [ expression ]
```

Describing Module Behavior with Procedural Blocks

- Procedural Block starts with the keyword:
 - always
 - Synthesizable construct
 - Executes at start of simulation
 - Execution blocks and unblocks in accordance with timing controls
 - When at end, loops back to beginning*
- initial
 - Non-synthesizable or Testbench construct
 - Executes at start of simulation
 - Execution blocks and unblocks in accordance with timing controls
 - When at end, *terminates*
- Multiple statements in procedural blocks are enclosed between **begin** and **end** keywords.
- Multiple procedural blocks interact concurrently.



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



Previous examples described the half-adder behavior by making continuous assignment statements.

To describe complex behavior, Verilog provides procedural constructs. You place procedural constructs within a module declaration. Within the procedural constructs, you place procedural statements similar to the statements of a programming language.

Verilog provides two procedural blocks:

- The always construct starts executing at the start of simulation. Execution suspends and resumes in accordance with timing controls. Upon executing the last statement, execution loops back to the beginning of the construct. The always construct is ideal for describing hardware that always reacts to one or more of its inputs. The always construct always has at least one timing control to keep the construct from “hogging” all the execution cycles.
- The initial construct starts executing at the start of simulation. Execution suspends and resumes in accordance with timing controls. Upon executing the last statement, execution of that block terminates. The initial construct is ideal for describing a testbench that administers the test a finite, number of times, usually just once. The initial construct does not need any timing controls, but in a testbench at least one will likely have timing controls in order to “step” through the test.

Within a procedural block, you can choose to execute a group of statements sequentially in their order of appearance, as shown here, and you can choose to execute a group of statements in parallel in no particular order.

No execution order is implied between the procedural blocks. Procedural blocks scheduled to run in the same simulation cycle can execute in any order.

Synchronizing Module Behaviors

- Use the @ event control.
- Execution blocks until an event in the event expression occurs.
- An event is any transition of the specified nets and variables.
- Verilog-2001 and above added the comma and wildcard operators.
 - The wildcard operator adds all “signals” that go into the block and into any functions called from the block.

 Parentheses are optional for event expressions consisting of a single token.

1995 – use or operator

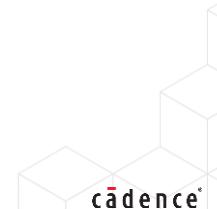
```
always @(a or b or sel)
  if (sel == 1)
    op = b;
  else
    op = a;
```

2001 and above – can use , operator

```
always @(a, b, sel)
  if (sel == 1)
    op = b;
  else
    op = a;
```

2001 and above – can use * wildcard for combinatorial logic inputs

```
always @*
  if (sel == 1)
    op = b;
  else
    op = a;
```



Verilog provides procedural timing controls for “stepping” execution of procedural blocks. The most common of these is the event control. An event control starts with the *at* (@) character and then follows with either a wildcard (*) character, a single event identifier, or a parenthesized event expression. The event expression can be a list of event expressions separated by the *or* keyword or by the comma (,) character – both are shown here. The *or* keyword in an event expression is a separator between event expressions and is not an operator in the usual sense. You can further qualify an expression with the *posedge* or *negedge* keywords that you will see later.

A timing control is not itself a statement, but precedes a statement. In the case of an event control, it blocks the execution of that statement and subsequent sequential statements until one of those events occurs.

```
event_control ::= 
  @ event_identifier
  | @ ( event_expression )
  | @*
  | @ (*) 

event_expression ::= 
  expression
  | hierarchical_identifier
  | posedge expression
  | negedge expression
  | event_expression or event_expression
  | event_expression , event_expression
```

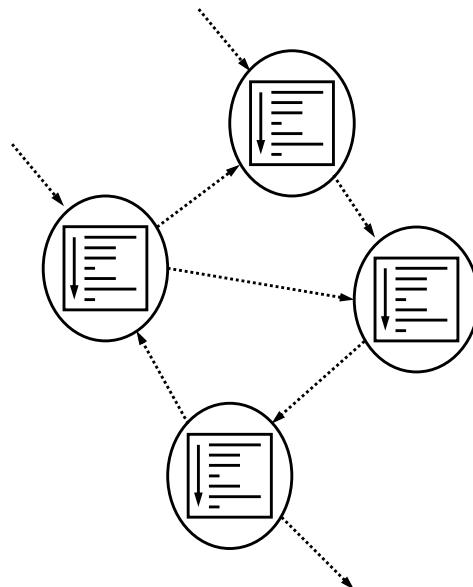
Communicating Between Behaviors

A design contains one or more modules (usually several):

- Reminder: Modules communicate through ports connected to external nets and variables.

A module may contain any number of procedural blocks:

- Each executes their own statements sequentially (like conventional programming languages).
- Multiple blocks execute concurrently (like hardware).
- Multiple blocks communicate using nets and variables.



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

This capability to have many procedural blocks communicating concurrently with each other through variables is the basic model which Verilog uses to describe hardware. The procedural blocks can be in a single module or partitioned into a number of modules in a hierarchical structure. However, the basic model remains the same.

Rules for Naming Identifiers

- Identifiers start with a letter or an underscore (_).
- Subsequent characters may be letter, digit, dollar sign (\$) or underscore.
- Verilog does not restrict name length.
- Tool or methodology may restrict name length.
- Identifiers are case sensitive.
 - ABC, Abc, abc are all different legal names.

Not Legal

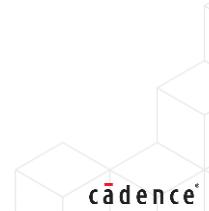
unit-32
16_bit_bus
\$abc

Legal

unit_32
bus_16_bits
abc\$

Escaped

\unit-32
\16_bit_bus
\\$abc



An identifier is a name you provide for Verilog objects and instances.

You must start an identifier with an alphabetical character (a-z, A-Z) or an underscore (_) character. Your later characters can include any alphanumeric character, the dollar sign (\$) character, and the underscore character.

You will later see that built-in and user-defined system tasks and system functions start with a dollar sign character. These are not user identifiers in the normal sense.

The Verilog language is for the most part sensitive to character case. Numbers and radices can be in either case. Keywords and the built-in system tasks and system functions are all lower case. The case of your identifier characters is exactly as you specify. You can declare two identifiers that differ only in case, and you can declare an identifier that differs from a keyword only in the case of one or more of its characters. These are both extremely poor practice.

An escaped identifier accepts any printable ASCII character in any position. You escape an identifier using a backslash (\) character prefix and a whitespace (space, tab, newline) character suffix. The backslash and white space are not part of the identifier.

Escaped identifiers exist to provide compatibility with tools and technology libraries. Some tools, for example, break vector variables into scalar variables that include their bracketed index in their name. The name must be escaped to allow the brackets. Some libraries, for example, start cell names with a digit to indicate the number of inputs. The name must be escaped to allow the initial digit. You should in general not declare escaped identifiers yourself because they make your code less readable and there is seldom good reason to do so.

Rules for Comments and White Space

```
// A one-line comment starts with // and ends with newline character
/* A block comment starts anywhere with /*
   and ends anywhere with */

module muxadd (a, b, sel, sum, carry, y);
  input a, b, sel;           // module inputs
  output /* module outputs */ sum, carry, y;
  ...

// Verilog is a free-format language
// White space is needed only to separate some language tokens
// Use additional white space to enhance readability
  assign sum    = a ^ b;
  assign carry  = a & b;

// Also use indentation (2 space is best) to enhance readability
  always @ (a or b or sel)
    if (sel == 1)
      y = b;
    else
      y = a;
  ...

```

Do not clutter your source
with comments that state
the obvious!

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

cadence®

Verilog is a free-format language. You can use white space to organize the code to enhance readability. Verilog ignores these characters except where needed to separate other language tokens.

Verilog has single-line comments and block comments:

- Single-line comments start with two consecutive slash (/) characters and terminate at the end of the line. The comment can include the whole line or any final part of it.
- A block comment starts with a slash character and following asterisk (*) character /*) and terminates with the reverse – an asterisk character followed by a slash character (*/). As a block comment ignores newline characters, you can place a block comment in a part of a line and across multiple lines. You do have to be careful that you do not use those character sequences within a block comment, as you cannot nest block comments.

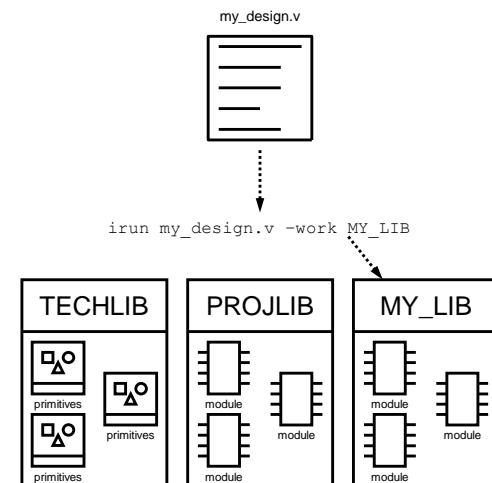
You should in general not clutter your source with unneeded comments. Describe your module in a comment header at the top of the file and otherwise use comments only where absolutely necessary to clarify obscure fragments of your code.

Use indentation and line breaks to make your code readable. You should indent your code a consistent two spaces for each indent. One space is easily missed and more than two consume too much white space and eventually make many of your statements wrap over to a second line. You should in general place no more than one executable statement on a line.

Using Design Libraries

Some Verilog tools use compilation libraries:

- A collection of already-compiled modules or primitives.
 - Stored in file system.
 - Referred to by library name.
 - Library map file maps name to location.
 - Library name -> directory name
 - Compiler places output in “WORK” library.
 - “WORK” library specified by user for each compilation session.
 - Configuration in Verilog provides a prioritized list of libraries from which to bind compiled cell descriptions to cell instances.



Not all Verilog simulators use compilation libraries.

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



Simulating an HDL design and testbench is a process that first converts the source files into a binary form that the elaborator and simulator can recognize. This process of checking the syntax and producing the binary file is known as ***compilation***.

Verilog tools that use libraries compile a design into a file system data structure known as a library. A vendor-specific mechanism selects one of those libraries as the *current working library*, and compilation by default places the results of compilation into that library. The elaboration process later selects design and testbench components from the libraries to construct the design and test configuration for simulation.

Compiling the Design

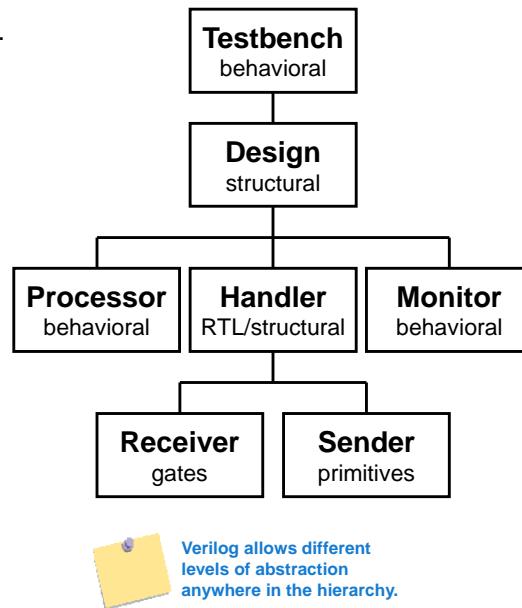
- Verilog compilation order is generally not important.
- Exception: compiler directives:
 - In source file(s).
 - Tell compiler how to interpret subsequent code.
 - Not part of design but can effect how compiler creates the design!



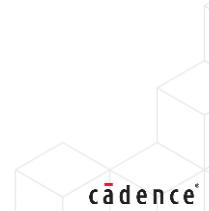
Scope of compiler directive is from point read to wherever overridden or reset until end of compilation session – across multiple files.



Use compiler directives infrequently and carefully!



Verilog allows different levels of abstraction anywhere in the hierarchy.



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

This diagram shows a complete hierarchical design and testbench.

- The behavioral testbench sits at the top level of the hierarchy. The testbench instantiates the design module and contains test code.
- The structural design module instantiates the three primary design blocks and contains no design code of its own.
 - The behavioral processor module fully describes the processor functionality by itself.
 - The behavioral monitor module fully describes the monitor functionality by itself.
 - The RTL handler module partially describes the handler functionality in synthesizable code and instantiates two submodules to complete the description. The receiver submodule is an already-synthesized netlist of ASIC vendor macros and the sender submodule is a hand-crafted netlist of Verilog primitives.

You can normally compile these modules in any order. The exception is that if any file contains compiler directives like 'ifdef, ifndef etc. that another file depends upon, then you must compile the files together and compile first the file containing the directives. Creating such cross-file dependencies is considered poor practice.

The elaborator later links together the compiled module descriptions to configure a design and testbench for simulation.

Module Summary

You should now be able to use basic Verilog constructs to describe a simple design.

This module very briefly introduced:

- Describing design modules (the **module** keyword)
- Representing hierarchy (instantiation and port connection)
- Describing module behavior (procedural blocks)
 - Synchronizing module behaviors (event controls)
 - Communicating between behaviors (shared nets and variables)
- Rules for identifiers, comments, white space
- Compiling a design (“work” library and library map file)



This module examined the fundamental language constructs and how you use them to describe a design:

- The basic building block of design hierarchy is the module declaration.
- You create hierarchy by declaring ports, nets, variables and module instances and connecting ports of the module instances to the locally declared ports, nets, variables.
- To describe complex behavior, Verilog provides procedural constructs.
- Verilog provides procedural timing controls for *stepping* execution of procedural blocks. The most common of these is the event control.
- An identifier starts with an alphabetical character (a-z, A-Z) or an underscore (_) character and can contain alphanumeric characters, the dollar sign (\$) character, and the underscore character.
- Single-line comments start with two consecutive slash (/) characters and terminate at the end of the line.
- A block comment starts with a slash character and following asterisk (*) character /*) and terminates with the reverse – an asterisk character followed by a slash character (*/).
- Verilog is a free-format language. You can use white space to organize the code to enhance readability.
- The Verilog 2001 update provides a standard means to specify a design configuration.

Module Review

1. What is the basic building block of a Verilog design?
2. How do Verilog procedures communicate or how is data passed between Verilog Procedures?
3. When compiling a set of Verilog files, which is normally compiled first?



This page does not contain notes.

Module Review Solutions

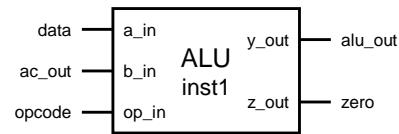
1. What is the basic building block of a Verilog design?
 - The Verilog basic building block is the module.
2. How do Verilog procedures communicate or how is data passed between Verilog Procedures?
 - Procedural blocks communicate by passing events, and by passing data via nets and shared variables informally called "signals".
3. When compiling a set of Verilog files, which is normally compiled first?
 - This matters only if a later file utilizes compiler directives set by an earlier file.



This page does not contain notes.

Module Exercise

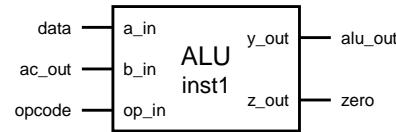
Write a module instantiation statement to instantiate this module:



This page does not contain notes.

Module Exercise Solution

Write a module instantiation statement to instantiate this module:



Solution:

ALU inst1

```
(  
    .a_in  (data)  ,  
    .b_in  (ac_out) ,  
    .op_in (opcode) ,  
    .y_out (alu_out) ,  
    .z_out (zero)  
) ;
```



This page does not contain notes.

Video: Simulation Using Xcelium



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

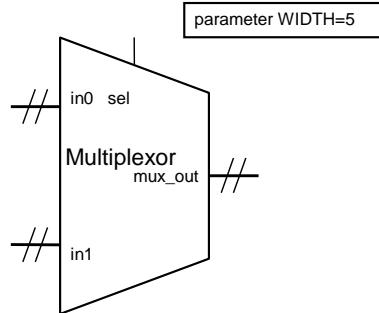
This page does not contain notes.



Lab

Lab 3-1 Modeling an Address Multiplexor

- Use basic Verilog constructs to describe a simple design.

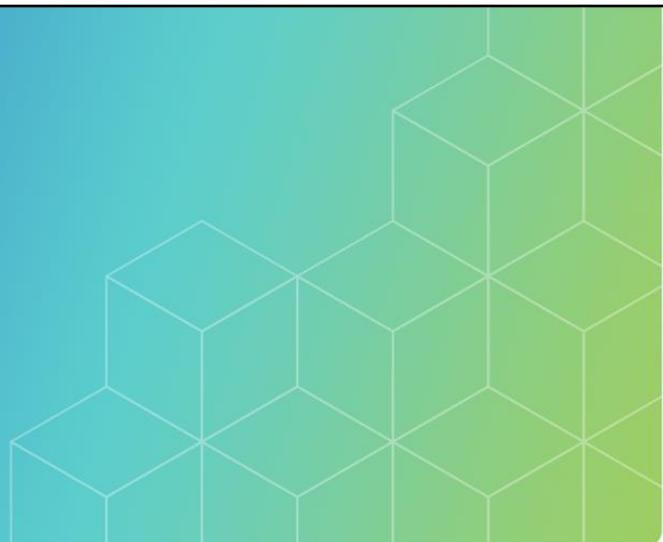
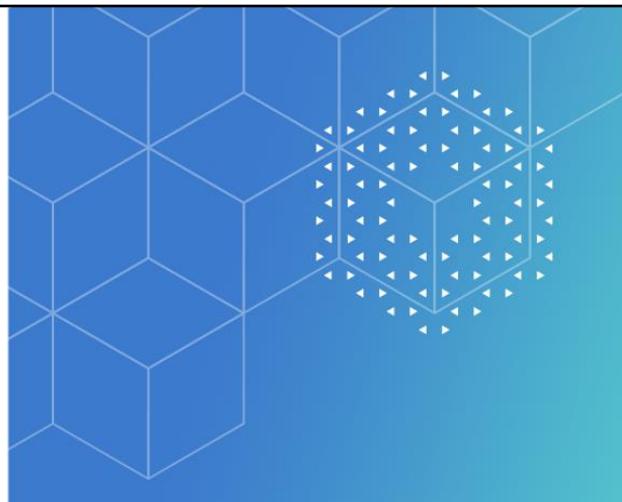


45 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to use basic Verilog constructs to describe a simple design.

For this lab, you use basic Verilog constructs to describe a parameterized-width two-to-one multiplexor. The lab instructions explain how to declare module parameters and vector ranges.



Module 4

Choosing Between Verilog Data Types

cadence®

This module examines the Verilog value set and data types. It explores nets, variables, and a form of constant called a parameter.

Module Objective

In this module, you:

- Choose and use the Verilog data types correctly

Topics

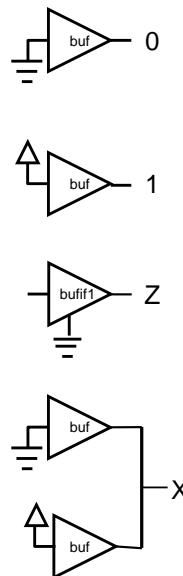
- Logic values
- Net and Register type rules and example
- Declaring vectors
 - Truncation and padding
- Defining literal values
- Declaring nets
- Declaring variables
- Declaring arrays of nets and variables
- Declaring module parameters



Your objective is to appropriately choose and effectively use the Verilog data types. To do that, you need to know what values can be represented, and how to represent those values using literals, constants, variables, and nets and how to create aggregates.

Value Set

Value	Associated Informal Terms
0	Zero, Low, False, Logic Low, Ground, VSS
1	One, High, True, Logic High, Power, VDD, VCC
Z	HiZ, High Impedance, Tri-State, Undriven, Unconnected, Driver Disabled
X	Uninitialized, Unknown (bus contention)



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



The Verilog value set consists of the four basic values:

- 0 – To represent a logic zero, low, or false condition;
- 1 – To represent a logic one, high, or true condition;
- z – To represent a high-impedance state; and
- x – To represent an unknown logic value.

The simulator initializes most nets to the high-impedance state. The exception is nets of the trireg type, which because they represent capacitive nets, initialize to the unknown value. Upon commencing the simulation, the simulator propagates the values of net drivers onto the nets. A net that has no drivers will remain at its initialized value throughout the simulation. This situation is usually the result of user error, as there is seldom good reason to leave a net undriven.

The simulator initializes most variables to the unknown value. The exception is variables of the real type, which it initializes to 0 because it is the only type that cannot hold high-impedance or unknown values. A variable that is never assigned a value will remain at its initialized value throughout the simulation. This situation is usually the result of user error, as there is seldom good reason to declare a variable and not use it.

The appearance during simulation of a high-impedance value on a net is usually due to its drivers being disabled. In real hardware this situation either has a short duration or does not occur because bus keepers pull the net to a high or low logic state.

The appearance during simulation of an unknown value on a net is usually due to a clash between drivers driving different values. In real hardware, this situation will either not exist or have an extremely short duration.

The appearance during simulation of a high-impedance value on a variable is due to an assignment of that value to the variable, either deliberately because the variable represents one of the drivers of a bus net, or by assigning the value of a net to the variable.

The appearance during simulation of an unknown value on a variable is due to an assignment of that value to the variable, either deliberately because the simulator cannot resolve the value of the assigned expression, or by assigning the value of a net to the variable. In real hardware the variable will assume the 0 or 1 state.

Data Types

Verilog provides three *groups* of value objects and different types in each group:

- **Nets**
 - Represent physical connection between structures and objects
 - supply0, supply1, tri/wire, tri0, tri1, triand/wand, trior/wor, trireg
- **Variables**
 - Represent abstract storage elements
 - integer, real, reg, time, realtime
- **Parameters**
 - Run-time constants
 - localparam, parameter, specparam



Procedures communicate by passing events, and by passing data via nets and shared variables informally called *signals*. Verilog does not actually have things called *signals*. Verilog has three *groups* of value objects and only a very few types in each group:

- Verilog has *nets* to represent physical connections between structures and objects.
 - **tri/wire triand/wand trior/wor trireg ...**
- Verilog has *variables* to represent abstract storage elements.
 - **integer real reg time realtime**
- Verilog has two simulation-time constants and an annotatable constant.
 - **Localparam, parameter and specparam**
 - These are constants and is illegal to modify their value at run time.

Net and Register Type Rules

You specify the type upon declaration.

- **Nets** and **reg** are one-bit wide unless you also specify their range.
- A port declaration implicitly declares a one-bit **wire** net unless you explicitly declare it otherwise.

```
// 1995 list of ports syntax
module mux (a, b, sel, op);
    input a, b, sel;
    output op;
    reg op;
    ...

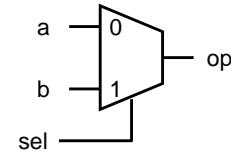
```

Must be variable to assign in procedure

```
// 2001 list of port declarations
module mux (
    input a, b, sel,
    output reg op
);
    ...

```

```
always @*
    if (sel == 1)
        op = b;
    else
        op = a;
```



Rules govern your use of data types:

- Variables can only be driven inside procedures.
- Nets are driven everywhere else (outside procedures).
- Constants are for unchanging or instance-specific values.



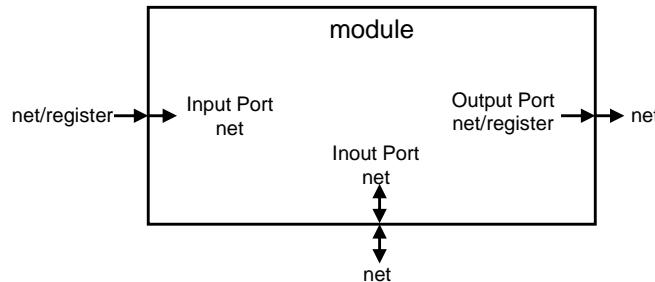
- A data item has associated with it the data values it can have and rules for how you use it.
- A net is the recipient of its drivers' values.
- A variable is an item you procedurally assign values to.
- A port is a net or variable that the instantiating module can connect its own net or variable to.
- A parameter is like a variable but has a constant value.
- Ports, nets, and variables of the **reg** type are a single bit unless you declare them with a range.

These module headers show two ways to declare ports:

- You can list the ports in the module header and later declare them as a module item; and
- As of the Verilog 2001 update, you can directly declare the ports in the module header.

Type Rules in Connectivity

- Module input ports are always nets.
- Module output ports are variables if driven by a procedural block, or nets in all other cases.
- Connections to the input ports of a module instance are variables if driven by a procedural block, or nets in all other cases.
- Connections to the output ports of a module instance are always nets.
- Connections to bidirectional inout ports are always nets.



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

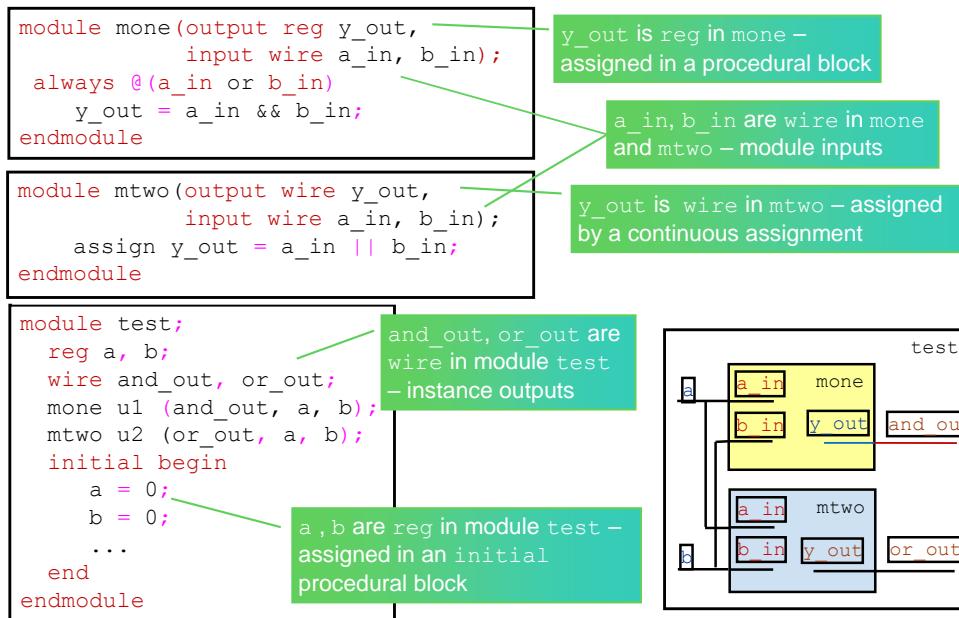


A port permits an external net or variable to connect to an internal net or variable, with restrictions.

Here are some common user errors, along with their typical error messages:

- You make a procedural assignment to a signal that you either declared as a net or you forgot to declare so it is thus implicitly a net.
 - This is an illegal assignment.
- You connect an output from an instance to a signal declared as a register. This is illegal as a module output can only drive a net.
 - This is an illegal output port specification.
- You declare a signal as a module input and as a register. This is illegal as inputs can only be nets.
 - These are incompatible declarations.

Net and Register Type Example



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

cadence®

Here, we are using fully defined Verilog-2001 ANSI-C syntax for module port declarations for clarity.

In both module mone and mtwo, the input ports a_in and b_in must be declared as net data types.

In module mone, the output y_out must be declared as a register data type as it is assigned from the always procedural block.

In module mtwo, the output y_out must be declared as a net data type as it is driven from the continuous assign statement.

In module test, a and b must be declared as register data types as they are assigned from the initial procedural block, but and_out and or_out must be declared as net data types as they are driven from the instance output ports.

Therefore a connection like a -> a_in, or y_out -> and_out actually changes data type as it crosses the module boundary.

Declaring Vectors

A vector is a net or reg with a range specification.

You specify the vector's range when declaring the variable:

- [msb_constant_expression : lsb_constant_expression]
- Range can be ascending or descending.
- Bounds can be negative, zero or positive.
- Bounds must be constant expressions.
- Individual bits can be selected from a vector.

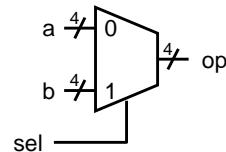
```
module mux4 (
    input wire [3:0] a, b,
    input wire sel,
    output reg [3:0] op
);
    4-bit vector reg /

always @ (a or b or sel)
    if (sel == 1)
        op = b;
    else
        op = a;

endmodule
```



[width-1:0] is the de-facto standard.



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

Ports, nets, and variables of the reg type are a single bit unless you declare them with a range.

To declare a multiple bit port, net, or reg variable, you need to specify a range. The range provides addresses for the individual bits. The only restriction on the range bounds is that they must be constant expressions. Either or both bound can be negative, zero, or positive, and the range can be ascending or descending.

Using Vector Ranges

Access vector selections whatever way you declared the range.

- Select one or more contiguous bits.

```
input  [3:0] inp;
output [3:0] outp;
assign outp = inp;

// outp[3] <- inp[3]
// outp[2] <- inp[2]
// outp[1] <- inp[1]
// outp[0] <- inp[0]
```

```
input  [3:0] inp;
output [3:0] outp;
assign outp[3] = inp[0];

// outp[3] <- inp[3]
// outp[2] <- inp[2]
// outp[1] <- inp[1]
// outp[0] <- inp[0]
```

```
reg [3:0] ireg;
reg [0:3] oreg;
always @(...) begin
    oreg = ireg;

// oreg[0] <- ireg[3]
// oreg[1] <- ireg[2]
// oreg[2] <- ireg[1]
// oreg[3] <- ireg[0]
```



This page does not contain notes.

Assigning Between Different Widths

Vector widths do not need to match in an assignment!

- Source wider than target truncates value from msb.
- Unsigned source shorter than target zero-extends value.
- Signed source shorter than target sign-extends value.
 - Selections and concatenations are not considered signed.

```
reg [3:0] zbus;      // 4 bits
reg [5:0] widebus; // 6 bits
always @(...) begin
  zbus = widebus; // same as
  zbus = widebus[3:0];
  //           ← widebus[5]
  //           ← widebus[4]
  // zbus[3] ← widebus[3]
  // zbus[2] ← widebus[2]
  // zbus[1] ← widebus[1]
  // zbus[0] ← widebus[0]
```

```
reg [3:0] zbus;      // 4 bits
reg [5:0] widebus; // 6 bits
always @(...) begin
  widebus = zbus; // same as
  widebus = {2'b00, zbus};
  // widebus[5] ← 0
  // widebus[4] ← 0
  // widebus[3] ← zbus[3]
  // widebus[2] ← zbus[2]
  // widebus[1] ← zbus[1]
  // widebus[0] ← zbus[0]
```

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



You can assign between vectors of different widths:

- Verilog truncates the leftmost bits when assigning a wider vector to a narrower vector. If you want to select some range other than the rightmost bits then you must specify that range.
- Verilog pads the leftmost bits with 0 when assigning a narrower vector to a wider vector. If you want to assign something other than 0, then you need to use the concatenation operator to construct a wider expression.

Defining Literal Values

You can specify a literal value as:

`<size>'<base><value>`

- **size** is an optional positive decimal number of bits:
 - If unsized is at least 32 bits.
- **base** is a character to indicate binary, octal, decimal, or hexadecimal radix.
 - B/b, O/o, D/d, H/h
 - Verilog-2001: Optionally preceded by an “s” character to indicate a signed value.
- **value** is legal digits for base:
 - Can include “_” if not 1st character.
 - Can include Z/z and X/x digits if base is binary, octal, hexadecimal.
 - Verilog-2001: can be single Z/z or X/x digit if decimal base.
 - **value** is itself an unsigned number.

```
...
reg [3:0] zbus;
...
zbus = 4'b1001; // 1001
zbus = 4'o05; // 0101
zbus = 4'd14; // 1110
zbus = 4'h2f; // 1111
...
```



Appropriately sizing the literal avoids padding and truncation!

More Examples

8'b1100_0001	8-bit binary
9'o017	9-bit octal
10'd1000	10-bit decimal (unsigned)
16'hff01	16-bit hexadecimal
12	32-bit decimal (signed)
'h83a	32-bit hexadecimal

You specify integer literals in binary, octal, decimal or hexadecimal format.

- You can specify a decimal integer literal as an optional sign followed by a sequence of decimal digits. A number in this format is a signed number.
- You can specify any integer literal as an optional sign followed by an optional size followed by a single quote followed by one or two base format characters and then followed by a sequence of digits appropriate to the radix.
 - A literal that you do not size has a size of at least 32 bits and most implementations make it exactly 32 bits.
 - There must be no white space between the single quote character and the base format.
 - The base format is not sensitive to character case and consists of a b, o, d or h character to indicate the radix. The Verilog 2001 update allows an optional preceding s character to indicate a signed value.
 - The digit sequence is not sensitive to character case and consists of digits appropriate for the radix. For the binary, octal or hexadecimal radices, you can use the z and x characters for any or all digits. The Verilog 2001 update also permits a decimal value to be a single z or x character to indicate a value that is all high impedance or all unknown. Except for the first digit, you can insert underscore (_) characters anywhere you need them to improve readability. You can substitute the question mark (?) character for a z character. The reason for this will become obvious when you study the case statement.

Automatic Extension of Unsigned Literals

For **sized literals** (e.g., 1'b1):

- Verilog pads to left with 0 to match size of wider expression.

For **unsized literals** (e.g., 'b1):

- Size is 32 bits.
- If leftmost digit is 1 or 0:
 - Verilog left-fills with 0 to make 32 bits and pads to left with 0 to match size of wider expression.
- If leftmost digit is Z or X:
 - Verilog left-fills with that leftmost digit to make 32 bits.
 - Verilog-1995 then pads to left with 0 to match size of wider expression.
 - Verilog-2001 then pads to left with that leftmost digit to match size of wider expression.

```
reg [7:0] a;
...
a = 8'b11; // 00000011
a = 8'o11; // 00001001
a = 8'd11; // 00001011
a = 8'h11; // 00010001
a = 'b11; // 00000011
a = 'o11; // 00001001
a = 11; // 00001011
```

```
reg [11:0] b;
...
b = 12'hzzz; // zzzzzzzzzzzz
b = 12'hz; // zzzzzzzzzzzz
b = 12'h0z; // 00000000zzzz
b = 12'hz0; // zzzzzzzz0000
```

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



The Verilog 2001 standard permits signed based literals. The sign bit can be 0,1, z or x. Verilog 2001 sign-extends signed based literals to match the width of the enclosing expression.

Verilog zero-extends unsigned based literals except in one situation. If the provided value has fewer digits than the size indicates and the leftmost bit is z or x, then Verilog extends the provided value with that z or x bit value up to the size of the literal. This made it easy to fill an entire vector with z or x. The standard guarantees that the size of an unsized literal is at least 32 bits and most implementations made it exactly 32 bits. What follows is the tricky part: The Verilog 1995 standard then zero-extended that 32-bit expression to match the width of the enclosing expression. The Verilog 2001 update continues to extend the z or x up to the width of the enclosing expression.

Variable Vector Selection

Bit-Select

- A single bit-select index may be variable.

Range-Select

- In Verilog 1995, range-select indices must be constant.
- In Verilog 2001, a range-select can use a variable index and a constant width:
 - Base expression (variable)
 - Width expression (constant)
 - Offset direction
 - Positive +
 - Negative -
 - Offset indicates if the width is added or subtracted from the base.

```
reg [7:0] vect;
reg [1:0] slice;
reg [2:0] index;
initial begin
  vect = 8'b10011001;
  index = 4;
  slice[0] = vect[index];
  ...

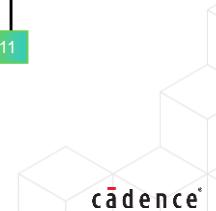
```

```
index = 4;           11
slice = vect[4:3];  ✗ Error Range select bounds
slice = vect[index:index-1]; cannot be variable
```

```
index = 4;
slice = vect[index -: 2];
```

Verilog 2001 allows constant width from variable index

$[base_expr +: width_expr]$
 $[base_expr -: width_expr]$



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

Individual bits can be selected from a vector using a variable or expression index.

However in extracting a range of bits from a vector, Verilog 1995 requires that both bounds of the range are constants or constant expressions. So a variable cannot be used in the extraction of a range of bits.

- identifier [msb_constant_expression:lsb_constant_expression]

Verilog 2001 provides an alternative syntax that permits the starting point of a range select to be a variable or variable expression, as long as the number of bits selected is a constant. In effect, to select a constant-width range from a variable starting bit position. The constant offset can be positive or negative from the starting point.

*identifier [base_expression +: width_constant_expression]
 identifier [base_expression -: width_constant_expression]*

Where:

base_expression is the variable starting bit position.

width_constant_expression is a constant plus or minus offset.

Declaring Nets

Nets behave like physical wires.

Values are driven onto them.

Verilog has several net types.

- Most commonly used is **wire**.

Verilog implicitly declares wires:

- Verilog-1995: Undeclared identifiers connected to instance ports.
- Verilog-2001: Also undeclared identifier driven by continuous assignment.
- Continuous assignments and instance output/inout ports drive nets.
- Change default with compiler directive.
 - **`default_nettpe <nettpe>**

Net Types

```
wire, tri           // float to z
wor, trior         // wired-or
wand, triand       // wired-and
tri0, tril         // float to 0,1
trireg            // capacitive
supply1, supply0  // power rails
uwire              // unresolved
```

Examples

```
wire a, b, sel;    // Scalar wires
wire [31:0] w1;   // Vector wire
wand c;          // Scalar wired-and
tri [15:0] busa; // Vector bus
```

```
module halfadd (
  input a, b, output sum, carry );
  // a & b are wire by default
  // drive by continuous assign
  assign sum = a ^ b;
  assign carry = a & b;
endmodule
```

Nets are continuously driven by their drivers. Verilog automatically propagates a new value onto a net when the drivers of the net change value. A net driver can be a continuous assignment or a module or primitive output.

Verilog has various net types for modeling design-specific and technology-specific functionality. The most common net type is a wire.

Verilog implicitly declares wire nets in some contexts in which you use a net without first declaring it. You can override this using the **`default_nettpe <net_type>** compiler directive. With this directive, such undeclared nets default to the net type you specify in the compiler directive. For this directive, Verilog 2001 adds the none net type. In other words, do not allow implicit net declarations.

Net Types	Functionality
wire, tri	For standard interconnection wires (default)
supply1	For power rails only in netlists
supply0	For ground rails only in netlists
wor, trior	For multiple drivers that are Wire-ORed
wand, triand	For multiple drivers that are Wire-ANDed
trireg	For nets with capacitive storage
tri1, tri0	For nets that pull up or down when not driven
uwire	Verilog 2005 unresolved wire accepts only one driver

Undeclared Identifier in Verilog

- A net is inferred for undeclared identifier by default in Verilog 2001.
- What's the issue?
 - `svm` is undeclared, so defaults to a single bit wire (implicit wire).
 - No compilation error/warning message.
 - `sum` output port is un-driven and so becomes value Z.
 - Mistyped identifier defaults to net/wire.

- What is the solution?
 - Check simulation carefully!
 - Use Verilog2001 ``default_nettpe none` which gives compilation error on `svm`.

```
module halfadd (
    input a, b, output sum, carry );
    // a & b are wire by default
    // drive by continuous assign
    assign sum = a ^ b;
    assign carry = a & b;
endmodule
```

Undeclared identifier
defaults to wire.

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



- Along with a typo, even lower case and uppercase identifiers (user-defined name of some such as the module, wire, variable, or the name of a function) are perceived as different in Verilog. If an identifier is typed in uppercase by mistake, and is used on the left-hand side of a continuous assignment, then an implicit net declaration is inferred, and no error or warning is reported.
- No error/warning is reported if undeclared identifier is used for connecting an instance of a module (makes design debug difficult).

For the compiler directive ``default_nettpe <net_type>`, Verilog 2001 adds the `none` net type for compiler directive. In other words, it does not allow any implicit net declarations.

Usage of `default_nettype none

- Implicit wires can be avoided by using the `default_nettype none compiler directive.
- Any undeclared identifier is a compilation error/warning when this directive is used.
- Directive can affect the compilation of other files once turned on.

```
'default_nettype none
module halfadd (
    input a, b, output sum, carry );
    // a,b,sum,carry are implicit wires
    assign sum = a ^ b;
    assign carry = a & b;
endmodule
```

Compilation warning on a, b, sum and carry

ERROR: Undeclared identifier

- Explicit declaration of identifier type is needed to avoid the error. This can be:

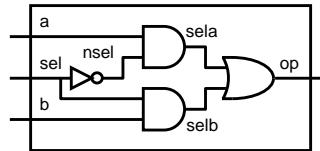
```
input wire a, b, output wire sum, carry
```



When ‘**default_nettype none**’ compiler directive is used, implicit data types are disabled. This can make any undeclared identifier name a syntax error. Hence, as a limitation of this directive, benefits of implicit data types declarations are lost. Another limitation is that once this compiler directive turned on in one file, then the compilation of other files also gets affected. To avoid this, the directive ‘**default_nettype wire**’ should be added at the end of each file where implicit nets have been turned off using `default_nettype none`.

Merging Net Declaration and Assignment

You can merge the net declaration and assignment – known as “net declaration assignment”.



```

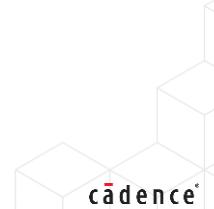
module mux (a, b, sel, op);
  input sel, b, a;
  output op;
  wire nsel, sela, selb;
  assign nsel = ~sel; Bitwise NOT
  assign selb = sel & b;
  assign sela = nsel & a;
  assign op = sela | selb;
endmodule
  
```

Bitwise OR

```

module mux (a, b, sel, op);
  input sel, b, a;
  output op;
  wire nsel = ~sel;
  wire selb = sel & b;
  wire sela = nsel & a;
  assign op = sela | selb;
endmodule
  
```

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



You can merge the net declaration with an assignment to make a net declaration assignment.

The first example explicitly declares the wires `nsel`, `sela` and `nselb` and later makes continuous assignments to them.

The second example merges the wire declaration with a continuous assignment. You can still make additional continuous assignments to the nets. Remember that a net can have multiple drivers.

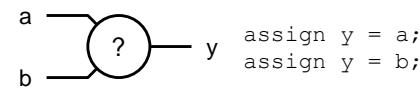
Ports that you do not declare otherwise are by default single-bit wire nets. Both examples also show a continuous assignment to the output port wire net.

Resolving Multiple Drivers of a Net

Multiple drivers can drive a net.

- Exception: Verilog-2005 **uwire**

Only a net can resolve the value of multiple drivers.



Two names
- same type
wire y;
tri y;

		b		
		0	1	Z X
a	0	0	X	0 X
	1	X	1	1 X
	Z	0	1	Z X
	X	X	X	X

Two names
- same type
wand y;
triand y;

		b		
		0	1	Z X
a	0	0	0	0 0
	1	0	1	1 X
	Z	0	1	Z X
	X	0	X	X X

Two names
- same type
wor y;
trior y;

		b		
		0	1	Z X
a	0	0	1	0 X
	1	1	1	1 1
	Z	0	1	Z X
	X	X	1	X X

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



Verilog resolves the value of a net driven by multiple drivers. Assuming that the drivers all have the same strength:

- The conflict of 0 and 1 values on a wire net results in an unknown value; and
- The conflict of 0 and 1 values on a wired-logic net results in either a 0 or a 1 value.

The wire, wired-and (wand) and wired-or (wor) net types have two names. You can alternatively refer to them as tri, triand, and trior types, respectively. Net type wire and tri have identical functionality. They can be used in different names for better readability. That is, tri can be used in places where multiple drivers are present. Urban legend claims that the “tri” names exist to remind the user that they can assign high-impedance values to these nets. The urban legend neglects to explain the tri0, tri1, and trireg nets that cannot have high-impedance values.

The wired-logic nets exist to model technology-dependent logic conflict resolution:

- The wired-AND net type to model open collector logic
- The wired-OR net type to model emitter-coupled logic

Verilog 2005 added the uwire net type, an unresolved wire that allows no more than one driver.

The uwire requires the –sv switch for the Cadence INCISIV Simulation tool.

Declaring Variables

Variables hold values that you procedurally assign. They keep the value until you assign a different value.

- **reg** – 4-state unsigned 1-bit
 - Can declare multi-bit vector
 - Verilog-2001 adds **reg signed**
- **integer** – 4-state signed 32-bit
- **time** – 4-state unsigned 64-bit
- **real** – Double-precision float
- **realtime** – Same as real

Examples

```
reg m, n, op; // Scalar reg
reg [31:0] rv; // Vector reg
integer i; // integer
time timeA; // time
real pi; // real
```

```
module mux (
    input a, b, sel,
    output reg op
);

always @ (a, b, sel)
    if (sel == 1)
        op = b;
    else
        op = a;

endmodule
```



A variable retains its value until you assign a new value to it.

- The reg type holds a four-state integral value. A reg is by default unsigned but with the Verilog 2001 update you can declare it signed. Arithmetic operations treat the value of a signed reg as a signed value. A reg is by default single bit wide, but you can declare an arbitrary range for it. The reg is the most common variable type for hardware design.
- The integer type holds a four-state signed integral value. The standard guarantees it to be at least 32 bits and most implementations make it exactly 32 bits. Arithmetic operations treat the value of an integer as a signed value.
- The time type holds a four-state unsigned integral value. The standard guarantees it to be at least 64 bits and most implementations make it exactly 64 bits. A time is always unsigned. It is exactly the same as a 64-bit unsigned reg vector. People typically use variables of this type to hold simulation time values.
- The real type holds a double-precision floating-point value as described by IEEE Std 754-1985 Binary Floating-Point Arithmetic.
- The realtime type is another name for real. If you use it that way, you can use a realtime to remind yourself that its value is a real version of the simulation time.

Making Assignments to Variables

You *read* a variable's value from inside or outside a procedure.

You *write* a variable's value only from *inside* a procedure.

Within a procedure you write *only* to variables.

```
module mux (
    input a, b, sel,
    output op // wire by default
);

    always @ (a, b, sel)
        if (sel == 1)
            op = b;
        else
            op = a; └─ ERROR: Attempt to
                      procedurally assign a
                      net!
endmodule
```

```
module mux (
    input a, b, sel,
    output reg op
);
    assign op = sel ? b : a; └─ ERROR: Attempt to
                           continuously assign
                           a variable!
endmodule
```

Conditional Operator

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



A variable can appear anywhere that you read its value.

Assignments to variables are procedural assignments. They exist only as a procedural statement.

The first example erroneously attempts to make a procedural assignment to a net. Remember that a port is implicitly declared as a single-bit net unless you declare it otherwise.

The second example explicitly declares the port to be a reg, but then erroneously attempts to make a continuous assignment to the variable.

Assigning Between Integers and Vectors

Reminder

- **reg** – 4-state unsigned any width
 - Verilog-2001 adds **reg signed**
- **integer** – 4-state signed 32 bit

Assignment between **reg** and **integer** follows previous rules:

- If value wider than target:
 - Truncates value
- If value shorter than target:
 - If *unsigned* value
 - Zero-extends
 - If *signed* value
 - Sign-extends

```
integer i;
reg [3:0] r;

initial
begin
    i = 17; // 00...10001
    r = i; // 0001
    i = r; // 00...00001
    i = -3; // 11...11101
    r = i; // 1101
    i = r; // 00...01101
end
```

Truncated
Truncated
Signage Lost



What is the result if "r" is signed?



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

You saw previously that Verilog truncates wide literal constants to fit shorter target widths and extends shorter literal constants to fit wider target widths, extending unsigned values with zero and signed values with the sign bit. This is exactly the case also for assignments between integers and vector reg variables.

This example assigns the value 17 to an integer variable and assigns the integer to a 4-bit unsigned vector reg variable. The assignment is a bit-for-bit replacement, so the vector reg variable gets the value 1. Upon assigning the value of the vector reg variable to the integer variable, Verilog first zero-extends the value to match the width of the integer variable, which is almost always 32 bits.

The example then assigns the value -3 to the integer variable and assigns the integer to the 4-bit unsigned vector reg variable. The assignment is a bit-for-bit replacement, so the unsigned vector reg variable gets the value 13. The signage is lost. Upon assigning the value of the vector reg variable to the integer variable, Verilog zero-extends the value to match the width of the integer variable.

Declaring Arrays of Nets and Variables

Verilog-1995 supports one-dimensional arrays of **integer**, **reg**, and **time**.

```
integer int_a [0:99]; // array of 100 integer  
reg [7:0] reg_a [0:99]; // array of 100 8-bit vectors
```

- You access one element (word) at a time by indexing to that word.
- You cannot directly take bit or part selects of a word (use an intermediate variable).

```
reg [7:0] word, array [0:255]; // memory word and array  
reg bit;  
...  
word = array[5]; // access address 5  
word = array[10]; // access address 10  
bit = word[7]; // access bit 7 of extracted word
```

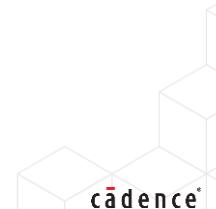
Verilog-2001 supports multi-dimensional arrays of **integer**, **real**, **realtime**, **reg**, **time**, and **nets**.

```
reg reg_b [0:99] [7:0]; // 100x8 array of 1-bit elements
```

- You can directly take bit and part selects of an indexed element.

```
reg [7:0] array [0:255]; // memory array  
bit = array[10][7];
```

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



As with programming languages, Verilog supports arrays:

- The Verilog 1995 standard permits one-dimensional arrays of integer, time and reg. The reg elements can be either single-bit elements or vector reg elements. People refer to an array of vector reg elements as a Verilog “memory”. The Verilog 2001 standard supports arrays of integer, time, reg, real and even nets, and they can have any number of dimensions.
- The Verilog 1995 standard did not accommodate directly taking a bit-select or a part-select of an indexed memory element. You had to extract the element to a variable and then take a bit-select or part-select of the variable. The Verilog 2001 standard does accommodate directly taking a bit-select or a part-select of an indexed memory element.

Declaring Module Parameters

A module **parameter** is an instance-specific constant:

- Parameterizes the module definition
 - Width, depth, frequency, time ...
- Can override for each individual instance

```
// Example parameter list
parameter INTEGER_P = 8,
          REAL_P    = 2.039,
          VECTOR_P  = 16'bx;
```



Uppercase constants improve readability!

```
// Verilog-1995 syntax
module mux (a, b, sel, op);
  parameter WIDTH = 2;
  input [WIDTH-1:0] a, b;
  input              sel;
  output [WIDTH-1:0] op;
  reg    [WIDTH-1:0] op;
  ...
endmodule
```

```
// Verilog-2001 alternative
module mux
#(parameter integer WIDTH = 2)
  (input wire [WIDTH-1:0] a, b,
   input wire              sel,
   output reg   [WIDTH-1:0] op);
  ...
endmodule
```

 Verilog-2001 allows you to specify a parameter type and to sign and size vector parameters.

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



Module parameters that you declare with the parameter keyword are constants that you can change for each instance of the module, thus you can use them to “parameterize” the instance, for example, to establish different widths and depths for each instance of a memory module. Parameters are not variables, so you cannot change them during the simulation run time.

The first example uses the Verilog 1995 list of ports syntax and declares the ports later as a module item. Before declaring the ports, it declares a module parameter to establish the port width and sets the parameter’s default value to 2. You can modify this parameter for each individual instance of the module.

The second example uses the Verilog 2001 list of port declarations syntax. The example needs to declare the parameter before the ports because it uses the parameter value to establish the port width. It declares the parameter using the Verilog 2001 module parameter port list syntax.

The Verilog 1995 syntax does not require a type declaration for a parameter – the parameter assumes the type of its initial value and you can change its type upon module instantiation by providing it a value of a different type. The Verilog 2001 syntax permits a type declaration for a parameter and any new value you provide it must be of that type.

Local Parameters and Parameter Passing

Localparams are true constants. Unlike parameters, localparams cannot be overridden from the next level of hierarchy.

- Use for constants that should never be overridden upon instantiation
 - For a module that is not instantiated (testbench?)
 - For “enumerations” (FSM states?)

Verilog-2001 provides three ways to override module parameters:

- Redefinition using `defparam`
- Positional parameter override during instantiation
- Named parameter override during instantiation
 - New in Verilog-2001



Parameter override using `defparam`
can be difficult to track.

```
Verilog-1995
module us_mult (a,b,product);
parameter width_a = 5;
parameter width_b = 5;
localparam op_width = width_a + width_b;

input [width_a-1:0] a;
input [width_b-1:0] b;
output [op_width-1:0] product;

assign product = a * b;

endmodule
```

```
Verilog-2001
defparam u2.width_b = 7;

us_mult #(5,7) u1 (.a (a_net),
    .b (b_net), .product (a_b_mult));

us_mult #(.width_a(5)) u1 (.a (a_net),
    .b (b_net), .product (a_b_mult));
```



The Verilog 2001 update provides the ***localparam*** construct – exactly like a parameter except that you cannot change it. For a module parameter that should *not* change on a per-module basis, you should use the ***localparam*** keyword instead of the ***parameter*** keyword.

To modify the parameters of a module instance, you make a *parameter value assignment* upon instantiating the module. A parameter value assignment is a parenthesized list of parameter assignments before the module instance name and prefixed with a hash (“#”) character. You may see the hash character used in other contexts to specify a propagation delay, but in the context of a module instantiation, it is a parameter value assignment or parameter value override during instantiation.

Your list of parameter assignments can use either the Verilog 1995 ordered parameter assignment syntax or the Verilog 2001 named parameter assignment syntax, but you cannot mix the two syntaxes in the same instantiation. These syntaxes are similar to the ordered and named port connection syntax, but with the ordered parameter assignment syntax, you cannot merely insert a comma as a place holder – you need to provide values for all earlier parameters if you want to provide a value for a later parameter. This example uses the ordered parameter assignment syntax, but you should in general use the named parameter assignment syntax to make your code more readable.

- You can also override a parameter value by using a ***defparam*** statement. A parameter can be modified using the ***defparam*** statement during compilation time. You can make this override from anywhere in the design by using a hierarchical parameter name. Your use of this override is superfluous when done locally and can potentially cause much confusion when not done locally as it might be difficult to track.
- ***localparam*** cannot be directly modified by ***defparam*** statements. ***Specparam*** provides timing and delay values and can be modified through SDF annotation only. (Refer Appendix C for SDF annotation.)

Module Summary

A net behaves like a physical wire driven by logic:

- Implicitly declared nets default to type **wire**.
- You drive nets by continuous assignments and by module and primitive outputs.

A variable stores a value:

- You update a variable only by a procedural assignment.
- You make a procedural assignment only to a variable.
- Assignments between **integer** and **reg** ignore signage.

Expressions that read nets and variables can exist inside or outside procedures.

A module port you do not declare otherwise is implicitly declared **wire**.

- Input ports are nets driven externally by nets and/or variables.
- Inout ports are nets connected externally to nets.
- Output ports are nets or variables externally driving nets.



This module examined the Verilog value set and data types. It explored nets, variables, and a form of constant called a parameter.

Module Review

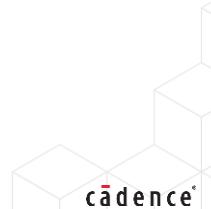
1. What are the four logic values in the Verilog value set?
2. Rewrite the binary value `8'b11010011` in hexadecimal.
3. How many bits wide is the integer type?
4. What is the primary difference between a net and a variable?



This page does not contain notes.

Module Review Solutions

1. What are the four logic values in the Verilog value set?
 - 0 1 z x
2. Rewrite the binary value `8'b11010011` in hexadecimal.
 - `8'hd3`
3. How many bits wide is the integer type?
 - 32
4. What is the primary difference between a net and a variable?
 - A net is continuously driven, behaving like a physical wire driven by logic. A variable represents storage, storing a value until a new one is assigned. In the language, you can assign variables only from procedural statements and nets only from continuous assignments.



This page does not contain notes.

Module Exercise

Code this two-input AND operation using a: (i) **wire** (ii) **reg**.



This page does not contain notes.

Module Exercise Solution

Code this two-input AND operation using a: (i) **wire** (ii) **reg**.



Solution:

```
wire y = a & b;
```

```
reg y;  
always @* y = a & b;
```



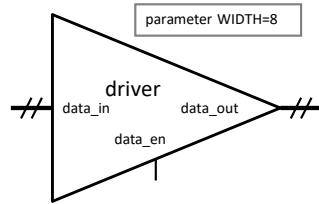
This page does not contain notes.



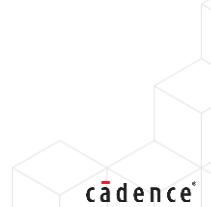
Lab

Lab 4-1 Modeling a Data Driver

- Use a Verilog literal value while describing a parameterized-width bus driver.

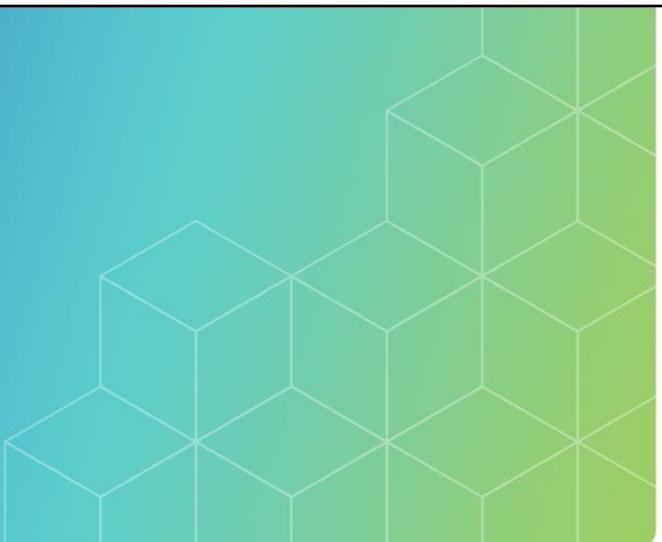
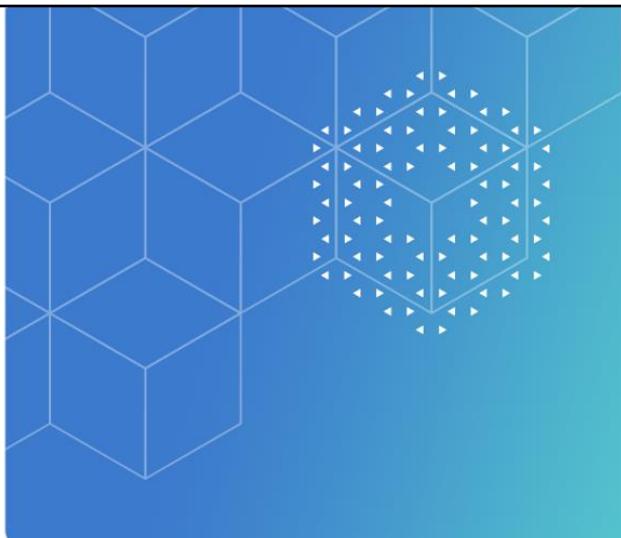


75 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to appropriately choose and correctly use the Verilog data types.

For this lab, you use a Verilog literal value while describing a parameterized-width bus driver.



Module 5

Using Verilog Operators

cadence®

This module provides an explanation and example for each Verilog operator.

Module Objective

In this module, you:

- Choose and use the Verilog operators correctly

Operators

Category	Symbol
Bit-wise	\sim & \wedge $\wedge\wedge$
Reduction	& $\sim\&$ $\sim $ \wedge $\wedge\wedge$ $\wedge\sim$
Logical	! &&
Arithmetic	$**$ * / % + -
Shift	$<<$ >> $<<<$ >>>
Relational	< > \leq \geq
Equality	\equiv != $\equiv\equiv$!=
Conditional	?:
Concatenation	{}
Replication	{()}

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



This page does not contain notes.

Bit-Wise Operators

not	<code>~</code>
and	<code>&</code>
or	<code> </code>
xor	<code>^</code>
xnor	<code>^ ^</code>
xnor	<code>^ ~</code>

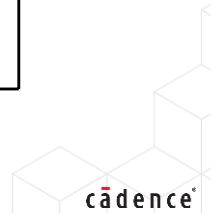
- Bit-wise operators operate on vectors.
- Operations are performed bit by bit on individual bits.
- Unknown bits in an operand do not necessarily lead to unknown bits in the result.

```
module bitwise;
  reg [3:0] rega, regb, regc;
  reg [3:0] num;

initial
  begin
    rega = 4'b1001;
    regb = 4'b1010;
    regc = 4'b11x0;

    num = ~rega;           // num = 0110
    num = rega & 0;        // num = 0000
    num = rega & regb;    // num = 1000
    num = rega | regb;    // num = 1011
    num = regb & regc;    // num = 10x0
    num = regb | regc;    // num = 1110
  end

endmodule
```



The bitwise operators perform logical operations in a bitwise manner.

The bitwise unary negation operator inverts the logical sense of each bit of its operand. Each 0 becomes 1 and each 1 becomes 0, and each high-impedance bit becomes unknown.

The bitwise binary operators first zero-extend a smaller operand to match the width of a larger operand, and then perform logical operations on individual bit positions. Depending upon the operation, bits in one operand that are 0 or 1 can mask bits in the same position of the other operand that are high-impedance or unknown, so unknown bits in an operand do not necessarily produce unknown bits in the result.

Unary Reduction Operators

not	&
or	
xor	^
nand	~&
nor	~
xnor	~^ ~~

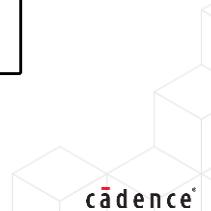
- Reduction operators perform a bit-wise operation on all the bits of a single operand.
- The result is always 1'b0, 1'b1 or 1'bx.

```
module reduction;
localparam [3:0] CONST_A = 4'b0100,
                CONST_B = 4'b1111;

reg val;

initial
begin
    val = &CONST_A; // 0
    val = |CONST_A; // 1
    val = &CONST_B; // 1
    val = |CONST_B; // 1
    val = ^CONST_A; // 1
    val = ^CONST_B; // 0
    val = ~|CONST_A; // 0
    val = ~&CONST_A; // 1
    val = ^CONST_A & &CONST_B; // 1
end

endmodule
```



Unary reduction operators operate on all bits of a single operand to produce a single-bit result. The effect is as if they first applied the logical operation to the first two bits of the operand, then iteratively applied the logical operation to the current partial result and the next bit. The result of the operation is always a single bit that is 0, 1, or unknown (x).

You will see these same operators also used as bitwise binary operators. When used with a single operand, they are reduction operators.

Logical Operators

not	!
and	&&
or	

Logical operators interpret their operands as either true (`1'b0`) or false (`1'b1`) or unknown (`1'bX`).

0 – If all bits 0

1 – If any bit 1

X – If any bit is Z or X and no bit is 1

```
module logical;
localparam integer FIVE = 5;
localparam [3:0] CONST_A = 4'b0011,
CONST_B = 4'b10xz,
CONST_C = 4'b0z0x;

reg ans;

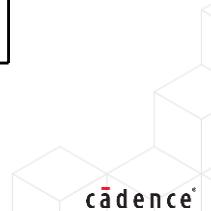
initial
begin
    ans = !CONST_A;           // 0
    ans = CONST_A && 0;       // 0
    ans = CONST_A || 0;       // 1
    ans = CONST_A && FIVE;   // 1
    ans = CONST_A && CONST_B; // 1
    ans = CONST_C || 0;       // X
end

endmodule
```



Logical and bitwise operators are identical for 1-bit expressions, but very different for vectors.

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



Logical operators reduce each operand to a single bit, and then perform a single bit operation.

The rules for reduction of an operand are as follows:

- If the operand contains all zeroes, it reduces to logic 0.
- If the operand contains any ones, it reduces to logic 1.
- If the operand contains no ones, but does contain one or more high-impedance or unknown values, it reduces to unknown, because its logical value is unknown.

The unary logical negation operator then inverts the logical sense of its operand. 0 becomes 1 and 1 becomes 0.

The binary conjunction and disjunction operators produce the logical conjunction and disjunction, respectively, of their operands.

Design Tip: Generally logical operators should be avoided unless explicitly required. It is very easy to get into the habit of using logical operators everywhere. They are identical to bitwise operators for single bit expressions, but give very different results when using vectors.

Arithmetic Operators

Verilog-1995:

Add	+
Subtract	-
Multiply	*
Divide	/
Modulus	%
Unsigned	Based literal, net, reg , time
Signed	Unbased literal, integer

Verilog-2001:

- Exponential Power **

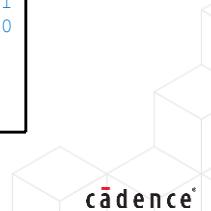
```
module arithops;
    localparam integer CONST_INT = -3,
                CONST_5   = 5;
    localparam [3:0] rega = 3,
                  regb = 4'b1010,
                  regc = 14;

    integer val;
    reg [3:0] num;

    initial
        begin
            val = CONST_5 * CONST_INT; // -15
            val = (CONST_INT + 5)/2; // 1
            val = CONST_5/CONST_INT; // -1
            num = rega + regb; // 1101
            num = rega + 1; // 0100
            num = CONST_INT; // 1101
            num = regc % rega; // 0010
        end

    endmodule
```

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



The Verilog 1995 arithmetic operators are add, subtract, multiply, divide and modulus. Verilog 2001 added the power operator.

This example declares two *integer* parameters, three 4-bit vector *reg* parameters, an *integer* variable and a 4-bit vector *reg* variable, and performs some operations with them. See where integer division discards the fractional part. See where assigning 3 to an unsigned 4-bit vector *reg* keeps the same rightmost four bits but now interprets the value as +13.

Some additional information about arithmetic operators:

- Any z or x bit in either operand produces an unknown (x) result.
- Integer division discards any remaining fractional part.
- Division or modulo by 0 produces an unknown (x) result.
- Raising 0 to a negative power produces an unspecified result.
- Raising a negative value to a real power produces an unspecified result.

Enhanced Signed Arithmetic

Verilog 2001

New reserved word: **signed**

- **signed** keyword treats literal, function, net, reg as signed.

```
32'shFDB97531
function signed [31:0] alu;
wire signed [15:0] addr;
reg signed [31:0] data;
```

- Arithmetic right shift operators maintain the sign of a value.
- **\$signed** and **\$unsigned** system functions cast value.



The parser interprets an integer with no base specifier as a signed value in 2's complement form.

- `intA = -12 / 3 ;`
- The left operand is a 32-bit signed value 12, negated.
- The expression result is -4 .

The parser interprets an integer with an unsigned base specifier as an unsigned value.

- `intA = -'d12 / 3 ;`
- The left operand is a 32-bit unsigned value 12, negated.
- The expression result is 1431655761.

The parser interprets an integer with an signed base specifier as an signed value.

- `intA = -4'sd12 / 3 ;`
- The left operand is a 4-bit pattern of 12 (1100) interpreted as a signed number (-4) and then negated (4).
- The expression result is 1.

Signed Vectors

signed defines vector types as signed quantities.

- 2's complement:
 - Also for function returns.
 - Signed arithmetic for vectors is easier.
 - Assignments between **signed reg** and **integer** types maintain sign information.
 - Vector types can also be defined as **unsigned**.
- By default vectors are **unsigned**.

```
reg [3:0] usreg;  
// 4-bit vector in range 0 to 15  
reg signed [3:0] sreg;  
// 4-bit vector in range -8 to 7
```

```
function signed [15:0] add_2_signals  
...  
endfunction
```

```
module smult (input signed [7:0] opa, opb;  
              output signed [15:0] product);  
  
  assign product = opa * opb;  
  
endmodule
```

```
reg signed [3:0] areg;  
integer aint = -5;  
  
initial begin  
  areg = aint; // areg = -5  
  aint = areg; // aint = -5  
  ...
```

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



For Verilog-2001, you can declare signed vectors. Vectors are by default unsigned. An unsigned vector is treated as signed for accesses through a signed port.

Shift Operators

Verilog-1995:

- **logical shift** `<< >>`
 - Ignores operand signs.
 - Fills extra bits with 0.
 - Implements division or multiplication by powers of two.

Verilog-2001:

- **arithmetic shift** `<<< >>>`
 - Ignores right operand sign.
 - Left shift operates like logical left shift.
 - Right shift preserves the left operand sign if the result is a signed expression.

```
module shift;
  reg [7:0] rega = 8'b10011001;
  reg signed [7:0] regs = 8'b10011001;
  reg [7:0] regb;

  initial
    begin
      regb = rega << 1; // 00110010
      regb = rega >> 1; // 01001100
      regb = regs <<< 1; // 00110010
      regb = regs >>> 1; // 11001100
      regb = rega << -1; // 00000000
    end
  endmodule
```

-1 is $2^{32}-1$

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



The logical shift operators shift the left operand by the number of bit positions given by the right operand interpreted as a positive number, filling vacated bit positions with 0. You can use the logical shift operators to implement integer division or multiplication by powers of 2.

Verilog 2001 added the arithmetic shift operators. The arithmetic left shift operator operates exactly as does the logical left shift operator. The arithmetic right shift operator preserves the sign bit if the resulting expression is signed.

The example initializes an 8-bit vector `rega` to value 153 (8'h99). Left-shifting this value once is equivalent to doubling its value, and placing the result back into an 8-bit vector `reg`. The data truncates and the value gets back down to 50 (8'h32). Right-shifting `rega` value once is equivalent to halving its value and losing the fractional part, so produces 76 (8'h4c). Left-shifting by `-1` is equivalent to left-shifting a very large number of times, so produces 0.

Relational Operators

less than	<
greater than	>
less than or equal to	<=
greater than or equal to	>=

The result is:

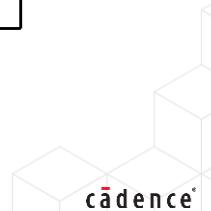
- 1'b0 if the relation is false.
- 1'b1 if the relation is true.
- 1'bX if either operand contains any "Z" or "X" bits.

```
module relational;
  reg [3:0] rega, regb, regc;
  reg val;

initial
begin
  rega = 4'b0011;
  regb = 4'b1010;
  regc = 4'b0x10;

  val = regc > rega ; // val = X
  val = regb < rega ; // val = 0
  val = regb >= rega; // val = 1
  val = regb > regc ; // val = X
end

endmodule
```



If at least one operand is unsigned, the comparison treats both operands as unsigned and zero-extends a smaller operand to match the width of a wider operand.

If both operands are signed integer quantities, the operation sign-extends a smaller operand to match the width of a wider operand and treats the comparison as a signed integer comparison.

If at least one operand is real, the operation if necessary converts the other operand to real and treats the comparison as a real comparison.

The result is 0 if the relation is false and 1 if the relation is true. Any high-impedance (z) or unknown (x) bit in either operand produces an unknown result regardless of the truth of the relation.

Logical Equality and Case Equality Operators

logical equality ==

- Result can be unknown.

	0	1	Z	X
0	1	0	X	X
1	0	1	X	X
Z	X	X	X	X
X	X	X	X	X

```
...
a = 2'b1x;
b = 2'b1x;

if (a == b)
    // values match & do not contain Z or X
else
    // values do not match or contain Z or X
    // above values execute this else branch
```

case equality ===

- Result is always known.

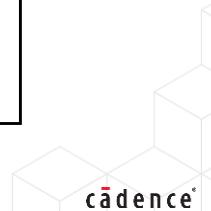
	0	1	Z	X
0	1	0	0	0
1	0	1	0	0
Z	0	0	1	0
X	0	0	0	1



Called "case equality" because the case statement matches items this way.

```
...
a = 2'b1x;
b = 2'b1x;

if (a === b)
    // values match exactly
    // above values execute this if branch
else
    // values do not match
```



The difference between logical equality and case equality is the handling of high-impedance and unknown values. The logical equality treats high-impedance and unknown bits as truly unknown bit values, while the case equality treats high-impedance and unknown bits as values to be matched. A case equality operation thus always produces a 0 or 1 result and never an unknown result.

The case equality operator gets its name from the fact that the case statement, which you will see later, matches items in the same manner. Some people informally call it the “identity” operator because it checks that the bits are identical instead of that their values are the same.

More About Equality Operators

logical equality	<code>==</code>
logical inequality	<code>!=</code>
case equality	<code>==></code>
case inequality	<code>!=></code>

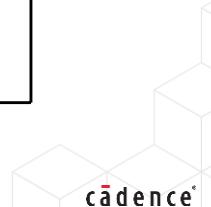
- For logical equalities, the result is always `1'b0`, `1'b1` or `1'bX`.
- For case equalities, the result is always `1'b0` or `1'b1`.

```
module equalities;
  reg [3:0] rega, regb, regc;
  reg val;

  initial
    begin
      rega = 4'b0011;
      regb = 4'b1010;
      regc = 4'b1x10;

      val = rega == regb; // val = 0
      val = rega != regb; // val = 1
      val = regb != regc; // val = X
      val = regc == regc; // val = X
      val = rega ==> regb; // val = 0
      val = rega !=> regb; // val = 1
      val = regb ==> regc; // val = 0
      val = regc ==> regc; // val = 1
    end

  endmodule
```

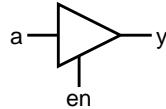


Here is another example illustrating the difference between the logical equality operators and the case equality operators.

The difference between logical equality and case equality is the handling of high-impedance (z) and unknown (x) values. The logical equality treats high-impedance and unknown bits as truly unknown bit values, while the case equality treats high-impedance and unknown bits as values to be matched. A case equality operation thus always produces a 0 or 1 result and never an unknown result.

Conditional Operator

Conditional ?:



```
module tribuf1 (a, en, y);
    input a, en;
    output y;
    assign y = en ? a : 1'bZ;
endmodule
```

```
module tribuf2 (a, en, y);
    input a, en;
    output reg y;
    always @ (a or en)
        y = en ? a : 1'bZ;
endmodule
```

```
module tribuf3 (a, en, y);
    input a, en;
    output reg y;
    always @ (a or en)
        if (en)
            y = a;
        else
            y = 1'bZ;
endmodule
```



Sometimes the conditional operator is more readable than the if statement.
Sometimes not...

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



The conditional operator is a ternary operator. It has three operands.

- If the first operand, the condition expression, is 1, then the operation result is the value of the second operand.
- If the first operand is 0, then the operation result is the value of the third operand.
- If the first operand is unknown (x), then the operation result is the value of the second operand merged with the value of the third operand so that if any bit position has the same value in both operands, then that bit position of the result also has that value.

So the conditional operator selects between two operands like a 2-to-1 multiplexor.

The operands can be any expression. It is very common to nest conditional operations.

Concatenation Operator

concatenation { }

- Can select and join bits from different vectors to form a new vector
 - Forms unsigned expression
- Can reorganize vector bits to form a new vector
 - Endian swaps / reverse / rotate
- Can use on either side of an assignment!



Literals in a concatenation must be explicitly sized so that all the bits go into the correct position.

```
module concatenation;
reg [7:0] rega, regb, regc, regd, new;
reg [3:0] nib1, nib2;

initial
begin
  rega = 8'b00000011;
  regb = 8'b00000100;
  regc = 8'b00011000;
  regd = 8'b11100000;

  new = {regd[6:5], regc[4:3], regb[3:0]};
  // new = 8'b11_11_0100

  new = {2'b11, regb[7:5], rega[4:3], 1'b1};
  // new = 8'b11_000_00_1

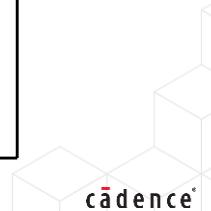
  new = {regd[4:0], regd[7:5]};
  // rotate regd right 3 places
  // new = 8'b00000_111

  {nib1, nib2} = rega;
  // nib1 = 4'0000, nib2 = 4'0011

end

endmodule
```

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



A concatenation operation joins together the bits of one or more comma-separated operands. You must size literal constant operands so that the operation can determine exactly where to place each bit.

Here are some examples that fail to size their operands:

```
a[7:0] = {5'b01010, 2}; //decimal value 2 unsized
c[3:0] = {3'b011, 'b0}; //binary value 'b0 unsized
```

Replication Operator

Replication {{}}

- Reproduces a concatenation a set number of times

- Syntax:

```
{const_expr {sized_expr}}
```

- The constant number of repetitions must not have Z or X values.

```
module replicate;
reg      rega = 1'b1;
reg [1:0] regb = 2'b11;
reg [3:0] regc = 4'b1001;
reg [7:0] bus;

initial
begin

// single bit rega replicated 8 times
bus = {8{rega}};
// bus = 11111111

// 4x rega concatenated with 2x regc[1:0]
bus = { {4{rega}}, {2{regc[1:0]} } };
// bus = 1111_01_01

// regc concatenated with 2x regb
bus = { regc, {2{regb}} };
// bus = 1001_11_11

// regc concatenated with 2x 1'b1
// and replicated 2 times
bus = { 2{regc[2:1]}, {2{1'b1}} } ;
// bus = 00_1_1_00_1_1

end

endmodule
```

A replication operation replicates a concatenation a nonnegative constant number of times. A replication operation applies only to a concatenation. You will not see it in any other context. The replication count cannot have any high-impedance (z) or unknown (x) values.

This example:

- 1st – Replicates the value of the single-bit rega variable value eight times to form an 8-bit value it assigns to the bus variable;
- 2nd – Concatenates four replications of the value of the single-bit rega variable with two replications of the value of the lowest two bits of the regc variable to form an 8-bit value it assigns to the bus variable;
- 3rd – Concatenates the value of the four-bit regc variable with two replications of the value of the two-bit regb variable to form an 8-bit value it assigns to the bus variable; and
- 4th – Concatenates the value of the middle two bits of the regc variable with two replications of the sized literal constant 1'b1, and replicates that 4-bit concatenation twice to form an 8-bit value it assigns to the bus variable.

Reference: Operator Precedence High to Low

Category	Symbol(s)
Unary	+ - ! ~ & ~& ~ ^ ~^ ~~
Exponential	**
Arithmetic	* / %
	+ - (binary)
Shift	<< >> <<< >>>
Relational	< <= > >=
Equality	== != === !==
Bit-wise	& (binary) ^ ~ ~^ (binary) (binary)
Logical	&&
Conditional	? :
Concatenation / Replication	{ } {{ }}
Reliance on operator precedence may make your code unreadable – use parentheses!	

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



Your reliance on operator precedence can make your code unreadable to the great majority of people that do not memorize this table. Your co-workers will appreciate your appropriate use of parenthesized expressions.

Quickly now, without looking at this table, insert parentheses to indicate the grouping of these expressions...

a ? ~ b * c << d : e < f ^ g || h

Give up? Don't you wish it was already parenthesized for you?

a ? ((~b * c) << d) : (((e < f) ^ g) || h)

Module Summary

Now you can appropriately and correctly use Verilog operators. This module presented the following Verilog operators:

- Reduction unary operators that perform a bit-wise operation across all bits of a single operand
- Arithmetic binary operators that preserve signage
- Shift binary operators – the right shift optionally preserves signage
- Relational binary operators
- Logical equality and case equality binary operators
- Bit-wise binary operators that perform individual logical operations on each bit position of two vectors
- Logical unary operator and binary operators that treat their operands as logical values
- Conditional ternary operator that selects between two operands based on the logical value of a third operand
- Concatenation operator that concatenates scalar or vector operands to create a new vector
- Replication operator that replicates a concatenation a fixed number of times



You can now appropriately and effectively use Verilog operators. To do that, you need to know what operators are available and what they do. This module provided an explanation and example for each Verilog operator.

Module Review

1. How wide is the result of a logical `&&` operation?
2. Explain the difference between the `&&` and `&` operators.
3. TRUE or FALSE: You must explicitly size literals in a concatenation.
4. Given the statement `regx = 4'b0101;` what is the value of:
`bus = { 2 {regx[3:1], {3{1'b0,regx[0]}}}} ;`



This page does not contain notes.

Module Review Solutions

1. How wide is the result of a logical `&&` operation?
 - The logical `&&` operator reduces each operand to a single bit (`1'b0`, `1'b1` or `1'bx`), then ANDs these bits together to give a single bit result.
2. Explain the difference between the `&&` and `&` operators.
 - The logical `&&` operator reduces each operand to a single bit, then ANDs these bits to give a single bit result. The bitwise `&` operator does a bit-by-bit AND of its operands, from LSB to MSB, producing a result which is the same width as the longest operand.
3. TRUE or FALSE: You must explicitly size literals in a concatenation.
 - TRUE. All operands in a concatenation must have a size. For example: `{3{'b1}}` is illegal because `'b1` is not sized, while `{3{1'b1}}` is legal.
4. Given the statement `regx = 4'b0101;` what is the value of:
`bus = { 2 {regx[3:1], {3{1'b0,regx[0]}} } };`
 - Bus = `18'b010_01_01_01_010_01_01_01`;



This page does not contain notes.

Module Exercise

Fix this code so that it preserves the sign of the value.

```
integer i;  
always @(...)  
begin  
    ...  
    i = -4; // -4  
    ...  
    i = i >> 1; // 2147483646  
    ...  
end
```



This page does not contain notes.

Module Exercise Solution

Fix this code so that it preserves the sign of the value.

```
integer i;
always @(...)
begin
...
i = -4; // -4
...
i = i >> 1; // 2147483646
...
end
```

Solution:

```
integer i;
always @(...)
begin
...
i = -4; // -4
...
i = i >>> 1; // -2
...
end
```



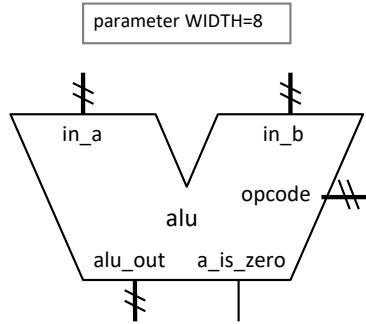
This page does not contain notes.



Lab

Lab 5-1 Modeling the Arithmetic Logic Unit

- Use Verilog operators while describing a parameterized-width arithmetic logic unit (ALU).

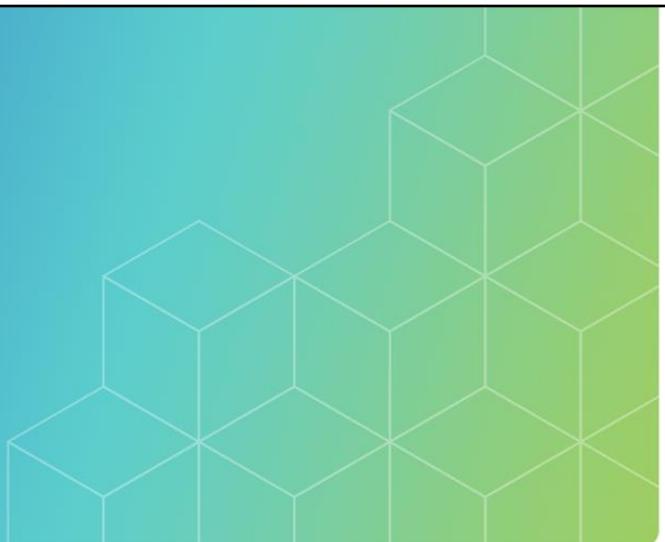
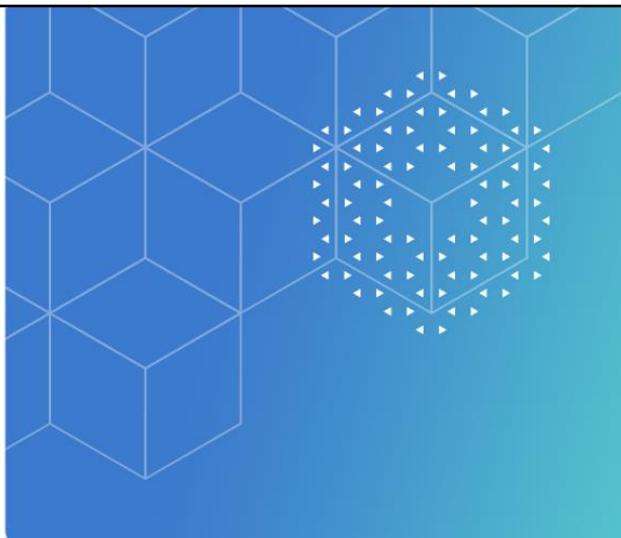


97 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to appropriately choose and correctly use the Verilog operators.

For this lab, you use Verilog operators while describing a parameterized-width arithmetic/logic unit (ALU).



Module 6

Making Procedural Statements

cadence®

This module describes the Verilog procedural programming statements.

Module Objective

In this module, you:

- Describe design behavior procedurally

Topics

- Procedural blocks review
- Making procedural assignments
- Making conditional statements
- Making case statements
- Making loop statements

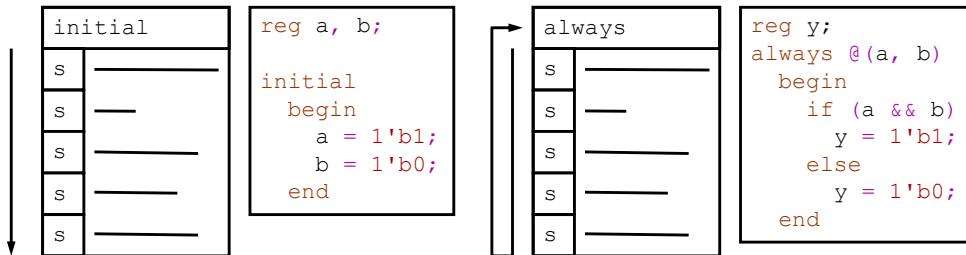


Your objective is to appropriately and effectively procedurally describe design behavior. To do that, you need to know generally about procedural blocks and procedural statements, and more specifically about the branching procedural statements. The Verilog behavioral modeling constructs are very similar to statements of the C programming language. They differ in subtle ways that can sometimes be irksome.

Describing Module Behavior

The **initial** construct starts execution at the start of simulation, and when complete, terminates the process.

The **always** construct starts execution at the start of simulation, and when complete, executes again.



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



Event simulation relies upon processes. A process is an object that reacts to input events and generates output events. The continuous assignment that you have already seen is a process – the simulator reacts to its input transitions to calculate a new output value and transitions the net to that new value.

Verilog has other kinds of processes – two most obvious to the user are the **initial** construct and the **always** construct, frequently referred to as the initial block and the always block. The **initial** and **always** keywords are not statements themselves – they apply to their following statement and make it execute as a process. Their following statement is usually a statement block, e.g., statements between the **begin** and **end** keywords, but does not have to be.

- The simulator executes each **initial** process at the start of simulation, and upon completing its execution, terminates the process.
- The simulator executes each **always** process at the start of simulation, and upon completing its execution, executes it again.

Synchronizing Module Behaviors

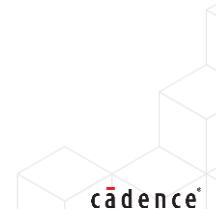
- Procedures “block” upon encountering an event control.
- Procedures “unblock” upon occurrence of any of the listed events.
- An event control can reference:
 - A single event identifier.
 - An event expression.
 - The “*” wildcard operator.

* Examples

```
@a
@(a)
@(a or b)
@(a, b)
@*
@(*)
@(posedge clk)
@(posedge clk or negedge rstn)
```

Event Expression

```
always @( a or b or sel )
  if (sel == 1)
    op = b;
  else
    op = a;
```



If an *always* construct has no construct to block its execution, then when the simulator started executing it at the start of simulation, execution would continue in a tight loop forever and the simulator would appear to hang.

Verilog provides procedural timing controls for “stepping” execution of procedural blocks. The most common of these is the event control. An event control starts with the *at* (@) character and then follows with either a wildcard character, a single event identifier, or a parenthesized event expression. The event expression can be a list of event expressions separated by the *or* keyword or by the comma (,) character. Both are shown here. The *or* keyword in an event expression is a separator between event expressions and is not an operator in the usual sense. You can further qualify an expression with the *posedge* or *negedge* keywords.

A timing control is not itself a statement, but precedes a statement, and in the case of an event control, blocks execution of that statement and subsequent sequential statements until one of those events occurs.

```
event_control ::==
  @ event_identifier
  | @ ( event_expression )
  | @@
  | @ (*)

event_expression ::==
  expression
  | hierarchical_identifier
  | posedge expression
  | negedge expression
  | event_expression or event_expression
  | event_expression , event_expression
```

Interactions Between Behaviors

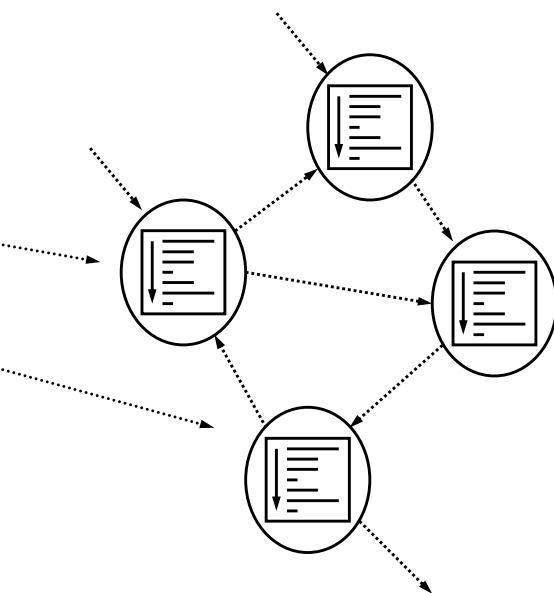
```
module design (
    list_of_port_declarations
);
    net_declaration(s)
    reg_declaration(s)

    always @*
        begin
            statement(s)
        end

    always @(posedge clk)
        begin
            statement(s)
        end

    assign
        net_assignment(s)

endmodule
```



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



A non-trivial design usually contains several modules, and a module at the RT level typically contains procedural blocks:

- Each sequential procedural block executes its statements sequentially like a conventional programming language;
- Multiple blocks execute concurrently, like hardware; and
- Multiple blocks communicate with each other using events, nets and variables.

This capability to have many procedural blocks communicating concurrently with each other is the basic model which Verilog uses to describe hardware.

Making Procedural Assignments

- Procedural assignments are assignments made inside procedures.
- You make procedural assignments only to variables.

```
module fulladder (
    input wire a, b, cin,
    output reg [1:0] out
);

    reg sum, carry;

    always @(a, b, cin)
        begin
            sum = a ^ b ^ cin;
            carry = (a & b) | cin & (a ^ b);
            out = {carry, sum};
        end

endmodule
```



The *initial* construct and the *always* construct are frequently referred to as *procedural blocks*, though as you have seen, they can apply to an atomic statement. Statements within a *procedural block* are procedural statements, and most of your procedural statements will be assignments.

- Recall that a continuous assignment is to a net and is its own process, so only exists *outside* any procedural blocks.
- A procedural assignment is to a variable and is not its own process, so only exists *inside* a procedural block.
- The right operand of either assignment can contain constants, variables and nets, as on the right side only their values are used.

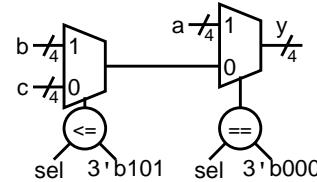
Making Conditional Statements

- ***if*** is a conditional statement.
- ***if*** tests the conditions in sequence.
 - Condition is Boolean expression.
- Conditions can overlap.
- Executes statement associated with first known true condition.
- Statement can be null “;”.

```
module if_example (
  input  [3:0] a, b, c,
  input  [2:0] sel,
  output reg [3:0] y
);

  always @ (a or b or c or sel)
    if (sel == 3'b000)
      y = a;
    else if (sel <= 3'b101)
      y = b;
    else
      y = c;

endmodule
```



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



if statement syntax:

```
if (expression) statement
[ { else if (expression) statement } ]
[ else statement ]
```

The conditional statement is a two-way branch. If the conditional expression evaluates to a known non-zero value then the first branch executes and otherwise the second branch executes if it exists. The conditional expression can be multiple bits.

As with other programming languages, in nested conditional statements, every ***else*** keyword is associated with the nearest previous ***if*** keyword that does not already have an ***else*** associated with it. If this association displeases you, then you can force association by using ***begin-end*** sequential blocks, or you can just toss in the ***else*** keyword where needed with a null following statement.

In this example, execution of the procedural block blocks at the event control until a transition occurs on at least one of the module inputs. When a transition occurs, the conditional statement executes. It evaluates the ***sel*** signal. If the value is exactly ***3'b000***, then it executes the first branch, which is an assignment of ***a*** to ***y***. If the value is anything else, it executes the second branch, which is another conditional statement. This conditional statement again evaluates the ***sel*** signal. If the value is less than or equal to ***3'b101***, then it executes the first branch. If the value is anything else, it executes the second branch. Note that such a chain of conditional statements implies priority. If the ***sel*** signal is exactly ***3'b000***, then the second test for less than ***3'b101*** is not made, even though the value would match that second test if it were made.

Conditional Statement Syntax

```
if (expression)  
    statement
```

if branch executes if condition is known true.

```
if (expression)  
    statement  
else  
    statement
```

if branch executes if condition is known true.
else branch executes if condition is false or unknown.

```
if (expression)  
    statement  
else if (expression)  
    statement  
else  
    statement
```

if branch executes if condition is known true.
else if branch executes for alternative condition known true.
else branch executes if no condition known true.

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



The 1st illustration omits the second branch.

The 2nd illustration includes the second branch.

The 3rd illustration is not a new syntax. Unlike some languages, Verilog does not have an *elsif* keyword, so this is simply chaining two conditional statements.

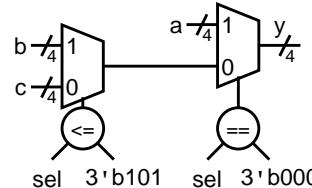
Making Case Statements: case

- The **case** statement is a multiway prioritized branching statement.
- case** item match expressions can overlap.
- Compares item match expressions to case expression in the order that they appear.
 - Uses case equality comparison
 - Bitwise 0,1,Z,X comparison
 - Executes statement associated with first match
 - Executes optional default item if no other item matches
 - In this example, inputs containing Z or X

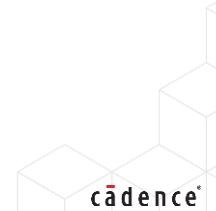
```
module case_example (
  input  [3:0] a, b, c,
  input  [2:0] sel,
  output reg [3:0] y
);

  always @(a or b or c or sel)
    case (sel)
      0 : y = a;
      1,2,3,4,5 : y = b;
      6,7 : y = c;
      default : y = 'bx;
    endcase

endmodule
```



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



case statement syntax:

```
case ( expression )
  expression { , expression } : statement
  {expression { , expression } : statement }
  [ default [:] statement ]
endcase
```

The **case** statement is a multiway prioritized branching statement. The case statement evaluates an expression and then evaluates a sequence of match expressions and executes the statement associated with only the first matching expression. The case statement considers high-impedance (z) and unknown (x) values and matches expressions that have a high-impedance or unknown bit in the same position in both expressions. You can comma-separate multiple match expressions associated with the same following statement. The following statement is often a statement block but does not have to be. You can optionally provide a **default** match item to be matched when no other match expression matches. It is common to place the default match item at the case statement end, but you can place it anywhere. You can have at most one default match item in each *case* statement.

In this example, execution of the procedural block is blocked at the event control until a transition occurs on at least one of the module inputs. When a transition occurs, the *case* statement executes. It evaluates the “sel” signal. If the value is 0 then it executes the first branch. If the value is between 1 and 5 then it executes the second branch. If the value is between 6 and 7 then it executes the third branch. If the value is anything else, it executes the default branch, which sets the output unknown. As an unknown value does not infer any particular hardware, the hardware has no implementation of this default branch.

Making Case Statements: casex

- Treats Z, X and ? characters as “don’t care” bit positions in either:
 - case expression (sel)
 - case item expression 3'b0XX
- Lets you group match values for more concise description
 - Encoders, decoders, etc.
 - Not very useful in testbenches
 - Not a preferred method usually
- casex treats X in case expression as meaningful in uninitialized logic
 - Hides initialization problems
 - Not recommended

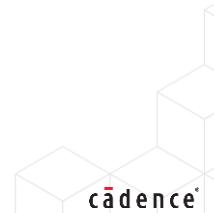
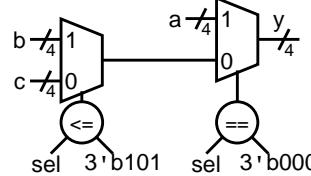


Can't match unknown or high impedance values coming out of DUT, so is less useful in testbenches.

```
module casex_example (
  input [3:0] a, b, c,
  input [2:0] sel,
  output reg [3:0] y
);

  always @(a or b or c or sel)
    casex (sel)
      3'b000 : y = a;
      3'b0??, 3'b?0? : y = b;
      3'b11X : y = c;
      default : y = 'bx;
    endcase
endmodule
```

Using ? is
more readable
than using X
or Z



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

The ***casex*** statement is a variation of the *case* statement that lets you specify bit positions it should not compare.

The *casex* statement treats the *x*, *z*, and question mark (?) characters as *don't-care* bit values. Recall that in a literal constant the ? character is equivalent to the *z* character and that only the based literal constants can use the *x*, *z*, and ? characters.

The *casex* statement does not compare bit positions of either the case expression or the case match items that have any of these *don't-care* values.

By convention, you might want to use the ? character to indicate *don't-care* bit positions in literal constants to avoid confusion with the high-impedance and unknown values. However, the *casex* statement still considers bit positions containing high-impedance (*z*) or unknown (*x*) values in expressions of nets and variables to be positions it should not match, so a *casex* statement in a testbench cannot test high-impedance or unknown values coming from the DUT.

This example is functionally identical to the previous example, but uses *don't-care* bit positions to represent a range of matching values.

Making Case Statements: `casez`

- Treats `z` and `?` characters as “don’t care” bit positions in:
 - `case expression (sel)`
 - `case item expression 3'b0zz`
- Performs definitive match for `x`
 - Useful in testbenches!
- `casez` is preferable to `casex`.
 - `X` in case expression is not wildcard
 - Safer for uninitialized RTL code



Can match unknown values so is more useful in testbenches.

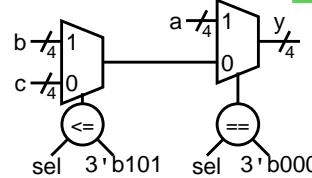


Use `?` instead of `z` – it's more intuitive and will cause less confusion!

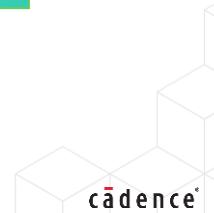
```
module casez_example (
  input      [3:0] a, b, c,
  input      [2:0] sel,
  output reg [3:0] y
);

  always @(a or b or c or sel)
    casez (sel)
      3'b000      : y = a;
      3'b0??, 3'b?0? : y = b;
      3'b11Z      : y = c;
      default     : y = 'bx;
    endcase
endmodule
```

Use `?` To avoid confusion with high impedance logic state Z



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



The `casez` statement is a variation of the `casex` statement and treats only the `z`, and question mark (`?`) characters as *don't-care* bit values and performs a definitive match for unknown (`x`) values. In a testbench, to test the existence of unknown values from the DUT, you should use the `casez` statement instead of the `casex` statement.

This example is functionally identical to the previous example, but uses the `z` character instead of the `x` character to represent the *don't-care* bit positions.

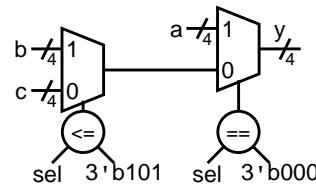
Alternate Case Statement Form

- **case** item match can be an expression as well.
- Case item match expression does not have to be a constant value!
- Can use case statement to match *variable* expressions.
- Case items are prioritized according to the order.

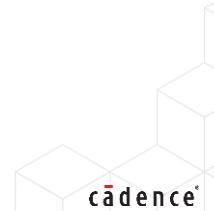
```
module finalcase (
    input      [3:0] a, b, c,
    input      [2:0] sel,
    output reg [3:0] y
);

    always @ (a or b or c or d)
        case (1'b1)
            sel == 3'b000 : y = a;
            sel <= 3'b101 : y = b;
            sel >= 3'b110 : y = c;
            default       : y = 'bx;
        endcase

endmodule
```



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



Alternate **case** statement syntax:

```
case ( expression )
    expression { , expression } : statement
    {expression { , expression } : statement }
    [ default [:] statement ]
endcase
```

Unlike most programming languages, the Verilog case item expression does not have to be constant.

This example is functionally identical to the previous example, but tests which match expression first matches the value 1, that is, which one is the first “true” expression.

```
case (1'b1)
    en_a : op = a;
    en_b : op = b;
    en_c : op = c;
endcase
```

In the above code, the case statement `en_a` has a highest priority. The statement checks for `en_a` first. If this is `1'b1`, then the output `op` is `a`. Otherwise it checks for `en_b`. If `en_b` is `1'b1` then output `op = b`. When both `en_a` and `en_b` not `1'b1`, then `en_c` is checked. If `en_c` is `1'b1` then output `op = c`.

Making Loop Statements: `while`

- `while` loop iterates and executes its following statement while the expression is known and non-zero
- Must previously declare any variables used in expression
- While expression is known and true, executes statement

```
integer count;
...
while (count < 10)
begin
    // statements
    count = count + 1;
end
```

```
reg [7:0] tempreg;
reg [3:0] count;
...
// Count the ones in tempreg
count = 0;
while (tempreg)
begin
    if ( tempreg[0] )
        count = count + 1;
    tempreg = tempreg >> 1;
end
```

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



while statement syntax:

```
while ( expression )
    statement
```

The **while** loop executes its following statement while the expression is known and non-zero. The following statement is usually a statement block but does not have to be. If the expression is not known or is zero, then the following statement never executes.

The 1st illustration executes a statement block while the count is known and less than 10. The last statement of the block increments the count. Some statement before the loop presumably initializes the count, but this is not shown.

The 2nd illustration counts the number of bits in the *tempreg* that are 1. While the “tempreg” is known and non-zero, if the least significant bit of the *tempreg* is known and non-zero it increments the count. In any case, each loop iteration shifts the *tempreg* one position to the right.

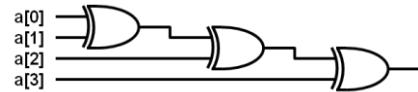
Making Loop Statements: `for`

- `for` loop requires prior declaration of any variables used in expressions.
- *Initialization* is done for the loop.
- Condition checked.
- *while* condition is known true, executes statement(s).
- *Update* expression is then executed at end of each iteration.

```
module parity (
    input [3:0] a,
    output reg odd
);

integer i;
always @*
begin
    odd = 0;
    for (i = 0; i <= 3; i = i + 1)
        odd = odd ^ a[i];
end

endmodule
```



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



`for` loop Syntax:

```
for ( initializer exp; condition exp;
      update exp)
    statement;
```

You can more concisely rewrite a while loop as a `for` loop that is not as complex. The `for` loop performs the initial variable assignment and then while the test expression is known and non-zero executes the following statement. The following statement is usually a statement block but does not have to be. After each iteration of its following statement, the `for` loop executes the second variable assignment. The two variable assignments are almost always to a variable that participates in the test expression, but they don't have to be.

This example calculates the parity of an input vector `reg`. The `for` loop initializes an index to 0 and while the index is less than or equal to 3 it exclusive-ORs the partial result with each successively indexed bit of the input, each time incrementing the index by 1.

If you think for a moment, you can think of an operator that will do this in one assignment without a loop. The same `for` loop can also be implemented using an equivalent while loop. The syntax is:

```
Initializer exp;
while ( condition exp)

    statement;
update exp;
```

Making Loop Statements: `repeat`

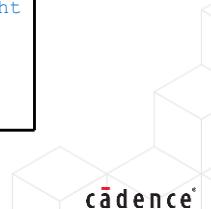
- `repeat` loop executes the statements number of times specified in the loop.
- Must previously declare any variables used in expression
- For expressed count executes statement.
- The `repeat` loop is equivalent to a `for` loop with the index variable automatically declared, initialized and incremented.

```
module multiplier (
    input      [3:0] a, b,
    output reg [7:0] result
);

reg [7:0] temp_a;
reg [3:0] temp_b;

always @(a, b)
begin
    temp_a = a;
    temp_b = b;
    result = 0;
    repeat (4)
        begin
            if (temp_b[0])
                result = result + temp_a;
            temp_a = temp_a << 1; // left
            temp_b = temp_b >> 1; // right
        end
    end
endmodule
```

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



`repeat` loop syntax:

```
repeat ( expression ) statement
```

You can more concisely rewrite a for loop as a `repeat` loop, where you do not need to use the index, for example, to select a bit of a vector. A `repeat` loop executes its following statement the number of times specified by its parenthesized expression.

This example multiplies two 4-bit inputs to produce an 8-bit output. Because it uses a shift multiplier implementation, it does not need a loop index, so uses a `repeat` loop instead of a `for` loop. Equivalent logic can be obtained using for loop as well. The syntax is:

```
integer h; // hidden index
for ( h=0; h < expression; h=h+1 ) statement
```

Making Loop Statements: **forever**

- **forever** loop executes statement forever.

```
reg clock;  
initial  
  forever  
    #10 clock = !clock ;
```

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



forever loop syntax:

forever statement

Verilog also provides a *forever* loop for when you want to repeat execution of a statement an essentially infinite number of times or until simulation ends. The same logic can be modeled using while loop as well. The syntax is:

```
while ( 1 )  
  statement
```

Module Summary

Now you can appropriately and correctly describe design behavior procedurally.

In this module, you studied these topics:

- The **initial** and **always** constructs introduce procedural blocks.
- The @ event control blocks further procedural block execution until an event occurs.
- The = procedural assignment that is inside a procedural block you make only to variables.
- The **if-else**, and **case/casex/casez** procedural branching statements.
- The **for**, **while**, **repeat**, and **forever** procedural looping statements.



You should now be able to appropriately and effectively procedurally describe design behavior. This module described Verilog procedural blocks and procedural programming statements.

Module Review

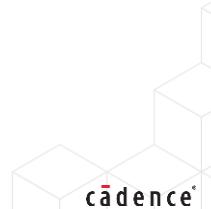
1. If more than one case item expression matches the case expression, which associated statement(s) execute?
2. Can you use the `case`, `if` and `for` statements in continuous assignments?



This page does not contain notes.

Module Review Solutions

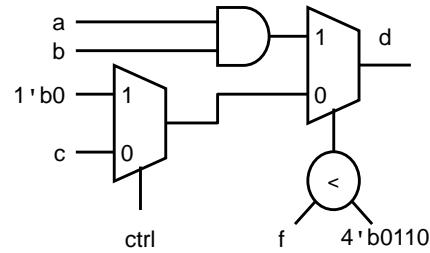
1. If more than one case item expression matches the case expression, which associated statement(s) execute?
 - The case expression is compared to the case item expressions in the order they appear and the statement associated with only the first match is executed.
2. Can you use the `case`, `if` and `for` statements in continuous assignments?
 - The `case`, `if` and `for` statements are procedural statements that cannot appear in a continuous assignment.



This page does not contain notes.

Module Exercise 1

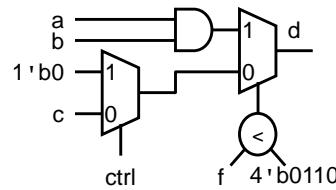
Code the following conditional logic as a procedural block:



This page does not contain notes.

Module Exercise 1 Solution

Code the following conditional logic as a procedural block:



Solution:

```
always @ (a or b or c or f or ctrl)
  if (f < 4'b0110)
    d = a & b;
  else if (ctrl)
    d = 1'b0;
  else
    d = c;
```

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



This page does not contain notes.

Module Exercise 2

```
// Find at least twelve things wrong!

module twelve_wrong (w, v, y, t, b, c, s)
input [3:0] a, b, c, s;
output [3:0] w, v, y, t;
reg [3:0] w, v, y, t;

assign w = s;

always @(a or b or c and s)
if (s == 0)
    v = a;
    y = b;
    t = c;
else if (s <= 4'b0101)
begin
    v = c;
    y = b;
    t = a;
    a = s;
end
else if (s == 6 or s == 7)
    v = c;
else v == 4'b'xxxx;
endmodule
```

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



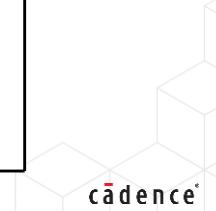
This page does not contain notes.

Module Exercise 2 Solution

```
module twelve_wrong (w, v, y, t, b, c, s) // Append semicolon
  input [3:0] a, b, c, s;                  // Include "a" in port list
  output [3:0] w, v, y, t;                // Change first "," to ":" 
  reg    [3:0] w, v, y, t;                // Change last "," to ";" 

  assign W = s;                         // Change "W" to "w"
  always @(a or b or c and s)           // Change to procedural assignment
    if (s == 0)                         // Change "and" to "or"
      v = a;
      y = b;
      t = c;
    else if (s <= 4'b0101)             // Insert "begin"
      begin
        v = c;
        y = b;
        t = a;
        a = s;                         // Change "a" to "w"
      end
    else if (s == 6 or s == 7)           // Change "or" to "||"
      v = c;
    else v == 4'b'xxxx;                 // Change "==" to "="
                                         // Remove second ''
  endmodule
```

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



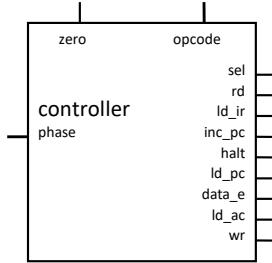
This page does not contain notes.



Lab

Lab 6-1 Modeling a Controller

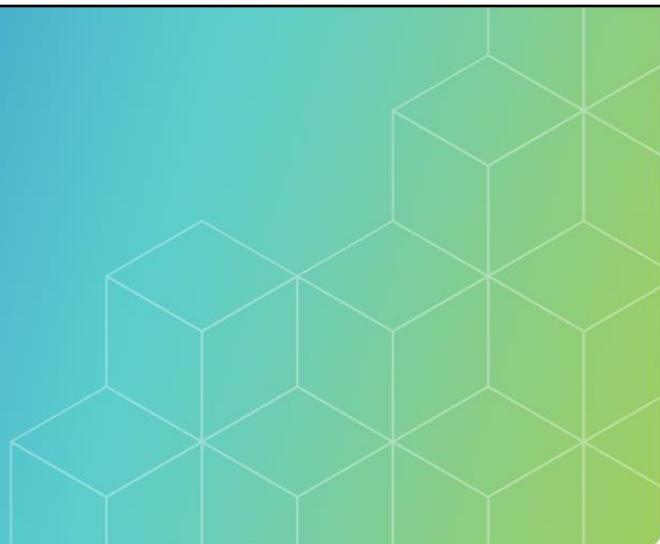
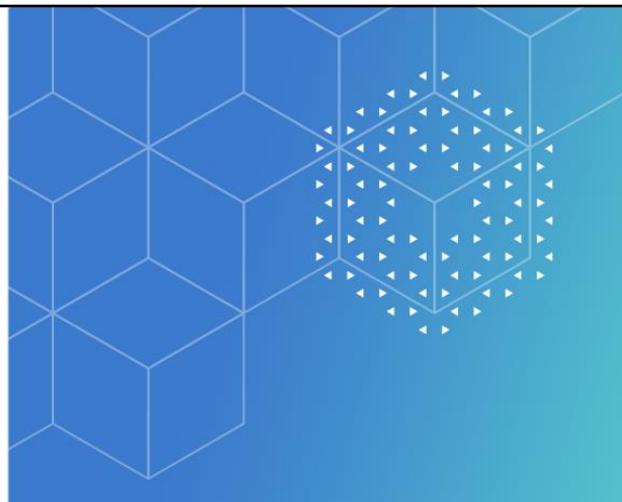
- Use the Verilog **case** statement to describe a controller.



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



This page does not contain notes.



Module 7

Using Blocking and Nonblocking Assignments

cadence®

This module reviews blocking and nonblocking assignments, and then examines how to use them in procedural blocks meant to represent combinational logic or sequential logic, and lastly explores issues with statement order and mixing blocking and nonblocking assignments in the same procedural block.

Module Objective

In this module, you:

- Use nonblocking assignments to model sequential design behavior

Topics

- Blocking assignment introduction
- Blocking assignments in sequential procedures
- Nonblocking assignment introduction
- Nonblocking assignments in sequential procedures
- Assignments in combinational procedures
- Mixing blocking and nonblocking assignment



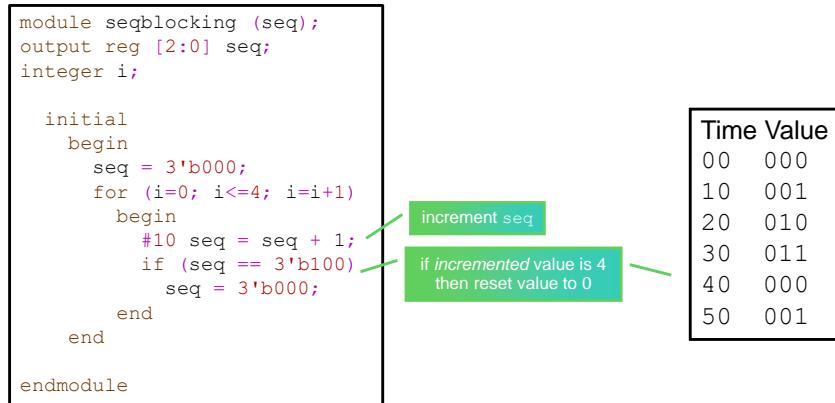
Your objective is to correctly choose between blocking and nonblocking assignments. To do that, you need to know more details about using blocking and nonblocking assignments in combinational and sequential procedures.

Blocking Assignment Introduction

A blocking assignment blocks further execution until complete.

```
variable = [delay_control] expression
variable = [event_control] expression
```

- By default completes immediately



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



The simulator executes a blocking assignment by evaluating the right-side expression, retaining its value, and blocking further execution of the block until it updates the left-side variable. If the assignment includes an intra-assignment timing control, then the simulator updates the variable after the timing control expires, and then continues block execution. If the assignment does not include an intra-assignment timing control, then the simulator updates the variable immediately.

In this example, the for loop increments the seq signal every 10 time units. The update occurs before the next statement executes. The next statement tests the new value, and if the new value is 4, immediately resets it to 0. The value 4 has no duration, so does not appear in the sequence of monitored values.

Blocking Assignments in Sequential Procedures

Blocking assignments can lead to race conditions, specifically when the same event triggers multiple procedures.

Example

- Both procedures execute on the positive clock edge.
- Blocking assignments to *a* and *b* finish immediately upon statement execution.
- But which statement executes first?
- The value of *b* depends on which procedure executes first.

```
initial
begin
    a = 0;
    b = 0;
end

always @ (posedge clock)
    a = a + 1;

always @ (posedge clock)
    b = a;
```



Procedures execute in indeterminate order. Code that inadvertently relies upon execution order will eventually break!

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



Blocking assignments can lead to race conditions specifically when the same event triggers multiple procedures, that then execute in a nondeterministic order, such that a procedure can read a variable that another procedure may or may not have already written.

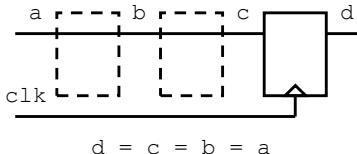
In this example, both procedural blocks unblock on the positive clock edge. The simulator can resume block execution in any order. The assignment statements use the blocking operator, so a race exists between the assignments to *a* and to *b*.

- If the simulator first executes the upper block, it updates *a* to get the incremented value and then executes the lower block where it updates *b* to get the new *a* value.
- If the simulator first executes the lower block, it updates *b* to get the not yet changed *a* value and then executes the upper block where it updates *a* to get the incremented value.

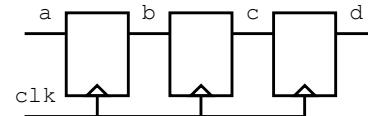
The final *b* value does depend upon procedure execution order.

Blocking Assignment Order Affects Functionality

```
always @ (posedge clk)
begin
    b = a;   b and c are intermediate  
(temporary) variables.
    c = b;
    d = c;
end
```



```
always @ (posedge clk)
begin
    d = c;
    c = b;
    b = a;
end
```



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



The left illustration uses blocking assignments:

- It assigns a to b and immediately updates the b value;
- Then it assigns b to c and immediately updates the c value; and
- Finally, it assigns c to d and immediately updates the d value.

As each variable is written before it is read, all the variables have the value of a. The b and c variables will not exist in a hardware implementation, as the d variable provides the same value.

The right illustration also uses blocking assignments, but it reorders the assignments:

- It assigns c to d and immediately updates the d value;
- Then it assigns b to c and immediately updates the c value; and
- Finally, it assigns a to b and immediately updates the b value.

As the variables are read before they are written, all can all have different values. All the variables will exist in a hardware implementation.

This is an example of position-dependent code. The statement order affects the functionality. You need to be aware of this issue, but it is not “bad” programming as such, because it is actually quite useful when deliberately done.

For this illustration, statement order would not affect functionality if the assignments were nonblocking.

Nonblocking Assignment Introduction

A nonblocking assignment schedules completion and does *not* block.

```
variable <= [delay_control] expression
variable <= [event_control] expression
```

- By default completes when all executing blocks have blocked.

```
module seqnonblock (seq);
output reg [2:0] seq;
integer i;

initial
begin
    seq <= 3'b000;
    for (i=0; i<=4; i=i+1)
        begin
            #10 seq <= seq + 1;
            if (seq == 3'b100)
                seq <= 3'b000;
        end
end
endmodule
```

Time	Value
00	000
10	001
20	010
30	011
40	100
50	000

Increment seq
If current value is 4 then reset value to 0

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



The simulator executes a nonblocking assignment by evaluating the right-side expression, retaining its value, scheduling the update, and continuing to execute the block until it encounters a blocking construct. The simulator schedules the variable update for a point in the simulation where all currently active blocks have executed up to the point where they are all blocked. This ensures that any other active block that reads the variable reads the old value and not the new value.

In this example, the *for* loop increments the *seq* signal every 10 time units. The update does not immediately occur. The next statement tests the old value, and if the old value is 4, resets it to 0. The value 4 has a duration of 10 time units, so *does* appear in the sequence of monitored values.

Making Nonblocking Assignments in Sequential Procedures

Nonblocking assignments *avoid* race conditions,
here for example:

- Both procedures execute on the positive clock edge.
- Assignments to “a” and “b” are scheduled.
- Assignment to “b” assigns the value of “a” from before the positive clock edge.

```
initial
begin
    a = 0;
    b = 0;
end

always @ (posedge clock)
    a <= a + 1;

always @ (posedge clock)
    b <= a;
```



Use nonblocking
assignment for
sequential logic!

In this example, both procedural blocks unblock on the positive clock edge. The simulator can resume block execution in any order. The assignment statements use the nonblocking operator, so no race condition exists.

- If the simulator first executes the upper block, it schedules an a variable update to get the incremented value and the a variable value does not yet change. The simulator then executes the lower block where it schedules a b variable update to get the old unchanged a variable value.
- If the simulator first executes the lower block, it schedules a b variable update to get the not yet changed a variable value. The simulator then executes the upper block, where it schedules an a variable update to get the incremented value.
- After executing these and all other triggered blocks to the point where they block, the simulator completes the nonblocking assignments to update the variable values.

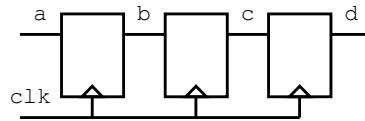
The final b variable value does NOT depend upon procedure execution order.

Can Nonblocking Assignment Order of Appearance Affect Functionality?

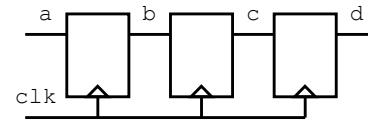
Nonblocking assignment order of appearance cannot affect functionality if:

- Statements are executed in the same simulation cycle.
- Each target is assigned only once.
- No target is assigned in any other procedural block.
- No blocking assignments are mixed with the nonblocking assignments.

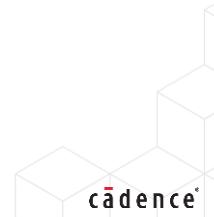
```
always @ (posedge clk)
begin
    b <= a;
    c <= b;
    d <= c;
end
```



```
always @ (posedge clk)
begin
    d <= c;
    c <= b;
    b <= a;
end
```



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



Here is the same illustration as previously, but utilizing nonblocking assignments. If you simply adhere to good programming practices, then the order of assignment appearance cannot affect functionality.

Making Assignments to Temporary Variables

- You can use blocking assignments to intermediate (temporary) variables within sequential procedures.
 - Declare temporary variables locally to discourage their use outside the block.
 - Assign inputs to temporary variables with blocking assignment.
 - Perform algorithm with temporary variables and blocking assignment.
 - Assign temporary variables to outputs with nonblocking assignment.

```
always @ (posedge clk)
begin : BLK
  integer temp;
  temp = a + b;
  q <= temp + c;
end
```

```
always @ (posedge clk)
begin : BLK
  integer tempa,tempb;
  tempa = in_p;
  ...
  tempb = f (tempa);
  ...
  out_p <= tempb;
  ...
end
```



Do not mix blocking and nonblocking assignments to the same variable.

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



You have previously seen that position dependent code is not “bad” programming as such, because it is actually quite useful when deliberately done.

You can use blocking assignments to strictly temporary variables within sequential procedural blocks meant to represent sequential logic.

The usage model is to use blocking assignments, which complete immediately, to break a complex expression representing combinational logic into a series of simpler assignments to temporary variables. Your final assignment uses the nonblocking operator to assign the temporary result to the variable that represents the sequential hardware.

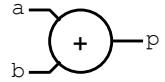
For the temporary variables to be truly temporary, they cannot be used anywhere other than that one block. This is impossible to enforce in the Verilog language, but you can make them less likely to be used elsewhere by declaring them locally. Recall that to declare local variables you need to name the block.

Multiple Assignments and Assignment Type

You can make multiple assignments to a variable within one procedure:

- All assignments to a variable should be the same type.
- Subsequent assignments override previous assignments.
- Watch out for unintended results!

```
always @ (a, b, c)
begin
    m = a;      — m and n are temporary variables.
    n = b;
    p = m + n;
    m = c;
    q = m + n;
end
```



```
always @ (a, b, c, m, n)
begin
    m <= a;      — m and n are not temporary variables.
    n <= b;
    p <= m + n;
    m <= c;
    q <= m + n;
end
```



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



You have previously seen that statement order affects functionality.

Assignment type can also affect functionality.

For blocking assignments, the simulator immediately updates the variable before continuing block execution. New values are immediately available until overwritten. In the blocking illustration, the simulator immediately updates the assignment of *a* to *m* and calculates a new value for *p* using the *a* value of *m*. It then immediately updates the assignment of *c* to *m* and calculates a new value for *q* using the *c* value of *m*.

For nonblocking assignments, the simulator schedules the variable update. New values are available only after the update occurs. In the nonblocking illustration, the simulator schedules the assignment of *a* to *m* and immediately replaces it with a scheduled assignment of *c* to *m*. Upon updating the *m* variable it gets the value of *c* and not that of *a*. The simulator re-executes the procedural block due to the transition of *m* and *n* and calculates new values for *p* and *q* that use the *c* value of *m*.

Module Summary

Now you can use nonblocking assignments to model sequential design behavior.

This module described:

- Blocking and nonblocking procedural assignment in sequential procedures
 - In sequential procedures, the order of blocking procedural assignments affects the result.
- Blocking and nonblocking assignment in combinational procedures
 - In combinational procedures, blocking assignments are sufficient, but use them carefully!
- Mixing blocking and nonblocking assignment
 - In a sequential procedure, make blocking assignments only to *temporary* variables.



You can now correctly choose between blocking and nonblocking assignments. This module reviewed blocking and nonblocking assignments, and then examined how to use them in procedural blocks meant to represent combinational logic or sequential logic, and lastly explored issues with statement order and mixing blocking and nonblocking assignments in the same procedural block.

Module Review

1. What is the primary difference between blocking and nonblocking assignments?
2. Where should you use blocking assignments?
3. Where should you use nonblocking assignments?
4. Where can you legitimately mix blocking and nonblocking assignments?



This page does not contain notes.

Module Review Solutions

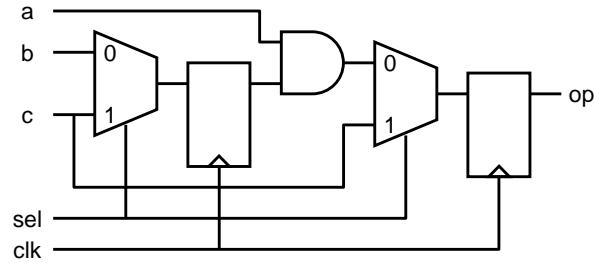
1. What is the primary difference between blocking and nonblocking assignments?
 - The blocking assignment operator blocks execution of subsequent statements until the assignment completes.
 - The nonblocking assignment operator calculates the RHS expression and schedules an update to the LHS variable.
2. Where should you use blocking assignments?
 - Use blocking assignments primarily in procedures that represent purely combinational logic.
3. Where should you use nonblocking assignments?
 - Use nonblocking assignments for variables that represent storage, primarily in a procedure that represents sequential logic.
4. Where can you legitimately mix blocking and nonblocking assignments?
 - You can make blocking assignments to intermediate (temporary) variables in a sequential procedure.



This page does not contain notes.

Module Exercise

Code this circuit in two procedures in such a manner that execution order does not affect the result.



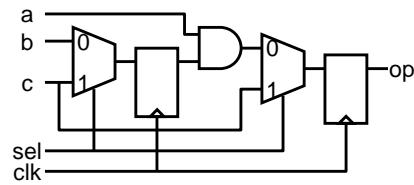
135 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Module Exercise Solution

Code this circuit in two procedures in such a manner that execution order does not affect the result.



Solution:

```
always @ (posedge clk)
    rg <= sel ? c : b;

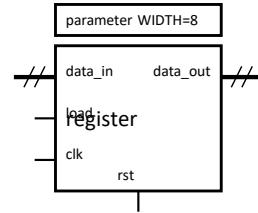
always @ (posedge clk)
    op <= sel ? c : a & rg;
```



Lab

Lab 7-1 Modeling a Generic Register

- Use nonblocking assignments while describing a register.

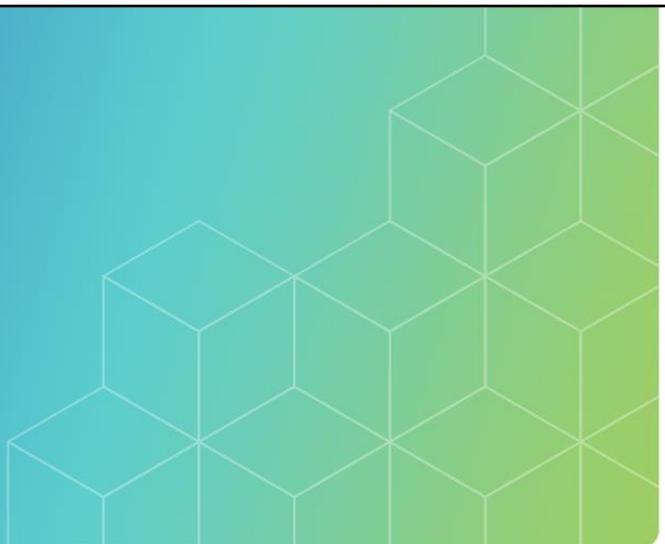
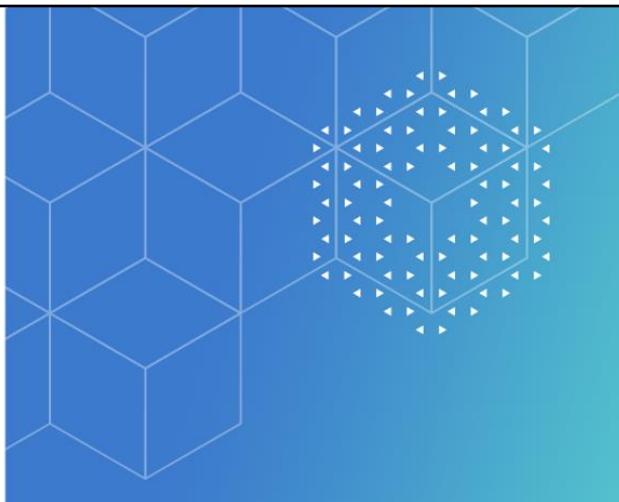


137 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to use nonblocking assignments to model sequential design behavior.

For this lab, you use nonblocking assignments while describing a register.



Module 8

Using Continuous and Procedural Assignments

cadence®

This module differentiates in more detail between continuous statements and procedural statements.

Module Objective

In this module, you:

- Appropriately choose between continuous and procedural assignments

Topics

- Continuous assignments review
- Multiple continuous assignments to a single net
- Procedural assignments review
- Multiple procedural assignments to a single variable
- Understanding the simulation cycle
- Conditional operator revisit
- Feedback loops
- Generate Statements



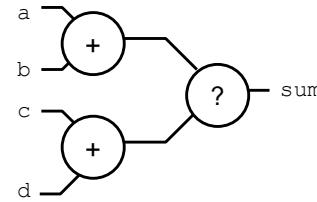
Your objective is to appropriately select between continuous and procedural statements. To do that, you need to know in more detail how these two different kinds of assignments work.

Continuous Assignments Review

- Only outside procedural blocks
- Continuously drive nets
- Order of declaration does not affect functionality

```
wire [8:0] sum;
assign sum = a + b;
assign sum = c + d;
```

Can declare
in either order



A continuous assignment is its own process – the simulator automatically updates the driven value and resolved net value when any of the inputs transition.

You can place a continuous assignment in a module anywhere after you declare the primaries that are its inputs. For tools compliant with the Verilog 2005 standard, you do not need to declare the target. The target of a continuous assignment is implicitly declared as a scalar net of the default net type, usually a wire. Reliance upon implicit declarations is generally considered poor practice.

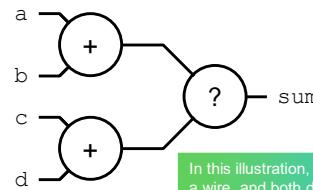
The simulator updates continuous assignments in any simulation cycle in which their inputs transition. No order is implied among multiple continuous assignments updated in the same simulation cycle. Relative position in the source code has no effect.

Multiple Continuous Assignments

Multiple continuous assignments to a single net are “wired together”.

- The **wire**, **wand** and **wor** net types all resolve value conflicts differently.

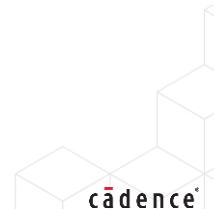
```
wire [8:0] sum;
assign sum = a + b;
assign sum = c + d;
```



In this illustration, if “sum” is a wire and both drivers are opposite values of strength, the “sum” value is X.

	wire				wand				wor			
	0	1	Z	X	0	1	Z	X	0	1	Z	X
0	0	X	0	X	0	0	0	0	0	1	0	X
1	X	1	1	X	1	0	1	1	1	1	1	1
Z	0	1	Z	X	Z	0	1	Z	Z	0	1	X
X	X	X	X	X	X	0	X	X	X	X	1	X

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

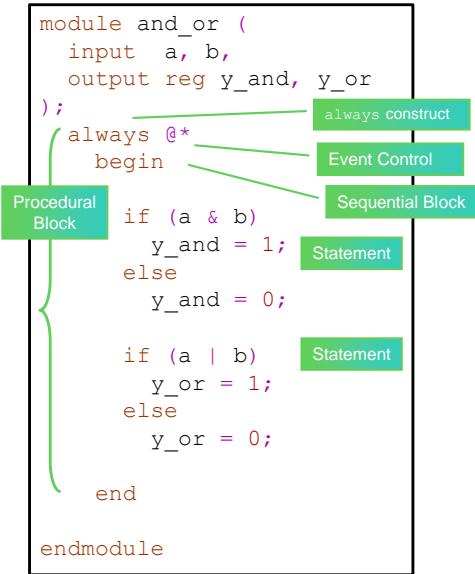


Except for the Verilog-2005 unresolved wire (**uwire**) type, the simulator resolves the value of a net driven by multiple drivers. The **uwire** type does not accept multiple drivers and reports an error.

- If one driver is stronger than the others, the strong driver “wins”. You can, for example, pull up a wire and still drive it to the 0, 1 and unknown (x) values. When you drive it to the high-impedance (z) value, the pullup takes over to maintain a pull-strength 1 on the wire. *Appendix B: Modeling with Verilog Primitives and UDPs* examines drive strengths in detail.
- If more than one driver is equally stronger than the remaining drivers, the simulator resolves the value of the net. The resolution function depends upon the net type and is different for nets that do and do not represent wired logic.
 - A **wire** net will attain the 0 or 1 value when all such strong drivers drive either a high-impedance value or the same 0 or 1 value and none of them drive an unknown value.
 - A wired-AND (**wand**) net will attain the 0 value when any such strong drivers drive a 0 value.
 - A wired-OR (**wor**) net will attain the 1 value when any such strong drivers drive a 1 value.

Procedural Assignments Review

- Procedural blocks start with **always** or **initial**.
- An event control almost always immediately follows the **always** keyword.
 - Blocks further statement execution until an event occurs.
- Statements within a sequential block (*begin-end*) execute sequentially.
- Assignments are only to variables – not nets.
- Multiple procedural blocks execute “concurrently”.



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



A procedural block is a process. It reacts to input transitions and generates output transitions.

You start a procedural block with an *always* keyword or an *initial* keyword. The *always* keyword starts a block that executes continually and the *initial* keyword starts a block that executes once. You control the execution of a procedural block by using timing controls. The most common of these is the event control, which blocks further statement execution until an event occurs.

The *always* and *initial* keywords apply to a following statement. That following statement is often a statement group between the *begin* and *end* keywords. Statements between the *begin* and *end* keywords execute sequentially. The previous statement finishes execution before the next statement starts. The Verilog for Verification section in this book examines a similar construct that encloses statements executing concurrently.

The procedural assignments you have seen up to this point are blocking assignments (=). They are called blocking assignments because they block execution of the next statement until they complete. This ensures that future statements can use the new value of the updated variable. You will often see it written that these assignments are “immediate,” but that is not necessarily true, as you will later see how you can deliberately delay it.

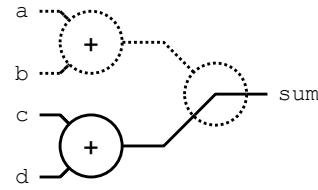
You will later see nonblocking assignments that use the same token as the less-than-or-equal-to (<=) operator. They are called nonblocking assignments because their completion is scheduled and they do not block execution of the next statement.

Multiple Procedural Assignments

Statements within a sequential block (*begin-end*) execute sequentially.

- Subsequent assignments override previous assignments.

```
reg [8:0] sum;
...
always @*
begin
    sum = a + b;
    sum = c + d;
end
```



Statements within a sequential block execute sequentially. Subsequent assignments override previous assignments. The latest assignment to a variable overrides all previous assignments to the variable.

This example first assigns to the variable the sum of a+b. Then without any delay it assigns to the variable the sum of c+d. The first assignment has no duration, so no other block can observe its results. No glitch will occur. Smart simulators will see this and optimize away the first assignment.

Example Multiple Procedural Assignments

- Subsequent assignments override previous assignments.
- Both these code fragments are equivalent.
- The version with the default assignment statement will be preferred for synthesis.

```
always @(a or b or sel)
```

```
begin
  if (sel)
    y = b;
  else
    y = a;
end
```

```
always @(a or b or sel)
```

```
begin
  y = a;
  if (sel)
    y = b;
end
```



Statements within a sequential block execute sequentially. Subsequent assignments override previous assignments. The latest assignment to a variable overrides all previous assignments to the variable.

This semantic is quite useful. You can write a code block that first provides default values for all the variables it updates, and then executes its algorithm to conditionally update some variables with various values. Upon exiting the block, you know that it has provided values for all of its variables. None has been missed regardless of what path the execution took through the block.

This illustration replaces the default branch of an *if* statement with a default assignment. The illustration is admittedly trivial, but does serve to illustrate the concept. Imagine a larger block having perhaps 50 statements and several possible paths through the statements.

Conditional Operator Revisited

- Can replace a simple combinational procedure with a continuous assignment.
- Target must be a net!
- Event control is assumed.
- May be more “readable” than procedural equivalent.

Both these modules are equivalent.

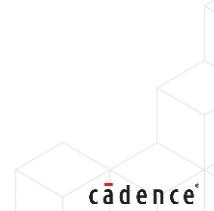
```
module procedural_if (
    input      [3:0] a, b, c,
    input      [2:0] sel,
    output reg [3:0] y
);
    always @ (a or b or c or sel)
        if (sel == 3'b000)
            y = a;
        else if (sel <= 3'b101)
            y = b;
        else
            y = c;
endmodule
```

```
// USING CONDITIONAL OP
y = (sel == 3'b000) ? a
: (sel <= 3'b101) ? b
: c;
```



```
module continuous_if (
    input      [3:0] a, b, c,
    input      [2:0] sel,
    output reg [3:0] y
);
    assign
        y = (sel == 3'b000) ? a
        : (sel <= 3'b101) ? b
        : c;
endmodule
```

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



For assignments to a single target, a conditional operator may be more readable than a conditional statement, and with the conditional operator, you can alternatively make the assignment a continuous assignment instead of a procedural assignment. Remember that continuous assignments are to nets and procedural assignments are to variables.

Combinational Feedback Loops – Be Careful

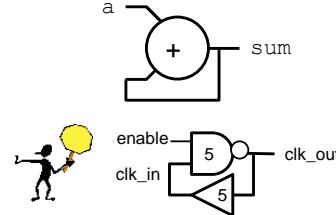
Zero-delay feedback loops may cause the simulator to appear to “lock up”.

- The process never finishes or suspends.
- The simulator never gets to do anything else.

Here is a short feedback loop deliberately generating a clock:

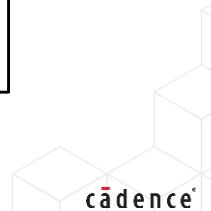
- For illustrative purposes only – more elegant ways exist to generate a clock.
- A continuous assignment is its own process. Whenever clk_out changes, the value is updated continuously.

```
always @(a , sum)
sum <= sum + a;
```



```
always @*
clk_out = #5 !(clk_in && enable);

assign #5 clk_in = clk_out;
```



The semantics of a Verilog process is that it reacts to its inputs and generates outputs. This is a different semantic than a procedural programming language such as C that does not autonomously react to inputs.

In continuous assignment, the simulator automatically updates the driven value and resolved net value when any of the inputs transition. For this illustration, the process is sensitive to its output, so the simulator could enter a tight loop where it does nothing but continually update the net.

You may understand this behavior better by instead making a procedural assignment to a variable within an always block and triggering the assignment with the same set of events. The assignment occurs when any of the inputs transition, so again the simulator could enter a tight loop where it does nothing but continuously update the variable.

Generate Statements in Verilog-2001

Verilog 2001 adds new reserved words: `generate endgenerate genvar`

- Defined within a module.
- Used to generate code dynamically within a module using conditional statements.
- Genvar is a positive integer value, used only inside a generate block.
- Generate_block_name (optional) is used to create an unique instance name for each generated item.
- Conditional (`case`, `if`) generation.
 - Instances, functions, tasks, variables, and procedural blocks.
- Iterative (`for`) generation.
 - Instances, variables, and procedural blocks (no functions or tasks).
- The keywords `generate – endgenerate` are optional in Verilog-2005.



Place your generated instances, functions, tasks, variables, and procedural blocks between the `generate` and `endgenerate` reserved words (you may not include parameters, ports, or specify blocks).

Declare `genvar` index variables for your generate `for` loops. You can declare them either inside or outside the `generate` statement. You can assign only integer values to them, and only within a `for` loop. These variables disappear after elaboration and are not available during simulation.

Iteratively generate statements using the `for` construct. Declare a named block within the `for` construct – this becomes the name of an array of scopes to match the iterations of the `for` construct. Inside this named block place your generated instances, variables, and procedural blocks. (You cannot define tasks and function within a generate `for` construct.)

Conditionally generate statements using `case` and `if` constructs.

Generate Statement – Conditional if Example

- Conditional *if* generation:
 - Instances, functions, tasks, variables, and procedural blocks
- Label not required to create generate-if scope.

```
module multiplier(a,b,product);
  parameter a_width = 8, b_width = 8;
  localparam product_width = a_width+b_width;
  input [a_width-1:0] a;
  input [b_width-1:0] b;
  output [product_width-1:0] product;
  generate
    if ((a_width < 8) || (b_width < 8)) begin: mult
      CLA_multiplier #(a_width,b_width) u1(a, b, product);
      // instantiate a CLA multiplier
    end
    else begin: mult
      WALLACE_multiplier #(a_width,b_width) u1(a, b, product);
    end
  endgenerate
endmodule
```

An implementation of a parameterized multiplier module

Cannot be modified directly with the defparam statement or the module instance statement #

The hierarchical instance name is mult.u1

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



These are the examples for usage of the *generate* statement using the conditionals *if-else*, case and the iterative *for*.

A *generate-for* loop permits one or more generate items to be instantiated multiple times. The index loop variable must be a *genvar*. This example shows a parameterized gray-code to binary-code converter using a loop to generate a continuous assignment for each bit of the converter.

A generate *if-else* or case permits generate items to be conditionally instantiated based on an expression that is deterministic at the time the design is elaborated.

The *if-else* generate example shows the generation of instances of a carry-look-ahead multiplier. If the input bus widths are greater than 8 bits, then an instance of a wallace-tree multiplier is generated.

Generate Statement – Conditional case Example

- Conditional case generation
 - Instances, functions, tasks, variables, and procedural blocks
- Label not required to create *generate-case* scope.

```
generate
  case (WIDTH)
    1: begin: adder // 1-bit adder implementation
        adder_1bit x1(co, sum, a, b, ci);
      end
    2: begin: adder // 2-bit adder implementation
        adder_2bit x1(co, sum, a, b, ci);
      end
    default:
      begin: adder // others - CLA
        adder_cla #(WIDTH) x1(co, sum, a, b, ci);
      end
  endcase
endgenerate
```

Generate with a case to handle widths less than 3

The hierarchical instance name is adder.x1



The *case-generate* example shows the instantiation of an appropriate adder depending on the case index *WIDTH* given as a *genvar*.

Generate Statement – Iterative Example

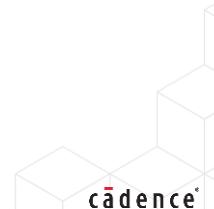
- Iterative (for) generation
 - Instances, variables, and procedural blocks (no functions or tasks)
- Label is required to create *generate-for* scope

```
module gray2bin1 (bin, gray);
  parameter SIZE = 8; // this module is parameterizable
  output [SIZE-1:0] bin;
  input [SIZE-1:0] gray;

  genvar i;
  generate
    for (i=0; i<SIZE; i=i+1) begin:bitnum
      // required label for scope naming
      assign bin[i] = ^gray[SIZE-1:i];
    end
  endgenerate
endmodule
```

A parameterized gray-code-to-binary-code converter module using a loop to generate continuous assignments.

i refers to the implicitly defined localparam whose value in each instance of the generate block is the value of the genvar when it was elaborated.



These are the examples for usage of the *generate* statement using the conditionals *if-else*, *case* and the iterative *for*.

A *generate-for* loop permits one or more generate items to be instantiated multiple times. The index loop variable must be a *genvar*. This example shows a parameterized gray-code to binary-code converter using a loop to generate a continuous assignment for each bit of the converter.

A generate *if-else* or *case* permits generate items to be conditionally instantiated based on an expression that is deterministic at the time the design is elaborated.

The *if-else* generate example shows the generation of instances of a carry-look-ahead multiplier. If the input bus widths are greater than 8 bits, then an instance of a Wallace-tree multiplier is generated.

The case generate example shows the instantiation of an appropriate adder depending upon the case index *WIDTH*.

Module Summary

Now you can appropriately choose between continuous and procedural assignments.

This module described:

- Continuous assignments using **assign** that you make only to a *net* and only *outside* procedural blocks. The simulator resolves a net value due to multiple continuous assignments.
- Procedural assignments using **=** that you make only to a *variable* and only *inside* procedural blocks. The last such assignment “wins” (which makes multiple such assignments in different procedural blocks problematic!).
- The conditional operator **?:**, which is useful in both continuous and procedural assignments.
- Inadvertently coding combinational feedback loops.
- The generate statements introduction: Verilog-2001 construct.



You should now be able to appropriately select between continuous and procedural statements. This module described in more detail the difference between continuous statements and procedural statements.

Module Review

1. What is the result of multiple continuous assignments to a net?
2. What is the result of multiple procedural assignments to a variable?
3. Explain how the conditional operator contributes to the usefulness of continuous assignments.
4. What causes combinational feedback loops?



This page does not contain notes.

Module Review Solutions

1. What is the result of multiple continuous assignments to a net?
 - Verilog resolves the value of multiple drivers of a net.
2. What is the result of multiple procedural assignments to a variable?
 - The last procedural assignment to the variable “wins”.
3. Explain how the conditional operator contributes to the usefulness of continuous assignments.
 - A continuous assignment can sometimes more intuitively describe simple combinational logic than can a procedure. With the conditional operator, you can concisely describe more complex logic.
4. What causes combinational feedback loops?
 - A combinational feedback loop occurs when a transition on the output of a process propagates to an input of the process. A combinational feedback loop is not a digital construct, so is a coding error in a digital design.

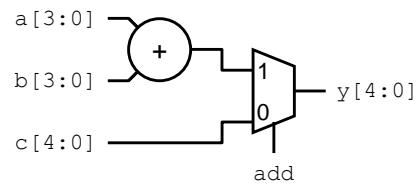
153 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Module Exercise

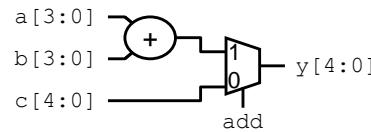
Code the following logic using a conditional operator in a (i) continuous assignment and (ii) procedural assignment.



This page does not contain notes.

Module Exercise Solution

Code the following logic using a conditional operator in a (i) continuous assignment and (ii) procedural assignment.



Solution:

```
always @* y = (add == 1) ? a + b : c ;
```

```
assign y = (add == 1) ? a + b : c ;
```



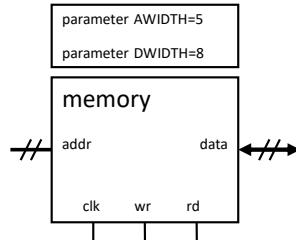
This page does not contain notes.



Lab

Lab 8-1 Modeling a Single-Bidirectional-Port Memory

- Use continuous and procedural assignments while describing a memory.



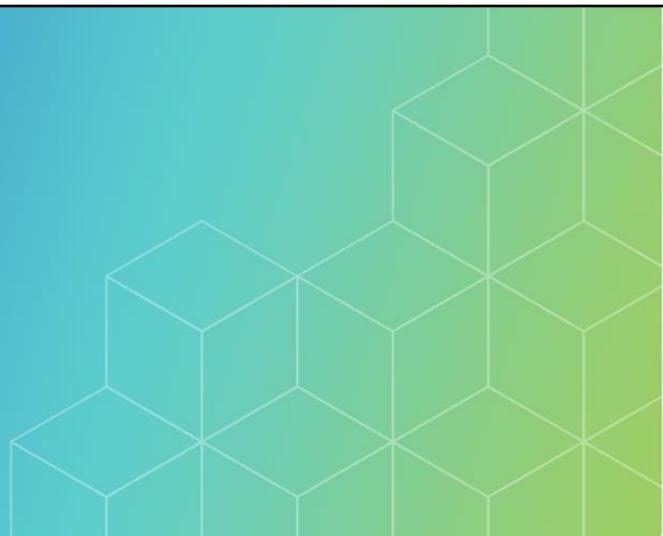
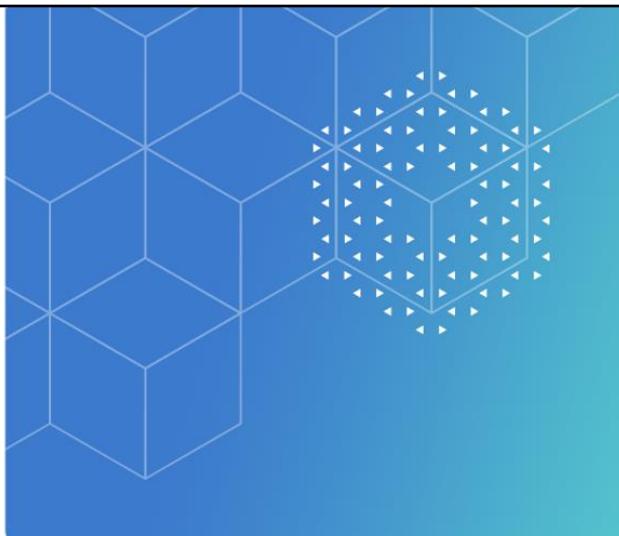
156 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to choose between continuous and procedural assignments.

For this lab, you use continuous and procedural assignments while describing a memory:

- You use procedural assignment for the write operation; and
- You use continuous assignment for the read operation.



Module 9

Understanding the Simulation Cycle

cadence®

This module examines in detail simulator execution of procedural blocks. It reviews procedural blocks and event controls, looks again at blocking and nonblocking procedural assignments, introduces the simulation cycle, and presents level-based event controls and simple delays.

Module Objective

In this module, you:

- Produce higher quality code that is less subject to nondeterminism and race conditions

Topics

- Procedural blocks and event control review
- Blocking procedural assignment review
- Making nonblocking procedural assignments
- Nonblocking assignments and the simulation cycle
- Synchronizing procedures
 - With event controls
 - With level-sensitive event controls
 - With procedural delay controls



Your objective is to produce higher-quality code less subject to indeterminacy and race conditions. To do that, you need to understand something about how the simulator executes processes and their statements. Some terminology definitions are as below:

- Simulation Time: Time value maintained by the simulator to model the actual time it would take for the circuit being simulated
- Process: Verilog code to evaluate, e.g., always and initial blocks and continuous assignments
- Evaluation Event: Evaluation of a process
- Update Event: Change in value of a net or variable
- Event Queue: Queue of events, ordered by simulation time
- Scheduling an Event: Putting an event onto the event queue
- Simulation Cycle: Processing all active events in the event queue

Procedural Blocks and Event Control Review

An **always** construct is a procedural block that loops continuously throughout the simulation.

An **initial** construct is a procedural block that executes once.

The “**@**” token introduces an event control, blocking further execution for either:

- A single event identifier.
- An event expression.
 - Can create an event list with **or** and comma “,” operators
 - Can qualify an expression term with **posedge** or **negedge**

Can give a block a name.

Combinational logic can use the blocking assignment “**=**” operator.

Sequential logic must use the nonblocking assignment “**<=**” operator.

Multiple procedures execute “concurrently”.

```
module adder (
  input  [3:0] a, b, c,
  output reg [4:0] o, p
);
  always @ (a or b or c)
    begin : ADDING
      o = a + b;
      p = a + c;
    end
endmodule
```

```
module flop (
  output reg q,
  input wire d, clk, clr
);
  always @ (posedge clk)
    if (clr) // CLOCKING
      q <= 0;
    else
      q <= d;
endmodule
```

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



An always construct is a procedural block that loops continuously throughout the simulation.

An initial construct is a procedural block that executes once.

The at (@) token introduces an event control, which blocks further execution until an event occurs. The event control can utilize a single event identifier or it can utilize an event expression that may be a list of event expressions separated by the or token or separated by commas with terms that can be qualified with posedge or negedge qualifiers.

You can name a sequential block by appending a colon and identifier to the begin keyword. A named block creates another level of scope that can declare its own variables. Of course you can always just make a pertinent comment about the statement’s purpose.

A procedural block that represents combinational logic can use the blocking assignment (=) operator.

A procedural block that represents sequential logic must use the nonblocking assignment (<=) operator.

Multiple procedural blocks execute in a manner that appears concurrent to the user. Multiple procedural blocks scheduled to execute in the same simulation cycle can execute in any order.

Blocking Procedural Assignment Review

- Blocking assignment “`=`” operator
- Blocks execution of subsequent statements until assignment completes:
 - By default immediately
 - Later slides show how to delay
- Almost all examples so far have used blocking assignments.
- This causes problems with some code structures.

```
reg [7:0] byte=8'b00001111;
...
// try to swap nibbles
byte[3:0] = byte[7:4];
byte[7:4] = byte[3:0];
// what is the result?
...
```

Byte is now
00000000



The procedural assignments you have seen up to this point are blocking assignments (`=`). They are called blocking assignments because they block execution of the next statement until they complete. This ensures that future statements can use the new value of the updated variable. You will often see it written that these assignments are immediate, but that is not necessarily true, as you will later see how you can deliberately delay it.

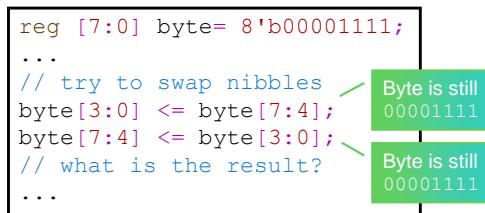
This illustration attempts to use blocking assignments to swap the upper and lower halves of a byte. It first assigns the upper half to the lower half and then assigns the lower half to the upper half. This does not have the desired affect, as the simulator completes the first assignment before it starts the second assignment. At the end of the first assignment, both halves have the same value.

You can work around such affects by utilizing a temporary variable, but an even better solution performs both assignments in a way that appears concurrent.

Nonblocking Procedural Assignment Review

Nonblocking assignment “`<=`” operator:

- RHS expression value is calculated.
- LHS variable update is *scheduled*.



Byte is 11110000
on next delta cycle.

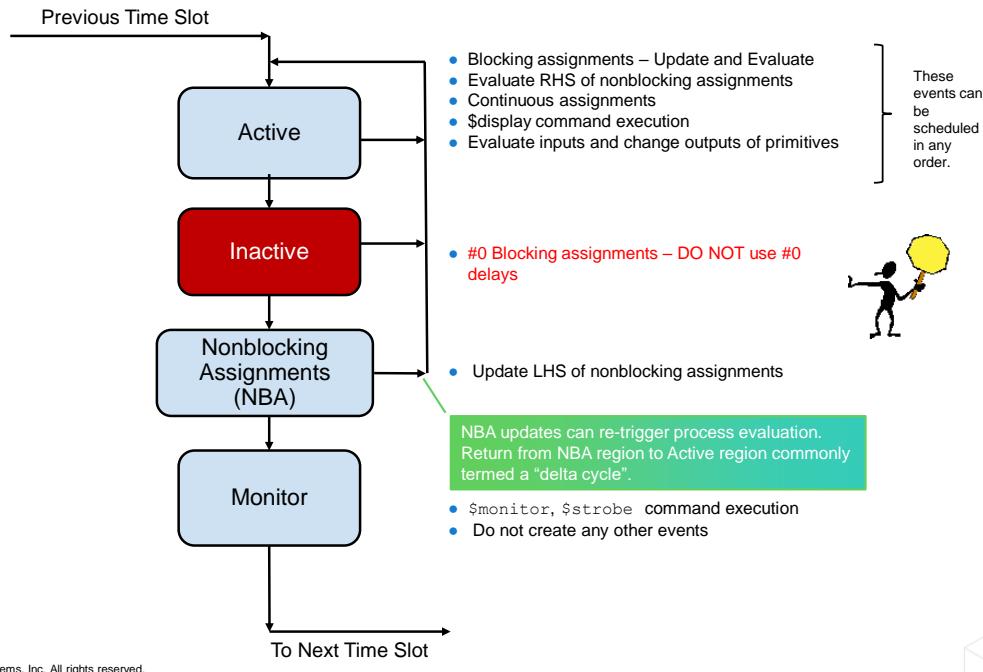


You can alternatively make nonblocking assignments. They are called nonblocking assignments because their completion is scheduled and they do not block execution of the next statement. The simulator instead calculates and retains the new value, but schedules the actual variable update for a point in the simulation where all currently triggered blocks have executed up to the point where they are all now all blocked. This ensures that any other triggered block that reads the variable reads the old value and not the new value.

Nonblocking assignments use the same token as the less-than-or-equal-to (`<=`) operator. It may be helpful to you to remember that the nonblocking word is longer than the blocking word and that the nonblocking operator is longer than the blocking operator.

This illustration uses nonblocking assignments to swap the upper and lower halves of a byte. It first assigns the upper half to the lower half and then assigns the lower half to the upper half. This does have the desired affect, as the simulator evaluates the first assignment after completing the second assignment. The simulator does not update the value of the variable until the procedural block next blocks at the event control.

Simplified Stratified Event Queue



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



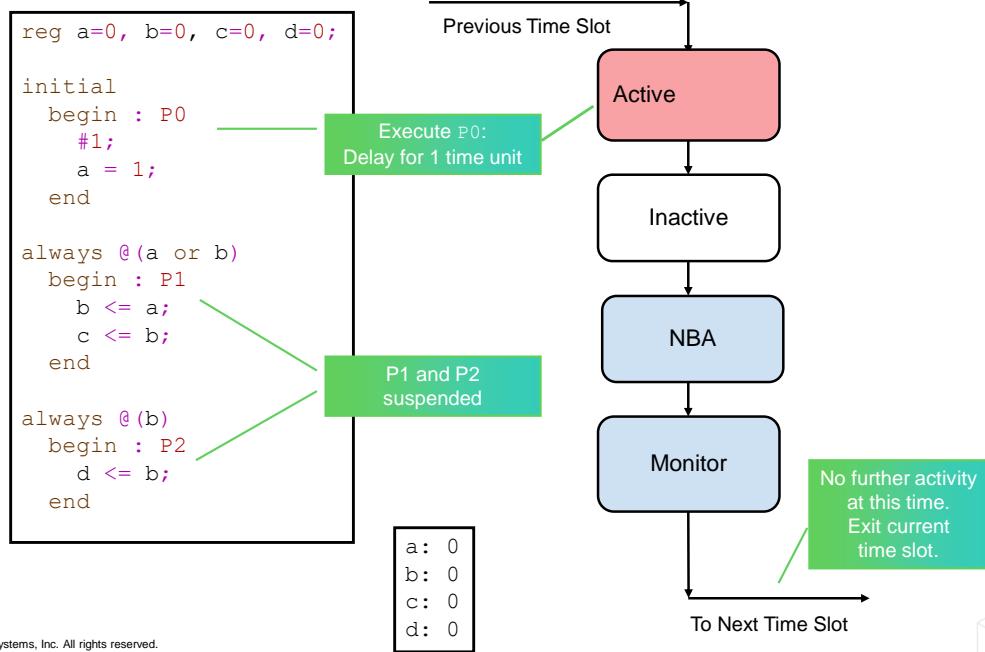
A simulation timeslot is divided into ordered regions to provide a predictable interaction between design constructs.

The Verilog event schedule has four regions for each simulation time:

- The Active region is for executing process statements.
- The Inactive region is for executing process statements postponed with a zero (#0) procedural delay.
- The NBA region is for updating nonblocking assignments.
- The Monitor region is for executing \$monitor and \$strobe and for calling user routines registered for execution during this read-only region. You cannot create additional events within this region.

The first three of these regions are iterative. They can schedule events that require return to the Active region. When no more events exist for the current simulation time, the simulator executes Monitor statements and then advances simulation time to the next time for which events are scheduled. The simulation terminates when no such future events exist.

Simulation Cycle: 1/6



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

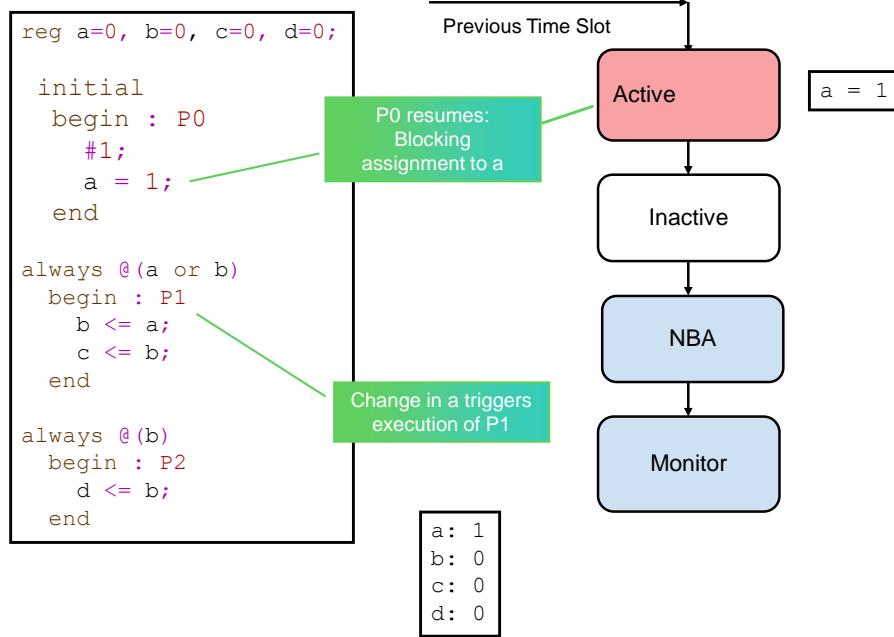


To demonstrate the behavior of nonblocking assignments and the simulation cycle, consider a module with three procedures, P0, P1 and P2. P1 is an always procedure triggered by an event on a or b. When either event occurs, procedure P1 makes nonblocking assignments to variable b and variable c, and returns to the top of the loop to again block. P2 also is an always procedure triggered by an event on variable b. When the event occurs, procedure P2 makes a nonblocking assignment to variable d, and returns to the top of the loop to again block. When an event occurs on variable b, the P1 and P2 processes will resume in either order. This order is predetermined by the simulator implementation and you cannot know or affect it.

As the simulation starts at time 0, the simulator moves all procedural blocks to the active evaluate events queue, and evaluates them in any order:

- As the simulator evaluates procedure P0, it immediately encounters a simple delay of #1, so schedules P0 resumption for the later time.
- As the simulator evaluates procedures P1 and P2, it immediately encounters event controls, so blocks further execution until one of the associated events occurs.

Simulation Cycle: 2/6



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



After exhausting all events at the current time, the simulator jumps to the next event time, and resumes execution of P0.

Here, the P0 procedure makes an immediate blocking assignment to register a, after which P0 completes.

When the simulator reaches the end of the active region, it checks to see if any procedural blocks have been triggered by the executed blocking assignments. Here a change in the value of a triggers P1.

Simulation Cycle: 3/6

```

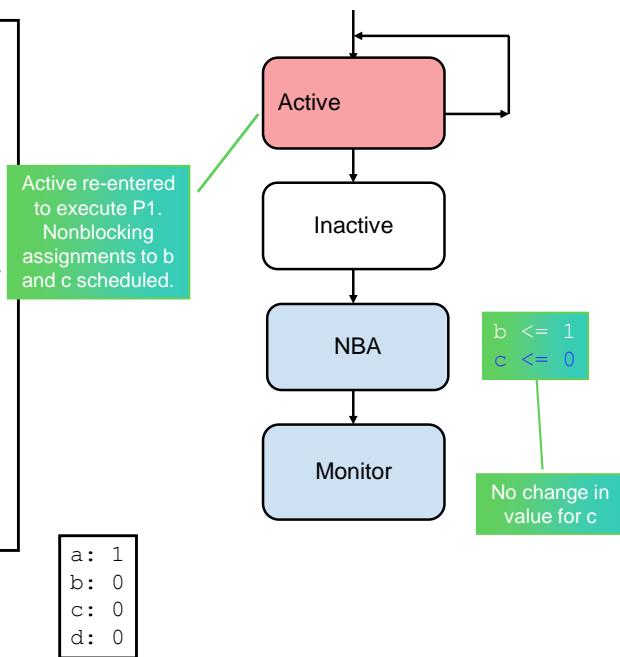
reg a=0, b=0, c=0, d=0;

initial
begin : P0
#1;
a = 1;
end

always @ (a or b)
begin : P1
  b <= a;
  c <= b;
end

always @ (b)
begin : P2
  d <= b;
end

```



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



The simulator re-enters the Active region to execute P1. P1 makes nonblocking assignments to b and c. These are scheduled for the NBA region. Remember nonblocking assignments are evaluated using the current values of variables, therefore c is assigned to the current value of b which is 0. This is not a new value for c, so the assignment can be ignored.

Simulation Cycle: 4/6

```

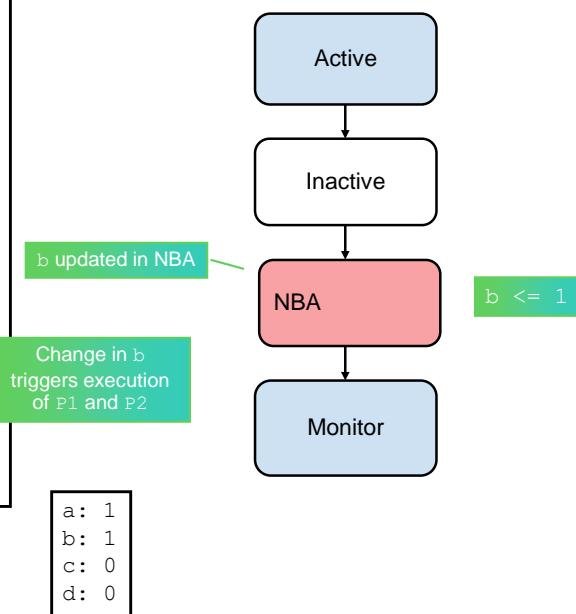
reg a=0, b=0, c=0, d=0;

initial
begin : P0
#1;
a = 1;
end

always @ (a or b)
begin : P1
b <= a;
c <= b;
end

always @ (b)
begin : P2
d <= b;
end

```



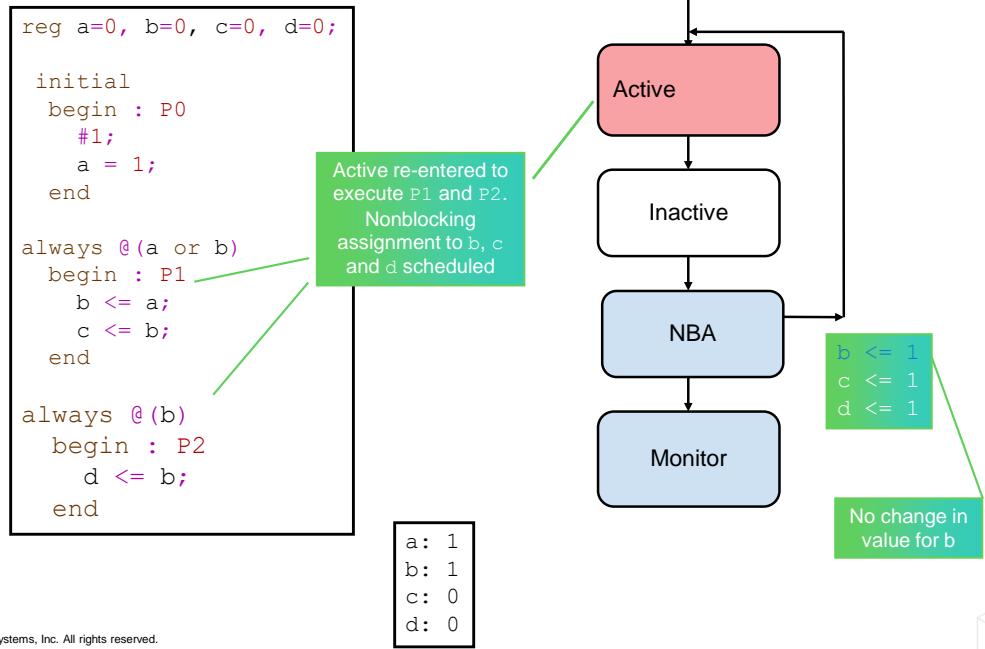
166 © Cadence Design Systems, Inc. All rights reserved.



There are no more procedural blocks triggered by activity in the Active region, therefore the simulator can advance to the NBA region to make the scheduled assignment to b.

When the simulator has completed all the nonblocking assignments in the NBA region, it checks to see if any procedural blocks have been triggered by the assignments. Here a change in the value of b triggers P1 and P2.

Simulation Cycle: 5/6



The simulator re-enters the Active region to execute P1 and P2.

P1 makes nonblocking assignments to b and c. These are scheduled for the NBA region. Only c receives a new value, therefore the assignment to b can be ignored.

P2 makes nonblocking assignments to c. This is scheduled for the NBA region.

The execution loop of re-entering the Active region from the NBA region is called a delta cycle. There may be many delta cycles at a given time slot before the simulating reaches a steady state.

Simulation Cycle: 6/6

```

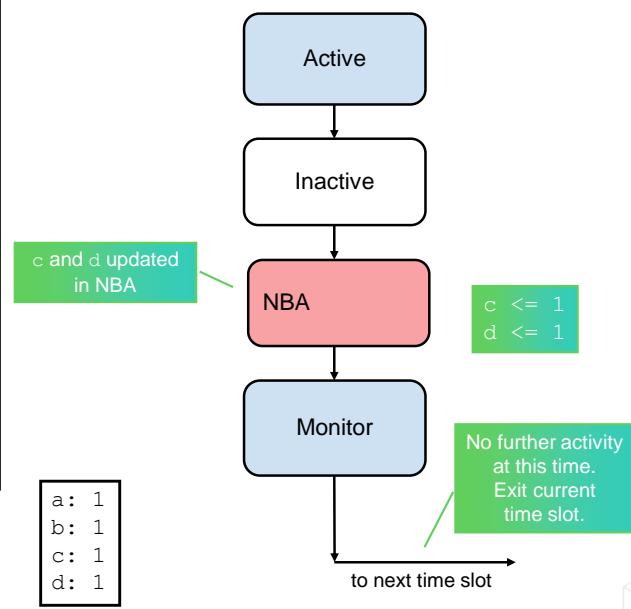
reg a=0, b=0, c=0, d=0;

initial
begin : P0
#1;
a = 1;
end

always @ (a or b)
begin : P1
b <= a;
c <= b;
end

always @ (b)
begin : P2
d <= b;
end

```



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



There are no more procedural blocks triggered by activity in the Active region, therefore the simulator can advance to the NBA region to make the scheduled assignments to c and d.

When the simulator has completed all the nonblocking assignments in the NBA region, it checks to see if any procedural blocks have been triggered by the assignments. Here there are no triggered blocks, therefore the simulator can exit the current time slot.

Simulator time will advance to the next nearest event, or if no events are scheduled, the simulator will terminate.

Simulation Cycle Summary

```

reg a=0, b=0, c=0, d=0;

initial
begin : P0
#1;
a = 1;
end

always @ (a or b)
begin : P1
b <= a;
c <= b;
end

always @ (b)
begin : P2
d <= b;
end

```

```

...
always @ (a or b)
begin : P1
b = a;
c = b;
end

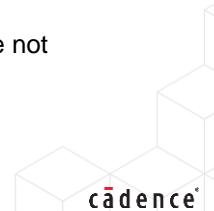
always @ (b)
begin : P2
d = b;
end

```

Cycle	Eval	Event	Update
cyc	P0	a=1	
	P1		b=1
cyc+1	P1		c=1
	P2		d=1

Note P1 is executed twice to reach a steady state.

- For combinational code, blocking assignments are sufficient.
- Nonblocking assignments are not needed and thus inefficient.

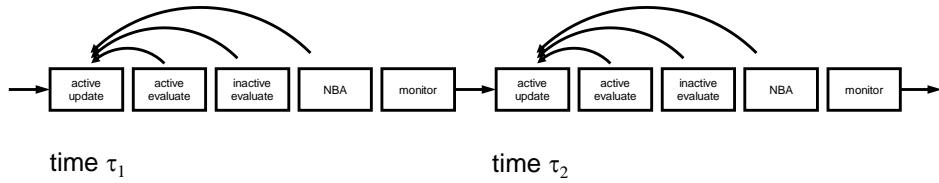


Here is a summary of the two delta cycles which propagate the transition of a to 1. The first delta cycle evaluates procedure P0, which generates an update event for the assignment to a. That update event triggers evaluation of procedure P1, which schedules a nonblocking update to b which occurs later in the same delta cycle. The nonblocking update to b triggers evaluation of procedures P1 and P2, which occur in the next delta cycle. These evaluations schedule nonblocking assignments to c and d that do not affect these procedures.

Synchronizing Procedures

You suspend execution of a procedural statement with:

- The `@` event control (wait for simulation events)
- The `wait` level-sensitive event control (conditionally wait for events)
- The `#` delay control (wait for simulation time)
 - Schedules process resumption into the inactive evaluate queue for the current or some future simulation time



Timing controls are not statements. They apply to the one immediately subsequent statement.

Verilog provides procedural timing controls for stepping execution of procedural blocks. Timing controls are not statements. Timing controls apply to the one immediately subsequent statement. In a sequential block, this of course also delays all following statements. Because a timing control is not a statement, it cannot appear at the end of a block unless a statement follows it. Conveniently, that following statement may be a null statement. A null statement is simply a semicolon (;).

Synchronizing Procedures: Event Controls

Use an event control for edge-sensitive timing control in behavioral code:

```
@ event_identifier
@ ( event_expression )
@*
@ (*)
```

- An event is any transition of the specified ports, nets or variables.
- Execution resumes when any of the events occur.
- Verilog-2001 added the comma and wildcard operators.
 - The wildcard operator adds all signals read within the block and any arguments of functions called from the block.

```
module reg_adder (
  output reg [3:0] out,
  input  [2:0] a, b,
  input          clk
);
  reg [3:0] sum;

  always @ (a or b)
    sum = a + b;

  always @ (negedge clk)
    out <= sum;

endmodule
```

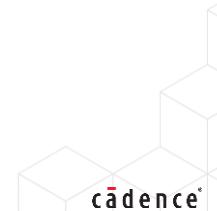
Wait for a transition on a or b

Wait for a positive transition on clk

```
initial
begin
  repeat (12)
    begin
      @ (posedge clk)
        tog <= ~tog;
    end
  end
```

Timing control can be anywhere in block!

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



Use an event control for edge-sensitive timing control in behavioral code. This is the most commonly used timing control. An event control starts with the at (@) character and then follows with either a wildcard character, a single event identifier, or a parenthesized event expression. The event expression can be a list of event expressions separated by the or keyword or by the comma (,) character. The or keyword in an event expression is a separator between event expressions and is not an operator in the usual sense. You can further qualify an expression with the posedge or negedge keywords.

This example uses a combinational block to maintain the value of an intermediate sum variable and a sequential block to assign the sum value to the output variable upon every negative edge of the clock.

- event_control ::=
 @ event_identifier
 | @ (event_expression)
 | @*
 | @ (*)
- event_expression ::=
 expression
 | hierarchical_identifier
 | posedge expression
 | negedge expression
 | event_expression or event_expression
 | event_expression , event_expression

Note: Wildcard event list adds all signals read within the block and any arguments of functions called from the block. If you read a signal in a function via side-effects (i.e., read directly and not passed by the argument list) then that signal is not included in the wildcard event list which is a feature of Verilog 2001.

Synchronizing Procedures: Level-Sensitive Event Controls

Use a **wait** statement for level-sensitive timing control in behavioral code:

wait (expr) statement

- When the statement executes:

- If the expression is known and true, it does not suspend the process.
- If the expression is unknown or false, it does suspend the process until the expression is known and true.

```
module latch_adder (
    output reg [3:0] out,
    input      [2:0] a, b,
    input                  enb
);

    always @ (a or b)
        begin
            // continue if/when enabled
            wait (enb)
            out = a + b;
        end
endmodule
```



Use a wait statement for level-sensitive timing control in behavioral code. If upon executing the wait statement the expression is already true the statement does not block. If upon executing the wait statement the expression is not true then the statement does block, and then unblocks if and when the expression becomes true.

In this example, the procedural block blocks upon every execution until an event occurs on its a or b inputs. When one of these events occurs, the procedural block tests the state of the enable (enb) input, and if the enable input is not 1 then the procedural block again blocks until the enable input becomes 1. While the procedural block is waiting for the enable input to become 1 it misses any further events on the a or b inputs, as it is not waiting for them.

-
- wait (expr) statement

The wait statement blocks if the expression is not true and then unblocks when the expression becomes true.

Synchronizing Procedures: Delay Control

Use a delay control for a simple time delay:

```
# delay_value
#( expr )
#( expr:expr:expr )
```

min:typ:max

- For modeling propagation delay
- For generating a clock
- For stepping the testbench

```
wire a, b; reg y;
...
always @(a or b) #10 y = a & b;
```

```
module clock (output reg clk);
parameter PERIOD = 10;

initial clk = 0;
always #(PERIOD/2) clk = ~clk;

endmodule
```

```
reg a, b;
...
initial
begin
    // wait 10 time units after
    a = 0; b = 1; #10;
    a = 1; b = 0; #10;
end
```

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



Use a delay control for a simple time delay for modeling propagation delay, generating a clock and stepping the testbench. If the delays are constants then you can use module parameters to propagate the constants through the hierarchy.

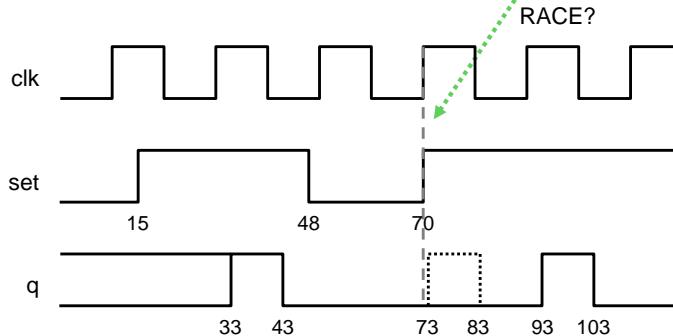
```
# delay_value
#( expr )
#( expr:expr:expr )
```

Example Timing Controls

```
// clock
parameter PERIOD = 20;
reg clk = 0;
always #(PERIOD/2)
  clk = ~clk;
```

```
// stimulus
reg set = 0;
initial
begin
  #15 set = 1;
  #33 set = 0;
  #22 set = 1;
end
```

```
// response
reg q;
always
begin
  wait (set);
  @ (posedge clk);
  #3 q = 1;
  #10 q = 0;
  wait (!set);
end
```



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



This example illustrates a collection of timing controls.

The response procedure does the following:

- 1st: It waits for “set” to be 1, ignoring the positive clock edge at time 10.
- 2nd: After “set” is 1 at time 15, it waits for the positive clock edge.
- 3rd: After the positive clock edge at time 30 it waits 3 time units before setting “q” to 1.
- 4th: After setting “q” to 1 at time 33 it waits 10 time units before setting “q” to 0.
- 5th: After setting “q” to 0 at time 43 it waits for “set” to be 0.
- 6th: After “set” is 0 at time 48 it again waits for “set” to be 1.
- 7th: “set” is 1 at time 70, coincident with the positive clock edge. Both assignment statements are on the active evaluate queue. Update to the “set” variable unblocks the response procedure and places it on the active evaluate queue, normally but not necessarily at the end of the queue, so the response procedure normally resumes execution after the positive clock edge has occurred and must wait for the next positive clock edge.

This illustration provides good arguments for synchronizing stimulus to the clock:

- 1st – The stimulus does not utilize the clock period constant. The stimulus causes greatly different results upon changing the clock period.
- 2nd – The stimulus is not synchronized to the clock. If the stimulus is synchronized to the clock, then we know that when the “set” signal transitions, the clock has already occurred and no race condition can exist.

Compiler Directive `timescale Preview

The `timescale compiler directive specifies the time unit and time precision for all following modules.

'timescale unit/precision

- Can appear only outside module descriptions.
- Time values in a module are rounded to the precision of that module.
- The overall simulation time precision is the smallest precision that any module specified.

```
'timescale 1ns / 100ps
module test;
reg a, b;
...
initial
begin
    // wait 10 ns after
    a = 0; b = 1; #10;
    a = 1; b = 0; #10;
end
...
```

Not a statement – no semicolon



The `timescale directive specifies the time unit and time precision for subsequent module declarations. Any expression used in a context to mean simulation time will assume the time unit of the current module and be rounded to the time precision of the current module. System tasks that display simulation time will scale the display to the time unit of the current module.

As the standard does not prescribe a default time scale, for interoperability, either all modules must be subject to a time scale or no module can be subject to a time scale. To ensure that all modules are subject to a time scale, you should get used to specifying a time scale at the top of every file, even if you not use time specifications in the file. To modify a time scale, you need to edit the file and recompile that file and all other files that utilized that directive. No method exists to modify a time scale during simulation.

'timescale time_unit / time_precision

- The *time_unit* argument specifies the assumed units for time values.
- The *time_precision* argument specifies the precision for time values.
- The *time_precision* argument cannot be larger than the *time_unit* argument.
- Time values are rounded to meet the current precision specification.
- The overall simulation time precision is the smallest specified time precision.
- The integer part of the arguments specify a magnitude {1, 100, 1000} and the character string argument suffix is the unit {s, ms, us, ns, ps, fs}.

Module Summary

Now you can produce higher quality code less affected by non-determinism and race conditions.

This module described:

- Procedural **initial** and **always** blocks
- Blocking = procedural assignment
- Nonblocking **<=** procedural assignment
- The simulation cycle (delta cycles)
- Timing control:
 - The @ event control
 - The **wait** level-sensitive event control
 - The # delay control



Your objective is to produce higher quality code less subject to indeterminacy and race conditions. To do that, you need to understand something about how the simulator executes processes and their statements.

Module Review

1. When does the variable value change when using a: (i) blocking assignment? (ii) nonblocking assignment?
2. What are the three Verilog timing controls?
3. What Verilog construct do you use in a procedure to advance simulation time?
4. Where can you place the `timescale compiler directive?



This page does not contain notes.

Module Review Solutions

1. When does the variable value change when using a: (i) blocking assignment? (ii) nonblocking assignment?
 - A blocking assignment blocks execution of subsequent statements of the procedure until the assignment completes, which is by default immediately.
 - A nonblocking assignment schedules assignment completion for the “update” phase of a delta cycle, by default the current cycle.
2. What are the three Verilog timing controls?
 - The `@` event control, the `wait` level-sensitive event control, the `#` delay control.
3. What Verilog construct do you use in a procedure to advance simulation time?
 - The `#` delay control suspends the procedure for a specified simulation time.
4. Where can you place the `'timescale` compiler directive?
 - In the source code outside any module definition. The Verilog standard implies but does not explicitly specify this by saying it applies to “modules that follow” and always showing it outside example modules.

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



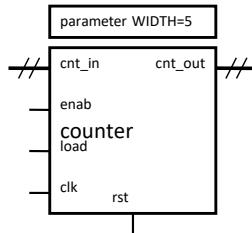
This page does not contain notes.



Lab

Lab 9-1 Modeling a Generic Counter

- Use blocking and nonblocking assignments while describing a counter.



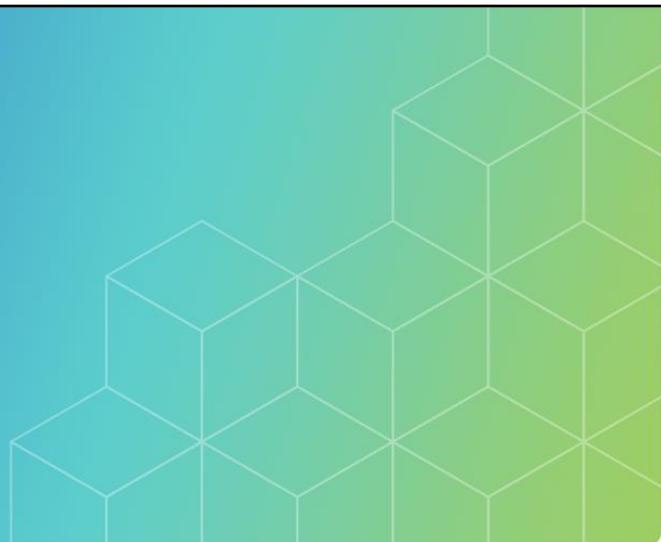
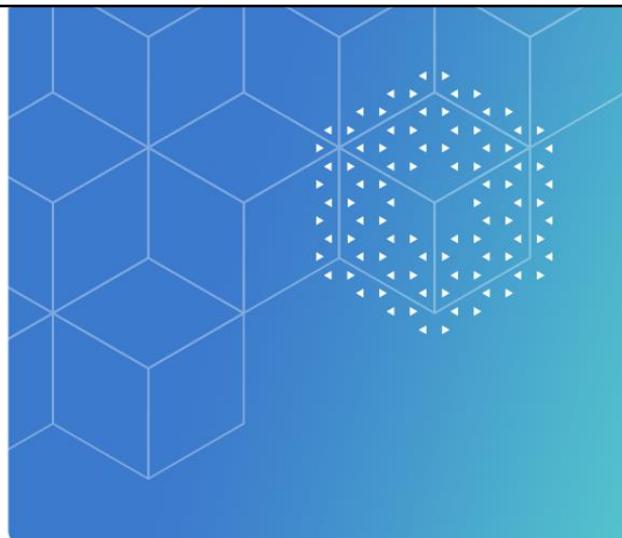
179 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to produce high-quality deterministic code.

For this lab, you use blocking and nonblocking assignments while describing a counter:

- You place combinational behavior in a combinational procedure.
- You place sequential behavior in a sequential procedure.



Module 10

Using Functions and Tasks

cadence®

This module describes Verilog subroutines, known as functions and tasks. It explains where and how to define them, how to invoke them, and some issues involved with using them.

Module Objective

In this module, you:

- Write Verilog subroutines to encapsulate functionality making your code more readable and reusable

Topics

- Verilog Subroutine Introduction
- Functions – Declaration and Calling
- Tasks – Declaration and Calling
- Issues with Functions and Tasks



Your objective is to effectively use subroutines to encapsulate functionality and thus to make your code more readable and reusable. To do that, you need to know what subroutines are available and how to use them.

Verilog Subroutines Introduction

Subroutines:

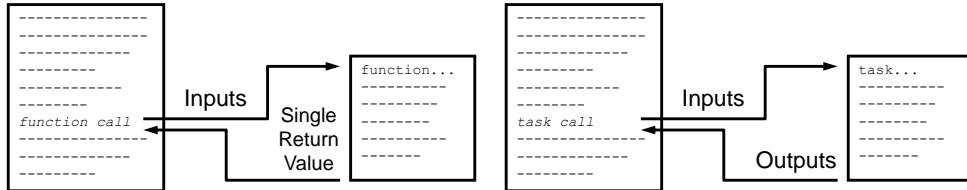
- Encapsulate code that might otherwise be duplicated.
- Contain statements that execute in sequence.

Function subroutines:

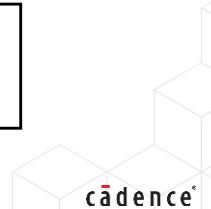
- Have one or more inputs and return a single value.
- Are invoked as an expression term.

Task subroutines:

- Have zero or more input/outputs.
- Are invoked as a procedural statement.



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

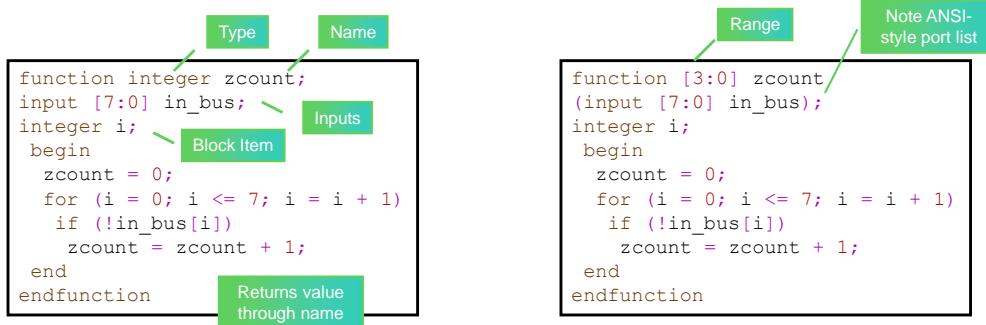


Let's first examine the purpose of subroutines:

- You can think of a subroutine as a level of hierarchy, or structure, in sequential code.
- You use subroutines to encapsulate some frequently used code in one place where you can more easily manage it. You can provide this code with inputs, to which you assign values as you invoke the subroutine. These input values can flexibly direct the subroutine to perform slightly differently for each invocation.
- Encapsulating sequential code in subroutines is similar to encapsulating design units in modules.

Declaring Functions

- Function declared only within a module
- Starts with **function** keyword
- Then the logic description of what the function does
- Ends with **endfunction** keyword
- Alternate syntax with the function port list



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



Syntax

```

function [automatic] [signed] [range_or_type] function_identifier;
  tf_input_declaration;
  {tf_input_declaration;}
  {block_item_declaration}
  function_statement
endfunction

```

Syntax Alternative

```

function [automatic] [signed] [range_or_type] function_identifier
  (function_port_list);
  {block_item_declaration}
  function_statement
endfunction

```

You declare a function only within a module. The declaration starts with the **function** keyword.

Functions are by default static. That means that exactly one copy of the function variables exists. All calls to the function utilize that one set of function variables, thus calls to static functions are generally not recursive. External code can access static function variables using hierarchical references. Verilog 2001 added the **automatic** declaration. Automatic functions create a new temporary set of their variables for each call. Automatic functions can be recursive. External code cannot access automatic function variables.

The return type defaults to a single bit. You can alternatively specify **integer**, **real**, **realtime**, or **time**, or a vector range. A vector is by default not signed. You can declare the vector signed. The **signed** declaration is a Verilog 2001 feature.

A function must have at least one input port and must have no output ports and no inout ports. You can declare the input ports using either syntax. With the function port list syntax, you can prefix port attributes. The input port type defaults to a single bit. You can alternatively specify **integer**, **real**, **realtime**, or **time**, or a vector range. A vector is by default not signed. You can declare the vector signed. The **signed** declaration is a Verilog 2001 feature.

You can declare additional block items. You cannot declare module items. For example, you can declare variables but not nets.

A function declaration contains a statement. The statement may be a statement block, for example grouped between the **begin** and **end** keywords. Functions cannot invoke the scheduler. That means function assignments are always blocking and functions always execute completely and instantly.

Functions can have side-effects, for example, a function can assign to a module variable but not to a module net. For a function to have side-effects is undesirable programming practice. Instead, declare the function return vector sufficiently wide to hold all the data you need to return, then deconstruct the vector value as needed upon return.

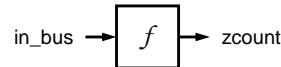
A function declaration ends with the **endfunction** keyword.

Calling Functions

You call a function as an expression term.

```
function_identifier (expression {,expression})
```

```
function integer zcount (
    input [7:0] in_bus
);
    integer i;
begin
    zcount = 0;
    for (i=0; i<=7; i=i+1)
        if (in_bus[i]==0)
            zcount = zcount + 1;
end
endfunction
```



```
module zfunc (
    input [7:0] a_bus, b_bus,
    output [31:0] account,
    output reg bzero
);

// function declaration here

assign account = zcount(a_bus);

always @(b_bus)
    bzero = zcount(b_bus)==0;

endmodule
```



You call a function as an expression term. The simulator assigns the values of the argument expressions to the input ports in the order in which they appear in the call.

When the function completes, the simulator uses the value of the function name variable where the function call appears in the calling expression. Functions return a single value that you use as an operand in an rvalue expression. An rvalue expression is one that may appear only on the right side of an assignment – you cannot assign a value to an rvalue expression. If the function never completes, for example, it executes a loop that loops forever, it never returns and the simulation “hangs”.

In this example, the zcount function returns an integer value representing the number of bits of the in_bus port that are zero. It returns a value between 0 and 8. The continuous assignment call assigns the returned value to the “account” module output port. The procedural assignment call uses the function return value as an operand in a comparison expression.

Constant Functions

Verilog-1995 allows simple constant expressions for defining limits for vectors, replicates, etc.

- Verilog-2001 allows limits to be defined by constant functions.
- Function value must be calculable at elaboration.
 - Usually inputs are constant.
- Greater flexibility for scalable reusable models.
- Multiple limits can be derived from a single constant.

```
parameter addw = 5;
parameter datw = 8;

reg [addw-1:0] address;
reg [datw-1:0] mem [1:memsize(addw)];

function integer memsize;
  input [15:0] width;
  begin
    case (width)
      ((width%2)==0): memsize = 1024;
      default: memsize = 512;
    endcase
  end
endfunction
```

Verilog 2001



Verilog 1995 module parameter values must be constant expressions, which are limited to operations on literals and previously declared module parameters.

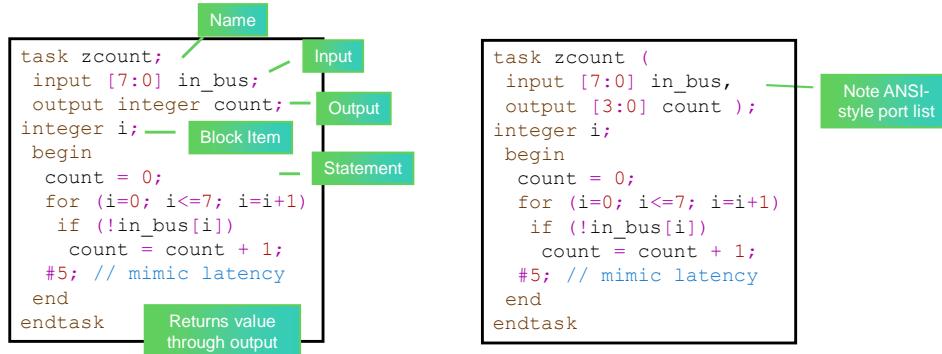
Verilog 2001 permits you to use a constant function call anywhere you are required to use a constant expression.

The standard lists restrictions upon constant function calls:

- Cannot be placed within any generate scope.
- Cannot contain hierarchical references.
- Cannot contain system function calls.
- Ignores system task calls (except will execute \$display in simulator but not elaborator).
- Cannot themselves make constant function calls in any context requiring constant expressions.
 - Can otherwise make constant function calls to functions local to containing module.
- Can access only functions and module parameters and nothing else declared outside the function definition.
- Module parameters must be previously assigned use of defparam can produce undefined results.

Declaring Tasks

- Task is declared only within a module.
- Declaration starts with task keyword.
- Logic description of what the task is supposed to do follows.
- Ends with endtask keyword.
- Alternate syntax with the port list.



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



You declare a task only within a module. The declaration starts with the task keyword.

Tasks are by default static. That means that exactly one copy of the task variables exists. All calls to the task utilize that one set of task variables, thus calls to static tasks are generally not re-enterable. External code can access static task variables using hierarchical references. Verilog 2001 added the automatic declaration. Automatic tasks create a new temporary set of their variables for each call. Automatic tasks are re-enterable. External code cannot access automatic task variables.

A task can have any number of input, output or inout ports. You can declare the ports using either syntax. Either syntax accepts port attributes. The port type defaults to a single bit. You can alternatively specify integer, real, realtime, or time, or a vector range. A vector is by default not signed. You can declare the vector signed. The signed declaration is a Verilog 2001 feature.

You can declare additional block items. You cannot declare module items. For example, you can declare variables but not nets.

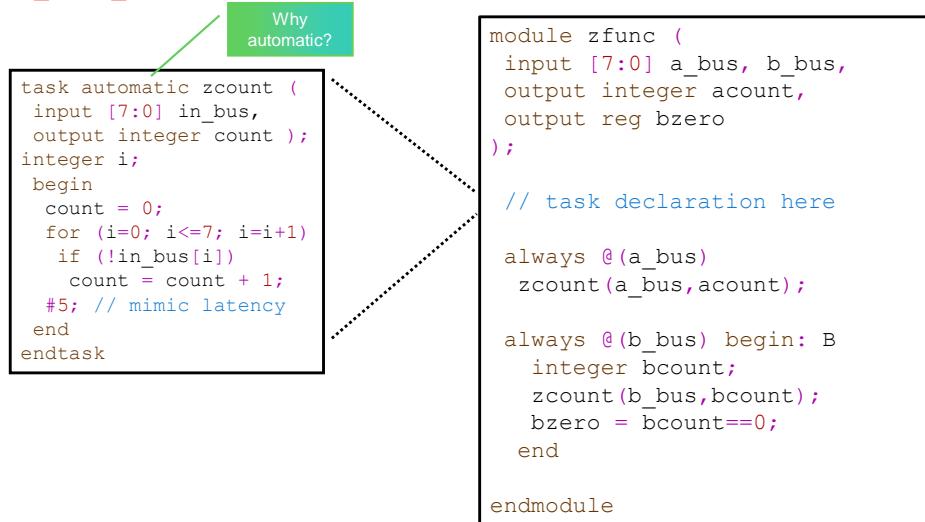
A task declaration contains a statement. The statement may be a statement block, for example grouped between the begin and end keywords. Tasks can invoke the scheduler. That means task assignments can be blocking or nonblocking and tasks can delay their completion for any amount of time. You can rewrite as a function any task that does not invoke the scheduler.

Tasks can have side effects, for example, a task can assign to a module variable but not to a module net. For a task to have side effects is a very useful feature and common programming practice. A task declaration ends with the endtask keyword.

Calling (Enabling) Tasks

You call (enable) a task as a statement.

```
hierarchical_task_identifier [(expression {,expression})];
```



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



You call a task as a procedural statement. The simulator assigns the values of the input and inout argument expressions to their ports. The standard refers to these “calls” as enables to remind you that a task can consume simulation time and thus can execute concurrently with other procedures.

When the task completes, the simulator assigns the values of the output and inout ports to their lvalue expression arguments. An lvalue expression may appear on the left side of an assignment – you can assign a value to an lvalue expression. If the task never completes, for example, in a loop that loops forever, it never returns and its calling procedure cannot continue. However, if the task “consumes” time, procedures other than the calling procedure can execute.

In this example, the zcount task returns an integer value representing the number of bits of the in_bus port that are zero. It returns a value between 0 and 8. Both task enables are procedural statements. As both procedures can concurrently have the task enabled, the task variables must be automatic variables so that each procedure gets their own copy.

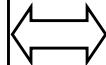
Tasks Without Timing Controls

Tasks can use the scheduler with delays and timing controls, for example.

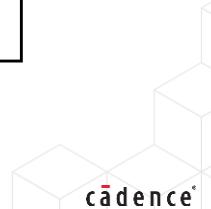
You can rewrite as a function a task that does not use the scheduler.

Here is such a situation:

```
task zcount (output integer count,
            input [7:0] in_bus);
    integer i;
    begin
        count = 0;
        for (i = 0; i <= 7; i = i + 1)
            if (!in_bus[i])
                count = count + 1;
    end
endtask
```



```
function integer zcount
(input [7:0] in_bus);
integer i;
begin
    zcount = 0;
    for (i = 0; i <= 7; i = i + 1)
        if (!in_bus[i])
            zcount = zcount + 1;
    end
endfunction
```



Tasks can invoke the scheduler. That means task assignments can be blocking or nonblocking and tasks can delay their completion for any amount of time. A task that does not invoke the scheduler operates very much like a function. You can rewrite as a function any task that does not invoke the scheduler.

Disabling Tasks

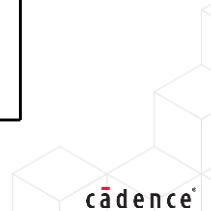
You can disable a task (force it to exit):

- In this example, detection of an interrupt disables any currently running *cpu_driver* task.
- Future statements can again enable the *cpu_driver* task.



You can also disable named blocks.

```
module busif_tb;
...
always #50 clk = ~clk;
initial
begin: STIMULUS
repeat(5) @(negedge clk);
wait(!interrupt);
cpu_driver(8'h00);
wait(!interrupt);
cpu_driver(8'haa);
...
end
...
always @(posedge interrupt)
begin
disable cpu_driver;
service_interrupt;
end
endmodule
```



The disable statement terminates activity associated with the task or named block. You can use it to work around the lack of C-like break and continue statements. You can use it to behaviorally model asynchronous activity such as interrupts.

The simulator discontinues any activity associated with the task or named block. If executing the task or named block, it resumes execution with the statement after the task call or after the named block. Implementations can choose whether to remove scheduled nonblocking assignments and whether to discontinue procedural continuous assignments, so this can be a source of differences between simulators.

Disabling a task or named block has no persistence. Subsequent statement execution can immediately again call the task and execution can immediately re-enter the named block, for example, if it is located in an always construct or loop construct.

This example repeatedly calls the *cpu_driver* task to apply stimulus. Upon arrival of an interrupt, this example terminates the current execution of the *cpu_driver* task to service the interrupt.

Issues with Functions and Tasks

Following pages address issues presented by functions and tasks:

- Function and task definitions and declarations are by default static.
 - Multiple execution threads within a function or task by default all use the same set of ports and variables.
 - As tasks can consume time, you can easily have multiple task calls simultaneously executing.
 - Functions and tasks can recursively call themselves.
- Function and task arguments are passed by value.
 - Input argument transitions are not visible to the task.
- A function or task can have side effects.
 - Can directly access variables in the scope of the module that defines it.
 - Can access (using out-of-module references) any static variable.



The following pages explore three characteristics of functions and tasks.

- The first issue is that by default there is only one copy of a subroutine's variables, so that multiple outstanding calls to the same subroutine clobber each other.
- The second issue is that arguments are passed and returned by value. The subroutines maintain local variables that get the input values upon invocation. A task cannot observe future transitions on the net or variable arguments from which the passed value was derived.
- The third issue is that subroutines can have side-effects, they can directly access the defining module's nets and variables, and can through out-of-module references access any nets and variables in the simulation. This is not an issue as much as it is a benefit, as it can greatly ease your testbench development effort.

Automatic Tasks: Re-Entering Subroutines

Verilog 1995

- All tasks and functions are static.
 - Concurrent task call and recursive function calls must be avoided.
 - Can cause conflict with internal variables and arguments.

Verilog 2001

- Can declare tasks and functions to be dynamic using keyword `automatic`.
- Allows concurrent task calls and recursive function calls.
 - These declarations cannot be accessed by hierarchical references.
 - These arguments cannot be updated with nonblocking assignment.

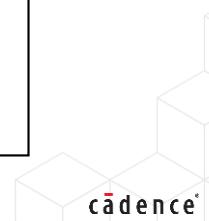
Verilog 2001

```
task automatic neg_clocks
    (input [31:0] number_of_edges);
begin
    repeat(number_of_edges)
        @(negedge clk);
end
endtask

initial begin
    neg_clocks(6);
    ...
end

always @(posedge trigger)
begin
    neg_clocks(10);
    ...
end
```

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



For Verilog 1995, all subroutines are static. Only one copy of a subroutine arguments and variables exists. Multiple concurrent task calls and recursive function calls all use the same copy of inputs, local variables, and outputs.

For Verilog 2001, you can declare an automatic subroutine. A separate copy of the subroutine arguments and variables exists for each invocation. Multiple concurrent task calls and recursive function calls all use their own copy of inputs, local variables, and outputs. As the automatic arguments and variables exist only for the duration of the subroutine execution, you cannot reference them hierarchically from outside the subroutine, cannot assign to them with nonblocking assignments, and cannot use \$monitor or \$strobe with them.

Arguments Are Passed by Value

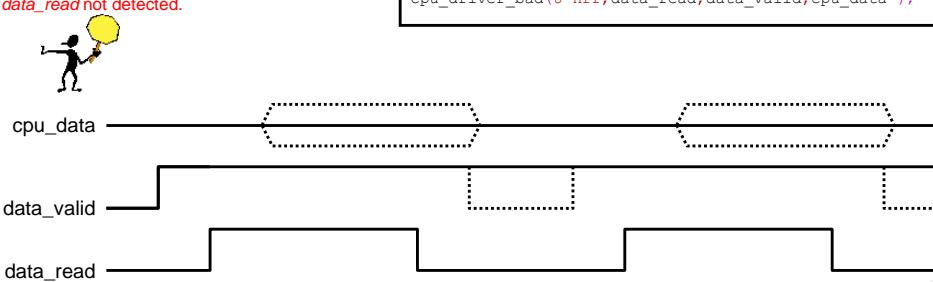
Arguments are passed by value.

- The simulator assigns expression values to input arguments upon invocation.
- The simulator assigns output values to output argument variables upon return.

Task never completes, *-data_valid* is never updated, and changes in *data_read* not detected.

```
task cpu_driver_bad;
  (input data_read,
   input [7:0] write_data,
   output data_valid,
   output [7:0] cpu_data);

begin
  #40 data_valid = 1'b1;
  wait(data_read == 1'b1);
  #20 cpu_data = write_data;
  wait(data_read == 1'b0);
  #20 cpu_data = 8'hzz;
  data_valid = 1'b0;
end
endtask
...
cpu_driver_bad(8'hff,data_read,data_valid,cpu_data );
```



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

cadence

Subroutine arguments are passed by value.

- The simulator assigns actual argument values to formal input arguments upon invocation.
- The simulator assigns output values to output argument variables upon return.

The problem with the `cpu_driver_bad` task is that the input argument `data_read` is sampled when the task is called and used throughout the task execution. External changes in `data_read` are not seen within the task during the lifetime of the task execution. Therefore the task hangs at the first wait statement.

Also the assignment to the output argument `data_valid` is not made – this will only be updated at the end of the task execution, but because the task never completes, `data_read` is never updated.

Example Task with Side-Effects

Subroutines can have “side effects,” can directly access items of their declaring module, and can access other design items **hierarchically**.

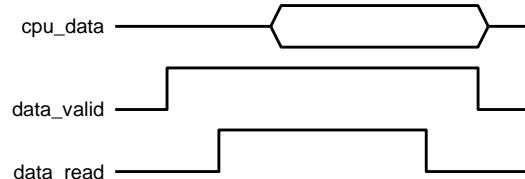
- Advantage – Allows encapsulation of a frequently used algorithm or testbench-DUT transaction.
- Disadvantage – More difficult to re-use the subroutine for other purposes.

```
task cpu_driver_good;
  input [7:0] write_data;
begin
  #40 data_valid = 1'b1;
  wait(data_read == 1'b1);
  #20 cpu_data = write_data;
  wait(data_read == 1'b0);
  #20 cpu_data = 8'hzz;
  data_valid = 1'b0;
end
endtask
...
cpu_driver_good(8'hff);
```

data_read, cpu_read, write_data
are the variables of declaring
module.



Direct references to module variables from within a function or task are resolved to variables within the *defining* module's scope and not the *calling* module's scope.



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

Subroutines can bypass their ports to read and write directly the variables of the defining module, and other design items by hierarchical reference.

This feature greatly simplifies your testbench.

This modified task directly references the data_valid and data_read module variables instead of having their values passed as argument. The wait statements inside the task can now detect transitions of the data_read variable and the task will operate correctly.

Keep in mind that direct subroutine references to module variables are to those within the defining module's scope and not the calling module's scope.

Subroutines Can Access Module Variables

```
module mytasks();

task clockit (input integer n);
  repeat(n) @(negedge busif_tb.clk);
endtask

task cpu_driver;
  input [7:0] write_data;
  begin
    #40 busif_tb.data_valid = 1'b1;
    wait(busif_tb.data_read == 1'b1);
    #20 busif_tb.cpu_data = write_data;
    wait(busif_tb.data_read == 1'b0);
    #20 busif_tb.cpu_data = 8'hzz;
    busif_tb.data_valid = 1'b0;
  end
endtask
...
endmodule
```

Upward Reference

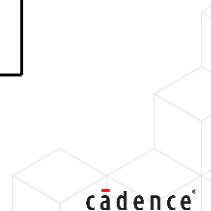
```
module busif_tb;
...
always #50 clk = ~clk;

// instantiate tasks module
mytasks m1 ();

initial
begin: STIMULUS
  m1.clockit(5);
  wait(!interrupt);
  m1.cpu_driver(8'h00);
  wait(!interrupt);
  m1.cpu_driver(8'haa);
...
end
...
endmodule
```

Downward Reference

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



At some point, you can find it convenient to declare a subroutine in one module and call it from another module.

Every Verilog object has a unique path name that starts at a top-level instance, and using the dot (.) character hierarchy separator, traverses down through instance names to the name of the object. You can use these hierarchical names for any Verilog 1995 object. You are not permitted to use the hierarchical name of a Verilog 2001 automatic subroutine variable.

- Hierarchical references can be absolute, starting at a top-level instance and traversing a downward path.
- Hierarchical references can be relative, starting at the scope of the reference and traversing a downward path.
- Hierarchical references can also be upward, but such references can have only a single instance or module name, which resolves to the nearest upward instance or module with that name.

Module Summary

You can now make your code more readable and reusable by encapsulating functionality in Verilog subroutines.

This module describes:

- Subroutine concepts:
 - Encapsulate code that might otherwise be duplicated.
- Functions:
 - Have one or more inputs and return a single value through their name.
 - Are invoked as an expression term.
- Tasks:
 - Have zero or more input and zero or more outputs.
 - Are invoked as a procedural statement.
- Issues:
 - Subroutine variables are by default static.
 - Subroutine arguments are passed by value.
 - Subroutines can directly access module nets and variables.

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



After completing this module, you can effectively use subroutines to encapsulate functionality and make your code more readable and reusable. This module described Verilog subroutines known as functions and tasks. It explained where and how to define them, how to invoke them, and some issues involved with using them.

Module Review

1. Which subroutine(s) can contain timing controls?
2. A call to which subroutine(s) can appear outside procedures?
3. What is the default type of a subroutine port?
4. By which method are subroutine arguments passed (value, pointer, reference)?



This page does not contain notes.

Module Review Solutions

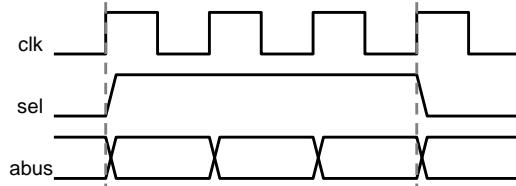
1. Which subroutine(s) can contain timing controls?
 - Only a task can contain timing controls. A function cannot invoke the scheduler in any way.
2. A call to which subroutine(s) can appear outside procedures?
 - You use a function call as a term of an expression, so a function call can appear on the right side of a continuous assignment.
3. What is the default type of a subroutine port?
 - A subroutine port is by default a single-bit reg. You can declare it as any variable type. Subroutines cannot declare nets.
4. By which method are subroutine arguments passed (value, pointer, reference)?
 - Subroutine arguments are passed by value.



This page does not contain notes.

Module Exercise

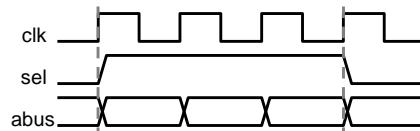
Write a task to accept a 32-bit input argument address and place that address on the “abus” and increment the address through a total of four cycles. The task also controls the “sel” signal. Refer to the diagram.



This page does not contain notes.

Module Exercise Solution

Write a task to accept a 32-bit input argument address and place that address on the “abus” and increment the address through a total of four cycles. The task also controls the “sel” signal. Refer to the diagram.



Solution:

```
task task4;
    input [31:0] addr;
    begin
        @(posedge clk)
        sel <= 1;
        abus <= addr;
        repeat(3)
            @(posedge clk)
            abus <= abus + 1;
        sel <= 0;
    end
endtask
```

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



This page does not contain notes.



Labs

Lab 10-1 Modeling the Counter Using Functions

- Encapsulate counter design combinational behaviors in a function.

Lab 10-2 Modeling the Memory Test Block Using Tasks

- Encapsulate memory test procedural behaviors in tasks.



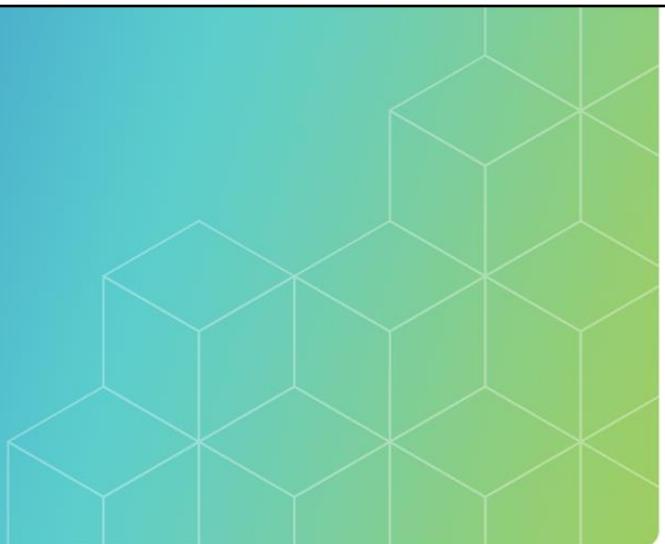
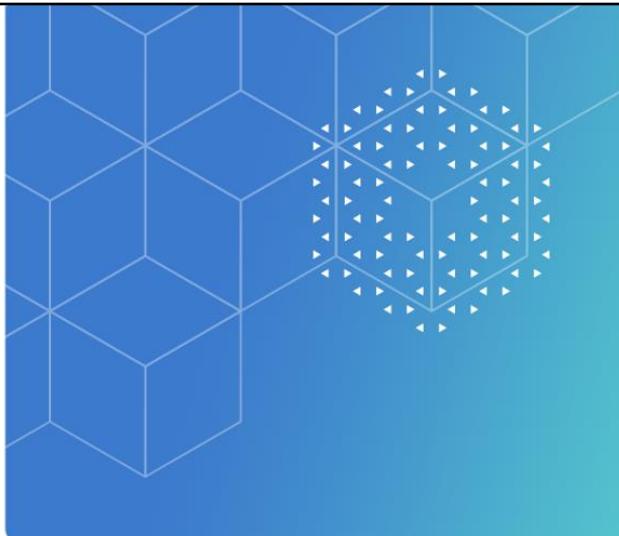
200 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to encapsulate functionality within Verilog subroutines.

For this lab, you:

- Encapsulate counter design combinational behaviors in a function; and
- Encapsulate memory test procedural behaviors in tasks.



Module 11

Directing the Compiler

cadence®

This module describes the Verilog compiler directives.

A compiler directive is a directive to the compiler. It is not syntactically a Verilog declaration or a Verilog statement. The scope of a compiler directive is from the point in the source stream input to the compiler to the point at which the directive is overridden or disabled or the end of the compilation session, probably across multiple files and multiple module descriptions, and sometimes affecting only parts of files or parts of module descriptions.

Verilog compiler directives are prefixed with the “back-tick” (`) character more formally called the accent grave. Wherever you see that character in Verilog code it is always associated with a compiler directive.

This module introduces compiler directives here. Although they are more closely related to lexical structure than to declarations and statements, you will understand them better if you have an understanding of Verilog declarations and statements. The course will later revisit many of these directives in an appropriate context that uses them.

Module Objective

In this module, you:

- Direct the compiler to interpret subsequent source code

Topics

- Substituting text
- Conditionally compiling code
- Including files
- Setting the timescale
- Reserving keywords
- Using pragmas



Your objective is to appropriately and effectively utilize compiler directives. To do that, you need to know what directives are available and what they do and how to use them.

Substituting Text: The `define Directive

Define a text substitution macro:

- `define name[(arguments)] text
- Text can use other text macros.

Use a text substitution macro:

- `name[(arguments)]
- Preprocessor substitutes text literally (WYSIWYG).

Undefine a text macro:

- `undef name

```
'define width 7
wire a, b, c;
reg [2:0] f;
reg ['width:0] mem [1:1024];

`define e {b, c, a};

always @ (posedge clk)
f<= `e ; //compiler replaces macro
           // with its definition
```



cadence®

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

You can use the text substitution facility anywhere that you want to write generic code that you can then easily replace during subsequent compilation sessions without the need to edit the target source file. You can define the text substitution macros in a separate file that you compile first during the compilation session, and many tools also provide a vendor-dependent command-line way to define text substitution macros.

As the scope of a compiler directive depends upon compilation order and thus can change between compilation sessions, you may want to instead use module parameters wherever possible for design constants such as delays and vector widths.

You can place a `define directive anywhere in your source code, but most people by convention place them all in a separate file, or at the top of the file if they are used in only one file.

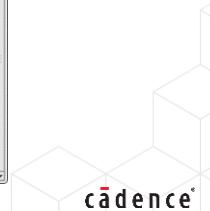
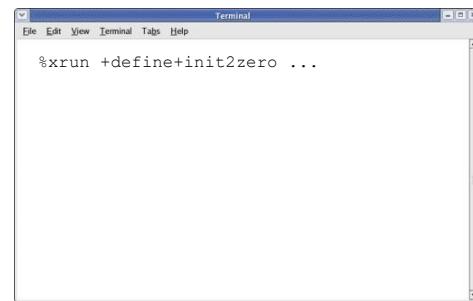
- Text macros have their own name space. You can define a macro with the same name as a module without causing confusion. You cannot name the text macro with the same name as an existing compiler directive, as that would definitely cause confusion.
- You can include a parenthesized list of formal arguments. The opening parenthesis must appear immediately after the macro identifier with no intervening white space.
- A newline character terminates the macro definition. You can escape the newline with a backslash (\) character to continue the definition on a subsequent line.
- Some constructs you cannot split across text macros. This means that if a macro definition starts the construct then it must complete the construct in that same macro definition. Constructs you cannot split across text macros are comments, identifiers, keywords, numbers, operators and strings.

Conditionally Compiling Code: The `ifdef Directive

Conditionally compile code regions:

- Verilog 1995: `ifdef, `else, `endif
- Verilog 2001: `ifndef, `elsif
- You can define the macros without providing a value
- Many tools provide a vendor-dependent command-line way to set text replacement macros

```
localparam integer SIZE = 256;
...
reg [31:0] mem [1:SIZE];
integer i;
initial
  for (i = 1; i <= SIZE; i = i + 1)
`ifndef init2zero
  mem[i] = 0;
`else
  mem[i] = i;
`endif
```



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

You can use the existence of a text macro to conditionally compile lines of source code.

You use these directives similarly in an algorithmic sense to the if-else statement, but of course keep in mind that these macros must each appear on their own line and it is the preprocessor that parses them during compile time.

This example provides the option to initialize the ROM to either all zeros or to values read from a file.

Including Files: The `include Directive

```
globals.txt  
// Clock and simulator constants  
// could also include tasks etc.  
localparam PERIOD = 10;  
localparam POSEDGE_FIRST = 1;  
localparam START_TIME = 2;  
localparam NUMBER_CYCLES = 100;
```

```
// Include global variables  
`include "globals.txt"  
reg clk;  
initial  
begin : CLKGEN  
clk = ~POSEDGE_FIRST;  
#START_TIME  
repeat(2*NUMBER_CYCLES)  
#(PERIOD/2) clk = ~clk;  
$display($time,, "Test Done");  
$finish;  
end
```

Insert the contents of a source file:

```
`include "filename"
```

- Use included files to ensure that team-wide module descriptions use the same declarations:
 - Constants
 - Tasks and functions
- The included file can contain another `include directive:
 - The standard guarantees nesting up to 15 deep.

The double-quoted argument is a literal string so cannot be a text macro.



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

The `include directive includes the contents of a file at the point where the directive appears. The effect is exactly as if the file contents were copied into the source at that point to replace the `include directive.

People use this directive to include declarations that are used by all team members and so for the sake of consistency should be maintained in one place.

The standard guarantees that you can nest included files at least 15 deep.

The argument to the `include directive is a quoted file path name and not a compiler directive or string literal. No standard way exists to generate the file name to be included.

Setting the Timescale: The `timescale Directive

Set time unit and time precision:

```
'timescale unit / precision
```

- 1 or 10 or 100 followed by unit.
- *unit* is one of fs, ps, ns, us, ms, s.
- *precision* cannot be larger than unit.
- The simulator rounds time specifications to the precision of the module and scales them to the time unit of the module.
- The overall simulation precision is the smallest of the defined precisions.
- Cadence requires either all modules to have a time scale:
 - Instead of specifying at each module we can as well give it as a command line option. The *-timescale* option provides a default timescale for the gate-level description.
 - `xrun -timescale 1ns/10ps`

Important: `timescale must appear outside of the module.

```
`timescale 10ns/1ns
module test;
localparam real delay = 2.55;
...
#delay; // 26 ns delay
...
endmodule
```

```
`timescale 1ns/100ps
module first (...);
...
#10; // 10 ns delay
...
endmodule
```

```
`timescale 1ps/100fs
module second (...);
...
#100; // 100 ps delay
...
endmodule
```

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

The `timescale directive specifies the time unit and time precision for subsequent module declarations. Any expression used in a context to mean simulation time will assume the time unit of the current module and be rounded to the time precision of the current module. System tasks that display simulation time will scale the display to the time unit of the current module.

As the standard does not prescribe a default time scale, for interoperability either all modules must be subject to a time scale or no module can be subject to a time scale. To ensure that all modules are subject to a time scale, you should get used to specifying a time scale at the top of every file, even if you not use time specifications in the file. To modify a time scale, you need to edit the file and recompile that file and all other files that utilized that directive. No method exists to modify a time scale during simulation.

In this example, the file containing the test module specifies that until further notice all subsequent time specifications are in units of 10ns and the time precision is 1ns. The 2.55 real value when used as a delay then means 25.5ns and is rounded to 26ns. The file containing the “second” module specifies that until further notice all subsequent time specifications are in units of 1ps and the time precision is 100fs. As this is the finest precision that any of these modules uses, if no other module requires a finer precision, this is the precision of the simulation.

Reserving Keywords: `begin_keywords, `end_keywords

Reserve keywords for identifiers:

```
`begin_keywords "version_spec"
`end_keywords
```

- Verilog-2005 directive for reserving keywords for user identifiers
- Place outside design element
 - config, module, primitive**
- Version specifier:
 - 1364-1995
 - 1364-2001-noconfig
 - 1364-2001
 - 1364-2005

```
`begin_keywords "1364-1995"
module m1995 (...);
task generate;
...
endtask
...
endmodule
`end_keywords
```

generate is a keyword new to Verilog-2001

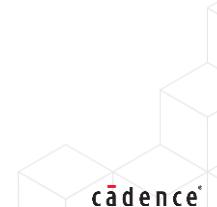
```
`begin_keywords "1364-2001-noconfig"
module design (...);
...
endmodule
`end_keywords
```

design is a configuration keyword new to Verilog-2001

```
`begin_keywords "1364-2001"
module m2001 (...);
wire unsigned uwire;
...
endmodule
`end_keywords
```

uwire is a keyword new to Verilog-2005

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



The `begin_keywords directive tells a parser to reserve for user identifiers all new keywords defined by a later standard. These keywords can appear only outside design elements, such as outside configuration, module, or primitive declarations.

If you have a legacy module that uses the generate word as a user identifier, then you can direct a Verilog 2005 compliant compiler to not use any keywords more recent than the 1995 standard for that module.

If you have a legacy module that uses the design word as a user identifier, you can direct a Verilog 2005 compliant compiler to not use any Verilog 2001 configuration keywords or any other keywords more recent than the 2001 standard for that module.

If you have a legacy module that uses the uwire word as a user identifier, you can direct a Verilog 2005 compliant compiler to not use any keywords more recent than the 2001 standard for that module.

Keywords new to IEEE Std 1364-2001

```
automatic cell config design endconfig endgenerate generate genvar incdir
include instance liblist library localparam noshowcancelled
pulsestyle_onevent pulsestyle_onedetect showcancelled signed unsigned use
```

Keywords new to IEEE Std 1364-2005

```
uwire
```

Using Pragmas: The `pragma Directive

Utilize a pragma:

```
`pragma pragma_name
[ pragma_expression
{, pragma_expression } ]
```

- Verilog-2005 structure for implementation-specific directives
- Verilog defines just a few:
 - `pragma reset
 - `pragma resetall
 - `pragma protect

```
module smux
#(parameter integer W=1)
(output [W-1:0] y, input s,
 input [W-1:0] a, b);
`pragma protect begin
assign y = s ? b : a ;
`pragma protect end
endmodule
```

Area to
protect

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



The `pragma directive provides a means for implementations to extend the set of compiler directives. A pragma name follows the directive and is followed by an optional list of pragma expressions. A pragma expression is a pragma keyword or a pragma value or an assignment of a pragma value to a pragma keyword.

The Verilog 2005 update defines only a few pragmas:

- The reset pragma resets the pragmas provided as pragma expressions;
- The resetall pragma resets all pragmas; and
- The protect pragma encrypts subsequent source code.

A implementation ignores pragmas it does not recognize.

```
pragma ::=

`pragma pragma_name [ pragma_expression { , pragma_expression } ] 

pragma_name ::= simple_identifier

pragma_expression ::=

  pragma_keyword
| pragma_keyword = pragma_value
| pragma_value

pragma_keyword ::= simple_identifier

pragma_value ::=

( pragma_expression { , pragma_expression } )
| number
| string
| identifier
```

Disabling Implicit Net Declarations

``default_nettype`

New default net type: **none**

- Verilog-1995 permits implicit net declarations.
 - Use a previously undeclared identifier in a port expression or terminal list.
 - Becomes the default net type (initially a **wire**).
 - You change the default net type with **`default_nettype`**.
 - **tri tri0 tri1 triand trior trireg wand wire wor**
- Verilog-2001 adds the default net type **none**.
 - Set this as the default net type with **`default_nettype none`**.
 - Undeclared signals become a syntax error.
 - Reduces potential for typographical error.

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



You implicitly declare a net when you use a previously undeclared identifier in a port expression or terminal list or as the lvalue of a continuous assignment. That net by default becomes a wire. You set the value of the ``default_nettype`` directive to change that default net type. The Verilog 2001 standard adds the `none` type. When you set the default net type to `none`, you require explicit declaration of all nets, thus reducing the potential for typographical error.

Note: All Verilog compiler directives start with the grave accent (sometimes called a “back-tick”). You should be aware that some fonts cannot render this character properly and display it as an acute accent (a “forward tick”).

Reference: Compiler Directives

<code>`celldefine</code>	Tags a library cell
<code>`endcelldefine</code>	
<code>`default_nettype</code>	Sets the net type for implicit net declarations
<code>`define</code>	Defines and undefines a text macro
<code>`undef</code>	
<code>`ifdef</code>	Conditionally compiles code depending upon text macro existence
<code>`else</code>	
<code>`endif</code>	
<code>`include</code>	Includes a source file
<code>`resetall</code>	Resets directives to their initial state
<code>`timescale</code>	Sets the time units and time precision
<code>`unconnected_drive</code>	Pulls up/pulls down unconnected module inputs
<code>`nounconnected_drive</code>	
<code>`ifndef</code>	Verilog-2001: More conditional compilation
<code>`elsif</code>	
<code>`line</code>	Verilog-2001: Overrides reported source file and line upon error
<code>`begin_keywords</code>	Verilog-2005: Reserves keywords for use as identifiers
<code>`end_keywords</code>	
<code>`pragma</code>	Verilog-2005: Changes how compiler interprets subsequent source

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



Use the `celldefine and `endcelldefine directives to tag modules as technology cells. PLI applications such as delay calculators and power calculators query this tag to determine the design hierarchy leaf cells that have a physical implementation.

Use the `default_nettype directive to set the default net type for implicitly declared nets. This is by default the wire type but you can set it to any net type. The Verilog 2001 standard adds the argument none which prevents implicitly declared nets.

Use the `define directive to define a text replacement macro and the `undef directive to undefine it.

Use the `ifdef, `else and `endif directives to conditionally compile source text depending upon the existence of a text macro. The Verilog 2001 standard adds the `ifndef and `elsif directives.

Use the `include directive to insert the contents of a source file at the point of the directive. The argument to the directive is a quoted file name. No standard way exists to generate the file name.

Use the `resetall directive to reset all compiler directives back to their initial state. This is very useful and recommended for placing at the beginning of every source file to ensure that only the directives set by that file are in effect during parsing of the file. This directive affects the `celldefine, `default_nettype, `timescale and `unconnected_drive directives. It does not undefine text macros, so your file can still be dependent upon text macros defined in some other file.

Use the `timescale directive to specify the time unit and time precision for subsequent module declarations. Any expression used in a context to mean simulation time will assume the time unit of the current module and be rounded to the time precision of the current module. System tasks that display simulation time will scale the display to the time unit of the current module.

Use the `pragma directive to change how the compiler interprets subsequent source. A pragma consists of a name and an optional keyword and an optional value. The standard defines only the reset and resetall keywords. Tool vendors define additional pragmas.

Module Summary

You should now be able to direct the compiler how to interpret subsequent source code.

This module described:

- The `define and `undef directives for substituting text.
- The `ifdef, `else, `endif, `ifndef, and `elsif directives for conditionally compiling code.
- The `include directive for including files.
- The `timescale directive for setting the timescale.
- The `begin_keywords and `end_keywords directives for reserving keywords.
- The `pragma directive for using pragmas.
- **The default_nettype directive from Verilog-2001.**



Your objective is to appropriately and effectively utilize compiler directives. To do that, you need to know what directives are available and what they do and how to use them.

Module Review

1. True or False: The `**undefall**` compiler directive undefines all defined text macros.
2. True or False: The standard pre-defines the VERILOG text macro that you can use to permit only the Verilog simulator to compile the code.
3. What file does this code attempt to include?
 - a. ``define ext inc`
 - b. ``define test(what) test.what.`ext`
 - c. ``include "`test(alu)"`
4. What is the IEEE-standard default timescale for a module not subject to a **`timescale`** directive?

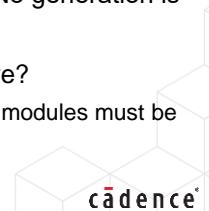


This page does not contain notes.

Module Review Solutions

1. True or False: The `undefall compiler directive undefines all defined text macros.
 - False. The standard does not describe an `undefall compiler directive. The `undef directive undefines one text macro.
 2. True or False: The standard pre-defines the VERILOG text macro that you can use to permit only the Verilog simulator to compile the code.
 - False. The standard does not pre-define any text macros. Vendors may pre-define their own text macros such as VERILOG.
 3. What file does this code attempt to include?
 - `define ext inc
 - `define test(what) test.what.`ext
 - `include "`test(alu)"
- It attempts to include the file "test(alu)". The compiler directive's argument is a string literal. No generation is permitted.
4. What is the IEEE-standard default timescale for a module not subject to a `timescale directive?
 - The standard does not define a default time scale. If ANY module is subject to a time scale then ALL modules must be subject to a timescale.

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



This page does not contain notes.



Lab

Lab 11-1 Verifying the VeriRISC CPU Design

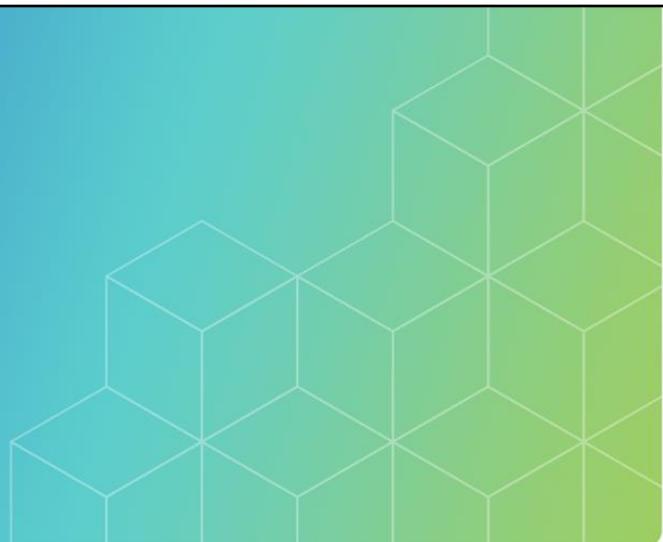
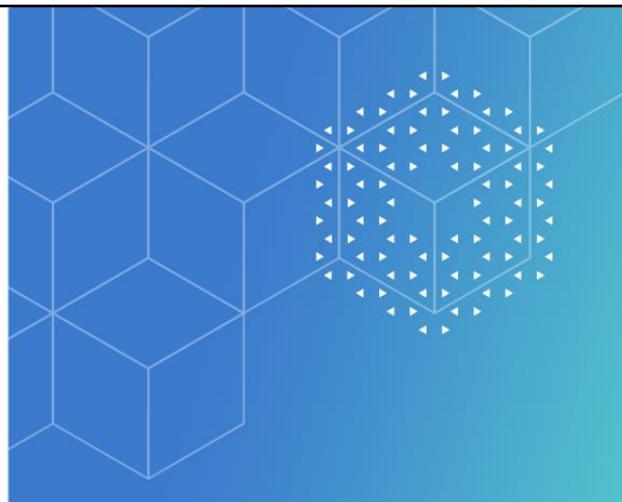
- You test or verify the VeriRISC CPU.

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



Your objective is to direct the compiler.

For this lab, you use compiler directives to define a mixed-level configuration of components.



Module 12

Introducing the Process of Synthesis

cadence®

This module explores the basics of synthesis – how it works, where it fits in your flow, its strengths and weaknesses, and some things to look out for.

Module Objective

In this module, you:

- Briefly examine the synthesis process and its results

Topics

- What is Logic Synthesis?
- How do you use synthesis?
- What does synthesis do?
- What synthesis sometimes can't do well
- Technology-specific issues
- Synthesis in your design flow
- Synthesizable Verilog

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



Your objective is to be able to briefly describe the synthesis process and its results to team members that are unable to take this course.

To do that you should know at least somewhat about:

- How you use synthesis.
- What the synthesis tool does.
- Synthesis tool strengths and weaknesses.
- Technology-specific issues.
- The Verilog language synthesis “subset”.

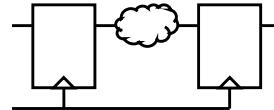
What Is Logic Synthesis?

Logic synthesis is a tool that:

- Infers storage elements (latches, registers).
- Infers combinational logic feeding storage elements.
- Optimizes generated structure to meet cost factors:
 - Area / speed / power / routability

A person new to synthesis might ask:

- How much of the design?
- How accurately?
- How optimally?



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

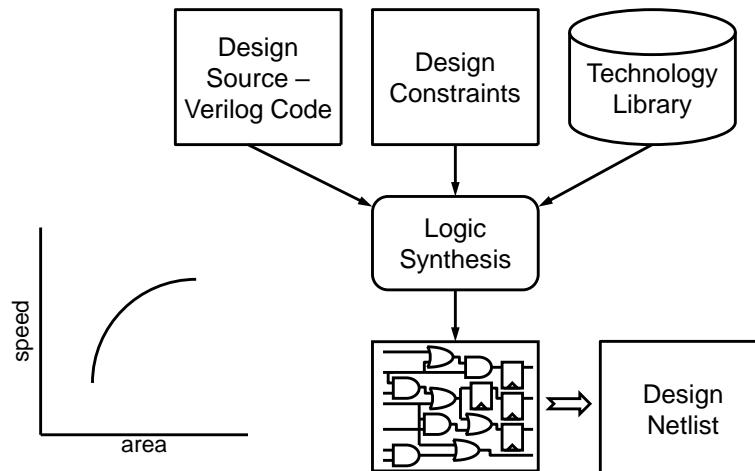
Synthesis automatically transforms your RTL Verilog design into an optimized netlist of predefined cells. Some obvious first questions might be the following:

- What parts of the design can synthesis handle?
- Is the result truly functionally equivalent to the RTL?
- Just how optimal is that result really?

How Do You Use Synthesis?

You provide design source, design constraints, and technology library.

The tool infers logic from the HDL source, maps the inferred logic to technology library macros, and optimizes the circuit to meet constraints.



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



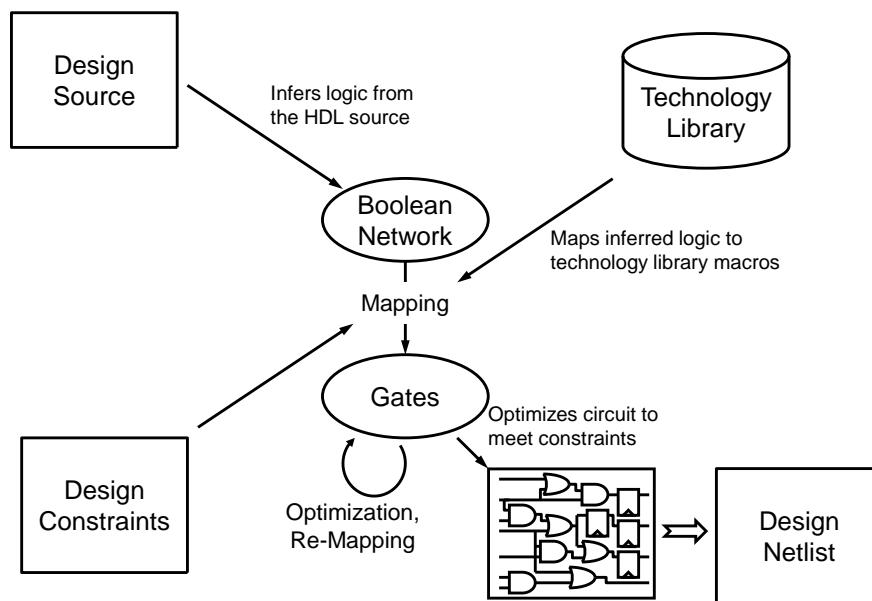
A synthesis tool can read your RTL Verilog code and convert it into a gate-level netlist, using gates and cells from a specified target technology library.

The synthesis tool will try to meet your design constraints for the area and performance, and will produce reports to tell you how successful it was.

Design constraints include the required clock speed, input drive strength and arrival time with respect to the clock, and output load and required arrival time with respect to the clock. Design constraints can also include power consumption and can sometimes include noise immunity, testability, and factors affecting ease of place and route.

The synthesis tool trades circuit speed and circuit area. As a general statement, it can produce a small design or a fast design or a compromise between small and fast. Several factors shift the area/speed curve. Among those factors are the target technology and the circuit architecture.

What Does Synthesis Do?



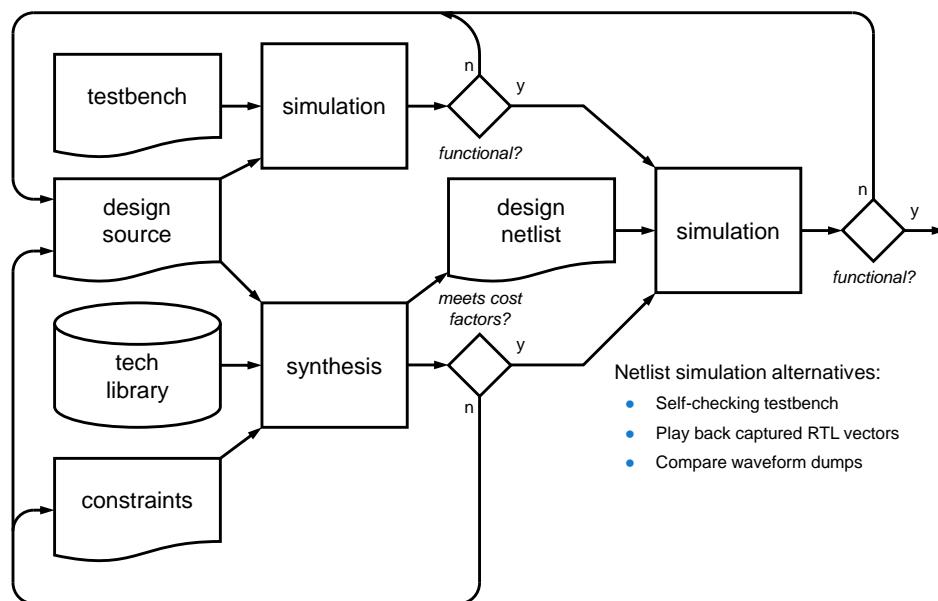
219 © Cadence Design Systems, Inc. All rights reserved.

cadence®

This diagram is of course very simplified. Synthesis technology continues to evolve and this course does not intend to describe its particulars anyway. In a very general sense:

- The tool first transforms HDL input into an internal Boolean network of sequential storage elements and the logic “cone” expressions that feed them.
- The tool then does some logic restructuring, such as to prune logic that does not drive anything and to share common subexpressions. It also at this point makes a guess about operator implementation, basing its choice on expression length. For example, for a add operation it might choose a carry look-ahead adder instead of a ripple-carry adder based on operand width.
- The tool next maps the design to the gates actually available in the target library and performs optimizations based on your constraints and the actual parameters of those gates.

Synthesis in Your Design Flow



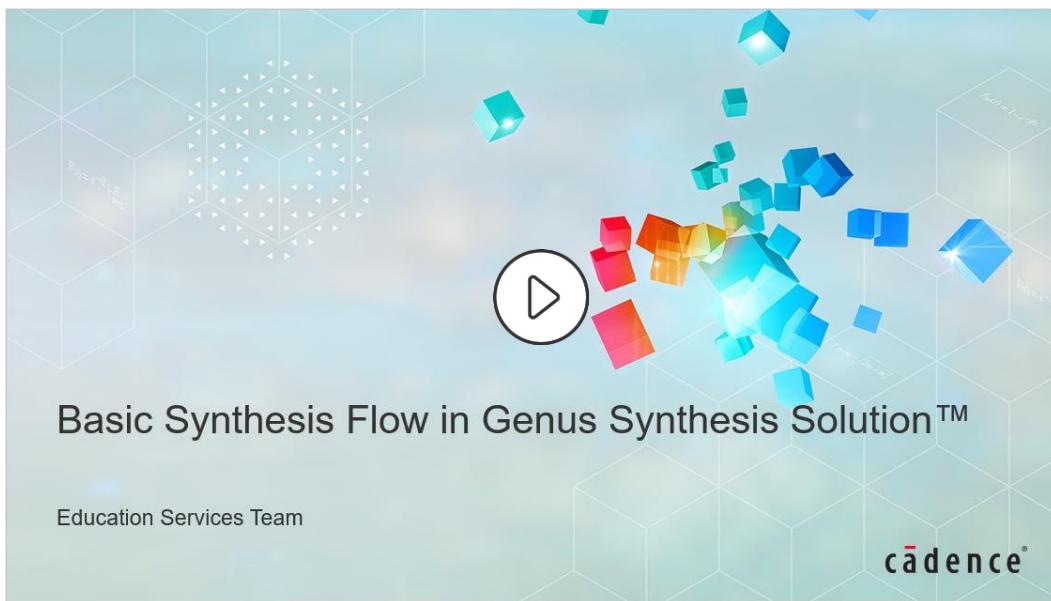
220 © Cadence Design Systems, Inc. All rights reserved.



This diagram is of course very simplified. Synthesis technology continues to evolve and this course does not intend to describe its particulars anyway. In a very general sense:

- The tool first transforms HDL input into an internal Boolean network of sequential storage elements and the logic “cone” expressions that feed them.
- The tool then does some logic restructuring, such as to prune logic that does not drive anything and to share common subexpressions. It also at this point makes a guess about operator implementation, basing its choice on expression length. For example, for a add operation it might choose a carry look-ahead adder instead of a ripple-carry adder based on operand width.
- The tool next maps the design to the gates actually available in the target library and performs optimizations based on your constraints and the actual parameters of those gates.

Video: Basic Synthesis Flow



Education Services Team

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

cadence®

Click below for a direct video link in Cadence Online Support or search for the video name in Support:

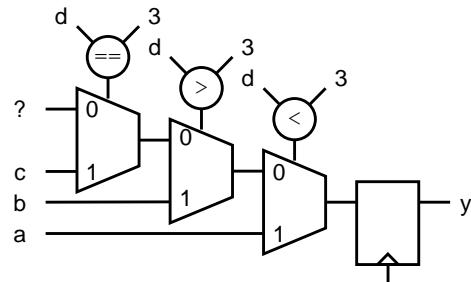
<https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1O3w000009frEfEAI&pageName=ArticleContent>

Logic Inference Is Literal

Synthesis tools infer logic from HDL structure and statements.

- They then optimize the logic as needed to meet constraints.
- Here the == operator is redundant but could still appear in the netlist.

```
always @ (posedge clk)
  if (d < 3)
    y <= a;
  else if (d > 3)
    y <= b;
  else if (d == 3)
    y <= c;
```



Synthesis tools do not interpret or analyze your code to understand your intention. They initially transform your RTL design to Boolean logic quite literally, and then optimize it. They stop optimization when the design meets your requirements or they run out of time. For any non-trivial design, synthesis produces an optimization that is hopefully “good enough” but not likely to be perfect.

You can help the optimization process by explicitly organizing and coding your design as closely as possible to what you want the optimized result to look like. Write RTL but “think” hardware. A later module discusses such RTL coding best practices for synthesis.

The design as first parsed and elaborated will reflect the RTL code. It will contain a priority structure of multiplexors and operators. If all inputs arrive at the same time, then the optimized design will likely consist of a sum of three products, the priority structure having been optimized.

Coding Style Affects Results

Synthesis tools infer logic from HDL structure and statements.

- Logic synthesis tools do not insert storage elements.
 - Pipeline considerations (number of registers) are totally your responsibility.
- Some logic synthesis tools can move storage elements upstream or downstream in combinational logic to balance delays.
- The quality of results depend mainly on code.



A synthesis tool does not optimize the registers or latches in your design. Be careful how you structure your registered processes so that you only infer the registers you need. Check synthesis reports carefully to make sure.

You also need to be careful how much combinational logic is placed between adjacent register banks. If you place a 32-bit multiplier between adjacent register banks with a 10ns clock period, don't be surprised if the synthesis tool struggles to produce a result.

Generally, you need to be careful how you partition and structure your design into combinational and registered logic.

What Synthesis Sometimes Can't Do Well

- Clock trees
 - Usually require detailed, accurate net delay information.
- Complex clocking schemes
 - Synthesis tools prefer simple, single clock synchronous designs.
- Memory, IO pads, technology-specific cells
 - You will probably need to instantiate these by hand.
- Specific macro-cells
- Synthesis tools can analyze hundreds of potential implementations in much less time than you need to analyze just one.



Logic synthesis primarily targets for optimization of the logic cones between registers. Some things that require special handling are, for example:

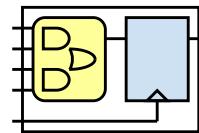
- Clock tree design is a global, chip-wide problem, which particularly in deep submicron designs requires detailed and realistic net delay information. No single clock tree generation method can apply to all designs. You will likely need to design the clock tree later as part of silicon layout.
- Synthesis tools typically have difficulty with complex clocking schemes. Synthesis works most reliably with a single clock per RTL module, and some tools restrict you to at most two clocks or require that the multiple clocks be harmonically related.
- You may have to manually instantiate I/O cells and large regularly structured cells that synthesis tools do not readily select. You may want to utilize the technology vendor's silicon compilers to generate such large regularly-structured datapath elements and large memory blocks.

For any nontrivial design, synthesis produces an optimization that is hopefully good enough but not likely to be perfect. For a critical path that does not meet timing requirements, it is quite possible that you can manually examine the path and find some way to improve it.

Technology-Specific Issues

- Each technology suggests its own optimal architecture, with architecture-specific “tricks” for good utilization and speed.
- Code becomes technology specific.
- Refer to vendor literature.

ASIC	FPGA
Technologies are similar	Technologies are not similar
Basic units are small and flexible	Basic units are large, not flexible
Optimization algorithms are similar	Optimization algorithms are not similar



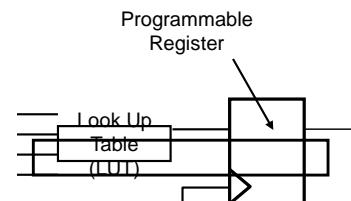
225 © Cadence Design Systems, Inc. All rights reserved.

Each technology suggests its own optimal architecture, with architecture-specific “tricks” for good utilization and speed. To understand these differences, let’s compare synthesis of an ASIC to synthesis of an FPGA:

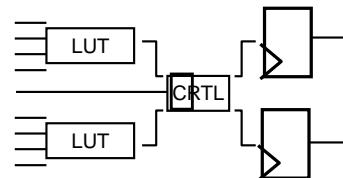
- Synthesis of an ASIC is similar for all technologies – the library cells are relatively small and similar between technologies, routing delays are relatively more predictable, and the tools use basically the same optimization algorithms.
- Synthesis of an FPGA can vary widely between technologies – the FPGA can be EEPROM, SRAM, or anti-fuse, each vendor has different building blocks that tend to be relatively large, and an FPGA architecture is relatively inflexible, leading to relatively widely-varying routing delays. FPGA design employs technology-specific and architecture-specific tricks, implemented as user-defined attributes, comments or direct instantiation, which are difficult for a synthesis tool to implement, thus making your Verilog code technology dependent and reducing its portability.

Programmable Logic Device Specific Issues

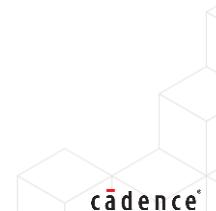
- Different architectures for different technologies
- Fixed architecture within a specific technology
- Architecture specific “tricks” for best utilization/speed
- Technology specific/generic code trade-offs
- How your synthesis tool handles your technology



Logic "Cell" – technology 1



Logic "Cell" – technology 2



Many of the comments so far have assumed synthesis to an ASIC. With an FPGA there are some different problems. ASIC synthesis tools all work in the same way, using the same optimization algorithms, and the target technology libraries all have similar cells. With FPGAs there are different technologies, EEPROM, SRAM and anti-fuse, and each vendor has different building blocks, which tend to be much larger than the gate-level primitives used in an ASIC.

For a particular FPGA, the architecture is fixed, limiting the usefulness of static timing analysis since routing delays can vary widely – with an ASIC they are much more predictable. What's more, to get a good “fit” the FPGA designer traditionally employs architecture specific tricks which are difficult to implement in a synthesis tool.

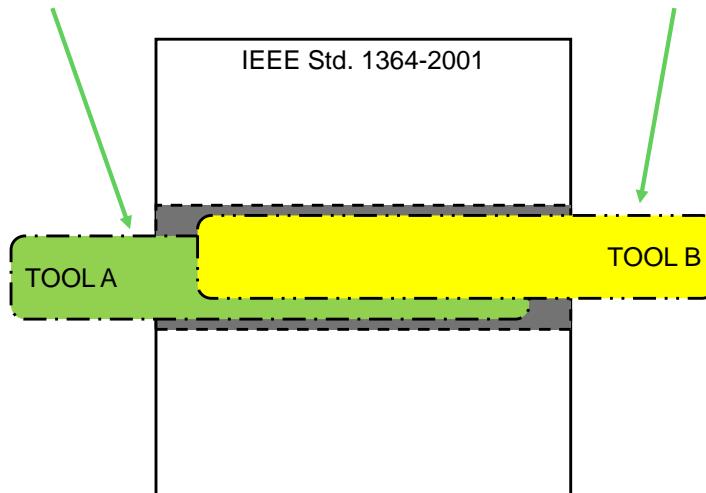
VHDL code may need to make use of technology- and architecture-specific techniques and tricks. These may be expressed with user-defined attributes, comments or direct instantiation of specific cells. Such techniques restrict the technology independence and portability of your code, but allow the most efficient implementation with a specific technology.

A key issue is how well your synthesis tool handles your chosen technology. To achieve a good result, it will need architecture specific algorithms.

Synthesizable Verilog

IEEE Std. 1364 defines the synthesizable Verilog constructs.

- Not all synthesis tools yet accept all the standard constructs
- Most synthesis tools still accept also their own legacy constructs



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



The IEEE Std. 1364.1-2002 for Verilog RTL synthesis describes use of the Verilog “synthesizable subset” in a manner from which logic synthesis tools can infer logic.

- You will rarely but perhaps occasionally use a logic synthesis tool that does not accept some standard synthesizable construct.
- As logic synthesis existed long before the standard, the predominant tools still support constructs that did not become standard.
- This course describes what is standard. If you write your RTL code according to this course, then your code will be portable between the maximum number of synthesis tools.

Module Summary

You should now be able to briefly describe what is synthesis.

This module described:

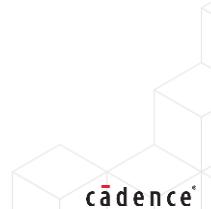
- What you provide to the synthesis tool:
 - Design source, design constraints, technology library
- What the synthesis tool does:
 - infers logic from the HDL source, maps the inferred logic to technology library macros, and optimizes the circuit to meet constraints
- What the synthesis tool might need help with:
 - Complex clocking schemes, large memories, I/O pads
- What part of Verilog is synthesizable:
 - Defined by IEEE Std. 1364.1



This module explored the basics of synthesis – how it works, where it fits in your flow, its strengths and weaknesses, and some things to watch for.

Module Review

1. What are the user-provided inputs to logic synthesis?
2. **True or False?** Given multiple different functionally accurate descriptions of a design function, a synthesis tool will for each produce the exact same one “best” implementation.
3. What design blocks might you need to instantiate rather than infer?
4. What entity determines how you must code Verilog designs for inference by a logic synthesis tool?



This page does not contain notes.

Module Review Solutions

1. What are the user-provided inputs to logic synthesis?
 - The logic synthesis user must provide the design source, design constraints, and target technology library.
2. True or False: Given multiple different functionally accurate descriptions of a design function, a synthesis tool will for each produce the exact same one “best” implementation.
 - False. A synthesis tool infers logic exactly as described, and then optimizes it to meet constraints. It stops when it meets the constraints or exceeds the computational limits you set.
3. What design blocks might you need to instantiate rather than infer?
 - You are likely to instantiate rather than infer large memories and I/O pads, and potentially at least partially instantiate clock trees.
4. What entity determines how you must code Verilog designs for inference by a logic synthesis tool?
 - The “IEEE Standard for Verilog Register Transfer Level Synthesis” determines how you must code Verilog designs for inference by a logic synthesis tool.

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



This page does not contain notes.



Lab

Lab 12-1 Exploring the Synthesis Process

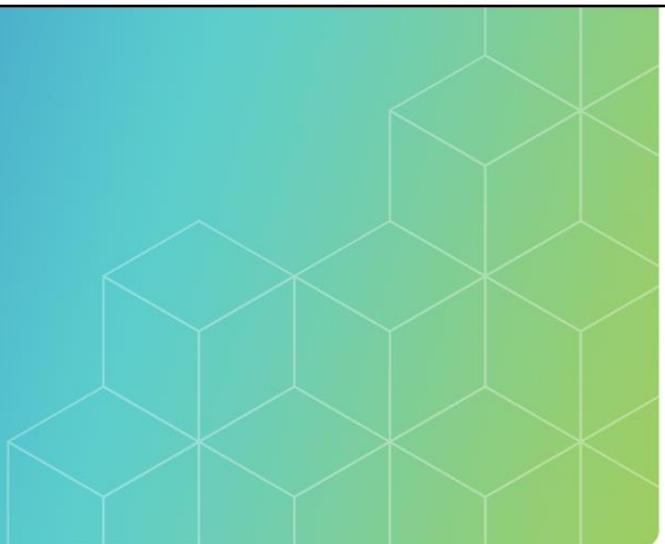
- For this lab, you synthesize a small multiplexor model and examine the results.

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



Your objective for this lab is to briefly observe the synthesis process and examine the results.

For this lab, you synthesize a simple design and examine the resulting netlist to associate the netlist components to the RTL operators and statements.



Module 13

Coding RTL for Synthesis

cadence®

This module intends to provide you with the information you need to code your RTL design in compliance with the IEEE Std 1364.1-2002 for Verilog RTL synthesis.

Module Objective

In this module, you:

- Code design behavior for logic synthesis

Topics

- Modeling combinational logic
- Modeling sequential logic
- Modeling latch logic
- Modeling three-state logic
- Using synthesis attributes

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



Your objective is to code design behavior for logic synthesis.

To do that, you need to know about:

- Modeling combinational logic;
- Modeling sequential logic;
- Modeling latch logic;
- Modeling three-state logic; and
- Using synthesis attributes.

Modeling Combinational Logic

Combinational Logic: Output is at all times a combinational function solely of the inputs.

- As a **net declaration** assignment
- As a **continuous** assignment
- As an **always** statement

```
wire w = expression;
```

```
wire w; assign w = expression;
```

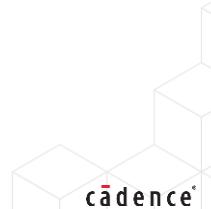
```
reg r; always @* r = expression;
```

- The event list must not contain a posedge or negedge event.
 - Include all procedure inputs to avoid mismatch between pre-synthesis and post-synthesis designs.
- Assignments must be blocking, they are sufficient and simulate more efficiently.

```
always @( all block inputs )
begin
  blocking assignments
end
```



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



You can model combinational logic with a net declaration assignment or a continuous assignment or an always statement.

When using an always statement, the single event list must not contain a posedge or negedge event. The event list does not otherwise affect the synthesis result, but to avoid incorrect RTL simulation results, you should include in the event list all inputs to the procedure. The easiest way to ensure this is to use the Verilog 2001 wildcard event control (@*).

You must not in an always statement assign a variable using both a blocking assignment (=) and a nonblocking assignment (<=). The blocking assignment is sufficient for a description of combinational logic and simulates more efficiently than the nonblocking assignment.

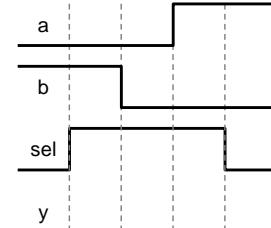
“Combinational logic shall be modeled using a continuous assignment or a net declaration assignment or an always statement. ... A variable assigned in an always statement shall not be assigned using both a blocking assignment (=) and a nonblocking assignment (<=) in the same always statement. ... The event list for a combinational logic model shall not contain the reserved words posedge or negedge.” – IEEE Std 1364.-2002 Section 5.1 Modeling combinational logic

Incomplete Event List

- For simulation, include all in the event list all signals that are input to the logic.
- The “*” wildcard sensitivity list automatically includes all input signals to the logic.

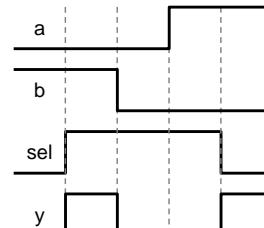
// Incomplete list

```
always @(a or b)
begin
    y = a;
    if (sel)
        y = b;
end
```



// Complete list

```
always @*
begin
    y = a;
    if (sel)
        y = b;
end
```



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



The event list does not affect the synthesis result, but to avoid incorrect RTL simulation results, you should include in the event list all inputs to the procedure. The easiest way to ensure this is to use the Verilog 2001 wildcard event control (@*).

This example illustrates the affect of an incomplete sensitivity list.

- If the event list omits the sel signal, the procedure executes upon transitions of only the a and b inputs – transitions of the sel signal have no effect.
- Debugging problems caused by an incomplete sensitivity list is difficult, so you might want to develop the habit of simply always using the wildcard event control for all combinational procedures.

“The event list does not affect the synthesized netlist.” – IEEE Std. 1364.1-2002 5.1 Modeling combinational logic

Complete Event List

Do not include temporary variables – those written and then read in the same procedure and nowhere else.

```
// Combinational logic

always @ (a or b or c)
    q = a + b + c;
```

The event list for a combinational procedure should contain all inputs to the logic.

```
// Combinational logic

always @ (a or b or c)
begin : comb_blk
    reg temp;
    temp = a + b;
    q = temp + c;
end
```



Do not place temporary variables in the sensitivity list.

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



The Verilog language lets you place any signals in the sensitivity list and to freely mix blocking and nonblocking assignments anywhere in the procedure. A simulation tool simply executes the procedure as you wrote it, using simulation semantics. However, to generate RTL simulation results consistent with those of the postsynthesis netlist, some guidelines exist:

- All edges of all signals input to combinational logic must be present in the sensitivity list. This is due to the rule that any changes on any inputs to the logic must have the opportunity to immediately affect the output. Do not place temporary variables in the sensitivity list. A temporary variable is one that the procedure writes only before it reads and that no other procedure uses. It is not an input to the logic.
- Do not mix blocking and nonblocking assignments to the same variable. Even better, you should use only blocking assignments within procedures that represent purely combinational logic. This is a recommendation to obtain higher simulation performance, as nonblocking assignments are not necessary and simulate more slowly.

The synthesis tool requires code that unambiguously states your design intentions. Code meant for the synthesis tool may use only a subset of the Verilog constructs and coding styles.

- For synthesis, it is an absolute requirement that you do not mix blocking and nonblocking assignments to the same variable!
- The synthesis standard states that the sensitivity list shall not affect the generation of combinational logic. That means that if the synthesis tool recognizes the block as combinational logic, it will proceed as if you had included all inputs to the logic in the sensitivity list. The synthesis tool may or may not warn you about missing inputs. The generated gates will simulate correctly, but very likely differently than the incorrect RTL simulation.

Incomplete Assignments

Combinational Logic

- Output is at all times a combinational function solely of the inputs.

```
// Incomplete assignment
```

```
always @ (a or b)
  if (b)
    y = a;
```

```
// Incomplete assignment
```

```
always @ (a or b)
  case (b)
    1: y = a;
  endcase
```



What is the value of `y` when `b` is 0?
How does logic synthesis implement
this behavior?

If an execution path through a combinational procedure exists that does not update the value of some output, then that output variable must retain its previous value. The synthesis tool infers a latch to implement this behavior. Latch inference is almost always not intended, and you can easily avoid it.

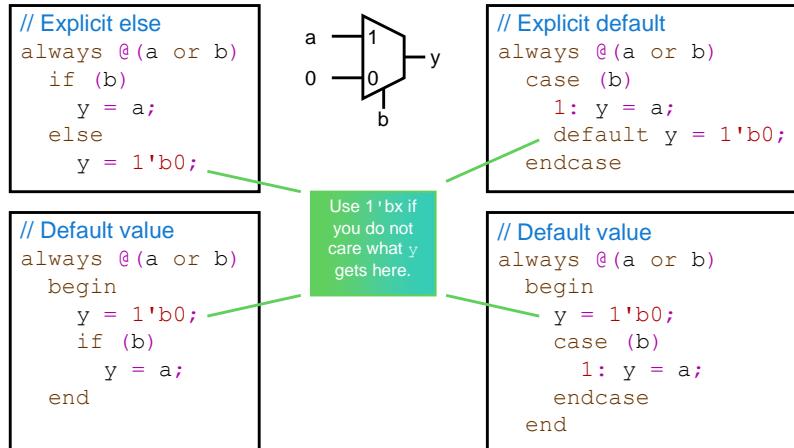
This example fails to update the `y` output variable when the `b` input is not 1.

How would you modify this code to ensure inference of purely combinational logic?

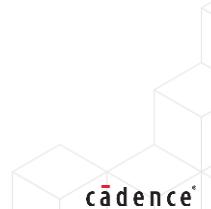
Complete Assignments

Avoiding latch inference is easy!

- Provide outputs with a value for every combination of inputs.
- This is most easily done with default assignments.



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



Here are two methods you can use to prevent latch inference:

- You can for an if statement use an explicit else clause and for a case statement an explicit default match item; or
- You can provide default values for all procedure outputs at the start of the procedure.

Which is the best technique in a real design?

If you have a procedure with a complex set of conditional assignments, you can easily miss an assignment for one or more of these branches. Making default assignments at the start of the procedure ensures that all procedure outputs have an assignment.

“The value x may be used as a primary on the RHS of an assignment to indicate a don’t care value for synthesis.” – IEEE Std. 1364.1-2002 Section 5.5 Support for values x and z

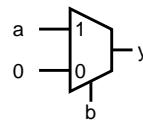
Continuous Assignments

Continuous assignments drive values onto nets.

Continuous assignments always represent combinational logic.

- Impossible to not assign a value for some input combination.
- Output is always a combinational function of current inputs.

```
assign y = b ? a : 1'b0;
```



Continuous assignments and net declaration assignments drive values onto nets. These assignments always represent combinational logic, as it is syntactically impossible to not assign a value for some input combination, thus the output is always a combinational function of the current inputs.

Modeling Combinational Logic Summary

Summary of the steps to procedurally describe purely combinational logic:

- Start the procedure with the **always** construct.
- Immediately follow the always construct with an event control.
 - Place all possible input events in the event expression.
- Group multiple following statements within a **begin...end** block.
- Use blocking assignments.
- Provide default assignments to prevent latch inference.
- Avoid combinational feedback loops.
- Assign a variable in only one procedure.

- Continuous assignments always synthesize to combinational logic.
- Subroutines almost always synthesize to combinational logic.



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

Here is a summary of the steps to procedurally describe purely combinational logic:

- Second level
- Start the procedure with the always construct and immediately follow the always construct with an event control, placing all possible input events in the event expression.
- Group multiple following statements within a begin...end block. You can omit the begin and end keywords if you have only a single statement.
- Make blocking assignments. You can equally validly make nonblocking assignments, but they simulate less efficiently and provide no additional benefit. In any case, do not mix blocking and nonblocking assignments to the same variable and do not make assignments to a variable from multiple procedures.
- Provide default assignments at the start of the procedure to prevent latch inference.
- Avoid combinational feedback loops.

Remember that:

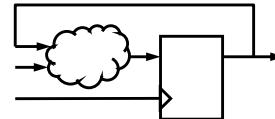
- Continuous assignments always synthesize to combinational logic.
- Functions almost always synthesize to combinational logic.

Modeling Sequential Logic

Sequential Logic

- Outputs are sampled in registers on a clock edge, thus storage is required.
- As an always statement:
 - The event list must contain only posedge and negedge events.
 - One represents the active clock edge and others represent asynchronous set and reset.
 - Model set/reset behaviors in an if statement early branches and normal behavior in the later branches.
 - Make nonblocking assignments to storage variables.

```
always @( clock edges )
begin
  nonblocking assignments
end
```



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



You model sequential logic using an always statement that has one or more posedge or negedge events in exactly one event list. Exactly one of those events represents the active clock edge that stores the value. Any additional posedge or negedge events represent asynchronous set and reset behaviors. The event list must not contain level-sensitive events.

You model the asynchronous set and reset behaviors by using an if statement. In the if statement, you model the asynchronous behaviors in one or more conditional branches. The unconditional last else branch models the normal sequential behavior.

To avoid simulation clock/data races, you should make only nonblocking assignments to variables that represent storage. You make blocking assignments to temporary variables. Temporary variables are those written and then read in the same procedure and nowhere else.

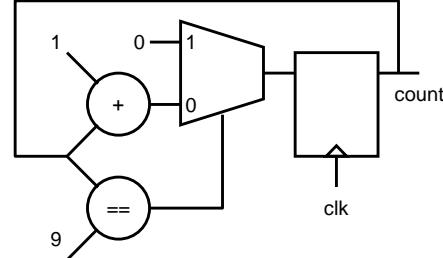
Normal Behavior

The event list for a sequential procedure must contain only single edges of clock signals.

- All events must be posedge or negedge qualified.
- The synthesis tool infers registers.
- For non-temporary variables assigned in sequential procedures:
 - Assigned with non-blocking statements.

// Sequential logic

```
always @ (posedge clk)
  if (count == 9)
    count <= 4'd0;
  else
    count <= count + 4'd1;
```



Something seems not quite right here...

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



Synthesis tools recognize sequential procedures by looking for a particular code template – in this case, a procedure with an event list containing only edge-qualified signals. Although some synthesis tools may support other coding styles for sequential procedures, your adherence to this standard produces code that you can port between all synthesis tools compliant with the standard.

This example has only the positive edge of the clock in its event list. All storage inferred for non-temporary variables that this procedure writes will have a rising active clock edge. The if statement describes the combinational logic calculating the new “count” value that is stored on the next rising clock edge.

Note that this example lacks a reset to initialize the count value!

Reset Behavior

Use an **if...else** statement to add set / reset to a procedure.

- Put the set / reset behavior in the first branch.
- Put the normal sequential behavior in the last branch.
- For asynchronous resets, add active set / reset edges to event list.

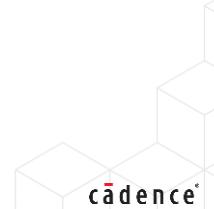
```
// Asynchronous reset
```

```
always @(posedge clk
      or posedge rst)
  if (rst)
    count <= 4'd0;
  else
    if (count == 9)
      count <= 4'd0;
    else
      count <= count + 4'd1;
```

```
// Synchronous reset
```

```
always @(posedge clk)
  // all else is same
  if (rst)
    count <= 4'd0;
  else
    if (count == 9)
      count <= 4'd0;
    else
      count <= count + 4'd1;
```

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



You can most easily provide set and reset behaviors by using an if statement. This is a requirement for asynchronous set and reset behaviors and strongly recommended for synchronous set and reset behaviors. The synthesis tool will treat synchronous set and reset behaviors you model outside an if statement as just that much more combinational logic.

In the if statement, place the set and reset behaviors in their order of priority in the first conditional branches, and place the normal synchronous behavior in the unconditional last else branch. Do not make assignments to the storage variable outside of the if statement.

For synchronous set and reset, trigger the procedure on only the clock edge.

For asynchronous set and reset, trigger the procedure also on the active set and reset edge(s).

Code synchronously reset, asynchronously reset and not reset registers in separate procedures.

Sequential Procedure Templates

Code must follow these templates:

- Procedure starts with an always construct.
- Followed by one event control containing only edge-qualified signals.
 - Clock
 - Asynchronous set or reset

```
always @ (posedge clk)
begin
    // normal
    // sequential
    // behavior
end
```

```
always @ (posedge clk)
if (!rst)
begin
    // synchronous
    // reset
    // behavior
end
else
begin
    // normal
    // sequential
    // behavior
end
```

```
always @ (posedge clk
         or negedge rst)
if (!rst)
begin
    // asynchronous
    // reset
    // behavior
end
else
begin
    // normal
    // sequential
    // behavior
end
```

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

cadence®

Sequential procedures have the clock edge in the event list, and also the set and reset edge(s) if there is asynchronous set or reset.

Separately code your not reset, synchronously reset and asynchronously reset procedures.

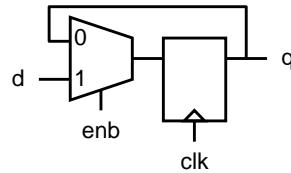
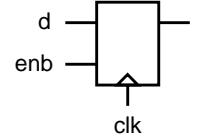
Incomplete Assignments

Sequential Logic

- Output is not at all times a combinational function solely of the inputs.
 - Implies some sort of storage.
- Incomplete assignment in a sequential procedure does not infer a latch.
 - Storage is already there!

```
always @ (posedge clk)
  if (enb)
    q <= d;
```

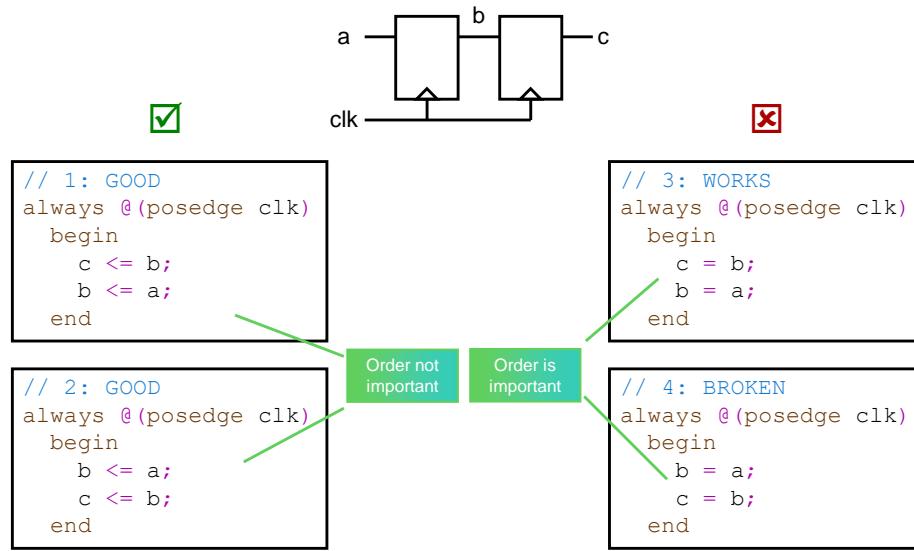
Potential Implementations



If procedure execution does not update a variable, then the variable value is not changed. In procedures representing combinational logic, this infers a latch to store the previous variable value. For procedures representing synchronous logic, the variable value is stored in the inferred register. For these procedures you do not use default assignment or else clauses to prevent latch inference.

Blocking vs. Nonblocking Assignment

Write a Verilog sequential procedure to codify this behavior.



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



Good coding practice demands that you make only nonblocking assignments to variables that represent storage elements. This is to avoid clock/data races in simulation and to avoid unintended logic inference in synthesis. The synthesis standard requires only that you not mix the two types of assignment to the same variable.

For examples 1 and 2, you can see that with nonblocking assignments the order of assignments is not important, as simulation schedules the actual assignments for the NBA region of the stratified event queue.

For examples 3 and 4, you can see that with blocking assignments the order of assignments is important, as simulation completes the assignments as it executes the statements. Different order generates different simulation results and different synthesis results.

- Example 3 reads variable **b** before writing it, so synthesis infers two registers;
- Example 4 reads variable **b** only after writing it. Variable **b** is a temporary variable. Synthesis infers one register.

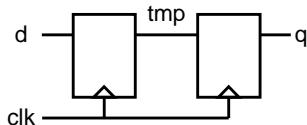
“Nonblocking procedural assignments should be used for variables that model edge-sensitive storage devices.” – IEEE Std. 1364.-2002 Section 5.2.2 Modeling edge-sensitive storage devices

Temporary Variables in Sequential Procedures

Temporary Variable

- Written first and then read (in same procedure).
- Variable is alias for expression so no register is inferred.

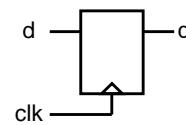
```
// 2 flop
always @(posedge clk)
begin: dff2
    reg tmp;
    q <= tmp;
    tmp = d;
end
```



Persistent Variable

- Read first and then written (in same procedure).
- Synthesis must infer a register to hold the value to the next read.

```
// 1 flop
always @(posedge clk)
begin: dff1
    reg tmp;
    tmp = d;
    q <= tmp;
end
```



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



You can use a temporary variable to hold the intermediate result of a calculation before you use the result later in the procedure. You use temporary variables to break up complex expressions and combinational logic into a series of smaller steps – making complex logic easier to describe, understand and maintain.

A temporary variable is one that a procedure writes with a blocking assignment only before the procedure reads it, and no other procedure uses it. You can declare temporary variables locally to ensure that no other procedure uses it.

A typical use model would be to make blocking assignments to temporary variables and then make nonblocking assignment of the temporary variable values to other variables.

“Blocking procedural assignments may be used for variables that are temporarily assigned and used within an always statement.” – IEEE Std. 1364-2002 Section 5.2.2 Modeling edge-sensitive storage devices

Modeling Sequential Logic Summary

Summary of the steps to procedurally describe sequential logic:

- Start the procedure with the *always* construct.
- Immediately follow the always construct with an event control.
 - Use only posedge or negedge events in the event expression.
 - Include only clock and asynchronous set / reset events.
- Group multiple following statements within a *begin...end* block.
- Place the asynchronous set / reset behavior first (in their order of priority) in the *if* statement.
- Place the normal sequential behavior in the last else branch.
 - Make blocking assignments to only the temporary variables.
 - Write temporary variables before and not after they are read.
 - Make nonblocking assignments to the register variable.
- Do not make assignments to a variable from multiple procedures.



Here is a summary of the steps to procedurally describe sequential logic:

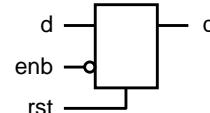
- Start the procedure with the always construct and immediately follow the always construct with an event control, placing only edge-qualified events in the event expression.
- Group multiple following statements within a begin...end block. You can omit the begin and end keywords if you have only a single statement.
- Make only blocking assignments to temporary variables and make only nonblocking assignments to variables representing storage. Write the temporary variables only before you read them in the same procedure. Do not make assignments to a variable from multiple procedures.

Modeling Latch Logic

But what if you want to infer a latch?

- A combinational block that for some combination of inputs does not provide an output value infers storage, i.e., latch.
- Make assignments for latch logic as you do for sequential logic.
 - Make blocking assignments to only the temporary variables.
 - Write temporary variables only before they are read.
 - Make nonblocking assignments to the latch variable.

```
always @(enb or rst or d)
begin: latch
  reg tmp;
  tmp = d;
  if (rst)
    q <= 0;
  else
    if (enb)
      q <= tmp;
end
```



Latch inference is almost always a mistake – be sure to document where and why you did it!



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

You deliberately infer a latch, if you truly want to, the same way you inadvertently infer a latch, with the exception that you remember to make nonblocking assignments to the variable that represents the storage.

A latch-based design can have higher performance than a register-based design, but can also be more difficult to correctly design and debug. Use of latches is typically infrequent and reserved for extenuating circumstances.

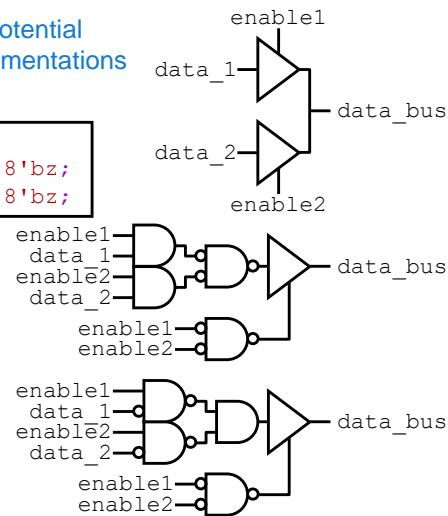
Modeling Three-State Logic

Synthesis infers three-state logic when you assign a net or variable the high-impedance value.

```
wire data_bus;
assign data_bus = enable1 ? data_1 : 8'bz;
assign data_bus = enable2 ? data_2 : 8'bz;
```

```
reg data_bus;
always @*
begin
    if (enable1)
        data_bus = data_1;
    else
        data_bus = 8'bz;
    if (enable2)
        data_bus = data_2;
    else
        data_bus = 8'bz;
end
```

Potential Implementations



The standard does not specify an implementation.



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

cadence

You model three-state logic by assigning the high-impedance (z) value to a net or variable. Any other assignments to that net or variable must also assign the high-impedance value. Further propagating the high-impedance value by use of net or variable assignments does not also make that downstream logic into three-state logic.

This example codes behavior representing three-state drivers in a form that synthesis can recognize. For this example:

- The “enable1” signal when high drives “data_1” onto the data bus; and
- The “enable2” signal when high drives “data_2” onto the data bus.

The synthesis standard does not address what the result should be if multiple enables are simultaneously true.

“Three-state logic shall be modeled when a variable is assigned the value z.” – IEEE Std 1364-2002
Section 5.4 Modeling three-state drivers.

Using Synthesis Attributes

Verilog attributes provide supporting information about HDL items.

Verilog defines only the attribute syntax and where they may appear.

```
(* name [= const_expr] {, name [= const_expr] } *)
```

- Does not define any attribute names or values

The synthesis standard defines some attributes and their purpose.

- For some the standard does not define what values to accept

Synthesis vendors add their own attributes.

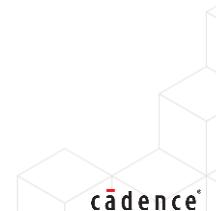
- Also other pragmas in the form of metacomments

```
(* synthesis, fsm_state = "gray" *)
reg [2:0] state;
// acme synthesis state_vector state -encoding gray
always @(posedge clk)
...
...
```

Standard Attribute

Nonstandard Metacomment

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



The IEEE Std. 1364-2001 Verilog HDL defines an attribute construct as a means to provide supporting information about an HDL construct to the various tools reading the HDL. The Verilog standard defines only the attribute syntax and where they may appear, and does not define any particular attributes.

The IEEE Std. 1364.1-2002 for Verilog RTL synthesis defines some synthesis attributes and requires that if a compliant tool supports any means to provide this information then it must support provision by means of the attributes.

As synthesis existed for many years prior to the Verilog RTL synthesis standard, synthesis vendors typically do still offer their own means to provide this information as metacomments. The industry collectively refers to these attributes and metacomments as “pragmas”.

This section will describe only a few of the standard synthesis attributes.

Attributes: “...properties about objects, statements and groups of statements in the HDL source...” – IEEE Std. 1364-2001 Section 2.8 Attributes

“An attribute instance can appear in the Verilog description as a prefix attached to a declaration, a module item, a statement, or a port connection. It can appear as a suffix to an operator or a Verilog function name in an expression.” – IEEE Std. 1364-2001 Section 2.8 Attributes

“A pragma is a generic term used to define a construct with no predefined language semantics that influences how a synthesis tool should synthesize Verilog HDL code into a circuit.” – IEEE Std. 1364.1-2002 Section 6 Pragmas

“If a synthesis tool supports pragmas to control the structure of the synthesized netlist or to give direction to the synthesis tool, attributes shall be used to convey the required information.” – IEEE Std. 1364.1-2002 Section 6.1 Synthesis attributes.

Pragma “full_case”

Synthesis tools accept a pragma to complete a **case** statement.

The directive applies to the case item expressions, not their statements!

```
module full (
    output reg a, b,
    input [1:0] sel
);
    always @*
        begin
            // a = 1'bx;
            // b = 1'bx;
            (* synthesis, full_case *)
            case (sel)
                2'b00: begin a = 1'b0; b = 1'b0; end
                2'b01: begin a = 1'b0; b = 1'b1; end
                2'b10: begin a = 1'b1; b = ...; end
                // default begin a = 1'bx; b = 1'bx; end
            endcase
        end
endmodule
```

Better Solution

Pragma

Without default value assignment you can get a latch anyway!

Equivalent to Pragma

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



The **full_case** attribute directs the synthesis tool to assign don't-care values to case statement outputs for all unspecified case choices. Synthesis can still infer a latch for outputs not assigned by all specified case items. Your use of the **full_case** attribute thus promotes inadvertent errors. A better solution is to explicitly provide default values for all procedure outputs.

Pragma “parallel_case”

Synthesis tools accept a pragma to suppress the priority structure.

```
module parallel (
    output reg a, b,
    input [1:0] sel
);
    always @*
        begin
            a = 1'b0;
            b = 1'b0;
            (* synthesis, parallel_case *)
            casez (sel)
                2'b?1: a = 1'b1;
                2'b1?: b = 1'b1;
            endcase
        end
endmodule
```



Are you *absolutely* sure that "sel" will never be 2'b11 ?



Avoid usage of "full_case parallel_case" directives with any Verilog case statements, make it correct by construction rather than pragma.

The parallel_case attribute directs the synthesis tool to disregard the implied priority of the case items and test all case items in parallel. This is in effect a declaration that the case items are mutually exclusive in normal operation, and can produce a more optimal postsynthesis netlist. You must be very careful to not declare case items to be mutually exclusive if they are not.

Pragma “implementation”

- Synthesis tools accept a pragma to recommend an operator architecture. The synthesis vendor defines the legitimate attribute values.

```
module adder (input [7:0] a, b, output [8:0] sum);
    assign sum = a + (* synthesis, implementation = "cla" *) b;
endmodule
```

Recommend
carry-lookahead
for performance

- It is generally a good coding style to give the synthesis tool and simulator same information about the functionality of a design.
 - Make the code correct by construction rather than using pragma. (Pragmas can be found majorly in some legacy code.)



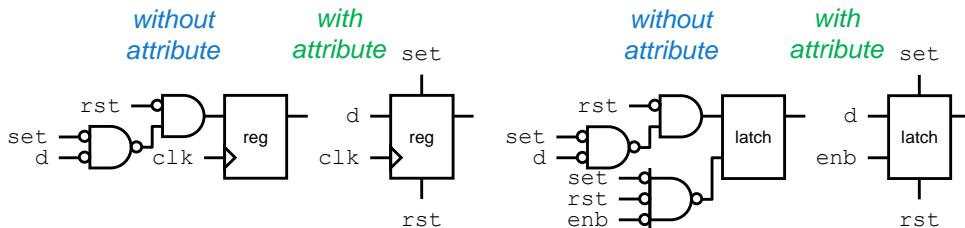
The implementation attribute recommends an architecture for operator implementation. The synthesis vendor defines the legitimate values. The synthesis tool can legally ignore the recommendation.

Pragma [a] sync_set_reset

Synthesis tools accept a pragma to cause direct connection of set / reset signals to set / reset terminals and optionally designate set / reset signals.

```
(* synthesis, sync_set_reset *)
always @ (posedge clk)
  if (rst)
    q <= 0;
  else
    if (set)
      q <= 1;
    else
      q <= d;
```

```
(* synthesis, async_set_reset *)
always @*
  if (rst)
    q <= 0;
  else
    if (set)
      q <= 1;
    else
      if (enb)
        q <= d;
```



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



The sync_set_reset attribute causes synthesis to directly connect synchronous set and reset signals to the device set and reset terminals, if a device with such terminals exists in the library. You can apply this attribute to a module or to a procedure. If you also designate the set or reset signals, then synthesis will not directly connect any other set or reset signals to the device set and reset terminals, but will instead route them into the data logic.

The async_set_reset attribute has exactly the same effect upon latch logic.

Reference: Synthesis Attributes

Attribute	Value	Description
(* synthesis, async_set_reset	[= "sig(.sig)"] *)	Identifies signals for device terminals
(* synthesis, black_box	[= value] *)	Ignores module or instance internals
(* synthesis, combinational	[= value] *)	Verifies nature of module or procedure
(* synthesis, fsm_state	[= encoding] *)	Specifies FSM state encoding
(* synthesis, full_case	[= value] *)	Unmatched expr is don't-care
(* synthesis, implementation	= value *)	Recommended operator architecture
(* synthesis, keep	[= value] *)	Leaves instance/module/net/reg "as is"
(* synthesis, label	= "name" *)	Labels an item for later reference
(* synthesis, logic_block	[= value] *)	Implements RAM/ROM as discrete logic
(* synthesis, op_sharing	[= value] *)	Enables module operator sharing
(* synthesis, parallel_case	[= value] *)	Suppresses case priority scheme
(* synthesis, ram_block	[= value] *)	Designates inferred RAM device
(* synthesis, rom_block	[= value] *)	Designates inferred ROM device
(* synthesis, sync_set_reset	[= "sig(.sig)"] *)	Identifies signals for device terminals
(* synthesis, probe_port	[= value] *)	Brings net or reg out to top level port
(* synthesis, translate off *)		Ignores portions of the design code not relevant to logic synthesis
(*synthesis, translate on *)		Resumes synthesis for the rest of the code

The `async_set_reset` attribute specifies that asynchronous set and reset signals of the module or procedure to which it applies shall directly connect to the asynchronous set and reset terminals of the library device if the library provides a device with such terminals. If the attribute specifies signal names, then any other set and reset signals shall not directly connect to asynchronous set and reset terminals of the device.

The `black_box` attribute specifies that synthesis shall ignore the internal contents of the instance or module to which it applies.

The `combinational` attribute specifies that synthesis shall report as an error any storage inferred by the module or procedure to which it applies.

The `fsm_state` attribute specifies a state encoding scheme for the register to which it applies.

The `full_case` attribute specifies that you don't care what the outputs are for unspecified case choices.

The `implementation` attribute recommends an operator architecture.

The `keep` attribute specifies that synthesis shall preserve an instance, module, net or register "as is".

The `label` attribute labels an item for future reference.

The `logic_block` attribute supports implementation of a RAM or ROM block in discrete gates.

The `op_sharing` attribute can disable operator sharing within a module. Operator sharing may occur anyway without the attribute. With the attribute enabled, operator sharing may still not occur.

The `parallel_case` attribute specifies that synthesis should suppress the normal case statement priority architecture.

The `ram_block` attribute supports selection of an inferred RAM device.

The `rom_block` attribute supports selection of an inferred ROM device.

The `sync_set_reset` attribute specifies that synchronous set and reset signals of the module or procedure to which it applies shall directly connect to the synchronous set and reset terminals of the library device if the library provides a device with such terminals. If the attribute specifies signal names, then any other set and reset signals shall not directly connect to synchronous set and reset terminals of the device.

The `probe_port` attribute causes synthesis to provide a top-level output port connected to the net or register.

Reference: Unsupported Verilog Constructs

Synthesis tools compliant with IEEE Std. 1364.1 do not support:

assign/deassign (as a procedural statement)	repeat
defparam	time
disable	tri0/tri1/trireg
event	cmos/nmos/pmos/rcmos/rnmos/rpmos
force/release	tran/tranif0/tranif1/rtran/rtranif0/rtranif1
forever	wait
fork/join	while
macromodule	-> (event emission)
primitive	==!=/!= (case identity)
pulldown/pullup	expression in event list
real	hierarchical identifiers
realtime	system functions (except \$signed/\$unsigned)
	** (power operator, except as power of 2)

Synthesis tools compliant with IEEE Std. 1364.1 ignore:

#delay (if occurs after event control)	system tasks
initial (except supported for ROM modeling)	variable declaration assignment
specify (entire block)	`celldefine, `endcelldefine, `line, `timescale, `unconnected_drive,
strength specifications	`nounconnected_drive

Here for your reference, are constructs that the synthesis tool does not support, and constructs that the synthesis tool ignores.

Synthesis supports the always keyword only with an immediately following event control (@).

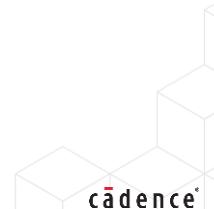
Module Summary

You should now be able to properly code hardware for logic synthesis.

This module discussed the following:

- Modeling combinational logic
 - In an **always** block with complete sensitivity list, blocking assignments, and all block outputs assigned each time the block is executed.
- Modeling sequential logic
 - In an **always** block sensitive to edge-qualified events, nonblocking assignments, and set/reset behavior in early conditional statement branches.
- Modeling latch logic
 - By deliberately omitting an output value for some combination of input values.
- Modeling three-state logic
 - By assigning the high-impedance (Z) state when disabled.
- Using synthesis attributes
 - E.g., (* synthesis, implementation = "cla" *).

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



This module explained how to describe hardware for logic synthesis. It addressed combinational, sequential, latch, and three-state logic. It introduced synthesis attributes that you embed in the source code to influence the synthesis process.

Module Review

1. For what conditions does logic synthesis infer a latch in logic you intend to be purely combinational?
2. Explain why a synthesis tool may or may not infer a register for a blocking assignment to a variable.
3. For what Verilog construct does synthesis infer a three-state gate?
4. In what most significant way does synthesis restrict a **for** loop?



This page does not contain notes.

Module Review Solutions

1. For what conditions does logic synthesis infer a latch in logic you intend to be purely combinational?
 - If you fail to specify an output value for at least one combination of input values then logic synthesis will insert a latch.
2. Explain why a synthesis tool may or may not infer a register for a blocking assignment to a variable.
 - Synthesis infers a register for a blocking assignment to a variable that occurs after the variable is already read in the execution of a sequential procedure. If the variable is written before it is read then it is only a temporary variable.
3. For what Verilog construct does synthesis infer a three-state gate?
 - Synthesis infers a three-state gate if you conditionally assign the high-impedance value to a variable.
4. In what most significant way does synthesis restrict a **for** loop?
 - You can synthesize a for loop if the number of iterations is a compile-time constant. The synthesis tool unrolls the loop so needs to know the exact number of iterations.

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



This page does not contain notes.

Module Exercise

Fix these procedures to make them legal for synthesis:

```
// 1  
always @ (posedge clk)  
  if (clk)  
    q <= d;  
  else  
    if (rst)  
      q <= 0;
```

```
// 2  
always @ (posedge clk  
          or rst)  
  if (rst)  
    q <= 0;  
  else  
    q <= d;
```

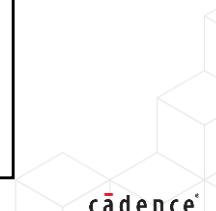
```
// 3  
always @ (posedge clk  
          or negedge rst)  
  if (rst)  
    q <= 0;  
  else  
    q <= d;
```

```
// 4  
always @ (posedge clk  
          || posedge rst)  
  if (rst)  
    q <= 0;  
  else  
    q <= d;
```

```
// 5  
always @ (posedge clk)  
  if (!rst)  
    q <= d;  
  
always @ (posedge rst)  
  q <= 0;
```

```
// 6  
always @ (posedge clk  
          or posedge rst)  
begin  
  if (clk)  
    q <= d;  
  else  
    q <= 0;  
  if (rst)  
    q <= 0;  
  else  
    q <= d;  
end
```

Refer to the synthesis templates



This page does not contain notes.

Module Exercise Solution

Fix these procedures to make them legal for synthesis:

```
// 1  
always @ (posedge clk)  
  if /*clk*/rst)  
    q <= /*d*/0;  
  else  
// if (rst)  
    q <= /*0*/d;
```

```
// 2  
always @ (posedge clk  
          or posedge rst)  
  if (rst)  
    q <= 0;  
  else  
    q <= d;
```

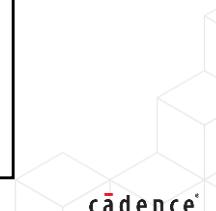
```
// 3  
always @ (posedge clk  
          or negedge rst)  
  if (~rst)  
    q <= 0;  
  else  
    q <= d;
```

```
// 4  
always @ (posedge clk  
/*||*/or posedge rst)  
  if (rst)  
    q <= 0;  
  else  
    q <= d;
```

```
// 5  
always @ (posedge clk)  
  if /*!*/rst)  
    q <= /*d*/0;  
  else q <= d;  
//always @ (posedge rst)  
// q <= 0;
```

```
// 6  
always @ (posedge clk  
          or posedge rst)  
begin  
// if (clk)  
//   q <= d;  
// else  
//   q <= 0;  
if (rst)  
  q <= 0;  
else  
  q <= d;  
end
```

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



This page does not contain notes.



Lab

Lab 13-1 Using a Component Library

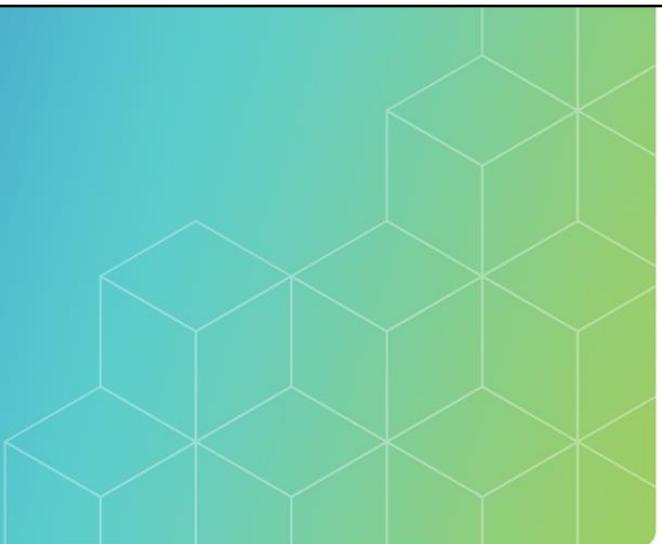
- For this Lab, you demonstrate mastery of basic coding styles.
- And you code and synthesize some representative models.

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



Your objective for this lab is to describe design behavior for logic synthesis.

For this lab, you code and synthesize several simple models and observe the results of different coding techniques.



Module 14

Designing Finite State Machines

cadence®

This module more specifically presents the various ways to code a state machine for synthesis.

Module Objective

In this module, you:

- Code state machines for synthesis

Topics

- FSM introduction
- Defining the FSM states
- Example read-write synchronizer FSM
- Coding FSMs in various styles
- Various ways to optimize FSMs



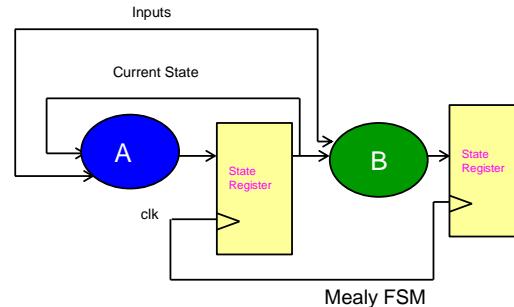
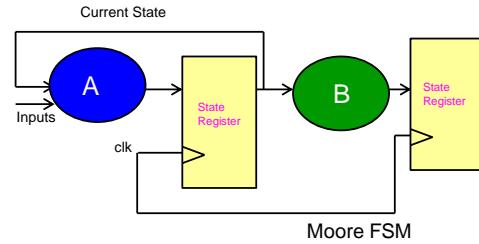
Your objective is to code state machines for synthesis.

To help you do that, this module discusses what an FSM is, how to define FSM states, various ways to code FSMs for synthesis, and various ways to optimize FSMs.

Finite State Machine (FSM) Review

FSM Structure

- **State Register**
 - Stores current state
- **Next State decode logic**
 - Decides next state based on current state and inputs
- **Output Logic**
 - Decodes state (or states and inputs) to produce outputs
- Outputs from the FSM can be a function of:
 - Current state only – **Moore**
 - Current state and the current inputs – **Mealy**



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



An FSM consists of state-encoding combinational logic, state-storing sequential elements, and output-decoding combinational logic.

- A Moore machine has no combinational path from inputs to outputs. The outputs are a function solely of the current state vector.
- A Mealy machine has at least one combinational from an input to an output. The outputs are a function of the current state and at least one current input. Mealy outputs can be available up to one clock earlier path than Moore outputs, but may complicate synthesis timing constraint definition.

If the current state of your design depends at least partially upon the previous state, then your design is an FSM.

Perhaps one way to remember which is the Mealy machine and which is the Moore machine is the phrase “Mealy is more and Moore is less”. This refers to the outputs, which in the Mealy machine can include unregistered inputs.

Defining the FSM States

You will likely see FSM descriptions that define state vector values with text replacement macros. This training does not do that and you should neither.

Macros

```
`define IDLE 2'd0
`define READ 2'd1
`define WRITE 2'd2
`define DONE 2'd3
reg [1:0] state, nstate;

always @*
  case ( state )
    `IDLE: nstate = do_write ? `WRITE
                                : `READ;
    ...
  end
```

- Scope is global – across files and modules from `define to `undef
- Accepted by synthesis tools but not for FSM optimization

Parameters

```
localparam IDLE  = 2'd0,
           READ   = 2'd1,
           WRITE  = 2'd2,
           DONE   = 2'd3;
reg [1:0] state, nstate;

always @*
  case ( state )
    IDLE: nstate = do_write ? WRITE
                               : READ;
    ...
  end
```

- Scope is local to declaring block
- Required by synthesis tools that perform FSM optimizations



The scope of any compiler directive is from the point of the direction to the point of its redirection or removal, potentially across multiple files and multiple modules. To inadvertently use a macro defined elsewhere or to define a macro inadvertently used elsewhere is very easy. To help prevent such erroneous use, designers typically place text macro definitions in a separate file that any compilation unit that uses the definitions includes during compilation. This encapsulates those definitions in a single easily-modified place.

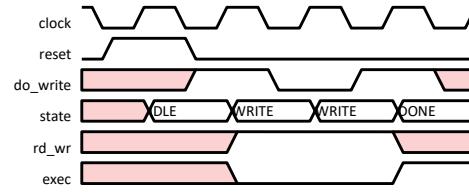
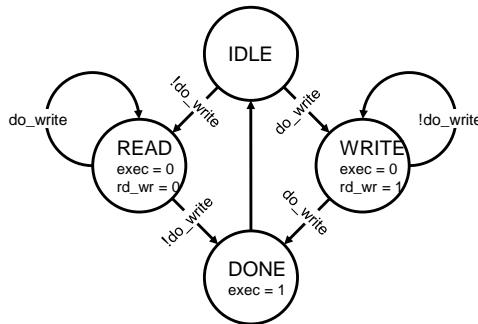
The scope of a local parameter is the declaring block. No code can change its initialized value.

Synthesis tools that perform state machine optimizations require you to define the states with parameters if you want the tool to perform the optimizations.

Example Read-Write Synchronizer FSM

The following examples refer to this FSM specification:

- If `do_write` is true, transition to `WRITE` and set `exec = 0` and `rd_wr = 1`.
 - When `do_write` is again true, transition to `DONE` and set `exec = 1`.
- If `do_write` is false, transition to `READ` and set `exec = 0` and `rd_wr = 0`.
 - When `do_write` is again false, transition to `DONE` and set `exec = 1`.



Outputs(`exec` and `rd_wr`) are considered unknown at reset for simplicity, in reality it may be either 1 or 0.

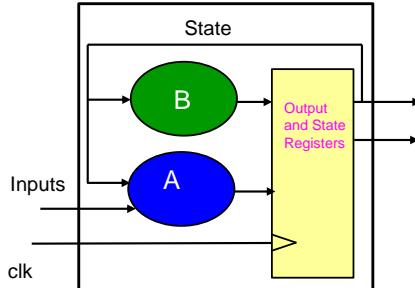
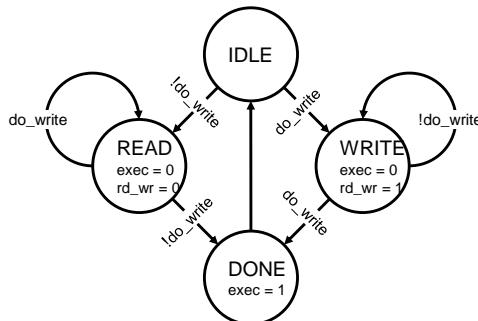


268 © Cadence Design Systems, Inc. All rights reserved.

The following examples refer to this state machine specification:

- If `do_write` is true, it transitions to the `WRITE` state and sets `exec` to 0 and `rd_wr` to 1. When `do_write` is again true, it transitions to the `DONE` state and sets `exec` to 1.
- If `do_write` is false, it transitions to the `READ` state and sets `exec` to 0 and `rd_wr` to 0. When `do_write` is again false, it transitions to the `DONE` state and sets `exec` to 1.

Coding the FSM in One Block: Sequential Outputs



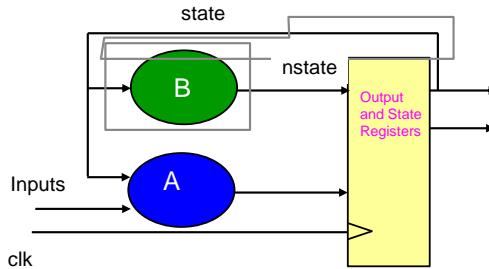
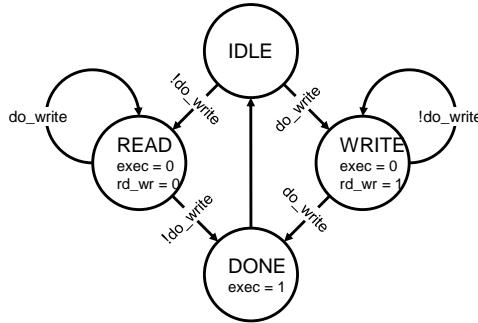
```

localparam IDLE = 2'd0, READ = 2'd1,
          WRITE = 2'd2, DONE = 2'd3;
reg [1:0] state;
reg exec, rd_wr;

always @(posedge clock)
  if (reset)
    state <= IDLE;
  else
    case (state)
      IDLE: begin
        exec <= 0;
        rd_wr <= do_write;
        state <= do_write ? WRITE
                           : READ;
      end
      READ: if ( !do_write )
        {state,exec}<={DONE,1'b1};
      WRITE: if ( do_write )
        {state,exec}<={DONE,1'b1};
      DONE: state <= IDLE;
    endcase
  
```

The one-block coding style generates the next-state, state, and outputs all in one sequential block.

Coding the FSM in Two Blocks: Sequential Outputs



```

localparam IDLE  = 2'd0, READ  = 2'd1,
          WRITE = 2'd2, DONE  = 2'd3;
reg [1:0] state, nstate;
reg exec, rd_wr;

always @*
  case ( state )
    IDLE: nstate= do_write? WRITE: READ;
    READ: nstate= !do_write? DONE : READ;
    WRITE: nstate= do_write? DONE : WRITE;
    DONE: nstate= IDLE;
  endcase

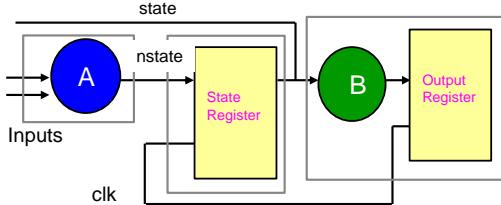
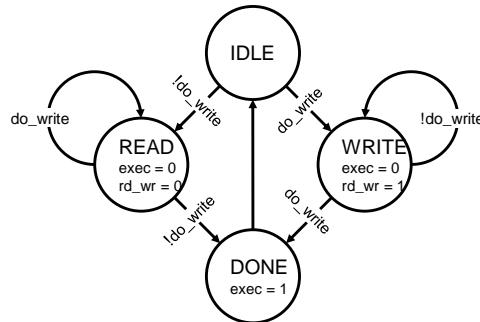
always @(posedge clock)
  if ( reset )
    state <= IDLE;
  else
    begin
      state <= nstate;
      case ( nstate )
        READ: {rd_wr,exec} <= 2'b00;
        WRITE: {rd_wr,exec} <= 2'b10;
        DONE: exec <= 1;
      endcase
    end
  
```

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

cadence®

The two-block coding style encodes the next state in a separate combinational block. You must place asynchronous reset in the sequential block. You can place synchronous reset in either block, but synthesis tools might more accurately identify the component's reset pin if you place the reset in the sequential block.

Coding the FSM in Three Blocks: Sequential Outputs



```

localparam IDLE  = 2'd0, READ  = 2'd1,
          WRITE = 2'd2, DONE  = 2'd3;
reg [1:0] state, nstate;
reg exec, rd_wr;

always @*
case ( state )
  IDLE: nstate= do_write? WRITE: READ;
  READ: nstate= !do_write? DONE : READ;
  WRITE: nstate= do_write? DONE : WRITE;
  DONE: nstate= IDLE;
endcase

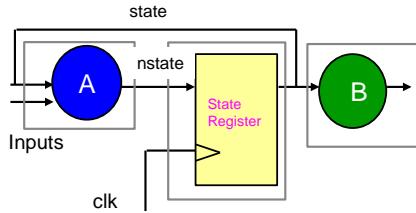
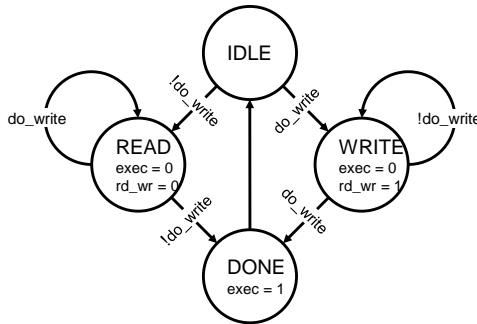
always @ (posedge clock)
if ( reset )
  state <= IDLE;
else
  state <= nstate;

always @ (posedge clock)
if ( !reset )
  case ( nstate )
    READ: {rd_wr,exec} <= 2'b00;
    WRITE: {rd_wr,exec} <= 2'b10;
    DONE: exec <= 1;
  endcase

```

The three-block coding style decodes the outputs in a third block. Here, that third block is sequential. Note that the outputs do not transition while the reset is active.

Coding in Three Blocks: Combinational Outputs



```

localparam IDLE  = 2'd0, READ  = 2'd1,
          WRITE = 2'd2, DONE  = 2'd3;
reg [1:0] state, nstate;
reg exec, rd_wr;

always @*
case ( state )
  IDLE: nstate= do_write? WRITE: READ;
  READ: nstate= !do_write? DONE : READ;
  WRITE: nstate= do_write? DONE : WRITE;
  DONE: nstate= IDLE;
endcase

always @(posedge clock)
if ( reset )
  state <= IDLE;
else
  state <= nstate;

always @*
case ( state )
  IDLE: {rd_wr,exec} = 2'bxx;
  READ: {rd_wr,exec} = 2'b00;
  WRITE: {rd_wr,exec} = 2'b10;
  DONE: {rd_wr,exec} = 2'b11;
endcase
  
```

The three-block coding style decodes the outputs in a third block. Here, that third block is combinational. The combinational block takes advantage of the specification that for some states we do not care what the output value is.

Optimizing Register Count

- Registering outputs is preferable for synthesis as it simplifies timing constraints.
- Examine the mapping between state and output.
- Can output registers replace state bits? If so then you can reduce your overall register count.

State	Encoding	rd_wr	exec
IDLE	01	?	?
READ	00	0	0
WRITE	10	1	0
DONE	11	?	1
match:		state[1]	state[0]

```

localparam READ  = 2'd0, IDLE = 2'd1,
          WRITE = 2'd2, DONE = 2'd3;
reg [1:0] state, nstate;
reg exec, rd_wr;

always @*
  case ( state )
    IDLE: nstate= do_write? WRITE: READ;
    READ: nstate= !do_write? DONE : READ;
    WRITE: nstate= do_write? DONE : WRITE;
    DONE: nstate= IDLE;
  endcase

always @(posedge clock)
  if ( reset )
    state <= IDLE;
  else
    state <= nstate;

always @*
begin
  rd_wr = state[1];
  exec  = state[0];
end

```



This page does not contain notes.

Optimizing Power and Noise: Gray Encoding

Where vectors regularly traverse a range of values, opportunities exist to encode values in consideration of power efficiency and noise reduction. Perhaps the most classic of examples is Gray-encoded counting.

Binary	Transitions
00	2
01	1
10	2
11	1

mean: 1.5

Gray	Transitions
00	1
01	1
11	1
10	1

mean: 1

Gray code effective for predictable state patterns with a specific number of state transitions.

State	Transitions
(00)IDLE	2
(01)READ	1
(11)DONE	1

mean: 1.3

State	Transitions
(00)IDLE	2
(10)WRITE	1
(11)DONE	1

mean: 1.3

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



Bell Labs researcher Frank Gray applied for patent 2,632,058 “Pulse Code Communication” in 1947, defining a “reflected binary code” for lack of a better term. Coincidentally with the patent award, the codes became popularly known as Gray codes. A Gray code is “a binary numeral system where two successive values differ in only one digit” – http://en.wikipedia.org/wiki/Gray_code

The tabulated sequence is the original “binary reflected Gray code” (that you can generate from binary using $g=b^{(b>>1)}$) but other Gray codes also exist.

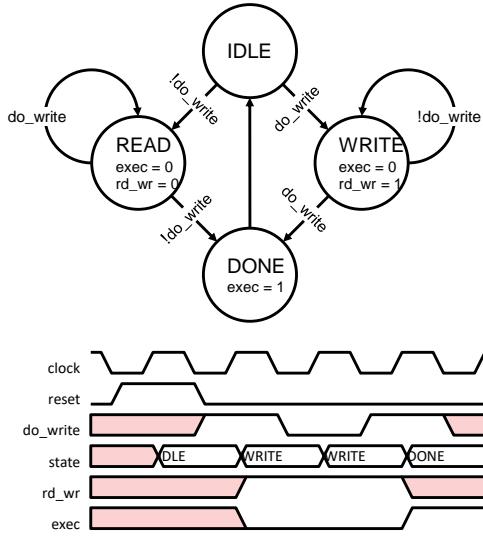
While initially used to work around switch-bounce, as the new vector value can be used immediately without further waiting for the value to “settle”, gray-encoding is indispensable for address counters that increment in one clock domain and are read in another, for example in asynchronous FIFOs.

Gray-encoding has more recently become valuable in the quest for reduced switching activity. The simple counting sequence displayed here averages 1.5 transitions per clock for binary encoding and 1 transition per clock for Gray encoding.

Paths through the example machine have a length of only three states, so present little opportunity to reduce switching activity.

Optimizing Performance: One-Hot Encoding

Reducing state encoding/decoding shortens timing paths to increase performance.



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

```

localparam IDLE  = 4'd1, READ  = 4'd2,
          WRITE = 4'd4, DONE  = 4'd8;
reg [3:0] state, nstate;
reg exec, rd_wr;

always @*
  case ( state )
    IDLE: nstate= do_write? WRITE: READ;
    READ: nstate= !do_write? DONE : READ;
    WRITE: nstate= do_write? DONE : WRITE;
    DONE: nstate= IDLE;
  endcase

always @(posedge clock)
  if ( reset )
    state <= IDLE;
  else
    state <= nstate;

always @*
begin
  rd_wr = state == WRITE;
  exec  = state == DONE;
end

```

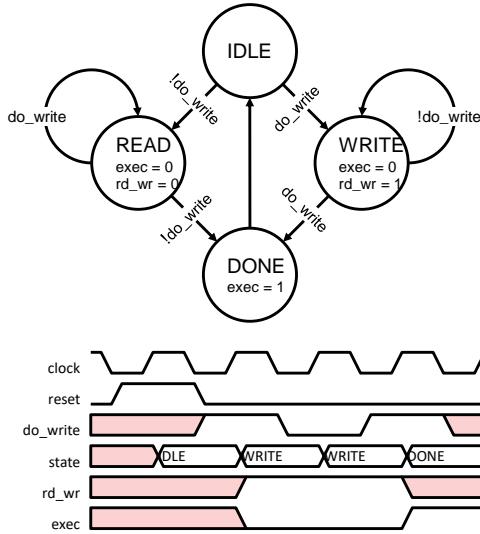
One-Hot is effective for
any number of arbitrary
transition patterns.

cadence®

One-hot state encoding provides a unique state bit for each state. This reduces the combinational logic required to encode and decode the state, thus shortens the timing path to permit increased clock speed.

One-Hot Encoding by Indexing State Bits

Indexing state bits more clearly reveals the reduced encoding.



```

localparam IDLE = 0, READ = 1,
          WRITE = 2, DONE = 3;
reg [3:0] state, nstate;
reg exec, rd_wr;

always @*
begin
  nstate[READ] = state[IDLE] && !do_write
    || state[READ] && do_write;
  nstate[WRITE]=state[IDLE] && do_write
    || state[WRITE] && !do_write;
  nstate[DONE]=state[READ] && !do_write
    || state[WRITE] && do_write;
  nstate[IDLE] = state[DONE];
end

always @ (posedge clock)
begin
  state[IDLE] <= reset || nstate[IDLE];
  state[READ] <= !reset && nstate[READ];
  state[WRITE]<= !reset && nstate[WRITE];
  state[DONE] <= !reset && nstate[DONE];
end

always @*
begin
  rd_wr = state[WRITE];
  exec = state[DONE];
end

```

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

cadence®

You can code the one-hot machine in a style that individually references each state bit. This style may more clearly reveal the reduced state encoding. Here, the longest path to any state bit is a sum of two products of three terms each.

Module Summary

You should now be able to code state machines for synthesis.

This module discussed:

- What is a FSM
 - State encoding, storage, and decoding
- How to define FSM states
 - By use of local constants
- Various ways to code FSMs for synthesis
 - 1-block, 2-block, 3-block
- Various ways to optimize FSMs
 - For area, power, noise, performance



You should now be able to code state machines for synthesis.

This module discussed what an FSM is, how to define FSM states, various ways to code FSMs for synthesis, and various ways to optimize FSMs.

Module Review

1. What is the difference between a Moore machine and a Mealy machine?
2. Why should you define state values using parameters instead of macros?
3. In how many blocks should you define the FSM to generate the best quality netlist?



This page does not contain notes.

Module Review Solutions

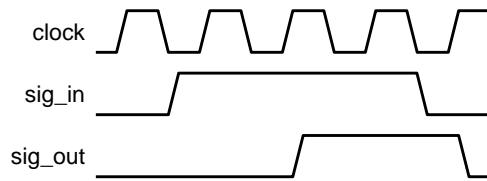
1. What is the difference between a Moore machine and a Mealy machine?
 - The outputs of a Moore machine are a function purely of the machine state. The outputs of a Mealy machine are a function partially or at least one machine input.
2. Why should you define state values using parameters instead of macros?
 - A synthesis tool is more likely to recognize a state machine if you define state values using parameters instead of macros. Recognizing the state machine permits optimizations such as Gray or one-hot encoding.
3. In how many blocks should you define the FSM to generate the best quality netlist?
 - The netlist quality does not generally depend upon the number of blocks. Choose a number that clearly communicates your design intention.



This page does not contain notes.

Module Exercise

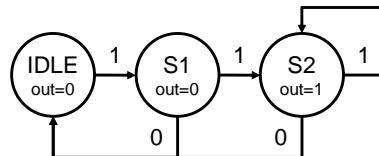
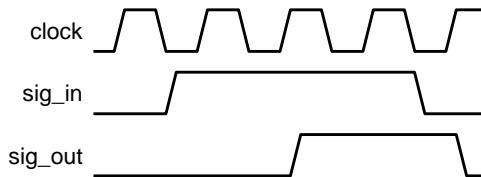
Architect and code a FSM that outputs a “one” for one clock whenever it recognizes two adjacent ones in the input stream.



This page does not contain notes.

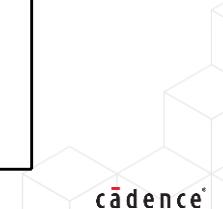
Module Exercise Solutions

Architect and code a FSM that outputs a “one” for one clock whenever it recognizes two adjacent ones in the input stream.



```
module fsm
(
    input wire clock, reset, sig_in,
    output wire sig_out
);
reg [1:0] state;
localparam IDLE=0,S1=1,S2=2;
always @ (posedge clock)
    if (reset)
        state <= IDLE;
    else
        case (state)
            IDLE: state <= sig_in ? S1
                    : IDLE;
            S1,S2: state <= sig_in ? S2
                    : IDLE;
        endcase
    assign sig_out = (state==S2);
endmodule
```

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



This page does not contain notes.



Lab

Lab 14-1 Coding State Machines in Multiple Styles

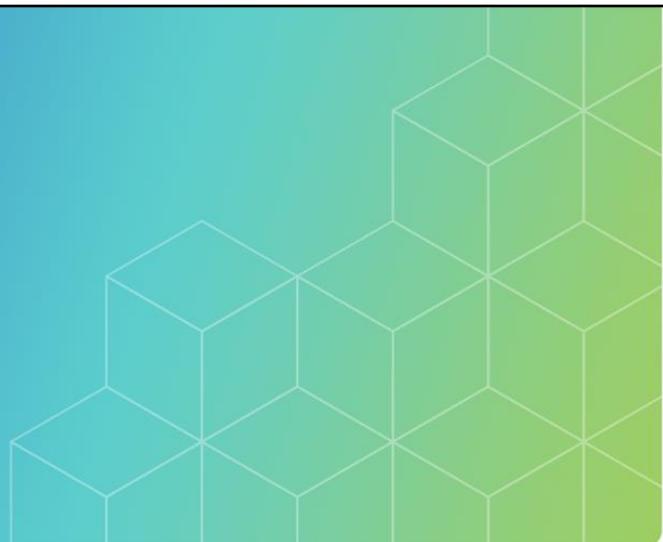
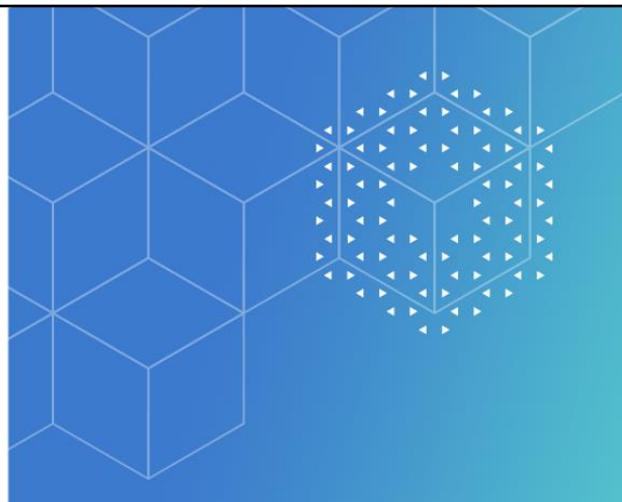
- Code state machines in various styles for synthesis.
- For this lab, you code the serial-to-parallel interface receiver as an FSM in different styles.

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



Your objective for this lab is to code state machines for synthesis.

For this lab, you practice recoding a provided FSM to the styles that this module presented.



Module 15

Avoiding Simulation Mismatches

cadence®

This module addresses the common and perplexing problem of the post-synthesis simulation results that do not match the pre-synthesis simulation results.

Module Objective

In this module, you:

- Avoid the common causes of mismatches between RTL simulation and post-synthesis netlist simulation

Topics

- Non-deterministic behavior
- Synthesis attributes
- Conditional compilation
- Incomplete sensitivity list
- Temporary variables
- Asynchronous set and reset
- RTL vs. Synthesis Models
- Incomplete Assignments
- Comparing unknown values
- Case statements: **casex** and **casez**
- Variable declaration assignment
- Delay controls

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

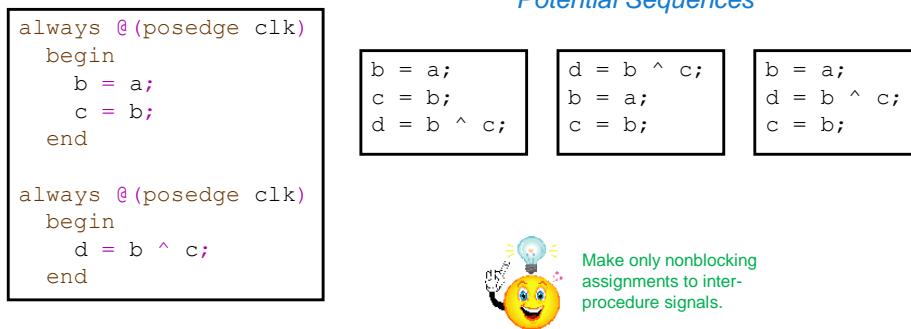


This page does not contain notes.

Avoiding Indeterminate Behavior

You have already seen that the Verilog standard permits the simulator to execute triggered procedures in any order.

The Verilog standard also permits the simulator to execute contiguous statements as multiple events, thus potentially interleaving statement execution from multiple blocks!



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



You have already seen that the Verilog standard permits the simulator to execute triggered procedures in any order.

The Verilog standard also permits the simulator to execute contiguous statements as multiple events, thus potentially interleaving statement execution from multiple blocks!

You can greatly reduce indeterminacy by making only nonblocking assignments to storage variables that communicate between procedures.

“At any time while evaluating a behavioral statement, the simulator may suspend execution and place the partially completed event as a pending active event on the event queue. The effect of this is to allow the interleaving of process execution. Note that the order of interleaved execution is nondeterministic and not under control of the user.” – IEEE Std. 1364-2001 Section 5.4.2 Nondeterminism

Applying Synthesis Attributes Carefully

Use only with great care those synthesis attributes that redefine functionality!

The **full_case** attribute is equivalent to a default case item assigning “don’t care” to all variables written in the case statement.

The **parallel_case** attribute removes the priority semantics from the case item matching.

Never Needed

```
module select (sel, a, b);
  input [1:0] sel;
  output a, b;
  reg a, b;
  always @*
    (* synthesis, full_case *)
    case ( sel )
      2'b00: begin a=1'b0; b=1'b0; end
      2'b01: begin a=1'b0; b=1'b1; end
      2'b10: begin a=1'b1;           end
    //default begin a=1'bx; b=1'bx; end
  endcase
endmodule
```

Equivalent to pragma



ALWAYS use a default and NEVER directly assign X values!

Seldom Needed

```
always @*
begin
  set = 0;
  rst = 0;
  (* synthesis, parallel_case *)
  casez (sel[1:0])
    2'b?1: set = 1;
    2'b1?: rst = 1;
  endcase
end
```

For sel==3 simulation does this
For sel==3 synthesis does both

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



The **full_case** attribute is equivalent to a default case item assigning “don’t care” to all variables written in the case statement. Simulation, of course, knows nothing about any synthesis attributes. For this example, when the select value is 3 or unknown, the output values are unknown for simulation and some undetermined binary value for synthesis.

The **parallel_case** attribute directs synthesis to not build a priority structure for testing case match items. Simulation, of course, knows nothing about any synthesis attributes. For this example, synthesis builds hardware assuming that if the select value is 3 then you really want both the set and the reset to occur simultaneously, while simulation, as usual, executes only the first matching statement.

Applying Conditional Compilation Carefully

Carefully consider what code you “hide” from simulation or synthesis!

```
`ifdef SYNTHESIS  
  // tons of code  
  a = 0;  
`else  
  // tons of code  
  a = 1; - difference  
`endif
```

Usage less
problematic

```
`ifndef SYNTHESIS  
  // testbench code  
  // diagnostic code  
`endif
```



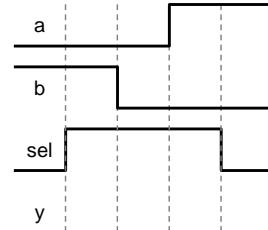
Carefully consider what code you hide from simulation or synthesis. You can reasonably safely hide from synthesis testbench-related code that you embed in an RTL module. Much less valid reason exists to hide design code from simulation. Complex, frequent and large conditionally compiled regions especially promote inadvertent coding errors that RTL simulation does not encounter.

Completing the Combinational Logic Sensitivity List

For simulation, include in the event list all signals that are input to the logic. For your convenience, you can use the “*” wildcard.

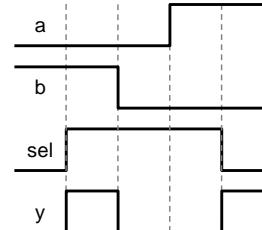
// Incomplete list

```
always @ (a or b)
begin
    y = a;
    if (sel)
        y = b;
end
```



// Complete list

```
always @*
begin
    y = a;
    if (sel)
        y = b;
end
```



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



The event list does not affect the synthesis result, but to avoid incorrect RTL simulation results, you should include in the event list all inputs to the procedure. The easiest way to ensure this is to use the Verilog 2001 wildcard event control (@*).

This example illustrates the effect of an incomplete sensitivity list. If the event list omits the sel signal, the procedure executes upon transitions of only the a and b inputs – transitions of the sel signal have no affect.

Debugging problems caused by an incomplete sensitivity list is difficult, so you might want to develop the habit of simply always using the wildcard event control for all combinational procedures.

“The event list does not affect the synthesized netlist.” – IEEE Std. 1364.1-2002 5.1 Modeling combinational logic

Ensuring Temporary Variables Are Truly Temporary

- Temporary variables are those written and then read in the same procedure and nowhere else.
- Here, “temp” is not a temporary variable so must appear in the event list.
- The event list for a combinational procedure should contain all inputs to the logic.
- Do not include temporary variables – those written and then read in the same procedure and nowhere else.

```
// Combinational logic

always @(a or b or c)
begin : comb_blk
    reg temp;
    temp = a + b;
    q = temp + c;
end
```

```
// Combinational logic

reg temp;
always @(a or b or c or temp)
begin
    q = temp + c;
    temp = a + b;
end
```



The synthesis standard states that the sensitivity list shall not affect the generation of combinational logic. That means that if the synthesis tool recognizes the block as combinational logic, it will proceed as if you had included all inputs to the logic in the sensitivity list. The synthesis tool may or may not warn you about missing inputs. The generated gates will simulate correctly, but very likely differently than the incorrect RTL simulation.

Modifying RTL Model to Match Synthesis Model

This RTL model embeds testbench constructs to correct the simulation.

```
always @(posedge clk
      or posedge set
      or posedge rst)
  if (set)
    q <= 1;
  else
    if (rst)
      q <= 0;
    else
      q <= d;
```

Best to simply
not do this!



This RTL model stays set if “set” and “rst” are both applied and then only “set” is removed.

```
always @(posedge clk
      or posedge set
      or posedge rst)
  if (set)
    q <= 1;
  else
    if (rst)
      q <= 0;
    else
      q <= d;

`ifndef SYNTHESIS
always @(set or rst)
  if (set)
    assign q = 1;
  else
    if (rst)
      assign q = 0;
    else
      deassign q;
`endif
```

Procedural
Continuous
Assignment

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



You are unlikely to model both asynchronous set and asynchronous reset, and even less likely to use them in a situation where they are both simultaneously active. If you do, though, the RTL synthesis template will not accurately model the behavior, as it does not respond to the inactive edge of the set or reset inputs.

You can work around this with embedded testbench code not meant for synthesis. You can use the procedural continuous assignment to override a normal procedural assignment. You apply a procedural continuous assignment only to a variable, unlike the normal continuous assignment you apply only to a net. While the procedural continuous assignment to a variable is in effect, the simulator ignores any normal procedural assignment to the variable, so be sure to deassign the variable when you want it to again behave normally.

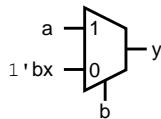
Do Not Confirm Expected Unknown Values

Synthesis of a “x” assignment is a “don’t care” condition that is “optimized away”.

Simulation of an “x” assignment makes the variable value unknown.

```
always @(a or b)
  if (b)
    y = a;
  else
    y = 'bx;
```

```
always @(a or b)
  if (b)
    y = a;
  else
    y = 'bx;
```



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



In the real world, the unknown state does not exist. Digital design nodes are almost invariably in a binary state. Synthesis accepts assignment of the x character to indicate a don’t care value.

Assigning the unknown value can help debug your RTL design. You can, for example, initially assign an unknown value to a variable that should always get assigned some other value before it is tested. Just don’t expect the unknown value to show up in simulation of the post-synthesis netlist.

Using `casez` and Not `casex`

- Simulation of the `casez` statement treats Z and ? as *don't care* bit positions.
 - In either the case expression or the case item expression.
- Unknown sel at start of RTL simulation matches no item.
- Simulation of the `casex` statement treats Z, X and ? as *don't care* bit positions.
 - In either the case expression or the case item expression.
- Unknown sel at start of RTL simulation matches first item.

```
always @*
begin
  set = 0;
  rst = 0;
  casex (sel[1:0])
    2'b?1: set = 1;
    2'b1?: rst = 1;
  endcase
end
```

```
always @*
begin
  set = 0;
  rst = 0;
  casez (sel[1:0])
    2'b?1: set = 1;
    2'b1?: rst = 1;
  endcase
end
```



In the real world, the unknown state does not exist and the high-impedance state exists only extremely rarely. Digital design nodes are almost invariably in a binary state.

Synthesis accepts:

- Assignment of the x character to indicate a don't care value;
- Assignment of the z character to infer three-state logic;
- The characters “x”, “z”, and “?” in the casex item expression to designate bit positions to not participate in the match, that is, “don't care” bit positions;
- The characters z, and ? in the casez item expression to designate bit positions to not participate in the match, that is, don't care bit positions;

RTL simulation accepts don't care bit designations also in the case expression. At the start of RTL simulation, much of the design will be in the unknown state and very likely little or none of it in the high-impedance state. The casex statement treats an unknown sel signal as don't care for all bit positions, thus matching the first case item, whatever that might happen to be. The casez statement does a definitive match for the unknown state, which is unlikely to appear in any of the case items, thus leaving the block outputs with their default values.

Avoiding Variable Declaration Assignment

Synthesis ignores variable declaration assignments.

- Hardware generally does not magically power-up into a known state.
- You must provide an explicit initialization path.

```
reg [3:0] count=4'd0; X

always @ (posedge clk)
  if (count == 9)
    count <= 4'd0;
  else
    count <= count + 4'd1;
```

```
reg [3:0] count; ✓

always @ (posedge clk)
  if (rst)
    count <= 4'd0;
  else
    if (count == 9)
      count <= 4'd0;
    else
      count <= count + 4'd1;
```



Synthesis ignores variable declaration assignments. With the exception of some PALs, manufacturers generally do not guarantee that their components power up in any particular state, so variable declaration assignments are rarely truly a hardware construct. As a general statement, you must provide an explicit initialization path for any RTL that you mean to synthesize.

The variable declaration assignment construct is new to the Verilog-2001 update. You may find it useful in your testbench but must not use it in RTL meant for synthesis.

Remembering Delay Controls Are Not Synthesized

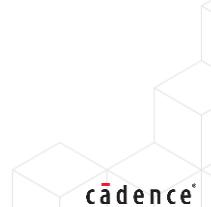
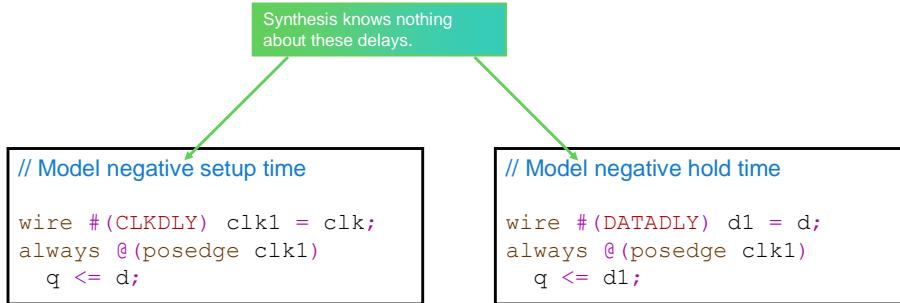
Synthesis *rejects* delay controls before the procedure event control.

Synthesis *ignores* delay controls after the procedure event control.

Synthesis *ignores* net delay and continuous assignment delay.

Time delay is not how you model timing for RTL synthesis.

- You specify arrival and departure times in a separate constraints file.



You can add timing delays to your RTL design to help you visualize the approximate delays of the post-synthesis netlist, but do not for even a moment imagine that this is how you specify the design timing for synthesis. Synthesis totally ignores RTL timing delays that do not violate the synthesis templates. You provide design timing information to synthesis in a separate file as constraints in the vendor's syntax. While simulating the post-synthesis netlist, you may find that some "guesstimates" of the design timing are not as close as you would have liked.

Module Review

1. State and defend your opinion that the **full_case** attribute has an overall positive or negative impact on designer productivity.
2. State and explain the consequence of making an incomplete assignment in a subroutine.
3. State and defend your selection between the **casex** and **casez** statements for designs meant for synthesis.



This page does not contain notes.

Module Review Solutions

1. State and defend your opinion that the `full_case` attribute has an overall positive or negative impact on designer productivity.
 - The `full_case` attribute is equivalent to a `default` case match item that assigns “don’t-care” values to all case statement outputs. However, it less clearly states the designer’s intentions and does *not* prevent latch inference.
2. State and explain the consequence of making an incomplete assignment in a subroutine.
 - Synthesis tools infer a latch if you specify no output value for at least one combination of input values. This latch inference may be inadvertent but it is at least functionally correct. Most synthesis tools do *not* infer such latch if the incomplete assignment is in a subroutine. They instead combinationally generate an undetermined logic output value for the missing input combination.
3. State and defend your selection between the `casedx` and `casez` statements for designs meant for synthesis.
 - Synthesis accepts in a `casedx` match expression the “x”, “z”, and “?” characters, and in a `casez` match expression the “z”, and “?” characters to indicate bit positions that do not participate in the match. Simulation accepts these designations also in the `casedx` and `casez` expression. At the start of RTL simulation, that expression is likely to be of unknown value, thus matching the *first casedx* item and *no casez* item. At the start of post-synthesis simulation, that expression is likely to be of arbitrary value, thus matching *some casedx* or `casez` item.



This page does not contain notes.

Module Exercise

Replace the *parallel_case* pragma with code that matches what synthesis infers.

```
module select (sel, a, b);
  input [1:0] sel;
  output a, b;
  reg a, b;
  always @*
    (* synthesis, full_case *)
    case ( sel )
      2'b00: begin a=1'b0; b=1'b0; end
      2'b01: begin a=1'b0; b=1'b1; end
      2'b10: begin a=1'b1;           end
      //default begin a=1'bx; b=1'bx; end
    endcase
endmodule
```

Replace the *full_case* pragma with functionally correct code that does not infer a latch.

```
always @*
begin
  set = 0;
  rst = 0;
  (* synthesis, parallel_case *)
  casez (sel[1:0])
    2'b1?: set = 1;
    2'b1?: rst = 1;
  endcase
end
```



This page does not contain notes.

Module Exercise Solutions

Replace the *parallel_case* pragma with code that matches what synthesis infers.

```
module select (sel, a, b);
    input [1:0] sel;
    output a, b;
    reg a, b;
    always @*
        begin
            a=1'bx;
            b=1'bx;
        /* (* synthesis, full_case *) */
        case ( sel )
            2'b00: begin a=1'b0; b=1'b0; end
            2'b01: begin a=1'b0; b=1'b1; end
            2'b10: begin a=1'b1; end
            //default begin a=1'bx; b=1'bx; end
        endcase
    end
endmodule
```

Replace the *full_case* pragma with functionally correct code that does not infer a latch.

```
always @*
begin
/*  set = 0;
rst = 0;
(* synthesis, parallel_case *)
casez (sel[1:0])
    2'b?1: set = 1;
    2'b1?: rst = 1;
endcase */
set = sel[0];
rst = sel[1];
end
```



This page does not contain notes.

Labs

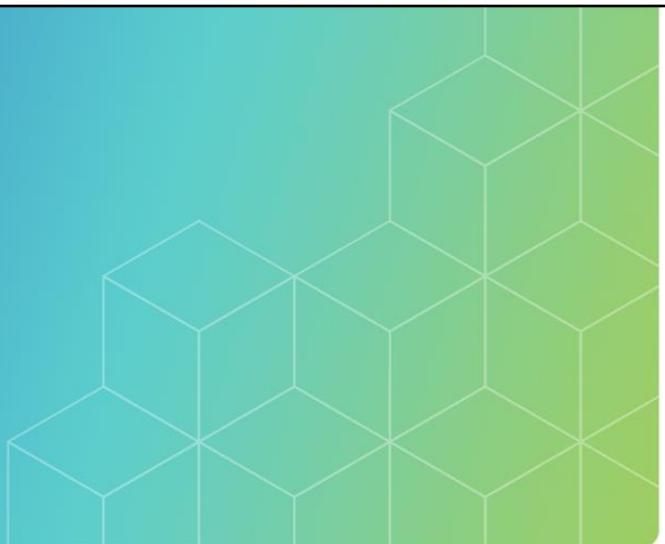
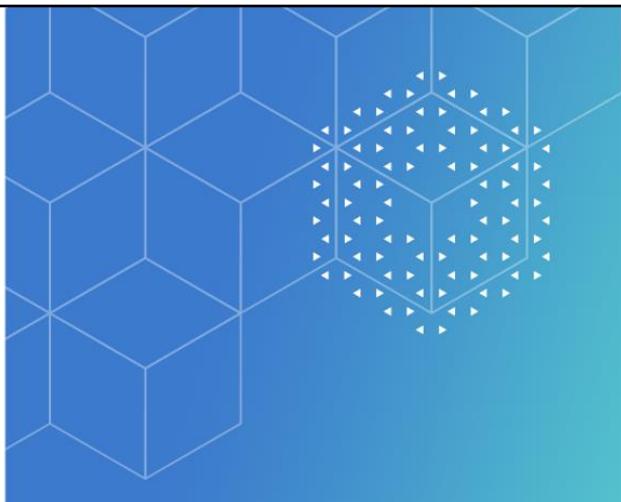


There are no labs in this module.

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



This page does not contain notes.



Module 16

Managing the RTL Coding Process

cadence®

This module suggests some RTL coding “Best Practices” for code meant for synthesis.

Module Objective

In this module, you:

- Manage the RTL coding process

Topics

- Managing the project
- Partitioning for synthesis
- Coding RTL for synthesis



This module examines some rules of thumb for preparing a design to ease the synthesis process. It discusses:

- Project management – Conventions for organizing and naming things.
- Partitioning for synthesis – Setting up the design hierarchy.
- Coding RTL for synthesis – Mostly a summary of rules from previous modules.

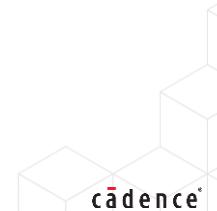
Managing the Project

Adopt a convention for identifiers:

- The name reflects the meaning
 - Without being overly long
- The suffix reflects the category
 - _i(input), _n(active low)
- Agree on capitalization, separation
 - my_name, MyName, myName
- Agree on tense (action, description)
 - grant_bus, bus_grant
- Avoid look-alike characters
 - 0(zero)/O(oh), 1(one)/l(eye)/l(ell)
- Avoid names differing only in case
 - Myname, myName, MyName
 - Especially if they look like keywords

Adopt a convention for project file names and locations.

```
ASIC
|_ Sources          (.v)
|_ |_ Archives
|_ Synthesis
|_ |_ Scripts       (.tcl)
|_ |_ Constraints   (.sdc)
|_ |_ Logfiles      (.log)
|_ |_ Reports       (.rpt)
|_ |_ Netlists      (.v)
|_ |_ Archives
|_ Simulation
|_ |_ Scripts       (.tcl)
|_ |_ Logfiles      (.log)
|_ |_ Archives
|_ OtherTool...
```



To facilitate project management, it is imperative that you organize your project data. This illustration is a starting point for your team discussions. Your ultimate project organization will very likely be different than this illustration and probably also more complex.

To facilitate communication among your team members, it is imperative that you adopt a convention for file names. This illustration suggests file extensions that are commonly used. It is also suggested that each file describe a single module having the same name as the file.

As your project evolves, it is imperative to maintain project archives and records. This allows you to back out of changes that break the project and allows management to utilize project history to estimate future developments. For these purposes, several popular version control mechanisms are available to control project file creation, maintenance, replacement, and deletion.

It is equally important to adopt a convention for identifiers:

- The name should reflect the purpose of the net, variable, constant, module or subroutine, yet without being overly long.
- Suffixes should reflect the name category, for example _i for inputs, _o for outputs, and _n for active-low signals.
- Capitalization and word separation should be applied consistently.
- How you use names should also be consistent, for example, which do you mean the name to reflect – an action that the signal does, or a description of the signal?
- Avoid look-alike characters, for example in some fonts the uppercase I (eye) looks identical to the lowercase l (ell) and the number 1 (one).
- Avoid names that differ only in case, especially names that would be keywords if in lower case. As output netlists are almost invariably Verilog, VHDL users should also avoid names that are Verilog keywords.

Partitioning for Synthesis

This section examines partitioning the design for synthesis:

- Registering block outputs
- What to keep together
- What to keep apart
- Partitioning for design reuse



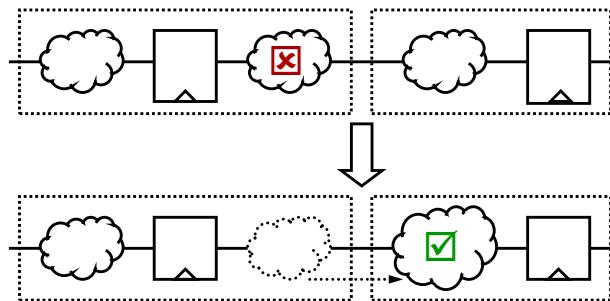
This section separately examines partitioning the design for synthesis. It discusses:

- Registering block outputs;
- What design parts to keep together;
- What design parts to keep apart; and
- Partitioning the design to facilitate its reuse.

Registering Block Outputs

Constraints are simple and identical for each module:

- Input drive strength is the drive strength of the preceding flip-flop.
- Input arrival time is the path delay through the preceding flip-flop.



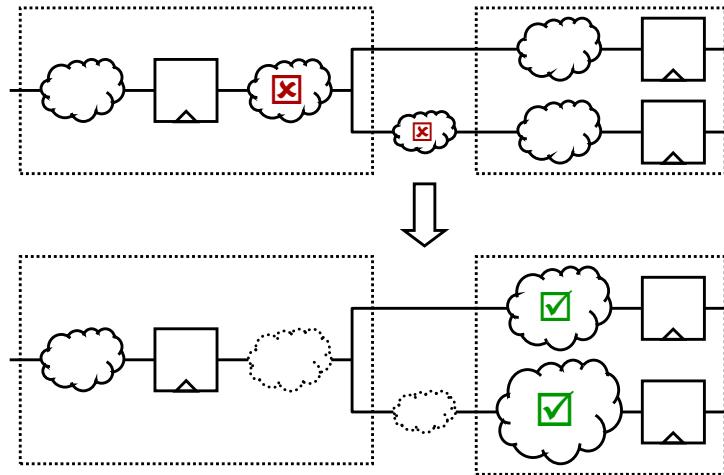
You will find the synthesis process is easier to manage if you have registers on the outputs of each of the hierarchical blocks in your design. This is because it is easier to set the constraints on each block, as the arrival time of signals at each input and output are well defined.

- With unregistered outputs, the designer must specify what proportion of the clock period is available to implement each combinational block. This is done by specifying input and output delays for each block. These delays must be realistic and reflect the comparative performance of the logic blocks.
- With registered outputs, the synthesis tool has almost a complete clock period to implement the combinational logic at the input of the downstream block.

Keeping Combinational Logic Together

Keep combinational logic in the block of its target register.

- This is especially true of “glue” logic.



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



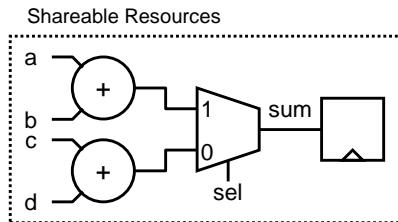
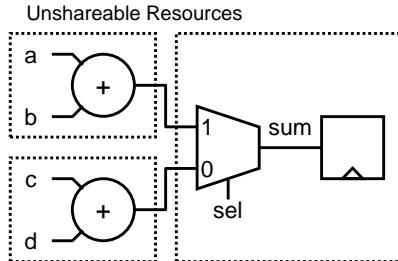
Synthesis tools do not generally, except perhaps for inverters, move logic across hierarchical boundaries. They thus cannot fully optimize combinational logic distributed among multiple blocks. This suggests two strategies you can utilize to facilitate optimization:

- 1st – As much as possible, group the combinational logic cone with the register that it feeds. This simplifies the timing constraints and permits the tool to fully optimize the combinational logic. Where combinational logic feeds registers in multiple blocks, consider merging portions of those blocks or duplicating the combinational logic.
- 2nd – Minimize or even eliminate the combinational “glue” logic at the parent level between instantiated blocks, as the tool can do little to optimize very small amounts of combinational logic.

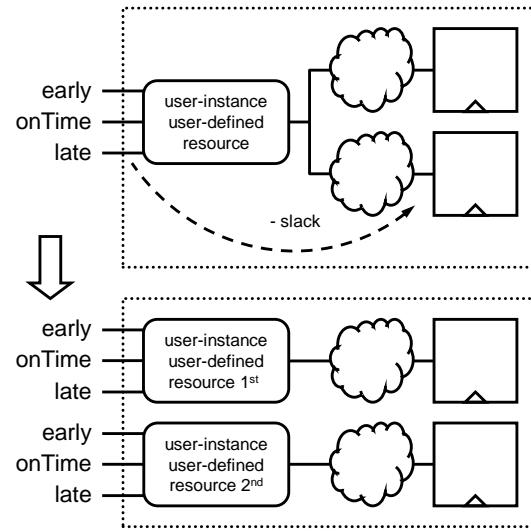
You should as much as practical move minor connective (glue) logic into one or the other instances that it connects.

Especially Keep Resources Together

Keep resources with the register that they feed.



Keep sharable resources together.



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

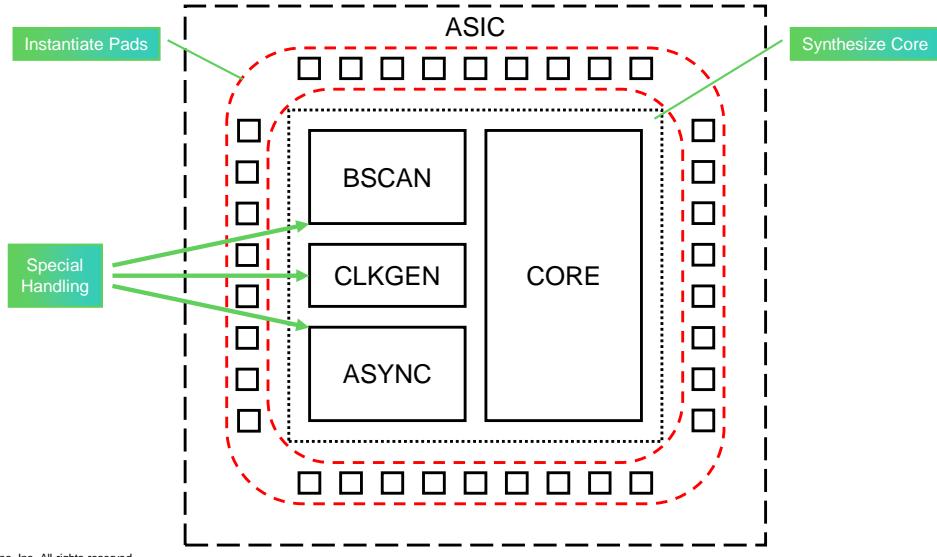


To share resources, the synthesis tool must perform a lifetime analysis of the operation results. The tool will likely be able to do this only if the operators are in the same procedural block. Keep the operators close together to facilitate this analysis.

Keeping the resource with the register that it feeds is an expansion of keeping combinational logic with the register that it feeds. To meet timing requirements, the tool can duplicate its own resources even if you manually instantiate them, like any combinational logic, as long as the resources are located in the scope of the registers that they feed. The tool cannot duplicate user-defined resources, but you will find it easier to duplicate an instance yourself if it is co-located with its register.

Separate Auxiliary Logic from Core Logic

Separate auxiliary logic (pad ring, clock generator, boundary scan, etc.) from core logic logically (and later) physically.



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



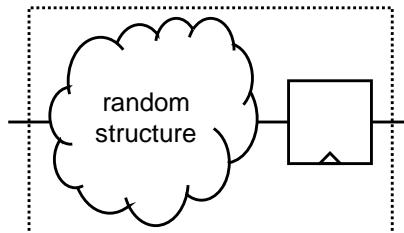
Separate auxiliary logic such as the pad ring, clock generator, and boundary scan from core logic both logically and later physically.

A middle level of hierarchy is recommended to clearly separate the instantiated pad ring from synthesized elements. If you utilize a clock backbone instead of a clock tree, you may want to place the backbone outside of the middle hierarchy of synthesized elements.

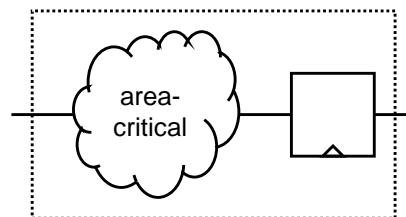
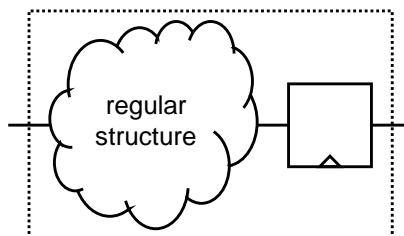
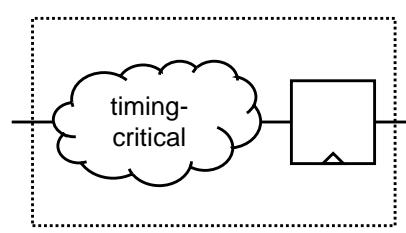
Within the middle hierarchy, you clearly separate the elements that require special handling from the bulk of sequential core logic for which synthesis is relatively straightforward.

Separate Blocks Needing Different Synthesis Techniques

Separate area-critical blocks from timing-critical blocks.



Separate regularly-structured blocks from randomly-structured blocks.



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



Place into separate modules those parts of your design that need special handling:

- The synthesis tool can perform 2-level minimization on random combinational logic that it can easily represent as a sum-of-products (SOP) or product-of-sums (POS). This technique may not be used by default. You may need to encourage the tool to use this technique, which you can most easily apply on a per-module basis. The tool likely cannot apply this technique to complex regular structures, especially those containing networks of exclusive-or operations.
- The synthesis tool will almost invariably favor timing over other cost factors as it optimizes the design. For a subdesign that you know has noncritical timing requirements, you can request that the tool favor some other cost factor, such as area, which you can most easily apply on a per-module basis.

Partitioning for Design Reuse

Keep reusability in mind throughout the design cycle:

- Adopt industry-standard interfaces and corporate-wide conventions.
- Partition in a manner that makes design pieces reusable.
- Parameterize your designs appropriately.



You will almost invariably during your career make frequent use of existing designs. You will probably at least occasionally think unhappy thoughts about the people who made some of those not-easily-reused designs. Keep reusability in mind throughout the design cycle – the next user could very easily be yourself:

- Adopt industry-standard interfaces and corporate-wide conventions;
- Partition in a manner that makes design pieces reusable; and
- Parameterize your designs appropriately. Appropriately means exactly as useful and no more. For example, a byte will always be 8 bits and a minute will always be 60 seconds, so those are constants you do not need to parameterize.

Coding RTL for Synthesis

Modules

- Avoid passing constants through ports.
 - Use parameters.
- Avoid port expressions.
 - Use port identifiers with/without a range.

Functions

- Avoid referencing values other than inputs.
- Synthesis generally make functions into combinational logic.
- Synthesis treats *static* functions as *automatic*.

Expressions

- Avoid unneeded parentheses in arithmetic expressions.
 - Prevents carry-save optimization.
- Use parentheses in non-arithmetic expressions to improve readability.

```
module my_mod (
  .i(a),
  .j1(b[1]), .j0(b[0]),
  .k({c,d}),
  y
);
  input      a,c,d;
  input [1:0] b;
  output     y;
  ...

```

y=(a+b) ? (c*d+e*f) : (c*e+d*f);
y=a&b^c&d|e&f^g&h;

y=a+b?c*d+e*f:c*e+d*f;
y=((a&b) ^ (c&d)) | ((e&f) ^ (g&h));

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



Passing constants through ports creates non-optimal code that synthesis must then optimize “away” and it may in some situation prevent that optimization. You should instead either declare the constant locally or pass it as a module parameter that the elaborator resolves.

A port can be just an identifier or it can be a port expression with or without an identifier. The port expression can be an identifier or a bit or part select of an identifier or a concatenation of these. Port expressions provide a way to have different external and internal names for ports and, and oh by the way, greatly complicate your debugging efforts.

The issues involved with functions are concerned mostly with simulation mismatches between the pre-synthesis RTL design and the post-synthesis gate-level design. These issues also apply to tasks, for synthesis tools handle tasks that do not contain event controls as if they were functions.

- The simulation tool most likely evaluates a function (used in a continuous assignment) only when one of its inputs transitions. If the function result is partly determined by a signal that is not an input to the function, synthesis of course includes that signal in the hardware representation of the function, thus potentially creating a mismatch between pre-synthesis simulation and post-synthesis simulation.
- The synthesis tool most likely infers combinational logic from a function assignment statement that in a procedural block would infer a latch due to incomplete assignment. The synthesis tool may choose a value for the missing assignment and might not even report that it is doing so. This can create a mismatch between pre-synthesis simulation and post-synthesis simulation.
- Synthesis expands the function at the point of the call, creating a new set of local variables for each call, thus treating the function definition as automatic. The standard requires that you declare functions automatic for synthesis but most tools do not enforce this requirement.

The issues involved with expressions are with the use of parentheses. You are encouraged to use parentheses to improve the expression readability, but they have the side affect of forcing term grouping that optimizations have to honor, thus significantly reducing optimization.

Module Summary

Adopting these suggestions will help you to manage the RTL coding process.

This module discussed:

- Managing the design project
 - Adopting a convention for project file names and locations
 - Adopt an identifier naming convention
- Partitioning for synthesis
 - What to keep together (what to keep apart)
 - Partitioning for design reuse
- Coding RTL for synthesis

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



This module examined some “rules of thumb” for preparing a design to ease the synthesis process.

It discussed:

- Project management – Conventions for organizing and naming things.
- Partitioning for synthesis – Setting up the design hierarchy.
- Coding RTL for synthesis – Mostly a summary of rules from previous modules.

Module Review

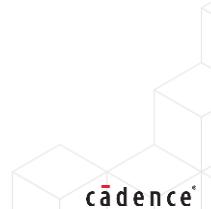
1. What can you do to promote reuse of your designs?
2. Suggest at least one preferred coding practice that you cannot always strictly adhere to.



This page does not contain notes.

Module Review Solutions

1. What can you do to promote reuse of your designs?
 - Adopt industry-standard interfaces and corporate-wide conventions
 - Partition in a manner that makes design pieces easy reused
 - Parameterize your designs appropriately
2. Suggest at least one preferred coding practice that you cannot always strictly adhere to.
 - You should in general partition your design to enable resource sharing, but in some situation, for example to decrease negative slack, you may need to prevent sharing of a specific resource.
 - You should in general parenthesize expressions to improve readability, but parenthesizing arithmetic expressions can prevent carrysave optimizations.



This page does not contain notes.

Labs

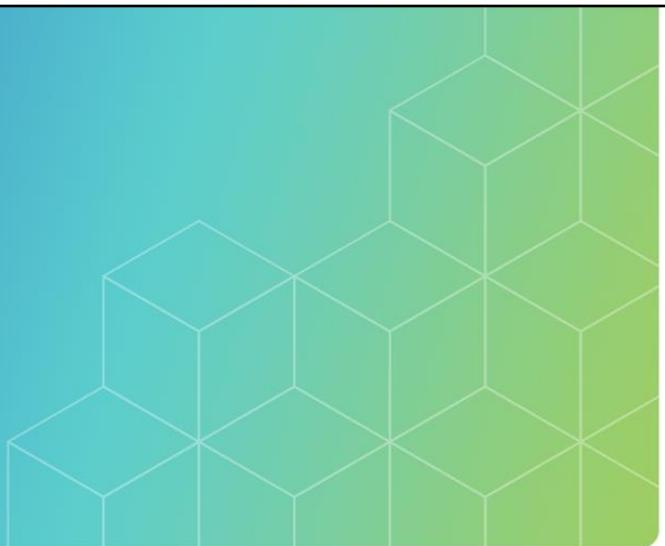
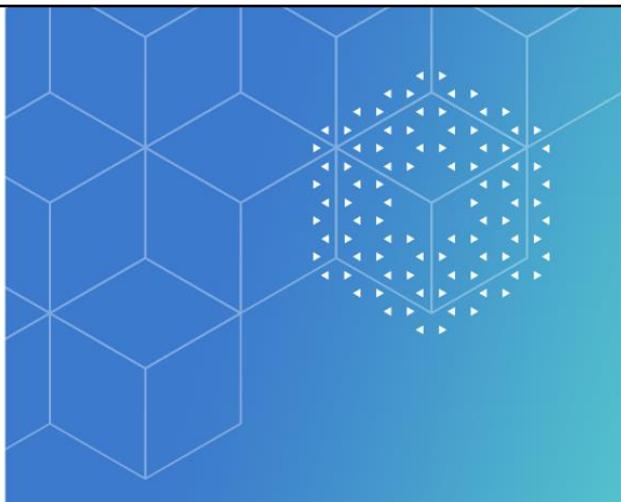


There are no labs in this module.

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



This page does not contain notes.



Module 17

Managing the Logic Synthesis Process

cadence®

This module just briefly presents the basic steps of the synthesis process and the associated Cadence Encounter® RTL Compiler commands.

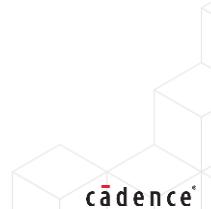
Module Objective

In this module, you:

- Go through the Synthesis Flow and manage the Synthesis Process

Topics

- Reading the HDL source
- Elaborating the design
- Applying constraints
- Mapping to technology cells
- Writing the netlist
- Optimizing Arithmetic Expressions
- Sharing/un-sharing Resources
- Boundary optimization, Retiming, Scan Insertion
- Analyzing Results, Report Timing



This module expands upon the synthesis process that the first module examined. The flow described here is similar to all popular synthesis tools. Any more exact examples derive from the Cadence Encounter RTL Compiler (RC).

Logic Synthesis Goals

Work fast

- Faster than the other guy's synthesis, definitely faster than manual

Work accurately

- Gate-level model functionally equivalent to RTL model
- Cost estimates correspond well with actual values measured later
 - Area, time, power

Maximize Quality of Silicon (QoS)

- Minimize time – maximize frequency and throughput
- Minimize area – cell count and cell size
- Minimize power – switching activity and leakage power



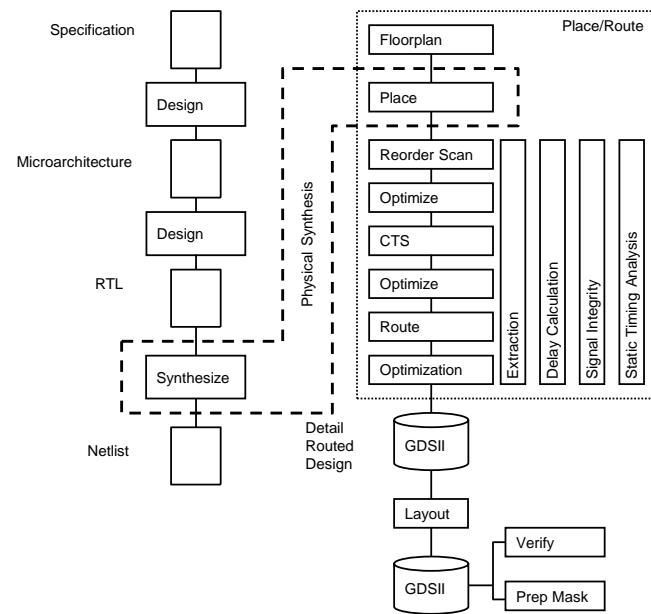
Synthesis tools, like all tools really, have only the two goals to work fast and work well. This slide further splits from the “work well” part the issue of accuracy, for if the tool does not evaluate function and timing well, who cares how wonderful are the false numbers?

- Working fast in a modern highly-competitive industry environment means not just optimizing the design faster than I could do it with pencil and paper but also faster than whatever tool my competitor uses. Here, “faster” means not just tool speed, but also tool usability and tool support that come to play.
- Working accurately, is at least for a modern tool, no longer an issue. Functional accuracy has not been an issue for decades and is now so rare that it is the highest level bug when encountered. Timing accuracy has for long been mainly an issue of how closely the chosen wire-load tables model actual parasitics of the routed design. Modern synthesis tools work in conjunction with layout tools to greatly improve timing accuracy.
- Producing high-quality silicon means minimizing the “cost” factors. Cost factors are foremost timing and only slightly secondarily area and power. All of these cost factors are really just different ways to provide more utility to the customer, i.e., to make the customer want to buy my product instead of the other guy's:
 - Timing is important because higher clock speed means that I can put more functionality into a smaller space and have it respond more quickly to my customer's inputs.
 - Area is important because less area means that I can put more functionality into a smaller space and I can also reduce the price because my own manufacturing and packaging costs are lower.
 - Power is important because less power dissipation means that my customer has lower energy costs, if it is a portable device then the battery can be lighter and stay charged longer, and if it is a handheld device then the customer does not cook their hand while holding it.

Logic Synthesis Basic Process

From RTL input:

- Parse
- Translate
- Optimize
- Map
- Optimize again
- Insert scan
- Optimize again



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



This module expands somewhat upon the synthesis process that the first module examined, for if you understand something about what's going on inside the flow, then you are likely to provide better-quality RTL inputs.

After every major operation the tool again optimizes the design. Optimization means calculating the design timing, area and power, and if not within your targets, swapping gates and restructuring logic in an attempt to meet your targets.

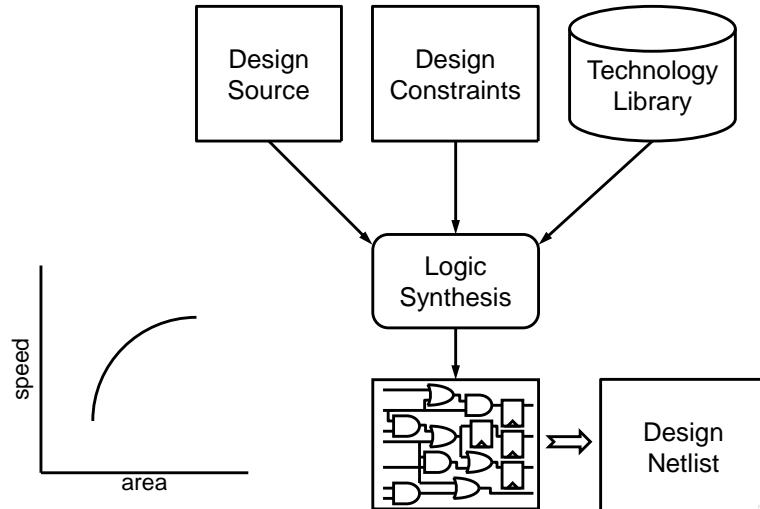
Logic Synthesis Inputs and Outputs

Input

- Synthesizable HDL
- Constraints – (SDC)
- Technology libraries

Output

- Netlist – (usually Verilog)
 - Of technology macros



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

cadence®

Inputs to the synthesis process include:

- The design itself, written in the “synthesizable subset” of the HDL.
- Design constraints, including the required clock speed, input drive strength and arrival time with respect to the clock, and output load and required arrival time with respect to the clock. Design constraints can also include power consumption and can sometimes include noise immunity, testability, and factors affecting ease of place and route. Design constraints may be in the Synopsys Design Constraint (SDC) format, and synthesis vendors also have their own proprietary formats.
- At least one technology library, including all pertinent information about the available cells, and factors for estimating interconnect delays.

Outputs from the synthesis process include:

- Most importantly, the design as a structural netlist of technology cells.
- Optionally, post-synthesis timing information for annotation to a gate-level simulation.
- A log file and any other reports you requested from the tool.

Logic Synthesis Basic Flow

Phase	Description	Example RC Command
Read HDL	Parse source code, check syntax	read_hdl
Elaborate	Build data structures and registers	elaborate
Apply constraints	Specify operating conditions, clocks, and I/O timing	define_clock set_input_delay
Map to generic cells	Map to generic cells. Technology-independent optimizations	synthesize to_generic
Map to technology cells	Map to technology cells. Technology-dependent optimizations	synthesize to_map
Optimize (optional)	Improve performance	retime prepare retime min_delay
Insert scan	Build the scan chain	insert_dft synthesize -to_map connect_scan_chains synthesize incremental
Analyze results	Create and analyze reports	report_area report_gates report_timing
Write netlist		write_hdl

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



The remainder of this module examines a generic flow from reading the HDL to writing the optimized netlist. Example commands are those of the Cadence Encounter RTL Compiler. Most of these commands have several options that this training does not show. This training also does not show the complete command set. For example, applying constraints is quite a bit more involved than simply defining the clocks.

Reading HDL

Read the HDL source and check syntax.

- Options can include the expected language and version.
- The tool generates an internal parse tree representing the HDL source.

Example

```
read_hdl my_design.v
```



The first step is to read the HDL source and check its syntax. Options might include to specify the expected language and language version, and for Verilog, to define text replacement macros. The tool generates an internal parse tree representing the HDL source.

elaborate

Check semantics and construct a design.

- Construct and connect hierarchy
- Infer sequential elements
- Expand function calls
- Propagate constants
 - Precompute results of expressions having only constant terms
- Remove dead code
 - Remove statements that cannot execute or whose results are not used
- Unroll loops
 - Replace each for loop iteration with its own statement

Example

```
elaborate my_top_module
```

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



Elaboration constructs a design.

The elaborator:

- Constructs and connects the hierarchy
- Infers sequential elements
- Expands functions at the point of each call
- Propagates constants
- Removes dead code
- Unroll loops

Example Elaboration Optimizations

Constant Propagation

- Original code

```
a = 0;
b = a + 1;
c = 2 * a;
```
- Optimized code

```
b = 1;
c = 0;
```

Dead Code Removal

- Original code

```
a = x;
b = a + 1;
c = 2 * a;
```
- Optimized code

```
b = x + 1;
c = 2 * x;
```

Loop Unrolling

- Original code

```
for (a=0;a<=2;a=a+1)
z[a] = x[a] + y[2-a];
```
- Optimized code

```
z[2] = x[2] + y[0];
z[1] = x[1] + y[1];
z[0] = x[0] + y[3];
```

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



The elaborator can do some optimization as it constructs the design:

- It can propagate constants to eliminate unneeded operators.
- It can remove code that is not executed or that sets variables that are not used.
- It unrolls loops.

Applying Constraints

Inform the tool of the expected operating environment and of any timing paths it can ignore or are longer than one clock cycle in normal operation.

- Clock signal attributes
 - Period, duty cycle, skew, latency
- Arrival times (w.r.t. clock) at I/O ports
- Environmental attributes
 - Input drive, output load
- Timing exceptions
 - False or multi-cycle timing paths

Example

```
define_clock -name 100MHz -period 10000
```

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


Timing and design constraints describe the “design intent” and the surrounding constraints, including synthesis, clocking, timing, environmental, and operating conditions.

Set these constraints on start points and end points to ensure that every path is properly constrained to obtain an optimal implementation of the RTL design. A path begin point is from either an input port or a register clock pin, while an end point is either an output port or a register data pin.

Use these constraints to:

- Describe different attributes of clock signals, such as the duty cycle, clock skew, and the clock latency.
- Specify input and output delay requirements of all ports relative to a clock transition.
- Apply environmental attributes, such as load and drive strength to the top-level ports.
- Set timing exceptions, such as multicycle paths and false paths.

Mapping to Generic Cells and Optimize

Do technology-independent RTL optimizations:

- Prune unloaded logic
- Optimize arithmetic expressions
- Share common sub-expressions
- Share and duplicate resources
- Carry-save adder transformations
- Merge and map operators to implementations

Example

```
synthesize -to_generic
```

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



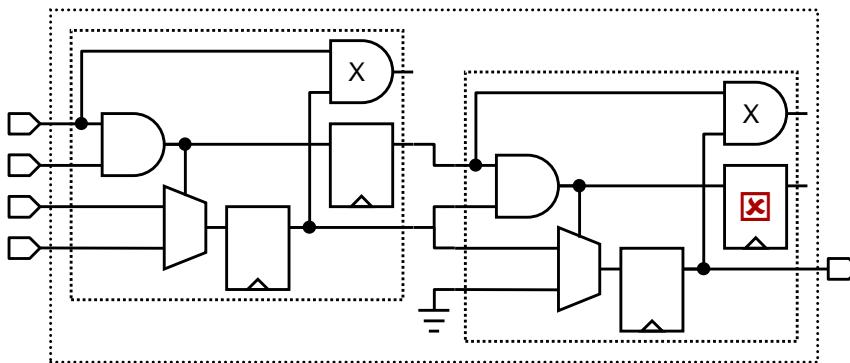
The synthesis tool does technology-independent RTL optimizations as a preliminary step, including:

- Datapath synthesis
- Resource sharing and speculation
- Multiplexor optimization
- Carry-save adder (CSA) optimizations

Pruning Unloaded Logic

Technology-independent optimizations remove unloaded logic.

- Removed cells marked X below do not transitively drive outputs.

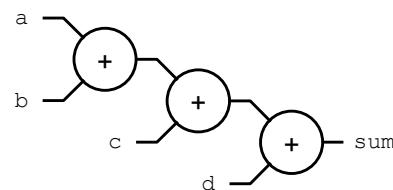


Technology-independent optimizations include the removal of unloaded logic. Here, the cells marked X do not transitively drive outputs so are removed.

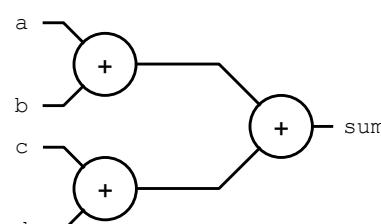
Optimizing Arithmetic Expressions

```
sum = a + b + c + d;
```

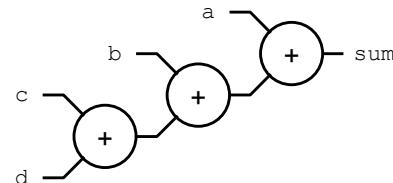
Initial implementation



Optimized for speed if all inputs arrive together



Optimized for speed if input "a" arrives late



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



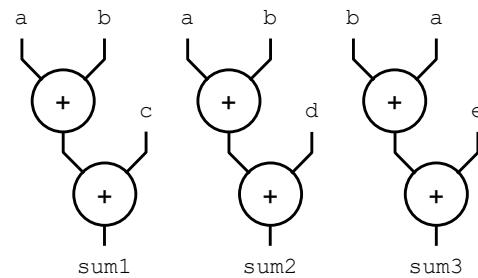
Technology-independent optimizations include the restructuring of arithmetic expressions. The logic tree is restructured to as much as practical balance the arrival times from all inputs. During RTL optimizations the tool estimates path delay based upon expression depth.

Tools typically honor explicit parentheses. Include parentheses to force structure when you know for example that some input will arrive late, and otherwise do not include parentheses.

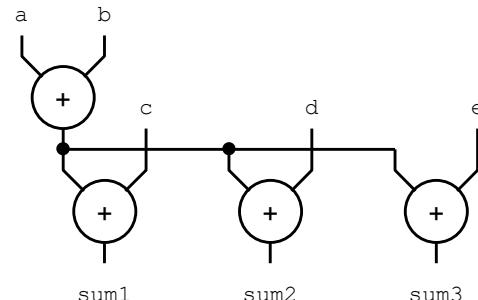
Sharing Common Sub-Expressions

```
sum1 = a + b + c;
sum2 = a + b + d;
sum3 = b + a + e;
```

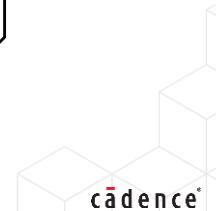
Initial Implementation



Optimized Implementation



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



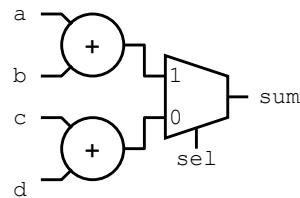
Technology-independent optimizations include sub-expression elimination. Here, the $(a+b)$ sub-expression is common to the three sums, so is shared. The tool may later duplicate sub-expressions to meet timing requirements. The order of the sub-expression operands is immaterial, but for most tools the subexpressions must be in same relative position in its enclosing expression.

Sharing Resources

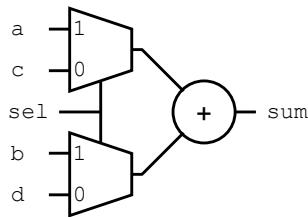
- A resource is a computational element, such as an adder, shifter, or multiplexor.
- Each HDL operator represents a unique resource type.
 - $+$ operator requires an adder.
 - $>$ operator requires a comparator.
- Without sharing, the number of resources is the number of operators.
- With sharing, the number of resources can be reduced.
 - Some operators can share the same resource in alternate clock cycles.
 - Some resources can implement more than one type of operation, e.g., $+$ and $-$.

```
if (sel==1)
  sum = a + b;
else
  sum = c + d;
```

Possible Implementation



Alternate Implementation



A resource is a computational element, such as an adder, shifter, or multiplexor.

Each HDL operator represents a unique resource type, for example the “plus” ($+$) operator represents an adder and the “greater than” ($>$) operator represents a comparator. Some tools can even merge some adjacent operators and map the merged operators to a more optimized resource.

Without sharing, the number of required resources is the number of operators.

With sharing, the number of required resources can be reduced:

- Similar operators that are utilized in non-overlapping clock cycles can be shared.
- Add and subtract operators, for example, can be considered to be similar.

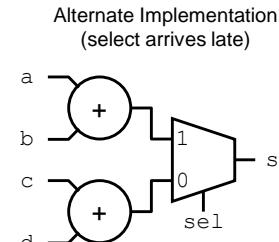
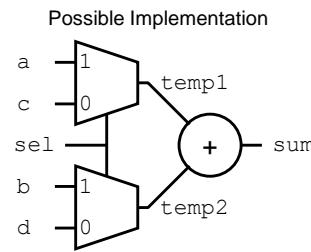
Depending upon the tool, resource sharing may occur by default or you may need to enable it.

Un-Sharing Resources (Speculation)

Can also add resources to meet timing requirements.

```

begin
  if (sel==1) begin
    temp1 = a;
    temp2 = b;
  end
  else begin
    temp1 = c;
    temp2 = d;
  end
  sum = temp1 + temp2;
end
  
```



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

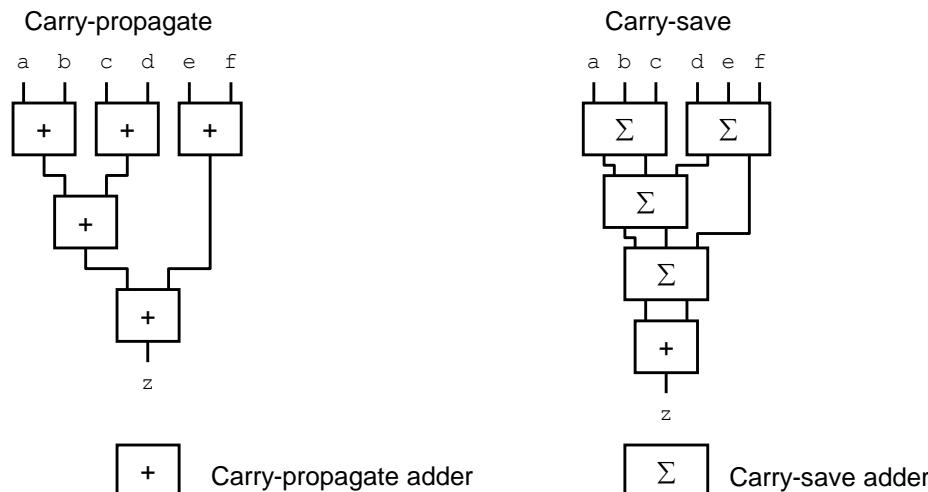


The tool can alternatively duplicate resources where needed to meet timing requirements. Here, because the select signal arrives late, synthesis duplicated the adder.

Carry-Save Adder Transformations

Carry propagation frequently appears on the critical timing path.

Carry-save adder transformations “save” carry generation until the end.



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



The most straightforward way to add a set of numbers is to employ an adder tree. Each adder consumes two numbers and produces one. The adder at the root of the tree generates the final sum.

A carry-save adder takes three numbers and produces two outputs, one formed with the sums and the other with the carryouts. Carry save transformation can greatly improve area and timing.

CSA transformation may apply to any datapath operators whose isolated implementation includes a carry propagate adder. This includes all discrete and relational operators. When the output of one such operator feeds into another datapath operator, it becomes a candidate for CSA transformation. When the output of one mergeable operator feeds into the input of another mergeable operator, it becomes a candidate for CSA transformation.

Datapath operators are merged in scenarios such as the following:

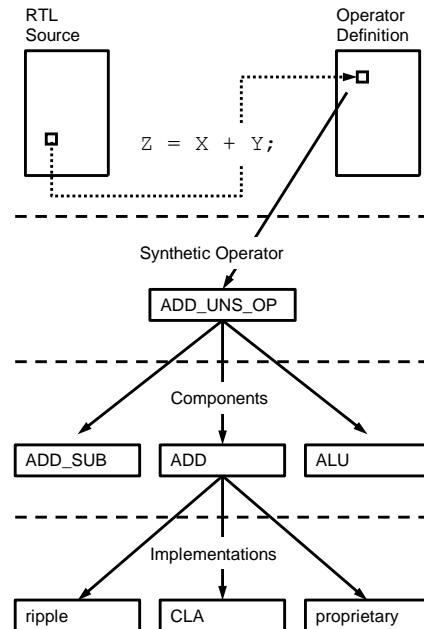
- Any combination of Vector Sum, Sum-of-Product, and Product-of-Sum, including intermediary inverted signals.
 - assign cs = $a + b + c + d$; assign y = $cs * p + q$;
- Comparator
 - assign p = $a + b$; assign q = $c + d$; assign is_greater = ($p > q$);
- Multiple Fanout
 - $cs = a * b$; $x = cs + c$; $y = cs + d$;
- Multiplexers
 - assign tmp = $s ? a*b + c*d : a*c + b*d$; assign z = $tmp + e$;
- Inverter
 - wire [16:0] tmp = $a + b$; assign z = $\sim tmp + c$;
- Truncated CSA
 - assign p = $a + b$; assign q = $c + d$; assign r = $p + q$;

assign y = $r[17:8]$;

Selecting the Architecture: ChipWare

Advanced synthesis tools such as the Cadence Encounter RTL Compiler (RC) have libraries of re-usable designs.

- The Cadence ChipWare (CW) library includes:
 - Common combinational and sequential components.
 - Arithmetic components (adders, subtractors, multipliers).
 - Memory components (flip-flops, FIFOs).
- Logic synthesis automatically maps operators to available CW components.
- Many CW components have multiple architectural implementations allowing logic synthesis to make area/time tradeoffs.



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



A ChipWare library is a set of re-usable functional blocks.

The Cadence Encounter RTL Compiler comes with a large set of pre-defined ChipWare components. You can add your own user-defined ChipWare components, and an IP vendor can package re-usable blocks as user-defined ChipWare components and provide them in a user-defined ChipWare library.

ChipWare involves a four-tier hierarchy:

- 1st – HDL operators (e.g., “+”) and HDL functions.
- 2nd – Synthetic operators, a language-neutral and technology-independent definition of a particular operator or function. For example, **ADD_UNS_OP** is a synthetic operator representing an addition between two unsigned operands. A synthetic operator can be bound to a selection of ChipWare components.
- 3rd – ChipWare components, which can have multiple implementations.
- 4th – The component implementations.

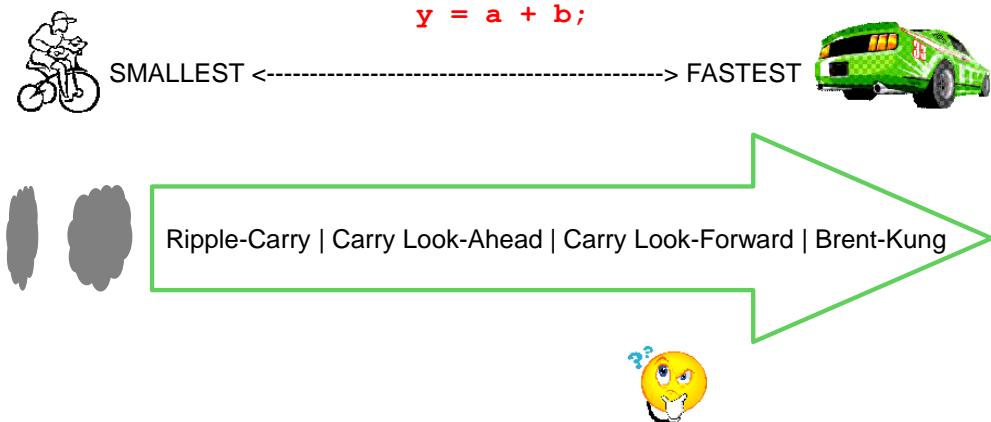
The Cadence RTL Compiler knows about the interface and functionality of these libraries, and for each operator, chooses an implementation based upon QoS calculations.

You can alternatively directly instantiate a ChipWare implementation in your RTL code.

Selecting the Architecture: Tradeoffs

Different architectures have different area and timing characteristics.

Logic synthesis chooses the smallest available architecture meeting timing.



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

Different resource implementations have different area and timing characteristics.

Logic synthesis chooses from the available implementations the smallest that it expects to meet the timing requirements.

Selecting the Architecture: Manual Instantiation

Sometimes you need to explicitly instantiate a specific technology vendor's library macrocell that the tool does not infer:

- I/O pad
- Memory
- Processor (etc.)

Explicit instantiation creates issues:

- Makes code technology dependent
- Can prevent some optimizations
- Can make the code less readable

```
module my_adder
#(W=4)
  (output [W-1:0] s,
   input  [W-1:0] a,b);
  assign s = a + b;
endmodule
```

```
module my_adder
#(W=4)
  (output [W-1:0] s,
   input  [W-1:0] a,b);
  CW_add #(W) u1 (.A(a), ...);
endmodule
```



Create a "wrapper" module around the macro so that "swapping in" some other IP is merely a configuration change.

Sometimes you need to explicitly instantiate a specific technology vendor's library macrocell that the tool does not infer, for example the I/O pads, a memory block, or an embedded processor. This explicit macro instantiation makes the code dependent upon the technology. To somewhat preserve technology independence, you can wrap the technology-specific component in a module for which you provide a technology-independent interface. You instantiate your technology-independent module where needed and configure the design to choose one or the other definition of the module.

Sometimes you want to explicitly instantiate a specific technology vendor's library macrocell because you want to force a specific implementation. This explicit macro instantiation can make the code less readable. To preserve readability you can instead retain the operator and utilize tool-specific pragmas to map specific HDL operators to specific implementations.

Map to Technology Cells and Optimize

Map “generic” logic functions to cells that are actually available in the technology library, including:

- Set target timing goals
- Map to technology cells to meet timing goals
- Iteratively remap cells on critical paths
- Incrementally reoptimize critical regions

Example

```
synthesize -to_map
```

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



Mapping the design to technology cells typically includes concurrent optimizations such as restructuring, splitting, pin swapping, buffering, pattern matching, and isolation.

The tool may then execute multiple remapping phases – some targeted at area optimization and others at timing optimization. The remapping operations are predominantly focused upon resizing the cells.

The tool lastly performs an incremental optimization to improve timing and area and fix design rule (DRC) violations. Timing by default is the highest priority. Incremental optimization can also include iterative resynthesis of fragments of the critical path to improve timing.

Technology-Dependent Optimization

Signal	Formula	Gate	Transistors
d	(a+b)'	nor	4
e	d'	not	2
f	(ab)'	nand	4
g	(ec)'	nand	4
h	f'	not	2
i	g'	not	2
j	(h+i)'	nor	4
k	(h+d)'	nor	4
y	j'	not	2
z	c'k+ck'	xor	8

Implementation of a full adder:

$$\begin{aligned}y &= ab + ac + bc \\z &= ab'c' + a'b'c' + a'b'c + abc\end{aligned}$$

- A design is technology dependent if implemented as interconnected technology library cells.
- Advantage: Library cells are predefined and highly optimized – area and delay are known, minimal, and accurate.
- Technology-dependent optimizations include:
 - Boundary optimizations.
 - Register retiming.

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



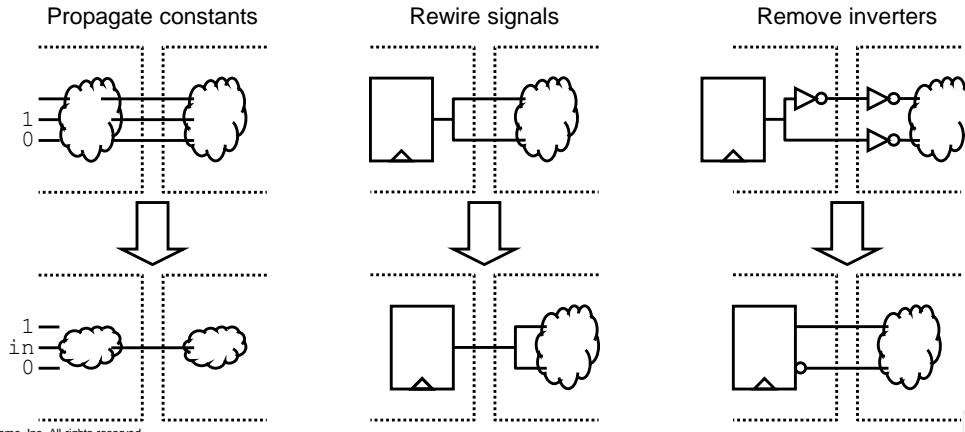
The predefined library cells are highly optimized – area and delay are known, minimal, and accurate.

With and after mapping the design to technology cells, the tool can perform technology-dependent optimizations, including:

- Boundary optimization
- Register retiming

Boundary Optimizations

- Remove gates driving or loading unconnected ports.
- Propagate constants across the hierarchical boundary.
- Rewire equivalent hierarchy-crossing signals.
- Remove inverters duplicated across the hierarchical boundary.



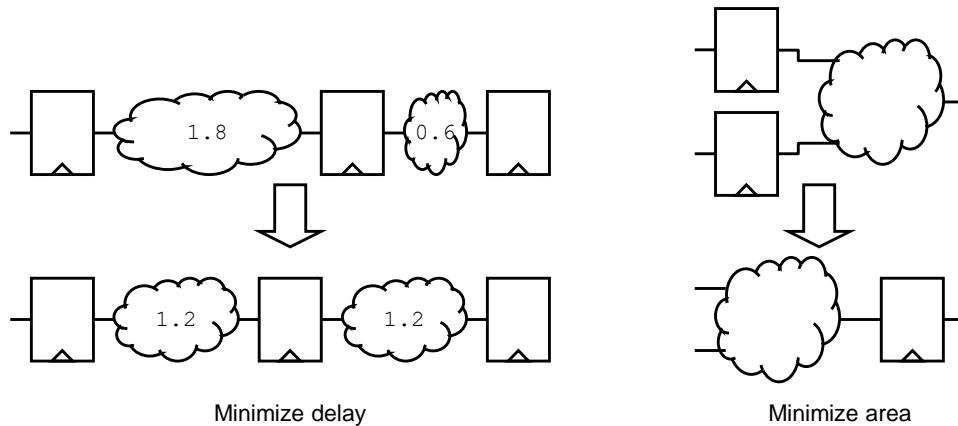
Boundary optimizations include:

- Removing gates driving or loading unconnected ports.
- Propagating constants across the hierarchical boundary.
- Rewiring equivalent hierarchy-crossing signals.
- Removing inverters duplicated across the hierarchical boundary.

Some of these, e.g., constant propagation, the synthesis tool can also do before technology mapping.

Register Retiming

Move registers upstream / downstream to improve results.



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



Retiming is a technique for improving the performance of sequential circuits by repositioning registers to reduce the cycle time or the area without changing the input-to-output latency. This technique is generally used in datapath designs. The retiming operation distributes the sequential elements at the appropriate locations to meet performance requirements. Thus, retiming allows you to improve the performance of the design during synthesis without having to redesign the RTL. Retiming does not change or optimize the existing combinational logic.

Improving the clock period or timing slack is the most common use of retiming. This can be a simple pipelined design, which contains the combinational logic describing the functionality, followed by the number of pipeline registers that satisfy the latency requirement. It can also be a sequential design that is not meeting the required timing. Retiming distributes the registers within the design to provide the minimum cycle time. The number of registers in the design before retiming may not be the same after retiming because some of the registers may have been combined or replicated.

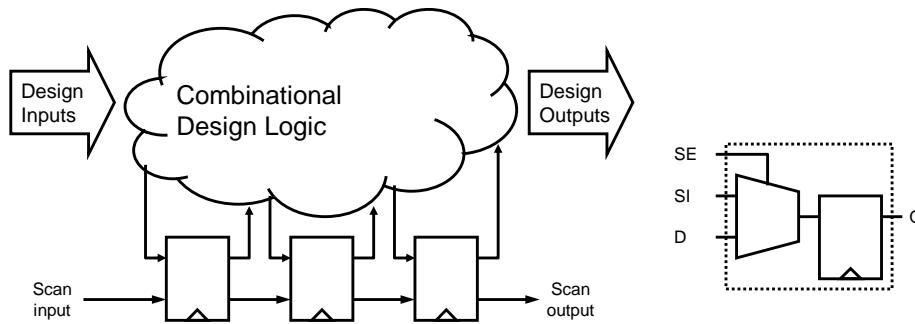
Retiming does not optimize combinational logic and hence the combinational area remains the same. When retiming for area, synthesis moves registers in order to minimize the register count without worsening the critical path in the design.

Retiming is typically not part of the generic flow. You inform the tool which hierarchies or blocks you think would benefit from retiming during synthesis. You can alternatively manually launch a retiming session on a specific block or block region.

If an equivalence checker is part of your flow then you need to inform the equivalence checker about the retimed registers.

Insert Scan

- Internal scan replaces each latch and flip-flop with a scan-equivalent unit.
- Scan-equivalent unit has multiplexor with scan enable and scan input.
- Test mode loads and unloads the scan “chain” serially.
- Makes every latch and flip-flop input and output a virtual I/O pin.
- Reduces test “problem” to just combinational logic.



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



Manufacturers test ASICs using automated test equipment (ATE), which presents stimulus vectors and clocks to the ASIC inputs and compares the ASIC outputs to expected response vectors.

To improve testability, ASIC developers typically insert internal scan. Internal scan replaces every sequential unit with a functionally equivalent unit having a multiplexed scan data input, and “stitches” the units together to form one or more scan “chains”. This greatly facilitates the automatic test-pattern generation (ATPG) of test vectors, as the tool has absolute control over every sequential unit and needs only to develop combinational test vectors.

Logic synthesis tools automatically insert and connect internal scan. They use ASIC vendor scan macros where available and otherwise build the scan from whatever macros are available. As the scan units are somewhat larger and slower, the logic synthesis tool reoptimizes the design after inserting scan.

Analyzing Results

Command	Description
report area	Area of the synthesized and mapped design
report clocks	Clocks of current design
report datapath	Inferred datapath operators
report design_rules	Design rule violations
report gates	Technology cells used, total area, instance count
report hierarchy	Design hierarchy information
report instance	Instances of current design
report memory	Platform memory used
report messages	Error message summary
report power	Leakage power
report qor	Quality of results
report timing	Timing of current design
report summary	Area, timing, and design rule violations

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



Logic synthesis tools report pretty much anything you want to know about the design in a wide selection of formats. This table suggests some of the more obvious things that you might want to report.

Report Timing

The timing report presents arrival time calculations for the critical path.

It reports the path as a series of instance pins with:

- Fanout for each output pin.
- Load and slew for each output pin.
- Incremental delay for each cell.
- Cumulative delay (or arrival time) at each cell.

Pin	Type	Fanout	Load	Slew	Delay	Arrival
		(fF)	(ps)	(ps)	(ps)	(ps)
(clock TST_CLK)	launch				0	R
(DSP.sdc_line_1512)	ext delay				+8000	8000 F
TST_DIN[0]	<<< in port	1	1.8 1000	+0	8000	F
p214748365A883/A					+0	8000

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



A timing report by default presents detailed information concerning the one most critical path. You can optionally request information for a number of the most critical paths, filter paths by start point, through points and end point, or specify the exact path to report. The example report is that provided by the Cadence Encounter RTL Compiler.

Write Netlist

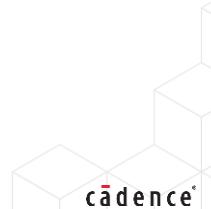
Write a (usually Verilog) netlist, most often for downstream tools.

Options can include:

- Whether to write the full netlist or just the header.
- Whether to write subdesigns or just the current scope.
- Whether to include tool-specific information for downstream tools.

Example

```
write_hdl
```



Write a netlist. The netlist is almost always a Verilog netlist. The netlist is usually meant for the next tool in the flow, for example verification tools and a place and route tool. Options might include:

- Whether to write the full netlist or just the header.
- Whether to write subdesigns or just the current scope.
- Whether to include additional tool-specific information for the downstream tools.

Module Review

1. Suggest some different general ways that the synthesis tool might implement mathematical operators.
2. Suggest reasons why you might want to use mathematical operators rather than directly instantiate a technology vendor's library macrocell.



This page does not contain notes.

Module Review Solutions

1. Suggest some different general ways that the synthesis tool might implement mathematical operators.
 - The synthesis tool may instantiate an appropriate macrocell that implements the operator, or it may implement the operator using discrete gates from the target technology library.
2. Suggest reasons why you might want to use mathematical operators rather than directly instantiate a technology vendor's library macrocell.
 - Directly instantiating macro-cells will make the code less readable, and dependent upon the technology that provides the macro-cell.



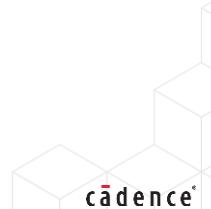
This page does not contain notes.

Module Exercise

Logic synthesis cannot add states to your design. Suppose that you know that the sum is not needed until two cycles later. Re-code this design to use two states and one adder. Assume that the inputs remain stable relative to clocks.

```
always @ (posedge clk)
  if (do_add)
    sum <= a + b + c;
```

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

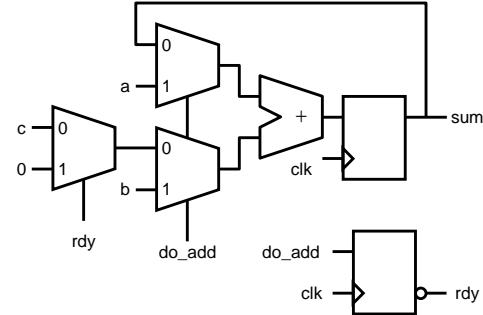


This page does not contain notes.

Module Exercise Solution

Logic synthesis cannot add states to your design. Suppose that you know that the sum is not needed until two cycles later. Re-code this design to use two states and one adder. Assume that the inputs remain stable relative to clocks.

```
always @ (posedge clk)
  if (do_add) begin
    sum <= a + b;
    rdy <= 0;
  end
  else
    if (!rdy) begin
      sum <= sum + c;
      rdy <= 1;
    end
  end
```



This page does not contain notes.



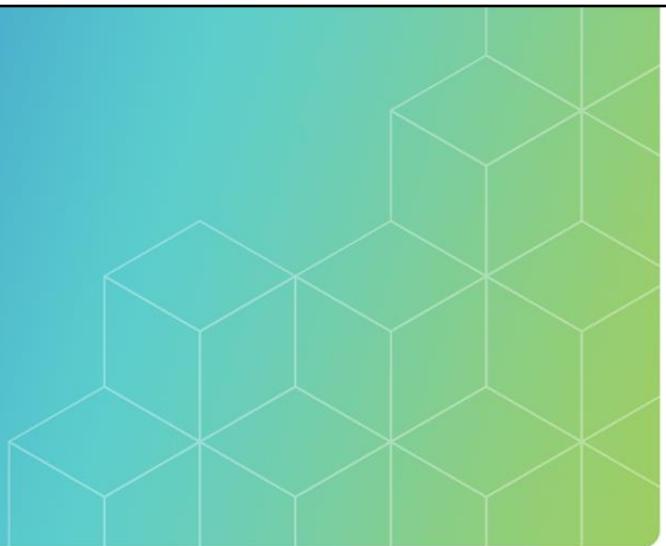
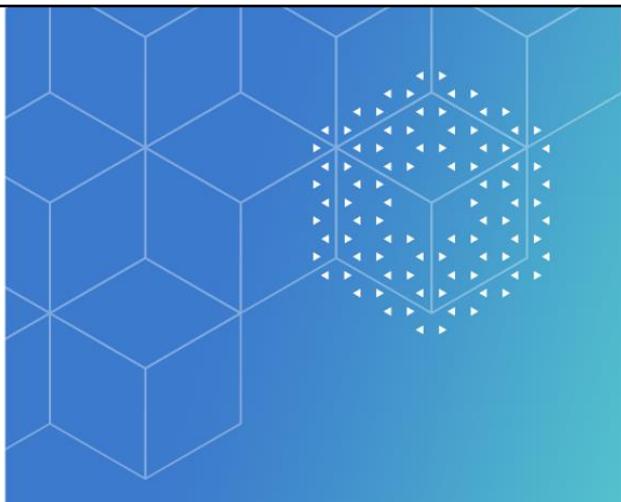
Labs

There are no labs in this module.

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



This page does not contain notes.



Module 18

Coding and Synthesizing an Example Verilog Design

cadence®

This page does not contain notes.

Module Objective

In this module, you:

- Construct a sample Verilog RTL design for synthesis

Topics

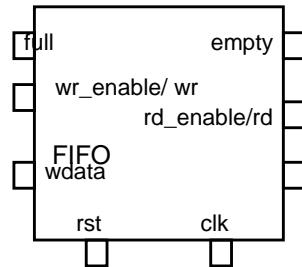
- FIFO specification
- FIFO implementation
 - Parameters
 - Ports
 - Variables
 - Functionality
 - Outputs
- Testbench design and implementation



This module applies the Verilog RTL for synthesis standards to construct a sample design.

FIFO Specification

- Parameterized for width and depth
 - Depth restricted to power of 2
- Asynchronous high-active reset
- Write/read synchronous to rising clock
- Ignores write when full
- Ignores read when empty



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



The sample design is a queue with a FIFO protocol, parameterized for width and depth, reset asynchronously and otherwise synchronized to a clock.

FIFO Implementation

Store FIFO contents in a register array.

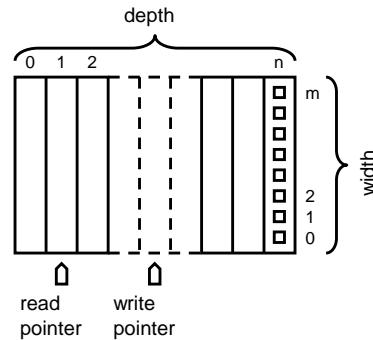
Access array with write/read pointers.

- For write:
 - Store data at current address
 - Increment pointer, maybe wrap
- For read:
 - Continuously drive current data
 - Increment pointer, maybe wrap

Derive the FIFO status from the pointer addresses.

Full – Addresses are equal and last operation was write.

Empty – Addresses are equal and last operation was read.



- Both pointers start at 0.
- Write pointer deposits value and increments.
- FIFO outputs data at read pointer.
- Read pointer increments to point at new data.
- Read pointer follows write pointer.
- Both pointers wrap around to array beginning.

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



Implement the FIFO by using a register array addressed by separate write and read pointers:

- Reset the write pointer to 0. Upon each write operation, if the FIFO is not already full, store the data at the current write address and increment the write pointer to point to the next location, wrapping back to the array beginning if necessary.
- Reset the read pointer to 0. Continuously present the array data at the current read address to the data output port. Upon each read operation, if the FIFO is not already empty, increment the read pointer to point to the next location, wrapping back to the array beginning if necessary.
- The read address thus follows the write address around the array. At any point where the read address is the same as the write address, you know that the FIFO is either empty or full. It is full if the most recent operation was to write and empty if the most recent operation was to read. Include a register to track whether the most recent operation was to write.

FIFO Module Structure

- Parameters
- Ports
- Variables
- Function

```
module fifo
#(
    // parameter declarations
)
(
    // port declarations
);

// variable declarations

// functional code

endmodule
```



The basic building block of the design hierarchy is the Verilog module. Every module description starts with the module keyword followed by the module identifier (“fifo”) and ends with the endmodule keyword. The module definition name must be unique within the definitions name space.

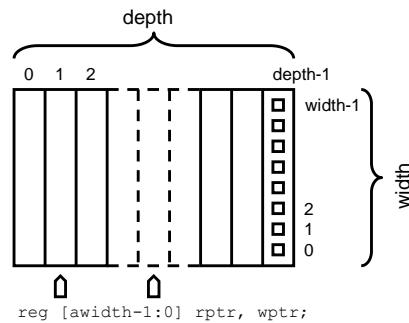
Following the module name is an optional module parameter port list, an optional list of ports or list of port declarations, and optional module items.

As for this design you will parameterized the input and output port width, you must declare the parameters before you declare the input and output ports of the module.

You must also declare all constants and variables before you use them.

FIFO Parameters

- Use the module **parameter** construct to define width and depth.
- Declare the parameters before you declare the ports.
- You can override module parameter values for each individual instance.



```
module fifo
#(
    // parameter declarations
    parameter awidth = 5,
    parameter dwidth = 8
)
(
    // port declarations
);

// variable declarations
// functional code

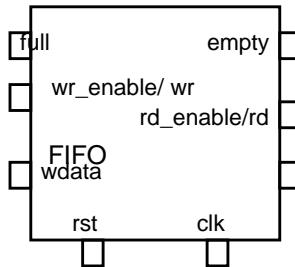
endmodule
```

```
// 16 wide by 16 deep FIFO
fifo #(16,4) fifo16x16 (...)
```



Use the module parameter construct to parameterize your module definition. You can override module parameter values on a per-instance basis. If you parameterize the input and output port widths, you must declare the parameters before you declare the input and output ports of the module. Module parameters are runtime constants that you can use wherever the syntax requires a literal value. You cannot change module parameters during the simulation.

FIFO Ports



```
module fifo
#(
    // parameter declarations
    parameter AWIDTH = 5,
    parameter DWIDTH = 8
)
(
    // port declarations
    input clk, rst, wr_en, rd_en,
    input [DWIDTH-1:0] data_in,
    output full, empty,
    output reg [DWIDTH-1:0] data_out
);

    // variable declarations

    // functional code

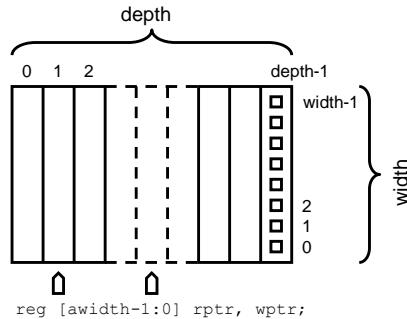
endmodule
```

Utilize a module parameter to define the width of the data input and output ports. Parameterizing the data width allows you to modify it for each instance of the FIFO.

FIFO Variables

- Register array
 - Write and read pointers
 - Most recent operation type

The diagram illustrates a memory structure with two dimensions: depth and width. The vertical axis is labeled "depth" and the horizontal axis is labeled "width". The memory is represented as a grid of cells. The depth dimension is indicated by a bracket above the first three columns, labeled "0 1 2" and "depth-1". The width dimension is indicated by a bracket on the right side, labeled "width-1". The grid contains 12 cells arranged in 3 rows and 4 columns. The bottom row has two empty cells, indicated by small white arrows pointing downwards.



```
module fifo
#(
    // parameter declarations
)
(
    // port declarations
);
// variable declarations
localparam depth = 2**awidth;
reg [dwidth-1:0] mem [0:depth-1];

reg [awidth-1:0] wptr;
reg [awidth-1:0] rptr;
reg wrote;

// functional code

endmodule
```

Verilog 2001 Power Operator



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

Utilize module parameters to define the width and depth of the register array and the width of the write address register and the read address register. Parameterizing the width and depth allows you to easily modify them for each instance of the FIFO.

FIFO Function

Asynchronous high-active reset

Write/read synchronous to rising clock

- For write:

- Ignore write when full
- Store data at current address
- Increment pointer, maybe wrap

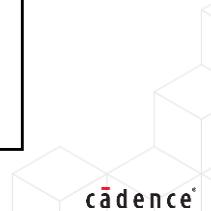
- For read:

- Ignore read when empty
- Increment pointer, maybe wrap

Track last operation

```
// functional code
always @(posedge clk or posedge rst)
begin
    if (rst)
        begin
            rptr <= 1'b0;
            wptr <= 1'b0;
            wrote <= 1'b0;
        end
    else
        begin
            if (rd && !empty)
                begin
                    rdata <= mem[rptr];
                    rptr <= rptr +1;
                    wrote <= 0;
                end
            if (wr && !full)
                begin
                    mem[wptr] <= wdata;
                    wptr <= wptr +1;
                    wrote <= 1;
                end
        end
end
```

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



Code a sequential procedure to describe the main functionality of the FIFO. The FIFO uses an asynchronous reset, so place the reset input as well as the clock input in the procedure event list. Remember that for synthesis all variables in an event list for a sequential procedure must be edge-qualified, so trigger the procedure by the positive edge of the reset and the positive edge of the clock.

While reset, initialize the write address and read address to location 0 and the tracking register to indicate that the most recent operation was to read.

Upon an active clock edge, check for a valid write or read operation:

- If the write enable is active and the FIFO is not full, store input data to the location of the current write pointer and increment the write pointer, wrapping back to the array beginning if necessary.
- If the read enable is active and the FIFO is not empty, increment the read pointer to address data in the next location, wrapping back to the array beginning if necessary.
- For this design, you can let the address registers wrap naturally. If the depth was not a power of 2 then you would need to explicitly wrap the address registers from their maximum value back to 0.

Upon an active clock edge, also update the tracking register.

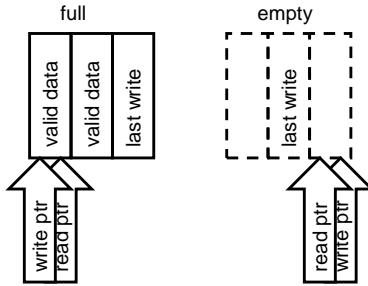
FIFO Outputs

Derive the FIFO status from the pointer addresses.

- Full – Addresses are equal and last operation was write
- Empty – Addresses are equal and last operation was read.

For read:

- Continuously drive current data.



```
// functional code  
...  
  
assign empty = (rptr == wptr) && !wrote;  
assign full = (rptr == wptr) && wrote;  
  
endmodule
```

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



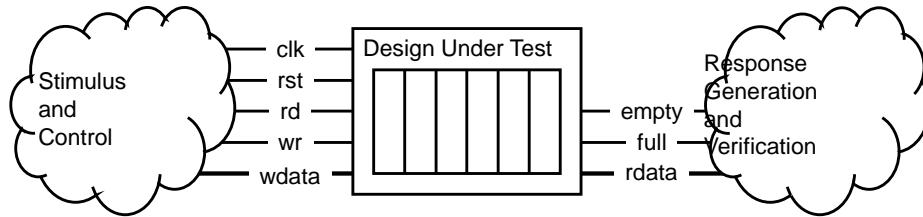
Continuously present to the data output port the register array data addressed by the read pointer.

Generate the FIFO status flags by comparing the read and write addresses. At any point where the read address is the same as the write address, you know that the FIFO is either empty or full. It is full if the most recent operation was to write and empty if the most recent operation was to read.

Note that signal “wrote” in the code is “last write” performed to the FIFO.

FIFO Testbench

- Instantiates and connects to the Design Under Test (DUT)
- Applies stimuli to the DUT input data and control ports
- Monitors the DUT output ports to verify correct behavior



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



To verify your FIFO model, you need to present it with stimulus and verify that its response is correct. In general, any testbench must do the following:

- Instantiate the Design Under Test (DUT) and connect its ports to local nets and variables.
- Apply stimuli to the input data and control (clock, reset) ports.
- Monitor the output ports to capture the response of the design to the applied stimuli.

A simple testbench records the stimulus and response vectors for subsequent analysis. Such analysis can for this design be a simple visual check of the transition data.

A more sophisticated testbench automatically compares the response to an internally generated response that is known to be correct, and reports whether the DUT failed or passed the test.

Modern testbenches tend to be extremely sophisticated. The majority of development effort is now utilized for verification.

Testbench Module Structure

- Constants
- Nets and variables
- Instances
- Function

```
module fifo_test;  
  
    // Constant declarations  
  
    // FIFO "signal" declarations  
  
    // FIFO instantiation  
  
    // FIFO stimulus  
  
endmodule
```



Why no ports?



The top level of the simulation model is a testbench module. The top-level module does not have module parameters or ports because it is not instantiated (otherwise it would not be the top level). The top-level module declares design and test configuration constants, signals to connect to the DUT instance, and the DUT instance. A top-level module can declare procedures to provide stimulus and monitor response, and can instantiate additional test modules to also do some of that work

Testbench Declarations

- Testbench constants
- FIFO *signals*

```
module fifo_test;

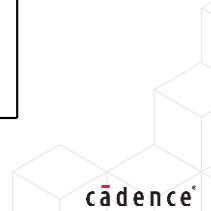
// Verilog-2001 local constants
localparam dwIDTH = 8;
localparam awIDTH = 5;
localparam DEPTH = 2 ** awIDTH;

// Variables for FIFO inputs
reg [dWIDTH-1:0] wdata;
reg clk, wr, rd, rst;

// Nets for FIFO outputs
wire full;
wire empty;
wire [dWIDTH-1:0] rdata;

// FIFO instantiation
// FIFO stimulus

endmodule
```



Declare local constants for the FIFO data width and address width and clock period.

Using these constants, declare variables to provide stimulus and nets to receive the response.

Testbench DUT Instance

- The module definition name is fifo.
- Override the module parameters.
- The module instance name is dut.
- Use named port connections to bind FIFO ports to local *signals*.

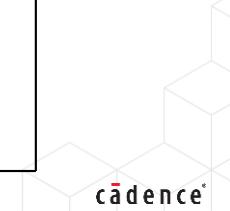
```
module test_fifo;
  // Declarations

  // FIFO instantiation
  fifo
  #(
    .awidth(awidth),
    .dwidth(dwidth)
  )
  fifo
  (
    .wdata  (wdata),
    .rdata  (rdata),
    .clk    (clk),
    .rst    (rst),
    .wr     (wr),
    .rd     (rd),
    .full   (full),
    .empty  (empty)
  );
  // FIFO stimulus
endmodule
```

 Verilog-2001 list of named parameter assignments

 Verilog-1995 list of named port connections

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



Instantiate the fifo DUT and override its module parameters and connect its ports to local nets and variables.

Testbench Stimulus

```

module test_fifo;
    // Stimulus for reading

    initial
    begin
        $monitorb("%d",$time,,clk,,rst,,wr,,rd,,           // set up monitor for signal value change
                  wdata,,full,,empty,,rdata);
        clk = 0; wr = 0; rd = 0; rst = 0; wdata = -1;          // initialize
        @(negedge clk) rst = 1;                                // assert reset
        @(negedge clk) rst = 0;                                // deassert reset
        @(negedge clk) wr = 1;                                wr signal is made high to write wdata to FIFO.
        @(negedge clk) begin
            wr = 0;
            rd = 1;                                           rd is asserted so that FIFO that was previously written can be read.
        end
        @(negedge clk) begin
            rd = 0;
            wr = 1;                                           wr signal is asserted so that WDATA can be written to the FIFO.
        end
        repeat (depth -1) @(negedge clk) wdata = wdata -1;
        @(negedge clk) begin
            wr = 0;
            rd = 1;                                           repeat loop is added to write the wdata content for (2 ** awidth-1)
        end                                              times, full signal can be seen high in the last clock pulse.
        repeat (depth +1) @(negedge clk);
        $stop;
    end
    always #10 clk = ~ clk;
endmodule

```

wr signal is made high to write wdata to FIFO.

rd is asserted so that FIFO that was previously written can be read.

wr signal is asserted so that WDATA can be written to the FIFO.

repeat loop is added to write the wdata content for (2 ** awidth-1) times, full signal can be seen high in the last clock pulse.

rd is asserted and kept high for (2 ** awidth+1) clocks for reading of FIFO, empty signal can be seen high in the last clock pulse.

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

cadence®

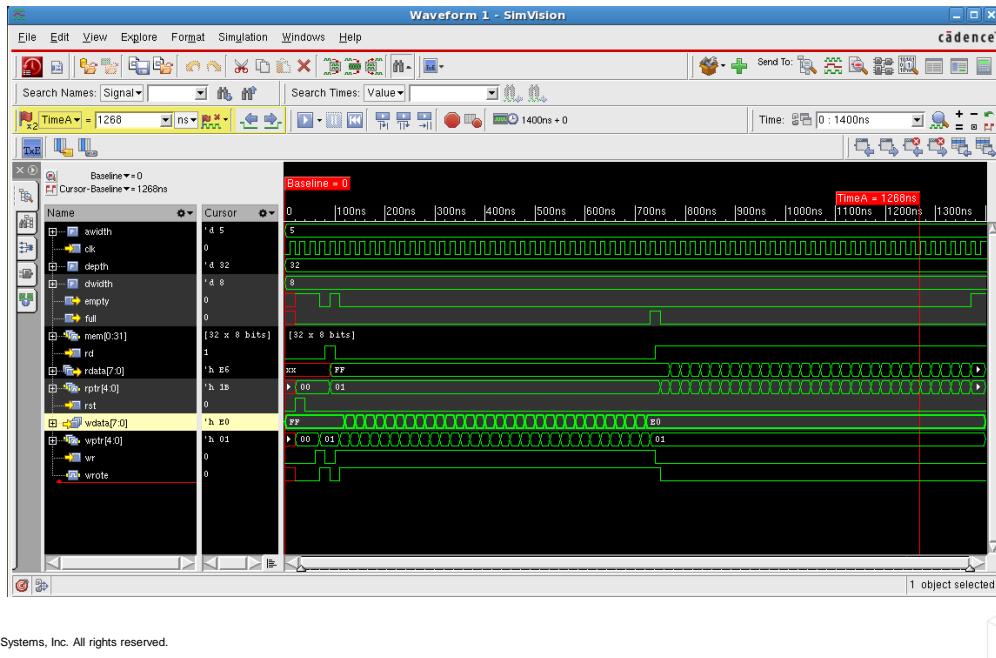
The \$monitor family of system tasks output the value of their arguments at the end of each unique simulation time in which any of them has changed.

As the fifo DUT is synchronized to the rising clock edge, the stimulus is synchronized to the falling clock edge. Having done that, the stimulus can utilize blocking assignments without causing clock/data races.

As is typical for a testbench, the stimulus tests the “corner” conditions of the DUT algorithm.

The \$stop system task stops the simulation.

Simulation Results



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



Here is a Waveform window of the Cadence SimVision graphical simulation analysis environment displaying the transition history of the testbench signals.

Module Review

1. What are module parameters?
2. What are local parameters?
3. How do you override the value of a module parameter?
4. Why do we size literals used in arithmetic operations?



This page does not contain notes.

Module Review Solutions

1. What are module parameters?
 - Module parameters are constants such as width and depth that you can override on a per-instance basis.
2. What are local parameters?
 - Local parameters are constants that you cannot override.
3. How do you override the value of a module parameter?
 - You override module parameter values as you instantiate the module by providing a list of parameter values.
4. Why do we size literals used in arithmetic operations?
 - Unsized literals are 32-bit values. A logic synthesis tool may initially assume that you need for example a 33-bit adder when adding an unsized literal to a much smaller value.



This page does not contain notes.



Lab

Lab 18-1 Coding a Serial-to-Parallel Interface Receiver

- In this lab, you code a moderately difficult Verilog RTL design for synthesis.

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



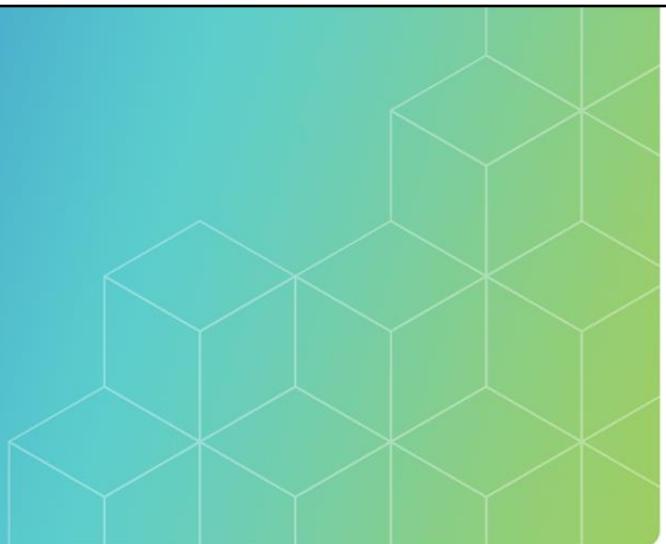
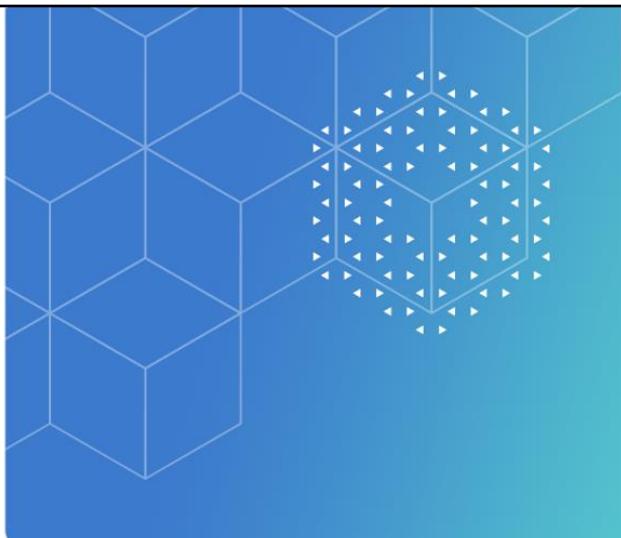
Your objective for this lab is to resolve a deadlock between two devices competing for two resources.

You will frequently encounter the situation where the test environment waits an undetermined time for the system being tested to respond to some stimulus. The system may malfunction such that it does not ever respond. The test must anticipate this failure mode and report it.

The lab model is an arbiter that arbitrates between two users of a resource. The arbiter prioritizes the requests for the resource. The lab does not model the resource.

The test environment is two instances of the arbiter and two instances of the device making the requests. The device at random intervals requires one or both resources. It requests the first resource, and upon being granted the first resource, sometimes also requests a second resource. Due to the random nature of the requests, the simulation will invariably eventually come to a point at which both devices have one resource and cannot continue until they have the second resource – hence both become “deadlocked”.

For this lab, you modify the requesting device so that after a reasonable amount of time waiting for the second resource, it cancels both current requests.



Module 19

Using Verification Constructs

cadence®

This page does not contain notes.

Module Objective

In this module, you:

- Choose and use Verilog verification constructs effectively

Topics

- Data types: real, realtime, time
- Case equality operators:
==, !=
- Procedural continuous assignments: force, release
- Looping statements:
forever, repeat, while
- Named events: event, ->
- Level-sensitive event control: wait
- Parallel blocks: fork, join
- Initial construct: initial
- Disabling tasks and named blocks: disable



This page does not contain notes.

Data Types: **real**, **realtime** and **time**

real – Double-precision float

- Cannot use posedge/negedge
- Cannot select bits or parts
- Cannot use to select bits/parts

```
real pi = 3.14
```

time – 64-bit unsigned integer

- Used almost exclusively to hold simulation time values, which can be quite large

```
time old, diff;
...
old_time = $time;
...
diff = $time - old;
```

System Function

realtime – Alias for **real**

- Used to store simulation time as a real value

```
realtime rtime;
...
rtime = $realtime;
```

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



A real variable holds a double-precision floating-point value. The simulator converts real values to integral values by rounding. Some operators you cannot use with real variables or values. For example, you cannot determine the posedge or negedge of a real variable, and you cannot select a bit or part of a real variable or real value and you cannot use a real variable or real value to select a bit or part of a variable or value.

A time variable holds a 64-unsigned integer value. People typically use it to hold simulation time values, perhaps for diagnostic and debugging purposes, but you can use it for other purposes as well.

The realtime type is an alias for the real type. People typically use it to hold simulation time as a double-precision floating-point value. Calling it a realtime variable helps them remember what they are using it for.

Case Equality Operators: === and !==

== (Logical equality)

- Result can be unknown.

	0	1	Z	X
0	1	0	X	
1	0	1	X	X
Z	X	X	X	X
X	X	X	X	X

```
...
a = 2'b1x;
b = 2'b1x;

if (a == b)
    // values match & do not contain Z or X
else
    // values do not match or contain Z or X
    // above values execute this else branch
```

===(Case equality)

- Result is always known.

	0	1	Z	X
0	1	0	0	0
1	0	1	0	0
Z	0	0	1	0
X	0	0	0	1

```
...
a = 2'b1x;
b = 2'b1x;

if (a === b)
    // values match exactly
    // above values execute this else branch
else
    // values do not match
```



Called "case equality" because the case statement matches items this way.

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



You recall that the equality operator returns a 1-bit result representing the truth of the comparison. If the equality operator cannot determine the truth of the comparison, it returns the unknown value.

The case equality operator, so called because it performs the comparison as the case statement matches item expressions, performs a definitive match for bit positions that contain high-impedance or unknown values. The case equality operator always returns a 0 or 1 result – the result is never unknown.

You will find the case equality operator to be very useful in a testbench. Let us suppose that you have a task that uses an if statement testing an expression that utilizes the logical inequality operator to compare the DUT response with the expected response. If the DUT response contains any nonbinary values, the operator returns the unknown value, so fails to execute the subsequent statement that would presumably alert you to the miscompare. In this situation, the case inequality operator does exactly what you need.

```
if ( response != expected )
begin // not executed if response is unknown!
```

Procedural Continuous Assignments: force and release

force – Forces a net or variable

- Single whole net or variable
- Bit or part of vector net
- Concatenation of above
- Not bit or part of vector variable
- Not array word

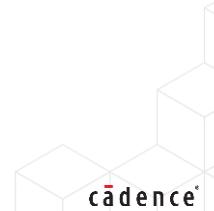
```
initial // QUICKDIRTPATCH  
force rts = rst;
```



release – Releases a force

- Net returns to assigned value
- Variable retains forced value until next assignment

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



The force statement procedurally sets up a continuous assignment of an expression to a net or variable. It is equivalent to a continuous assignment, but as you do it procedurally, you can also apply it to variables. You can also apply it to a constant bit-select of a vector net, a part-select of a vector net, or a concatenation. You cannot apply it to a bit-select or a part-select of a vector variable, or to an array word. The procedural continuous assignment overrides any other assignment to the net or variable until it is released or another procedural continuous assignment is made.

The release statement releases the forced continuous assignment. A net that is subject to a continuous assignment immediately resumes that value. A variable retains the forced value until it is next procedurally assigned.

The force statement is an anachronism. Early simulators provided an interpretive user interface that accepted interactive procedural statements from the keyboard. Verification personnel used the force statement interactively as a temporary patch mechanism during their debug efforts. You would be very unlikely to see it in source code as you do here.

This example forces the rts signal to follow the rst signal. The user hypothesized that the rts signal was implicitly declared when connected to an instance port, but spelled incorrectly. The user temporarily forced it to follow the rst signal to prove the hypothesis.

To prevent such implicit net declarations, use the `default_netttype compiler directive to set the default net type to none.

Loop Statements: while

```
while ( expression )
    statement
```

- Must previously declare any variables used in expression
- While expression is known and true, executes statement

```
integer count;
...
while (count < 10)
begin
    // statements
    count = count + 1;
end
```

```
reg [7:0] tempreg;
reg [3:0] count;
...
// Count the ones in tempreg
count = 0;
while (tempreg)
begin
    if ( tempreg[0] )
        count = count + 1;
    tempreg = tempreg >> 1;
end
```

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



The while loop executes a statement while its expression is true.

The first example tests a counter and executes a statement while the count is less than 10.

The second example counts the number of 1 bits in a bit mask.

Loop Statements: `repeat` and `forever`

`forever statement`

- The `forever` loop is equivalent to a `while` loop with the expression always true:

```
while ( 1 )
    statement
```

`repeat (expression)` `statement`

- The `repeat` loop is equivalent to a `for` loop with the index variable automatically declared, initialized and incremented:

```
integer h;//hidden index
for ( h = 0;
      h < expression;
      h = h + 1 )
    statement
```

```
module multiplier (
    input      [3:0] a, b,
    output reg [7:0] result
);

reg [7:0] temp_a;
reg [3:0] temp_b;

always @ (a, b)
begin
    temp_a = a;
    temp_b = b;
    result = 0;
    repeat ( 4 )
        begin
            if ( temp_b[0] )
                result = result + temp_a;
            temp_a = temp_a << 1; // left
            temp_b = temp_b >> 1; // right
        end
    end
endmodule
```

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



The forever loop is equivalent to a while loop with its expression always true.

The repeat loop executes a statement the number of times by the value of the parenthesized expression.

This example implements a multiplication algorithm that iterates `specified` through the bit width of its inputs.

Named Events: `event` and `->`

- **Event** has no Boolean value.
- An Event is defined using the event type.
- An Event is caused using the trigger `->`.

```
module event_example(a,b,c,int);
Input [3:0] a,b;
Output [4:0] c;
Event disp_c; Event declaration

Always @ (a or b)
begin
  c=a+b;
  -> disp_c; Event trigger
end

Always @ (disp_c) Event used
$display (c);
endmodule
```

 Event is not synthesizable.

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



The event is a very efficient Verilog type because it does not have a Boolean value nor does it need to be scheduled. It is used mostly in testbenches.

To trigger procedures in a behavioral design or testbench, if you do not need to communicate a value, use a named event instead of a variable. This prevents your co-workers from having to guess whether you actually used the value anywhere. The simulator handles named events more efficiently than variables, as it does not have to consider the value.

Level-Sensitive Event Control: wait

**wait (expression)
statement**

- The **wait** statement waits for the expression to become true.
- It does not wait if the expression is already true.

```
module runReqAck (
    input run,ack,rst,
    output reg req
);
    always @ (posedge run)
        begin
            req = 1;
            wait (ack || rst)
            req = 0;
            if (!rst)
                send_stuff;
        end
endmodule
```



The wait statement blocks if the expression is not true and then unblocks when the expression becomes true.

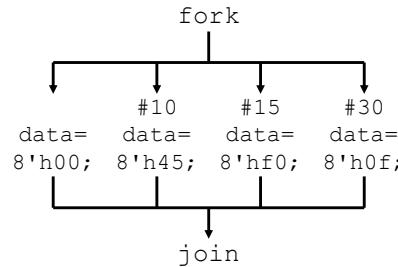
This example responds to the run positive edge by asserting the req output and waiting asynchronously for either the ack input or the rst input, upon which it deasserts the req output, and if the reset input is not true, calls the send task.

Parallel Blocks: fork and join

fork...join – Concurrent statements

- Enclosed statements start executing immediately.
 - Start in any order.
- Each statement is subject to only its own timing controls.
- The statement after join executes when all forked statements complete.

```
module forkjoin_tb;
  reg [7:0] data;
  // instance of DUT
  initial
    fork
      data = 8'h00;
      #10 data = 8'h45;
      #15 data = 8'hf0;
      #30 data = 8'h0f;
    join
endmodule
```



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



Statements enclosed by the fork...join construct execute concurrently, that is, they all start together at the moment simulation enters the block, and thereafter each statement is subject to only its own timing controls.

The parallel block completes when all forked statements have completed.

You can use a parallel block to start multiple concurrent processes. One common use is to start one process that takes an undetermined amount of time to complete and to start a second process as a watchdog timer that issues a message and disables the first process if the first process takes too long to complete.

This example forks four stimulus statements to apply data at different offsets from their starting points. Here the offsets are ordered for readability, but any such ordering is not necessary.

More About Parallel Blocks

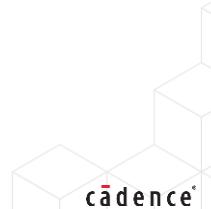
A parallel block can contain any procedural statement of any complexity.

- Task calls, loops, other blocks...

Issues

- You cannot predict the execution order of statements scheduled to execute at the same simulation time. This can cause race conditions.
- If any statement (e.g., forever loop) does not complete, the block never joins.

```
module concurrent_tb;
  reg [7:0] data;
  // instance of DUT
initial
  fork
    #10 data = 8'h45;
    #20 repeat (7)
      #10 data = data + 1;
    #25 repeat (5)
      #20 data = data << 1;
    #99 data = 8'h0f;
  join
endmodule
```



A parallel block is a set of statements that the stimulator starts executing simultaneously at the moment simulation enters the block. Each of those parallel statements is then subject to only its own timing controls and can be of unlimited complexity. Each can further contain parallel and sequential subblocks.

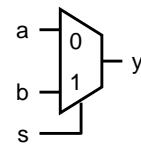
In this example, the two repeat loops start at different times and execute concurrently. This stimulus would be very difficult to apply in a single sequential block.

initial and always Constructs

always – Executes *always*

- Starts at beginning of simulation
- Continually loops
 - Must have a timing control
- Hardware construct

```
module mux (
  input a,b,s,
  output reg y );
  always @*
    y = s? b : a;
endmodule
```



initial – Executes *initially*

- Starts at beginning of simulation
- Executes exactly once
- Testbench construct

```
module mux_tb;
  wire a,b,s,y;
  mux m(a,b,s,y);
  reg [2:0] stim=0;
  assign {a,b,s}=stim;
  initial
    #1 repeat(7)
      begin
        stim=stim+1;
        #1;
      end
  endmodule
```



An always construct executes continuously. It always contains at least one timing control so that it blocks at some point to let the simulator do something else. The always construct is ideal for describing the behavior of hardware that always reacts to input transitions. If it describes hardware for synthesis, then it has only one event control, and places it immediately after the always keyword.

The initial construct executes exactly once, starting at the beginning of the simulation. It can contain any number of timing controls, including none at all. The initial construct steps through the timing controls, and when it executes its last statement, it is finished and never runs again during that simulation session. The initial construct is ideal for describing the behavior of a testbench that applies each stimulus just once.

You can also use the initial construct to describe system-level behavior that you do not want to synthesize. After executing statements to initialize various parts of the system, you can execute a forever loop in which you continually generate randomized stimulus.

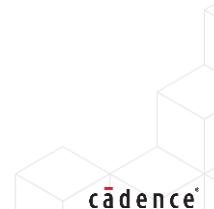
This example generates stimulus within an initial construct. In a repeat loop it generates all possible binary combinations of multiplexor inputs.

Disabling Tasks and Named Blocks: disable

disable – Disables task or named block

- Activity associated with task or named block ceases.
- Execution continues with statement after block or task call.
- Subsequent statement execution can immediately again call task or re-enter block.

```
module busif_tb;
...
always #50 clk = ~clk;
initial
begin: STIMULUS
repeat(5) @(negedge clk);
wait(!interrupt);
cpu_driver(8'h00);
wait(!interrupt);
cpu_driver(8'haa);
...
end
...
always @(posedge interrupt)
begin
 disable cpu_driver;
 service_interrupt;
end
endmodule
```



The disable statement terminates activity associated with the task or named block. You can use it to work around the lack of C-like break and continue statements. You can use it to behaviorally model asynchronous activity such as interrupts.

The simulator discontinues any activity associated with the task or named block. If executing the task or named block, it resumes execution with the statement after the task call or after the named block. Implementations can choose whether to remove scheduled nonblocking assignments and whether to discontinue procedural continuous assignments, so this can be a source of differences between simulators.

Disabling a task or named block has no persistence. Subsequent statement execution can immediately again call the task and execution can immediately re-enter the named block, for example, if it is located in an always construct or loop construct.

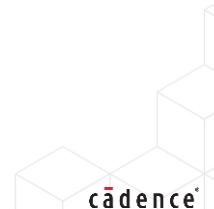
This example repeatedly calls the cpu_driver task to apply stimulus. Upon arrival of an interrupt, this example terminates the current execution of the cpu driver to service the interrupt.

Module Summary

Now you can appropriately choose and effectively use Verilog verification constructs.

This module discussed:

- Data types: **real**, **realtime**, **time**
- Case equality operators: **==**, **!=**
- Procedural continuous assignments: **force**, **release**
- Looping statements: **forever**, **repeat**, **while**
- Named events: **event**, **->**
- Level-sensitive event control: **wait**
- Parallel blocks: **fork**, **join**
- Initial construct: **initial**
- Disabling tasks and named blocks: **disable**



This module examined a collection of constructs you cannot use in a design meant for synthesis but that have more or less use in a behavioral design or testbench:

- The real data type is the equivalent of the C double-precision float.
- The case equality operator performs a definitive match for bit positions that contain high-impedance or unknown values, so always returns a known 0 or 1 result.
- The force statement procedurally sets up a continuous assignment of an expression to a net or variable, and the release statement releases it.
- The while loop executes a statement while its expression is true.
- The forever loop is equivalent to a while loop with its expression is always true.
- The repeat loop executes a statement the number of times specified by the value of the parenthesized expression.
- Use a named event instead of a variable to trigger procedures in a behavioral design or testbench if you do not need to communicate a value.
- The wait statement blocks if the expression is not true and then unblocks when the expression becomes true.
- Statements enclosed by the fork...join construct execute concurrently.
- The initial construct executes exactly once, starting at the beginning of the simulation.
- The disable statement terminates activity associated with the task or named block.

Module Review

1. Which inequality operator (`!=` or `!==`) would you use to detect that the DUT response is not what you expect it to be?
2. Verilog has a named event and at least one other popular HDL does not. Explain how it is useful but not really needed for RTL design and test.
3. Explain the difference between the edge-sensitive “`@`” event control and the level-sensitive “`wait`” event control.
4. Modify the **always** construct to act like an **initial** construct and modify the **initial** construct to act like an **always** construct:
 - a. `always begin statements end`
 - b. `initial begin statements end`



This page does not contain notes.

Module Review Solutions

1. Which inequality operator (`!=` or `!==`) would you use to detect that the DUT response is not what you expect it to be?
 - Use the case inequality operator (`!==`) because it performs a definitive match for the unknown state ("X") and always returns a known 0 or 1 result.
2. Verilog has a named **event** and at least one other popular HDL does not. Explain how it is useful but not really needed for RTL design and test.
 - The named **event** has no value or connectivity. It has no place in the implementation of or interface with a RTL design. For behavioral modeling, it simulates more efficiently than a **reg** variable and more clearly indicates the abstract nature of the model.
3. Explain the difference between the edge-sensitive "@" event control and the level-sensitive "wait" event control.
 - The edge-sensitive event control blocks execution of the subsequent statement until a listed event occurs. An event has no duration, so the control always blocks. The level-sensitive event control blocks the subsequent statement until its expression result is 1. If the expression result is already 1 when the control is executed, it does not block.
4. Modify the **always** construct to act like an **initial** construct and the **initial** construct to act like an **always** construct:
 - `always begin statements wait(0); end`
 - `initial forever begin statements end`

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



This page does not contain notes.



Lab

Lab 19-1 Resolving a Deadlocked System

- For this lab, you will resolve a deadlock between two devices competing for two resources.
- You modify the requesting device so that after a reasonable amount of time waiting for the second resource, it cancels both current requests.

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



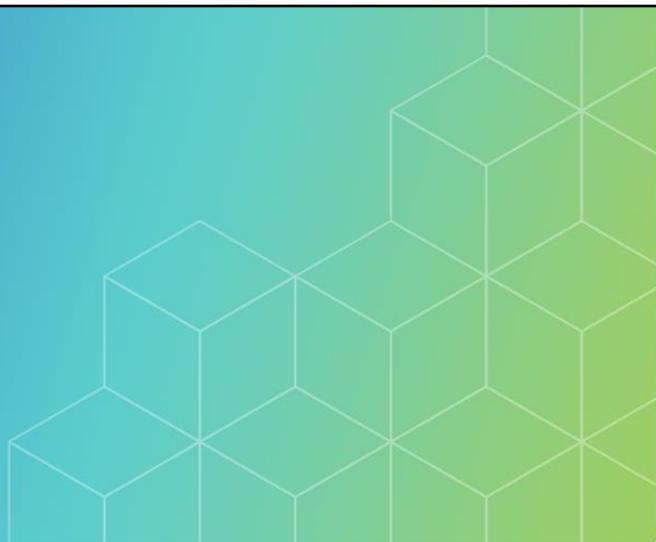
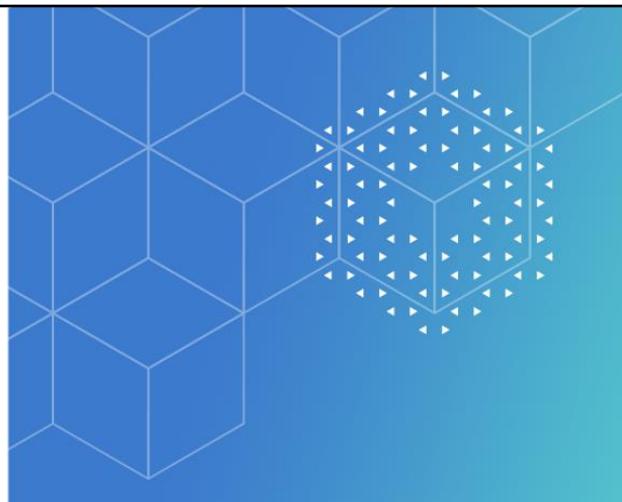
Your objective for this lab is to resolve a deadlock between two devices competing for two resources.

You will frequently encounter the situation where the test environment waits an undetermined time for the system being tested to respond to some stimulus. The system may malfunction such that it does not ever respond. The test must anticipate this failure mode and report it.

The lab model is an arbiter that arbitrates between two users of a resource. The arbiter prioritizes the requests for the resource. The lab does not model the resource.

The test environment is two instances of the arbiter and two instances of the device making the requests. The device at random intervals requires one or both resources. It requests the first resource, and upon being granted the first resource, sometimes also requests a second resource. Due to the random nature of the requests, the simulation will invariably eventually come to a point at which both devices have one resource and cannot continue until they have the second resource – hence both become “deadlocked”.

For this lab, you modify the requesting device so that after a reasonable amount of time waiting for the second resource, it cancels both current requests.



Module 20

Coding Design Behavior Algorithmically

cadence®

This module just briefly introduces testbenches and focuses primarily on the difference between behavioral and RTL modeling. It uses a bus interface controller model to illustrate these differences.

Module Objective

In this module, you:

- Model a design behavior algorithmically and are introduced to a testbench concept

Topics

- Comparing Behavioral and RTL modeling
- Example Bus Interface Controller Model
- Testbench Introduction



Your objective is to model a design at the behavioral level of abstraction and at the register-transfer level of abstraction. To do this, it is probably beneficial to first look at some simple examples.

Verilog Supports Multiple Levels of Abstraction

Most examples herein are coded at the Register Transfer Level (RTL).

- Coded in terms of registers values at each clock cycles.
- People tend to think of RTL as the “synthesis subset”:
 - Using only the constructs and templates accepted by synthesis.

```
always @ (posedge clk)
begin
  d_out <= d_tmp;
  d_tmp <= d_in;
end
```

Verilog also supports the *behavioral* level of abstraction.

- No regard for an actual hardware implementation.
 - All constructs usable.
- Useful for algorithmic or architectural exploration and for testbench.

```
always @ (d_in)
d_out <= repeat(2)
@ (posedge clk)
d_in;
```



Almost all examples you have seen here up to this point are coded at the Register Transfer Level (RTL). That means that the behavior is coded in terms of registers and the values they have at any particular clock cycle. People tend to think of RTL as the synthesis subset, but though the constructs and templates accepted by synthesis are definitely RT-level constructs, RTL coding can include non-synthesizable constructs.

Verilog also supports the behavioral level of abstraction, in which you code behavior with no regard for an actual hardware implementation, so can utilize any Verilog construct. The behavioral level of abstraction is useful for exploring design architecture, and especially useful for developing testbenches.

Comparing Behavioral and RTL Modeling

RTL Modeling

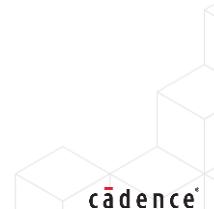
- Defines model architecture
- Clocks generally required
 - Activity synchronized to clock edge
 - Models timing with clock cycles
- Data stored in “registers”
- Contains code constructs and style supported by synthesis standard
- Used for all code to be synthesized

Behavioral Modeling

- Defines model behavior
- Clocks generally omitted
 - Activity synchronized to any event
 - Models timing with delay control
- Data stored in any manner
- Contains any Verilog construct
- Used for system-level modeling and for testbench

 RTL does not necessarily mean the “synthesis subset” but many people think of it that way.

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



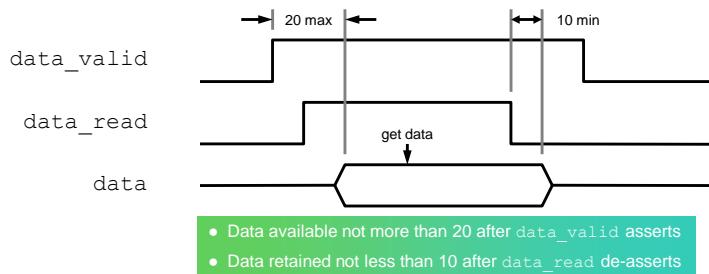
Verilog supports multiple levels of abstraction, from a stochastic system-level model for architectural exploration down to a switch-level model. People most often use the loosely defined behavioral level or the better-defined register-transfer level.

- The behavioral level defines only the model behavior. It can define the behavior totally algorithmically, it can “step” the algorithm by a series of ordered events, and it can provide inter-event delays for visualization purposes and for latency and throughput calculations. The behavioral level can store data in any manner and use any Verilog construct. Your testbench is an example of a behavioral model.
- The register-transfer level (RTL) defines the model in terms of sequential storage units referred to as registers and the synchronized transfer of data between the registers. It defines clocks, variables to represent the storage elements, and processes that react to active edges of the clock.

Since the advent of logic synthesis, the RT level has come to commonly mean the usage of an HDL in a structured manner from which a logic synthesis tool can infer combinational and sequential logic.

Hence, the standards defining the synthesizable subset of the HDL have become de facto definitions of the RT level of abstraction.

Example Bus Interface Controller Model



```
// Behavioral model
always
begin
    wait (data_valid);
    data_read = 1;
    #20;
    local_buffer = data;
    data_read = 0;
    #10;
    wait (~data_valid);
end
```

- Defines model behavior
- Clocks generally omitted
- Data stored in any manner
- Contains any Verilog construct

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

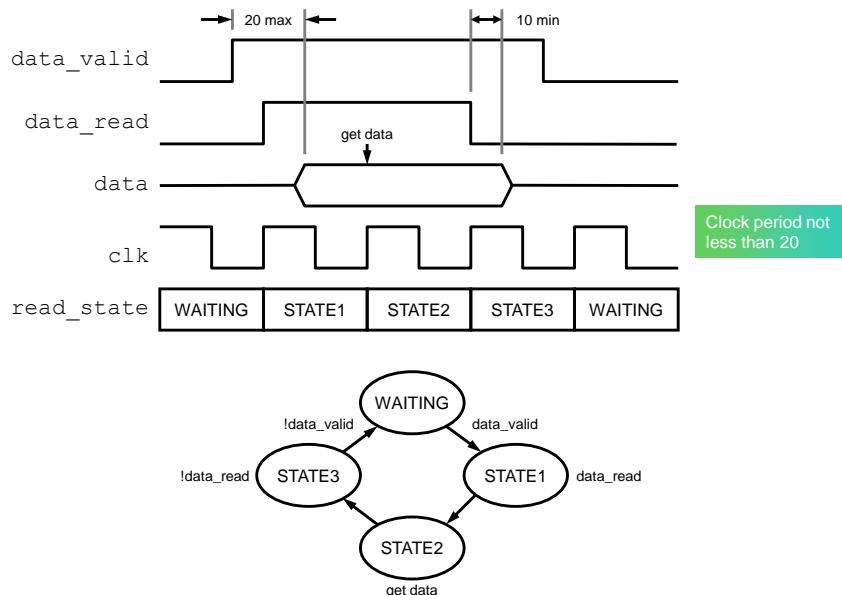


The waveform diagram depicts the Bus Interface Controller specification:

- The `data_valid` input indicates that new data is available.
- The `data_read` output indicates the model is ready to receive the data.
- The `data` input appears no later than 20ns after `data_valid` asserts.
- The model can drop the `data_read` output any time after capturing the data.
- The `data_valid` input drops to indicate the end of the transaction.

The behavioral model implements this behavior without regard to clocks.

Bus Interface Controller Implementation



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



The RTL model synchronizes its behavior to clock edges and defines a state machine to step through the behavior. The state machine steps through four states, stepping on the positive clock edge:

- The WAITING state waits for the **data_valid** input and transitions to STATE1.
- The STATE1 state asserts the **data_read** output and transitions to STATE2.
- The STATE2 state captures the **data** input and transitions to STATE3.
- The STATE3 state drops the **data_read** output and transitions back to WAITING.

RTL Procedures Review

Sequential Logic

- Storage is implied.
- Outputs are a function of current inputs and storage variables.
- Procedure is sensitive to clock edge and optionally to reset edge.
- Nonblocking assignments.

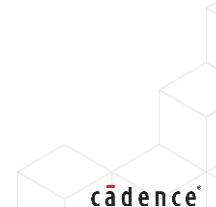
```
always @*
  if (sel == 0)
    y = a;
  else if (sel <= 5)
    y = b;
  else
    y = c;
```

Combinational Logic

- Storage is not implied.
- Outputs are a function of current inputs.
- Procedure is sensitive to all inputs.
- Blocking assignments.

```
always @ (posedge clk)
  if (rst || count==9)
    count <= 0;
  else
    count <= count + 1;
```

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



The definition of combinational logic is that if you account for propagation delay then you know what the outputs are based solely upon the state of the inputs, i.e., it has no internal storage. This requires that the simulator evaluate the inputs and update the outputs upon any transition of any input. You can guarantee this by doing two things:

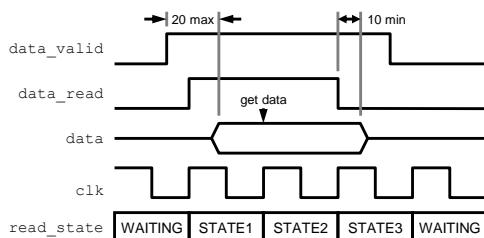
- You form a list of event expressions containing all nets and variables that the procedural block reads, or use the Verilog 2001 implicit event expression list wildcard (*) character.
- You ensure that each execution of the procedural block assigns a value to all variables that the block writes.

The left combinational block does both. It uses the implicit event expression list, and with the last else branch ensures that the d variable gets a value if it did not already get one. Note that this block uses blocking assignments.

The definition of sequential logic is that it must somehow imply storage. If it evaluates the inputs upon any transition of any input, but fails to update some output for some combination of inputs, then it implies storage for that output to hold its value until it is next assigned. It implies latch storage because it is not sensitive to any particular input edge. If the block evaluates its inputs and updates its outputs only upon a particular input edge, then it implies storage to hold all outputs until that next particular input edge. It implies flip-flop storage because it is sensitive to a particular input edge.

The right sequential block implies flip-flop storage because it updates its outputs only upon a particular input edge. Note that this block uses nonblocking assignments to prevent clock/data races.

Bus Interface Controller RTL

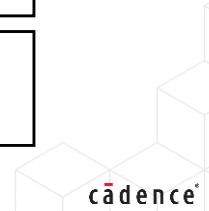


```
// FSM
always @(posedge clk or posedge reset)
  if (reset == 1)
    read_state <= WAITING;
  else
    case (read_state)
      WAITING: if (data_valid)
                  read_state <= STATE1;
      STATE1 : read_state <= STATE2;
      STATE2 : read_state <= STATE3;
      STATE3 : if (!data_valid)
                  read_state <= WAITING;
    endcase
```

```
// Constants and variables
localparam integer WAITING = 0,
            STATE1   = 1,
            STATE2   = 2,
            STATE3   = 3;
reg [7:0] local_buffer;
reg [1:0] read_state;
reg        data_read;
```

```
// Data store
always @(posedge clk)
  if (read_state == STATE1)
    local_buffer <= data;
```

```
// Output decode
always @ (read_state)
  data_read = read_state == STATE1
             || read_state == STATE2;
```



To code the RT-level model, we focus separately upon its four features:

- Firstly, we declare the model's constants and variables. We declare constants for the FSM states, a variable to hold the captured data, a variable to hold the current state, and a variable to hold the value of the `data_read` output.
- Second, we define the FSM. We make a sequential procedural block sensitive to the positive clock edge and sensitive to the positive edge of an asynchronous reset. The reset sets the state to `WAITING`, and then in a case statement, we step through the states, each step dependent on the current state and some steps depending also on the value of an input.
- Third, we define a sequential procedural block to capture incoming data upon the active clock edge that occurs while in the `STATE1` state. We could have merged this feature into the FSM block but chose to not do so because it is not really part of the FSM.
- Fourth, we define a combinational block to generate the `data_read` output as a function of the current state. We could also have merged this feature into the FSM block but chose to not do so because it also is not really part of the FSM.

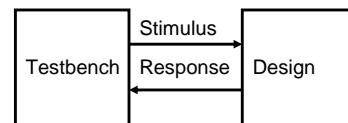
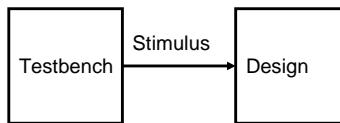
Testbench Introduction

Sophisticated Testbench

- Models operational environment
- Complex randomized stimulus
- Interacts with design
- Response checking during simulation

Simple Testbench

- Just stimulates the design
- Simple stimulus generation
- No interaction with design
- Response checking is a post-simulation activity



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



The majority of the project development effort is typically focused upon design verification.

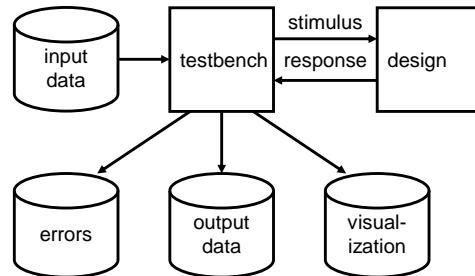
Testbenches tend to start off simple but quickly become sophisticated.

- A simple testbench might declare nets and variables, connect them to a DUT instance, provide stimulus to the DUT, and dump the stimulus and response to a file for later analysis. A simple testbench is appropriate for a small simple test case and not appropriate for a large complex design.
- A sophisticated testbench will at least check the DUT response spontaneously. It would ideally dynamically react to the DUT response, that is, communicate with the DUT. A truly powerful testbench would as much as practical mimic the expected DUT operating environment.

Testbenches and Files

Verilog can read and write external files:

- Input data generated by some other tool.
- Output data to be analyzed later.



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



Your testbench can read data from a file, for example a file of test patterns that some other program or tool such as a system modeling tool created.

Your testbench can write data to a file, for example:

- A record of what was tested and what failed.
- Stimulus and response data for later analysis.
- Pattern (image) data for a visualization application.

Module Summary

Now you can model design behavior algorithmically.

In this module, you learned to:

- Describe behavioral modeling, which is used for system-level modeling and for testbenches. The behavioral model stores the data in any manner, uses any Verilog construct, and generally omits clocks.
- Describe RTL modeling architecture, which is used for all code that you are going to synthesize. The RTL model includes the required clocks, data stored in registers, code constructs, and style supported by the synthesis standard.
- Write simple and sophisticated testbenches.



You can now model a design both behaviorally and at the register-transfer level. This module just briefly introduced testbenches and focused primarily on the difference between behavioral and RTL modeling. It used a bus interface controller model to illustrate these differences.

Module Review

1. What is behavioral modeling used for?
2. What is RTL modeling used for?
3. Explain the difference between the procedures that represent combinational logic and the procedures that represent sequential logic.
4. Suggest a way to define the constant values that represent FSM states.



This page does not contain notes.

Module Review Solutions

1. What is behavioral modeling used for?
 - Behavioral models are not restricted to synthesis-supported constructs, so can use any construct to model a system and to code a testbench.
2. What is RTL modeling used for?
 - The register-transfer level of abstraction has come to mean the use of synthesis-supported templates to represent digital hardware.
3. Explain the difference between the procedures that represent combinational logic and the procedures that represent sequential logic.
 - The outputs of combinational logic are a function purely of the current inputs, with no implied storage. The procedure is sensitive to all inputs and typically uses blocking assignments.
 - The outputs of sequential logic are a function of at least one storage variable and potentially also inputs. The procedure is sensitive to a clock edge and optionally sensitive to a reset edge, and must use nonblocking assignments for the storage variables.
4. Suggest a way to define the constant values that represent FSM states.
 - The most prevalent and a synthesis-supported way to define FSM state values is to use the parameter or localparam constructs.

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



This page does not contain notes.

Labs

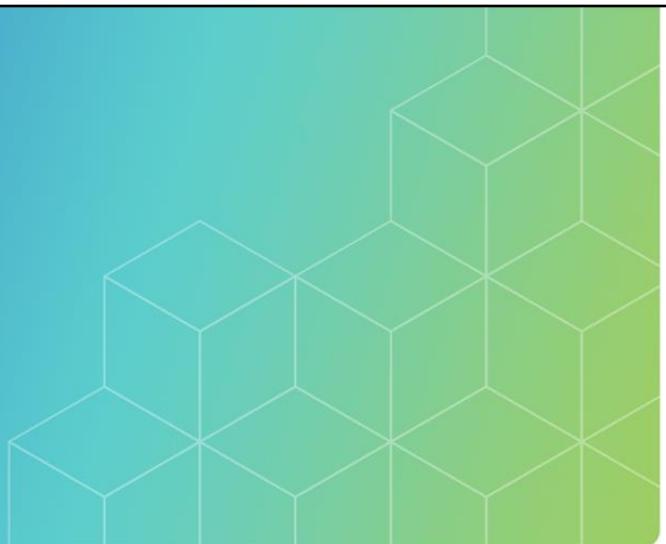
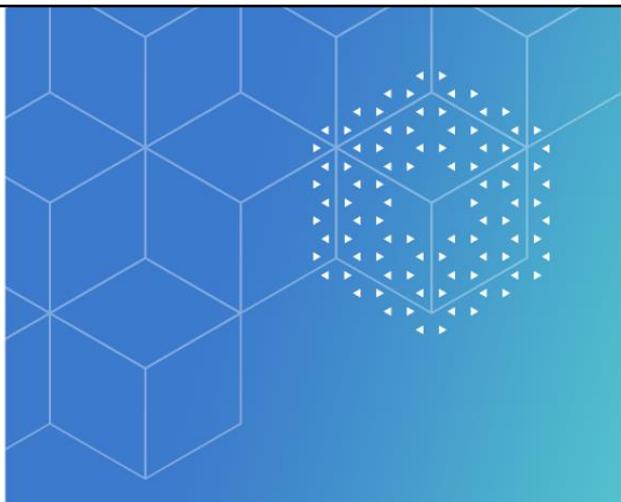


There are no labs in this module.

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



This page does not contain notes.



Module 21

Using System Tasks and System Functions

cadence®

This module examines system tasks and system functions commonly used for stimulus generation and debugging. Later modules will revisit some of these in more detail and introduce some more.

Module Objective

In this module, you:

- Use system tasks and functions for stimulus generation and debug

Topics

- Displaying messages
- Getting simulation time
- Formatting the time display
- File input and output
- Applying stimulus from a file
- Controlling the simulation
- Passing real values through ports
- Getting command-line values
- Dumping value-change data
- Checkpointing the simulation

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



This page does not contain notes.

Displaying Messages: \$display and \$write

The **\$display** and **\$write** system tasks print values to the standard output.

- **\$display** appends a newline
- **\$write** does not

The default base is decimal. Verilog supports other bases:

- **\$displayb**
- **\$displayo**
- **\$displayh**

Arguments can include formatting strings, which can include escaped characters:

`\n \t \ddd \\ \"`

```
module disp_wr;
reg [7:0] var1, var2, var3;
initial
begin
var1 = 8'h0F;           // 15
var2 = 8'b01010101; // 85
var3 = 8'b11001100; // 204
$display (var1,var2,var3);
$displayb (var1,, var2,, var3);
$displayh (var1); \ Null Argument
$write (var1);
$writeth (" var2(hex) is ", var2);
$writeo ("\\n var3(oct) is ", var3);
$write ("\\n");
end
endmodule
```

Output sized to variable – 8 bits always written in decimal as 3 characters

String Argument

15 85204 00001111 01010101 11001100 0f 15 var2(hex) is 55 var3(oct) is 314
--

Consecutive commas replaced by single space

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



The **\$display** and **\$write** system tasks are identical, except that **\$display** adds a newline character to the end of every output, and **\$write** does not.

The **\$display** and **\$write** system tasks support multiple default bases. The default base for **\$display** and **\$write** is decimal. Append the **b** character to make the default base binary, **o** for octal and **h** for hexadecimal. You can embed formatters in string arguments to override that default radix for individual subsequent arguments. Later pages demonstrate this.

The display width depends on the base and the maximum size of the variable. For example, the maximum value of an 8-bit vector is 256, which requires 3 decimal characters, so every 8-bit value written in decimal requires 3 characters. To display an 8-bit vector requires 8 characters in binary, 3 in octal and 2 in hexadecimal. The decimal radix replaces leading zeros by spaces, while the other radices display the leading zeros. When using a format string, you can override this automatic sizing. Later pages demonstrate this.

You can include string arguments to separate values and to make the output more meaningful. You can also separate values by including null arguments. A null argument is an extra comma between arguments. For a null argument, the simulator displays a single space.

Formatting Text Output

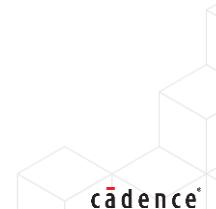
Each format specifier formats one argument value.

- Usually grouped into one first argument format string.
- Can be distributed among the task arguments.
- Matched to argument values in order – one to one match.
 - More values than format specifiers simply use default format.
 - More format specifiers than values is an error.

```
module disp_wr_fmt;
  reg [7:0] var1, var2, var3, var4;
  initial
    begin
      var1 = 8'h0F;           // 15
      var2 = 8'b01010101; // 85
      var3 = 8'b11001100; // 204
      $display ("%b", var1);
      $display ("var2 binary: %b \t", var2, "hex: %h", var2);
      $display ("%h \t %h \t %h", var1, var2, var3);
      $write ("var3 is hex %h \t octal %o \n", var3, var3);
    end
  endmodule
```

00001111
var2 binary: 01010101 hex: 55
0f 55 cc
var3 is hex cc octal 314

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



You can embed formatters in your string arguments to, among other things, override the default base. A formatter can apply to only one subsequent argument, but other than that, you can group the formatters any way you want. Many people put them all in one first argument string, and some people insert individual format strings just prior to the argument they apply to.

When using format strings, you must be very careful to not apply the formatters to null arguments. You might want to adopt the convention that you do not use both in the same statement.

Within any character string literal, you can place escaped character sequences to represent special characters such as newline and tab characters.

The 1st display statement overrides the default base to display the value of the var1 argument in binary. Format characters can be in either case, but most people use the lower case as is done here. The var1 argument is an 8-bit vector, so displays in 8 characters.

The 2nd display statement displays the value of var2 in binary and hexadecimal, with a space and tab between them.

The 3rd display statement displays the value of var1, var2 and var3 in hexadecimal with a space and tab and an extra space between them.

The 4th display statement displays the value of var3 in hexadecimal and octal, with a space and tab and extra space between them.

Reference: Format Specifications

Format	Description	Example
%b %B	Display in binary format	\$display("%b",1'b1); // 1
%o %O	Display in octal format	\$display("%o",3'o7); // 7
%d %D	Display in decimal format	\$display("%d",4'd9); // 9
%h %H	Display in hexadecimal format	\$display("%h",4'hF); // f
%c %C	Display in ASCII character format	\$display("%c",65); // A
%s %S	Display as a string	\$display("%s","foo"); // foo
%e %E	Display real in exponential format	\$display("%e",3.1); // 3.100000e+00
%f %F	Display real in float format	\$display("%f",3.1); // 3.100000
%g %G	Display real using %e or %f	\$display("%g",3.1);
%l %L	Display library binding information	\$display("%l"); // work.test
%m %M	Display hierarchical name	\$display("%m"); // top.test1
%t %T	Display in current time format	\$display("%t",\$time); // 1.1 ns
%v %V	Display net signal strength	\$display("%v",n1); // St1
%u %U	Unformatted 2 value binary data	
%z %Z	Unformatted 4 value binary data	



The integer formatters by default display values in a width sufficient to contain the maximum value the expression size can represent. For the decimal format they display leading spaces and for the other formats they display leading zeros. This tabular format makes vertically long displays more readable, but often unnecessarily wastes horizontal display space. An integer, for example, always requires 10 spaces no matter how small the value. You can override this automatic sizing. Placing a 0 character between the % character and the radix character reduces the displayed width to only the width actually required to display the value. Unlike the C language, in Verilog, you cannot specify the display format more precisely.

The binary formatter always displays high-impedance or unknown bits with lower case z and x characters. The octal and hexadecimal formatters use upper case Z and X characters where the character represents contain at least one bit with a binary value and at least one bit with a high-impedance or unknown value. In this case, an unknown bit trumps any high-impedance bits to display an upper case X. The decimal formatter does not try to compute the bit range in this case and displays the entire number using uppercase Z or uppercase X.

The %l formatter displays the library binding information – the module definition name preceded by a period (.) and the name of the library in which the elaborator found the compiled definition.

The %m formatter displays the hierarchical scope of the instance containing the display system task.

The %t formatter displays the argument according to the format you set with the \$timeformat system task. With this task, you set the format width and precision for a time argument. You can use this formatter for any integer value if you want to thoroughly confuse your co-workers.

The %v formatter displays the strength and value of a net. Net drivers have associated with them one of eight strengths, that the simulator uses to resolve the strength and value of the net.

The %u and %z formatters write binary data, presumably to a file, for an application that reads binary data.

Reference: Escape Sequences in Format Strings

Argument	Description
\n	The newline character
\t	The tab character
\\"	The \ character
\\""	The " character
\ddd	A character specified by 1 to 3 octal digits
%% [2001]	The % character

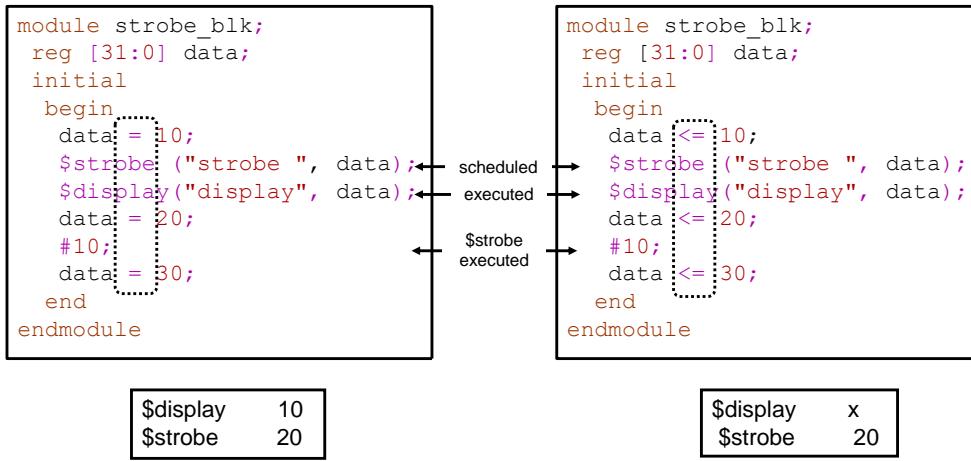


Here is the full set of escape sequences provided by the IEEE Std 1364-2001 Verilog HDL. The Verilog-2001 update added the escaped percent (%) character. Note that it is escaped by another percent character and not by a backslash (\) character.

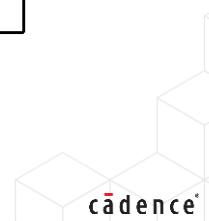
Displaying Messages: \$strobe

The **\$strobe** system task is similar to the **\$display** system task

- Execution of **\$strobe** is scheduled for the end of the current time “slice”.



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



The **\$strobe** system task evaluates and displays its arguments at the end of the current simulation time, and is otherwise identical to the **\$display** system task. Like the **\$display** system task, it is available in four versions with four different default bases.

The left example uses blocking assignments, so the **\$display** system task displays the assigned value 10 and the **\$strobe** system task displays the later assigned value 20.

The right example uses nonblocking assignments, so the **\$display** system task displays the not-yet-assigned default unknown value (x) and the **\$strobe** system task displays the later assigned value 20.

Getting Simulation Time: \$time, \$stime, \$realtime

The **\$time**, **\$stime** and **\$realtime** system functions return the simulation time:

- **\$time**
 - Returns time as a 64-bit integer
- **\$stime**
 - Returns time as a 32-bit integer
- **\$realtime**
 - Returns time as a real number

```
'timescale 1 ns / 100 ps

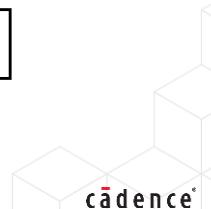
module time_sysf;
  reg [5:0] data;
  initial
    begin
      #10;
      data = 6'd20;
      $display ($stime,, data);
      #10.4;
      data = 6'd30;
      $write ("time: %0d", $time);
      $write (" realtime: %f", $realtime);
      $write (" data: ", data, "\n");
    end
endmodule
```

%0d removes leading spaces

```
10 20
time: 20 realtime: 20.4 data: 30
```

The simulator scales returned time values to the timescale of the calling module.

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



The `timescale directive specifies the timescale units for following modules.

The \$time system function returns 64-bit simulation time scaled to the time scale of the calling module and then if needed rounded to an integer value.

The \$stime system function returns the lower 32 bits of the \$time value. This might not for obvious reasons represent the actual simulation time. For short simulations, you can use \$stime to display a value that takes ten spaces to display instead of twenty.

The \$realtime system function returns 64-bit simulation time scaled to the time scale of the calling module and converted to a real value.

The \$display statement displays 64-bit simulation time and a 6-bit data value separated with a space character.

The \$write statements display formatted 64-bit simulation time followed by time as a real number followed by the data value, each separated by a space character.

Displaying Messages: \$monitor

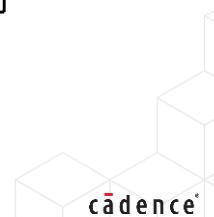
The **\$monitor** system task continuously monitors nets and variables:

- It acts like \$strobe when any argument (other than \$time) changes value.
- It supports the same default bases and formatters as the \$strobe system task.
- Only one \$monitor task can be active at a time.
 - A new \$monitor replaces any current \$monitor.
- You can disable monitoring with **\$monitoroff** and re-enabled it with **\$monitoron**.

```
module time_monitor;
reg [7:0] var1, var2, var3, var4;
initial
begin
$monitor("%0d \t %h %h %h",
$time, var1, var2, var3);
var1 = 8'h0F;
var2 = 8'b0101_0101;
var3 = 8'b1100_1100;
#5;
var1 = 8'h80;
#8;
var2 = 8'h66;
var3 = 8'h77;
#4;
var1 = 8'h55;
end
endmodule
```

0	0f	55	cc
5	80	55	cc
13	80	66	77
17	55	66	77

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



The \$monitor system task, once invoked, continually monitors its arguments, and displays their values at the end of the time “slice” in which any argument changed.

Thus the \$monitor system task, like the \$strobe system task, involves the scheduler.

Like the \$strobe system task, it is available in four versions with four different default bases.

At most one monitor is active at a time. A new call to \$monitor replaces the list of monitored signals. The simulator ceases monitoring the previous list of signals and instead monitors the new list of signals.

You can control monitoring with the \$monitoroff and \$monitoron system tasks. You can use these system tasks to monitor signal values only between certain time intervals of the simulation.

Formatting the Time Display: \$timeformat

The **\$timeformat** system task sets the display format for the **%t** formatter.

- Controls units, precision, suffix and field width.
- Format is consistent for **\$time**, **\$stime** and **\$realtime**.
- Format is consistent for all timescales.
 - **`timescale** is required.
- Supported by all variants of **\$display**, **\$monitor**, **\$strobe** and **\$write** tasks.

```
`timescale 1 ns / 10 ps
module time_fmt;
  wire o1;
  reg in1;
  assign #9.53 o1 = ~in1;
  begin
    $display("time \t realtime \t in1 \t o1");
    $timeformat(-9, 2, " ns", 10);
    $monitor("%0d \t %t \t %b \t %b",
             $time, $realtime, in1, o1);
    in1 = 0;
    #10;
    in1 = 1;
    #10;
  end
endmodule
```

time	realtime	in1	o1
0	0.00 ns	0	x
10	9.53 ns	0	1
10	10.00 ns	1	1
20	19.53 ns	1	0

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



You use the **\$timeformat** system task to specify how the **%t** formatter displays time values.

The **\$timeformat** system task takes four arguments:

- The 1st argument is an integer giving the time unit to which to scale the displayed time. The argument value must be between 0, meaning seconds, and -15, meaning femtoseconds.
- The 2nd argument is an integer giving the precision for the display, that is, how many digits to display after the decimal point.
- The 3rd argument is a string to append to the display. People typically use this string to show the time units they scaled the time to.
- The 4th argument is an integer giving the minimum field width for the display. The display inserts leading spaces to force this minimum field width.

The **\$timeformat** system task in this example scales simulation time to nanoseconds, displays it with two fractional digits, follows it with a string indicating its time units, and displays it with a ten-character minimum field width.

The IEEE Std 1364-2001 Verilog HDL Section 17.3.2 states that the **\$timeformat** applies to “all **%t** formats specified in all modules that follow in the source description until another **\$timeformat** system task is invoked”.

File Output: Opening Files with `$fopen`

You can write to files:

- Open the file with **\$fopen**.
 - Returns a 32-bit unsigned integer multi-channel descriptor (MCD) with one bit set to 1.
 - Returns 0 if unsuccessful.
 - Each successful \$fopen returns a different MCD bit set to 1.
 - Channel 0 is pre-opened to the standard output.
 - You can open up to 30 additional channels.
 - Bit 31 is reserved
always 0
 - Close the channel with **\$fclose**.
 - Allows channel reuse.

```

module file_write;
integer data_chan, warn_chan;
reg [7:0] var1, var2;
initial
begin
  data_chan = $fopen("data.txt");
  if (!data_chan) $finish;
  $displayb ("data_chan ", data_chan);
  warn_chan = $fopen("warn.txt");
  if (!warn_chan) $finish;
  $displayb ("warn_chan ", warn_chan);
  $fmonitor (data_chan, var1, var2);
  // do stuff ...
  $monitoroff;
  $fclose(data_chan);
end
endmodule

```

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



Use the \$fopen system function to open a file. The \$fopen system function returns a 32-bit unsigned multichannel descriptor (MCD) uniquely associated with the file. The system function returns 0 if it could not open the file for writing.

You must save the returned MCD as an integer or 32-bit reg vector.

Think of the MCD as a set of 32 flags, where each flag represents a single output channel:

- The most significant bit of an MCD is reserved for purposes we will discuss later and is always 0.
 - The least significant bit of an MCD is reserved for the standard output. Most simulators also write this output to a log file.
 - You can open up to 30 additional channels and simultaneously write to any combination of the open channels.

Use the \$fclose system task to close the channels specified in its MCD argument. The \$fclose system task closes the specified channels. Subsequent calls to the \$fopen system function reuse those released channels.

File Output: Writing to Files

The **\$fdisplay**, **\$fmonitor**, **\$fstrobe** and **\$fwrite** set of system tasks take a MCD first argument and can simultaneously write to multiple channels.

```
reg [7:0] x, y;
integer datafile, timefile, bothfile;
initial
begin
    datafile = $fopen("data.txt");
    timefile = $fopen("time.txt");
    bothfile = datafile | timefile;
    x = 77;
    y = 2;
    #50;
    $fdisplay(datafile, "X-axis %d\nY-axis %d", x, y );
    $fdisplay(timefile, "%0d X = %0d Y = %0d", $time, x, y );
    $fdisplay(bothfile, "Output to both files");
end
```

MCD Values
datafile 00010
timefile 000100
bothfile 00110

X-axis 77
Y-axis 2
Output to both files

50 X = 77 Y = 2
Output to both files

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



The four formatted display tasks (\$display, \$write, \$monitor, and \$strobe) have counterparts that write to a specific set of channels rather than to just the standard output.

The counterparts have an f prefix character and take an extra MCD first argument. The MCD can be any arbitrary expression that results in a 32-bit unsigned integer value. This value determines which open files the system task writes to. You simultaneously write multiple channels by bitwise ORing existing MCDs. You can set bit 0 to also write to the standard output.

This example opens the data.txt file and the time.txt file. It then bitwise ORs the two MCDs to create a third MCD representing both channels. It later writes to the channels individually and simultaneously.

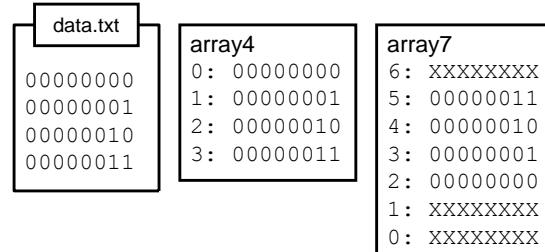
File Input: \$readmemb and \$readmemh

Verilog system tasks read from a file into a 1-D array of reg vector:

- Binary **\$readmemb**
- Hexadecimal **\$readmemh**

You can optionally specify start and end addresses (data at these addresses must exist in the file).

```
module readfile;
reg [7:0] array4 [0:3];
reg [7:0] array7 [6:0];
initial
begin
$readmemb("data.txt", array4);
$readmemh("data.txt", array7, 2, 5);
end
// other procedures ...
endmodule
```



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



The \$readmemb and \$readmemh system tasks load data from a text file into an array. Depending on the task, you provide the data in a binary radix or a hexadecimal radix. The system task has no versions to accept octal data or decimal data.

- The 1st argument is the data file name.
- The 2nd argument is the array to receive the data.
- The 3rd argument is an optional start address, and if you provide it, you can also provide –
- The 4th argument optional end address.

The data file itself can optionally specify to what addresses its data belongs.

If the memory addresses are not specified anywhere, then the system tasks load file data sequentially from the left memory address bound to the right memory address bound. The year 2005 standard changes this default loading order to be from the lowest address toward the highest address.

If the file does not specify addresses for its data and the system task specifies a starting address, then the system tasks load file data sequentially from that starting address toward the specified ending address, or if no ending address is specified, then toward the right memory address bound for Verilog 2001 tools and toward the high memory address bound for Verilog 2005 tools.

If the file does specify addresses for its data, then if the system task specifies an address range, that address range must encompass and exhaust the available data.

File Input Data Format

- You can embed hexadecimal address information in the text file.
- You can use comments, underscores and spacing for readability.

data.txt

```

0000_0000
0110_0001 0011_0010
// comments are ignored
// addresses 3-255 not defined
@100 // hex
1111_1100
/* addresses 257-1022 not defined */

@3FF
1110_0010

```

```

...
reg [7:0] mem1Kx8 [0:1023];
...
$readmemb("data.txt", mem1Kx8);
...

```

mem1Kx8
0: 00000000
1: 01100001
2: 00110010
...
256: 11111100
...
1023: 11100010

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



Here are the things that a data file can contain:

- White space, including spaces, newlines, tabs, and formfeeds.
- Comments, which can be single line or multiline.
- Binary numbers or case-insensitive hexadecimal numbers, which:
 - If address values, are hexadecimal digits following an at (@) character.
 - If data, are a sequence of binary or hexadecimal digits without base or size information but that might embed underscore (_) characters for readability.

This example data file contains 8-bit binary data for only the addresses 0, 1, 2, 256 and 1023. A system task call that specifies an address range must include these addresses in that range.

Enhanced C-Style File I/O

Verilog-2001 adds C-like file system operations to the Verilog language.

- With added type option, **\$fopen** returns a 32-bit File Descriptor (FD):
 - Type option is access, e.g., "r", "w", "a"
 - FD has bit 31 set to differentiate it from an MCD
 - Can represent $2^{**}31$ channels but never multiple channels
 - Channels 0, 1, 2 already opened to stdin, stdout, stderr
- Counterparts exist to almost all C file system functions:
 - Writing characters, lines, binary, or formatted
 - **\$fdisplay \$fmonitor \$fstrobe \$fwrite \$fflush**
 - Reading characters, lines, binary, or formatted
 - **\$fgetc \$ungetc \$fgets \$fread \$fscanf**
 - Manipulating the file pointer
 - **\$fseek \$ftell \$rewind**



The Verilog 2001 update added support for C-style file I/O routines, thus providing the capability to input something more than just a memory data. These routines utilize a 32-bit unsigned integer file descriptor. A file descriptor represents a single channel. The file descriptor has the most significant bit set to 1 to differentiate it from a multichannel descriptor, which has the most significant bit set to 0. As in C, the first three channels are pre-opened to the standard output, input, and error files.

The enhanced C-style file I/O re-uses the existing output system tasks and provides new system tasks for input and for file pointer manipulation.

Reference: Enhanced C-Style File I/O

System Task/Function	Returns	Description
<code>\$fopen ("filename", type) ;</code>	<code>fd</code>	Opens a file
<code>\$fclose (fd) ;</code>		Closes a file
<code>\$ferror (fd, str) ;</code>	<code>errcode</code>	Gets error code and description
<code>\$fgetc (fd) ;</code>	<code>char</code>	Gets a character
<code>\$ungetc (c, fd) ;</code>	<code>errcode</code>	Ungets (puts back) a character
<code>\$fgets (str, fd) ;</code>	<code>errcode</code>	Gets a string
<code>\$fflush ([fd]) ;</code>		Flushes output buffer(s)
<code>\$fread (reg, fd) ;</code>	<code>errcode</code>	Reads binary data to a reg
<code>\$fread (mem, fd [, [start] [, [count]]]);</code>	<code>errcode</code>	Reads binary data to a reg array
<code>\$fscanf (fd, format, args) ;</code>	<code>errcode</code>	Reads formatted data from a file
<code>\$ftell (fd) ;</code>	<code>position</code>	Gets the file pointer position
<code>\$fseek (fd, offset, operation) ;</code>	<code>errcode</code>	Repositions the file pointer
<code>\$rewind (fd) ;</code>	<code>errcode</code>	Rewinds the file pointer
<code>\$sscanf (str, format, args) ;g</code>	<code>errcode</code>	Reads formatted data from a string
<code>\$sprintf (reg, format, args) ;</code>	<code>length</code>	Formats data to a string
<code>\$fwrite (reg, args) ;</code>		Formats data to a string

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



This page intends to make you aware of what is available but encourages you to refer to the Verilog standard for full and complete documentation. The system tasks are similar, but not identical, to the C standard I/O library routines.

Example for Reading from a File

```

module testbench;
reg [8:1] result;
integer data_chan;

initial
begin
data_chan = $fopen("source.dat", "rb");

while (result != 255)
begin : get_char
  result = $fgetc(data Chan);
  if (result == 255)
    begin
      $display ("Finished!");
      //Prevent $display printing out EOF
      disable get_char;
    end
  $display("%s %0d %b", result, result, result);
end

fclose(data Chan);
end
endmodule

```



```

95 01011111
! 33 00100001
@ 64 01000000
% 37 00100101
^ 94 01011110
& 38 00100110

10 00001010
a 97 01100001
b 98 01100010
c 99 01100011
z 90 01011010

10 00001010
Finished!

```

Type field determines access mode.

Result is set to EOF (-1) if an error occurs.

"r" or "rb" – read (from existing file)
 "w" or "wb" – write (to new file)
 "a" or "ab" – append (to existing file)
 "r+", "r+b", "rb+" – read/write (existing)
 "w+", "w+b", "vb+" – read/write (new)
 "a+", "a+b", "ab+" – append (new or old)



An example for reading from a file using some of the C-style File I/O.

Simulation Control: \$finish and \$stop

- The **\$stop** system task interrupts simulation and enters the interactive mode.
 - You can continue the simulation from the stop point.
- The **\$finish** system task terminates the simulation and exits the simulator.

```

task expect (input [7:0] resp, exp);
  if (resp !== exp)
    begin
      $display("want %b - got %b",
               exp, resp);
      $display("TEST FAILED");
      $stop;
    end
  endtask

initial
begin
  wait (empty);
  $display("Data buffer empty");
  $display("TEST COMPLETE");
  $finish;
end

```

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



The \$stop system task interrupts the simulator and lets you enter commands interactively. You can continue the simulation from the stop point.

The \$finish system task terminates the simulation.

This example defines a task to compare the actual response with the expected response and interrupt the simulation if they are not identical. If the test completes successfully then the initial block detects the completion and terminates the simulation.

Passing Real Values Through Ports

Module input ports must be net types.

- E.g., **wire [range]**

Module output ports can also be integral types.

- I.e., **reg [range], integer, time**

Use these conversion functions as needed:

```
my_reg64 = $realtobits ( my_real ) ;
my_real  = $bitstoreal ( my_reg64 );
my_integer = $rtoi ( my_real ) ;
my_real    = $itor ( my_integer ) ;
```

```
module real_out ( ro );
  output reg [63:0] ro;
  real rv;
  always @rv ro = $realtobits(rv);
  // do real stuff with rv
endmodule
```

```
module real_in ( ri );
  input wire [63:0] ri;
  real rv;
  always @ri rv = $bitstoreal(ri);
  // do real stuff with rv
endmodule
```

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



You cannot directly pass a real value through a module port. You instead use the \$realtobits and \$bitstoreal system functions to convert a real value to and from a bit vector that you can pass through module ports. The Verilog bit vector representation of real values conforms to IEEE Std 754-1985 Binary Floating-Point Arithmetic. Verilog rounds the converted value to the nearest legal value.

The example module named `real_out` calls `$realtobits` in a procedure to convert an internal real variable value to a vector value for module output.

The example module named `real_in` calls `$bitstoreal` in a procedure to convert a vector module input value to a real value for the internal variable.

Getting Command-Line Values

Verilog-2001 adds two system functions for retrieving command-line options.

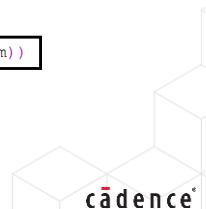
- **\$test\$plusargs("prefix")**
 - Returns 0 if no command-line “plusarg” starts with the provided prefix.
 - Returns 1 if a command-line “plusarg” starts with the provided prefix.
- **\$value\$plusargs("prefix%format",variable)**
 - Formatters are %b %o %d %h %e %f %g %s.
 - Returns 0 if no command-line “plusarg” starts with the provided prefix.
 - Returns 1 if a command-line “plusarg” starts with the provided prefix.
 - Places scanned command-line “plusarg” suffix in variable.

```
% verilog +Hello_There ...           % verilog +"stimulus=test2" ...
```

```
if ($test$plusargs("Hello")) // true
```

```
if ($value$plusargs("stimulus=%d",testnum))
```

Note: Does not require recompilation!

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


The Verilog 2001 update added two system functions for retrieving command-line “plus” options:

- The \$test\$plusargs system function returns 0 if no command-line “plus” argument has the provided prefix, and 1 if at least one command-line “plus” argument does have the provided prefix.
- The \$value\$plusargs system function does that, and also retrieves the formatted value of the character substring following the prefix. You can use any of the integer, real or string formatters.

The first illustration is determining whether the invocation argument prefix “+Hello_There” was provided.

The second illustration is determining whether the invocation argument prefix “+stimulus=test” was provided, and if it was, interprets the immediately following character substring as a decimal number and places it in the variable named “testnum”.

Dumping Value-Change Data (VCD)

The Verilog language provides testbench capture of signal waveform data.

- A Value Change Dump (VCD) file contains ASCII header information, variable definitions, and value change data.
- You manipulate a VCD database with these Verilog system tasks:

\$dumpfile ("filename")	Opens database. Optionally provides name. Standard default name is dump.vcd
\$dumplimit (size)	Stops recording after size bytes
\$dumpvars (...)	Selects signals for recording (see next page)
\$dumpoff	Stops recording
\$dumpon	Starts recording again
\$dumpall	Checkpoint the values of all recorded signals
\$dumpflush	Flushes database to disk

```
initial
begin
    $dumpfile(); // dump.vcd
    $dumpvars(); // current scope and down
end
```

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



The IEEE Std. 1364-1995 Verilog LRM describes testbench capture of Verilog signal waveform data to a Value Change Dump (VCD) database. A VCD file contains ASCII header information, variable definitions, and value change data. The VCD facility records signal value changes that occur during the simulation of your design. It records changes on only the signals that you select.

You can manipulate a VCD database with these standard Verilog system tasks:

- Use the \$dumpfile system task to open a VCD database. If you do not specify a filename argument, the simulator opens the default dump.vcd database. You can open only one database during simulation, and must open it before you invoke any other \$dump* tasks.
- Use the \$dumplimit system task to limit the size in bytes of the database file. The simulator stops dumping value change data when the database file reaches this limit.
- Use the \$dumpvars system task to add signal names for the simulator to dump. You can invoke \$dumpvars from several places in your simulation, but must execute them all at the same simulation time. You cannot add signals after the simulator starts dumping data. Following pages describe the levels and list_of_scopes_or_variables arguments you provide upon calling this task.
- Use the \$dumpoff system task to suspend dumping of value change data. The simulator writes a record to the database that gives all dumped signals the unknown value, then suspends dumping.
- Use the \$dumpon system task to resume dumping of value change data.
- Use the \$dumpall system task to create a checkpoint of the dumped signals. The simulator normally dumps individual signal values only as they change. You can use this task to update the database just prior to ending simulation. Some waveform display tools may require this update.
- Use the \$dumpflush system task during simulation to flush the dump file buffer to disk if you want some other application to be able to access all of the current data.



VCD Signal Selection: \$dumpvars

You can optionally provide `$dumpvars` with a levels argument and a list of scopes and variables. The levels argument applies to the subsequent scopes.

```
$dumpvars [ (levels [,list_of_scopes_or_variables]) ]
```

- You can provide no arguments, just the levels argument, or all arguments:

Task Call	Depth / Scope
\$dumpvars	all / all
\$dumpvars (1)	1 / current
\$dumpvars (1, top.u1)	1 / top.u1
\$dumpvars (2, top.u2)	2 / top.u2
\$dumpvars (0, top.u3, top.u1.r0.q)	all / top.u3, top.u1.r0.q

```
initial
begin
  $dumpfile("verilog.dump");
  $dumpvars(0,top);
end
```

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



Use the `$dumpvars` system task to add signal names for the simulator to dump. You can invoke `$dumpvars` from several places in your simulation, but must execute them all at the same simulation time. You cannot add signals after the simulator starts dumping data. You can optionally provide a levels argument and a `list_of_scopes_or_variables` argument:

- The levels argument applies to subsequent scopes. The levels argument defaults to 0, which means to probe all levels of the current hierarchy.
- The `list_of_scopes_or_variables` argument defaults to the current scope. If you provide a levels argument, then you can also provide a `list_of_scopes_or_variables` argument to probe modules, instances, and signals in the specified hierarchies.

The example opens a VCD file called `verilog.dump` and dumps to it value change data for all nets and variables in and under the scope named `top`.

Dumping Extended Value-Change Data (EVCD)

The Verilog language provides extended testbench capture of port data.

- An Extended Value Change Dump (EVCD) file contains ASCII header information, port definitions, and change data for port direction, strength, and value.
- Tools can “play” this data back for gate-level simulation (for example of an ASIC).
- You can manipulate an EVCD database with these Verilog system tasks:

<code>\$dumports (scopes, filename)</code>	Open database. Optionally provide scopes and file name. Standard default name is <code>dumports.evcd</code>
<code>\$dumportslimit (size, filename)</code>	Stop recording after <code>size</code> bytes
<code>\$dumpportsoff (filename)</code>	Stop recording
<code>\$dumpportson (filename)</code>	Start recording again
<code>\$dumpportsall (filename)</code>	Checkpoint the values of all recorded signals
<code>\$dumpportsflush (filename)</code>	Flush database to disk

```
initial
  $dumports (top.dut); // DUT scope to dumports.evcd
```



The Verilog-2001 update describes testbench capture of Verilog port data to an Extended Value Change Dump (EVCD) database. An EVCD database is a record of the port direction, strength, and value changes that occur during the simulation of your design. It records changes on only the ports that you select.

You can manipulate an EVCD database with these Verilog system tasks. You can provide each task with a filename argument. The filename argument can be a string literal, or a reg variable or expression representing the string’s ASCII value. If you do not specify a filename argument, the \$dumports system task by default opens the “dumports.evcd” database file, and the other system tasks apply to all open EVCD databases:

- Use the \$dumports system task to open an EVCD database and specify the scopes at which to dump port information. The default is the scope of the task call. You can invoke \$dumports from several places in your simulation, but must execute them all at the same simulation time, and may not reuse arguments. You cannot add scopes after the simulator starts dumping data.
- Use the \$dumportslimit system task to limit the size in bytes of the database file. The simulator stops dumping port change data when the database file reaches this limit.
- Use the \$dumpportsoff system task to suspend dumping of port change data. The simulator writes a record to the database that gives all dumped ports the unknown value, then suspends dumping.
- Use the \$dumpportson system task to resume dumping of port change data.
- Use the \$dumpportsall system task to create a checkpoint of the dumped ports. The simulator normally dumps individual port data only as it changes. You can use this task to update the database just prior to ending simulation. Some waveform display tools may require this update.
- Use the \$dumpportsflush system task during simulation to flush the dump file buffer to disk if you want some other application to be able to access all of the current data.

Checkpointing the Simulation: \$save and \$restart

Debugging long simulations can be tedious, especially if a problem occurs only after several minutes of simulation.

To quickly iterate through the debug/simulate cycle:

- Use **\$save** to save the simulation database at the point just before the problem occurs.
 - You can use **\$incsave** periodically to save an incremental database.
- Use **\$restart** to reload an incremental or full simulation database.
 - Or a invocation option if provided.

```
'ifdef SAVE
initial $save("fullsim.db");
always #5000 $incsave("incsim.db");
`endif

'ifdef RESTART
initial $restart("incsim.db");
`endif
```

```
% xrun -define SAVE
```

```
% xrun -define RESTART
```

The IEEE Std 1364-2001 Verilog HDL does not require implementations to support these system tasks.

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


These constructs are very useful for long simulations.

While debugging a problem, every time you need to add more signal information, you need to reset the simulation and simulate again up to the point of the problem.

If you save the state of the simulation just prior to the point of the problem, then you can quickly restart simulation at the point of the problem.

The \$save system task saves everything needed to restart the simulation.

The \$incsave system task saves only the current states of the nets, variables, and event queues. The incremental database is associated with the full database, so the full database must still exist to restart from an incremental database.

The IEEE Std 1364-2001 Verilog HDL does not require implementations to support these system tasks. Most simulator vendors provide alternative or additional proprietary ways to checkpoint the simulation that may have additional features. Check the vendor's documentation of their simulator. For example, Cadence® Xcellium™ simulation does not support these system tasks, but instead provides the save and restart interactive commands.

The first illustrated simulation session saves a full simulation database (fullsim.db) and then every 5000 time units saves an incremental simulation database (incsim.db).

The second illustrated simulation session restarts simulation using the incremental simulation database (incsim.db), which first loads the full simulation database (fullsim.db) and then overlays it with the incremental data.

Module Summary

You should now be able to use system tasks and functions for stimulus generation and debug.

This module discussed:

- Displaying messages with **\$display**, **\$write**, **\$strobe**, and **\$monitor**
- Getting simulation time with **\$time**, **\$stime**, and **\$realtime**
- Formatting the time display with **\$timeformat**
- File input and output with **\$fopen**, **\$fclose**, etc.
- Applying stimulus from a file with **\$readmemb** and **\$readmemh**
- Controlling the simulation with **\$finish** and **\$stop**
- Passing real values through ports with **\$realtobits** and **\$bitstoreal**
- Getting command-line values with **\$test\$plusargs** and **\$value\$plusargs**
- Dumping value-change data with **\$dumpvars** and **\$dumports**
- Checkpointing the simulation with **\$save** and **\$restart**

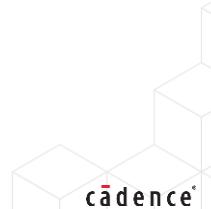


You can now use system tasks and system functions for stimulus generation and debug.

This module discussed a variety of system calls your testbench can make to support stimulus generation and debug.

Module Review

1. What output system task would you use to capture the “steady-state” settled values of a collection of nets and variables whenever any one or more of them transition?
2. Explain briefly the difference between **\$stop** and **\$finish**.
3. What is the decimal value of the integer “testnum” upon execution of
`if ($value$plusargs ("stimulus=test%d", testnum))`
if the simulator invocation includes the `+stimulus=test2` option?
4. Which VCD system task would you use to dump value change data for the ports of an RTL ASIC you are developing, assuming you might want to use that data to test the gate-level synthesis netlist?



This page does not contain notes.

Module Review Solutions

1. What output system task would you use to capture the “steady-state” settled values of a collection of nets and variables whenever any one or more of them transition?
 - This is probably most easily done with the **\$monitor** system task.
2. Explain briefly the difference between **\$stop** and **\$finish**.
 - The **\$stop** system task interrupts simulation, from whence you or a script can continue it. The **\$finish** system task irrevocably terminates simulation.
3. What is the decimal value of the integer “testnum” upon execution of:
`if ($value$plusargs ("stimulus=test%d", testnum))`
if the simulator invocation includes the **+stimulus=test2** option?
 - The **\$value\$plusargs** system function interprets the subsequent “2” substring using the specified decimal formatter.
4. Which VCD system task would you use to dump value change data for the ports of an RTL ASIC you are developing, assuming you might want to use that data to test the gate-level synthesis netlist?
 - You would probably use **\$dumpports** instead of **\$dumpvars** because you need direction information and you do not need the values of internal signals.



This page does not contain notes.



Lab

Lab 21-1 Adding System Tasks and System Functions to a Beverage Dispenser Design

- In this lab, you add system tasks and system functions to support debug efforts.
- This lab provides a rudimentary test that leaves much room for improvement.

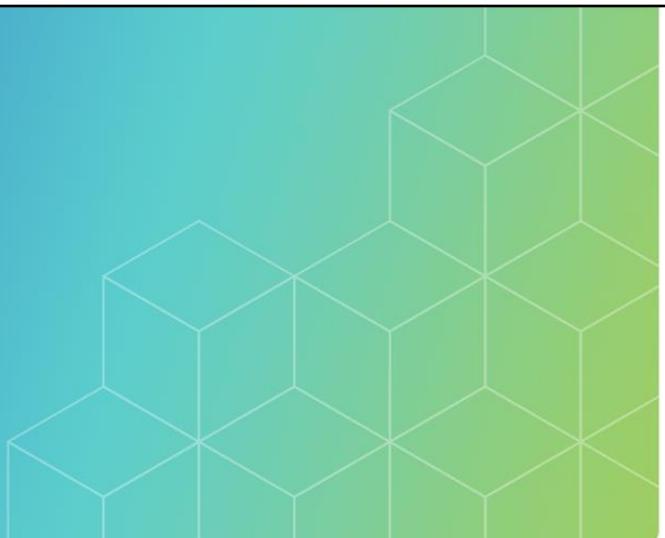
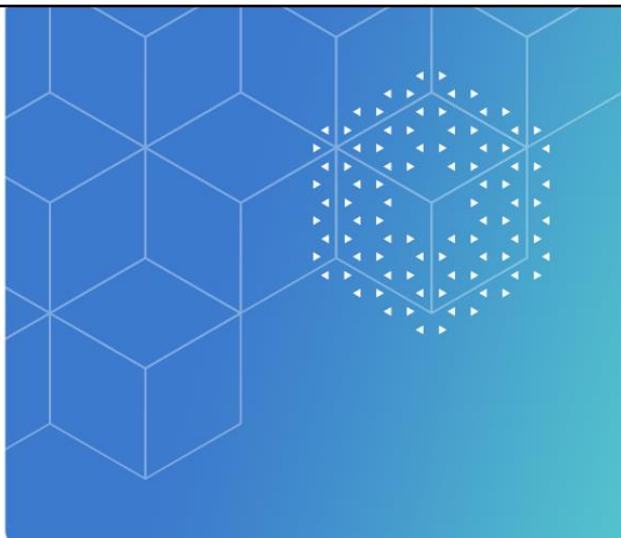
425 © Cadence Design Systems, Inc. All rights reserved.



Your objective for this lab is to add system tasks and system functions to support debug efforts.

The lab model is a beverage dispenser (drink machine). After resetting the machine, you load it with coins and cans. Then, as you insert nickels, dimes, and quarters, at 50 cents or above it dispenses a can and attempts to issue your change (if any). This drink machine has no coin return feature and inserted coins are not available later as change. If the EMPTY signal is true than you lose any coins you insert. If the USE_EXACT signal is true then you may find yourself short-changed. As the machine is currently defined, it does not attempt to issue lower-denomination coins if higher-denomination coins are not available.

The lab provides a rudimentary test that leaves much room for improvement. For this lab, you add system tasks and system functions to improve the test.



Module 22

Generating Test Stimulus

cadence®

This module examines some common coding styles and methods applicable to testbench generation.

Module Objective

In this module, you:

- Generate a test stimulus

Topics

- Simulation process review
- Organizing the testbench
- Using hierarchical names
- Generating clocks
- Generating stimulus
 - “In-line” stimulus
 - Stimulus using loops
 - Stimulus using tasks
 - Random Stimulus
 - Applying Stimulus from a File
 - Testing “corner” conditions
 - Testing protocol interactions
 - Capturing and playing back vectors

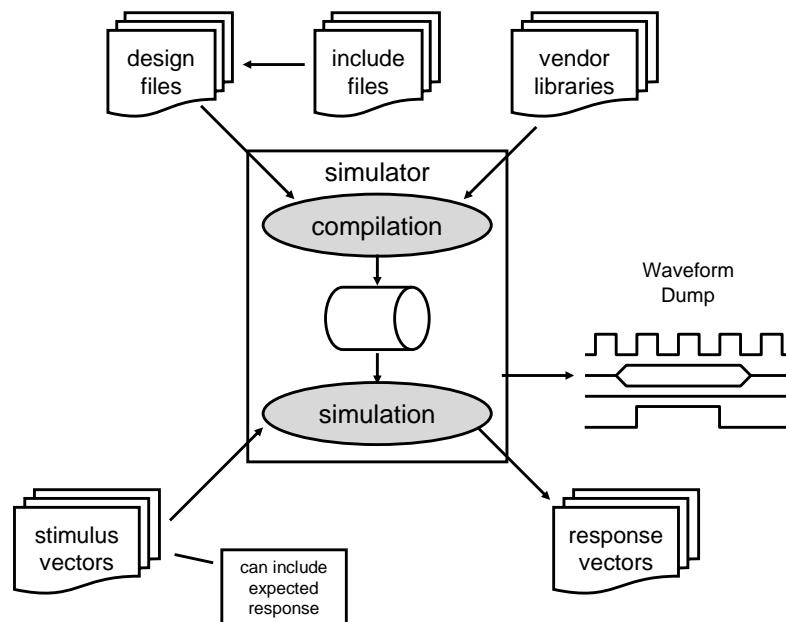
427 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to generate test stimulus.

To do that, you need to know how to create a testbench and some common ways to generate stimulus.

Simulation Inputs and Outputs Review



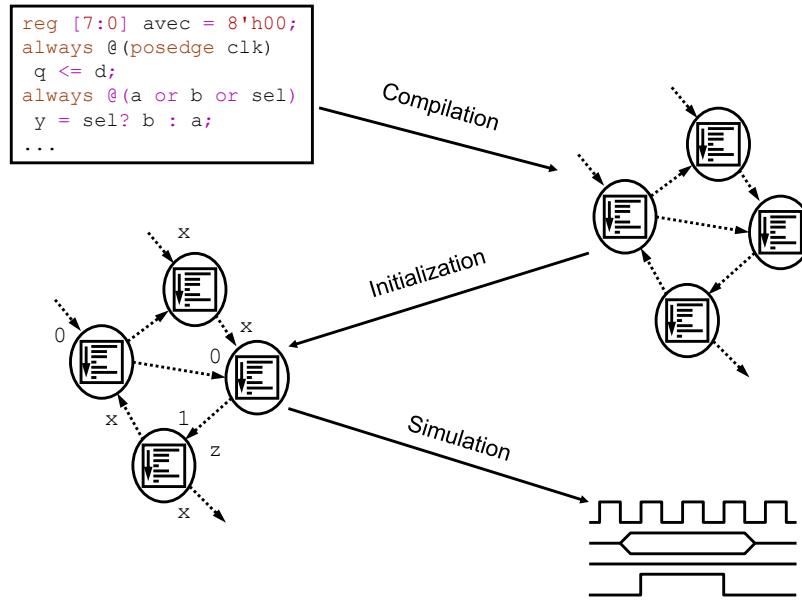
428 © Cadence Design Systems, Inc. All rights reserved.



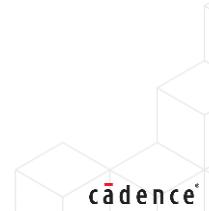
Input to the compiler is source text, which may reference third-party IP, and potentially other files to define the design configuration. Output from the compiler is a database representation of the design hierarchy.

Input to the simulator is of course the database representation of the design hierarchy, and can include stimulus vectors, may include expected response vectors, or perhaps some sort of pseudocode to drive an on-board stimulus generator. Output from the simulator is whatever diagnostic messages the user coded and value change data from whatever nets and variables the user probed.

Simulation Process Review



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



Simulation of the model takes the following steps:

- The 1st step is to compile the model:
 - The compiler parses the source text and constructs an internal database object for each design unit. A separate part of the compilation process, called elaboration, then links together the various parts of the design and builds a data structure representing the design hierarchy. For precompiled implementations, yet another process, called code generation, generates platform executable code for the design unit behaviors.
- The 2nd step is to initialize the elaborated data structure:
 - The initialization phase initializes most nets to the high impedance state and initializes most variables to the unknown state. It initializes trireg nets to the unknown state and real variables to 0.
- The 3rd step is the actual simulation, starting at time 0:
 - The simulation first propagates the design initialization events through the design hierarchy. The simulation then executes the statements in each initial and always construct up to the point at which a timing control blocks execution or the initial construct completes. Assignments in these statements can schedule events for the current and future times. As time advances, the simulator executes scheduled events, this execution often scheduling additional events. This process continues throughout the simulation.

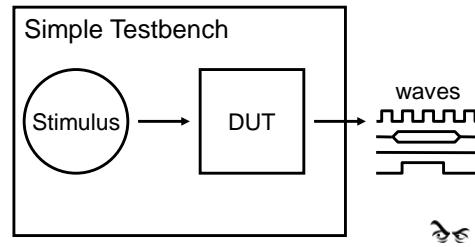
The simulator initializes most variables to the unknown state. Variables remain at the unknown state until an assignment statement assigns some other state. A variable remaining in the unknown state throughout the simulation is an indication of a failure to assign a known state to the variable, most likely a failure to reset the design.

Most nets initialize to the high impedance state. A net remaining at the high impedance state throughout the simulation is an indication that nothing drives the net.

Organizing the Testbench

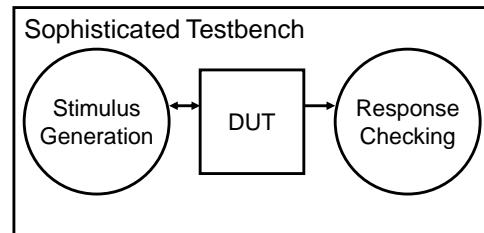
Simple testbench

- Just sends data to design
- Few procedures
- No interaction



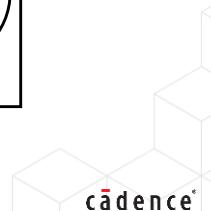
Sophisticated testbench

- Models system-level design environment
- Two-way communication
 - Stimulus generation
 - Response checking



We are most interested in sophisticated testbenches!

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



A simple testbench applies vectors to the design under test (DUT) and the user manually verifies the results.

Non-trivial projects use some form of “intelligent” or “smart” testbench that reacts to the DUT, e.g., for a bus protocol that requires some kind of handshake mechanism.

Such sophisticated testbenches are self-checking, i.e., they automatically verify the results.

Sophisticated testbenches typically instantiate modules for stimulus generation and response checking under a top-level “wrapper”, to more easily swap different tests and design configurations “in” and “out”.

Using Hierarchical Names

Every identifier has a unique hierarchical path name.

- Starts at a top module instance and downward traverses module instances, named blocks and tasks and functions.
- Each scope separated by “dot” (“.”).
- Hierarchical names can be used to access identifiers in other modules.
 - Also called Out Of Module References (OOMRs)
- There are restrictions:
 - Cannot access items of automatic tasks or functions
 - Cannot access items of unnamed blocks
- Hierarchical pathnames can be used anywhere but are primarily used in testbenches.

```
module fsm (
  input clk, rst, do,
  output done);
  reg [2:0] state;
  // assume fsm behaviors
endmodule

module fsm_tb;
  reg clk, rst, do; wire done;
  fsm u1 (clk, rst, do, done);
  initial begin
    $monitor($stime,,u1.state);
    // assume test behaviors
  end
endmodule
```

Path name can be:

- Downward relative to current scope.
- Downward starting at higher module or instance.



Every identifier has a unique hierarchical path name. The path name starts at a top module instance and traverses module instances, named blocks and tasks and functions, downward. Each such scope in the path name is separated from the next with the dot (.) character. You can make hierarchical references from anywhere in the design to almost anywhere else in the design. The exceptions are that you cannot make hierarchical references to items of automatic tasks and automatic functions, and you can make hierarchical references to items of an unnamed generate block only from within that generate block.

You can access items within the hierarchy rooted at the current scope by using relative path names. You access other items using absolute path names. The example testbench monitors the internal state of the FSM. This is a common use of hierarchical references – to gain testbench access to signals that are not available through module ports.

Hierarchical pathnames are primarily for testbench use, but can also be used in cross-hierarchy references, to access identifiers in one sub-module from another. As hierarchical pathnames cannot contain explicit “up-scope” references, cross-hierarchy references rely on the following resolution process:

- Hierarchical paths do NOT start at the top module instance.
- The paths are always relative to where they are used.
- If there is no match for the path downwards from the current scope, then the simulator automatically jumps up one scope and looks for a match from there.
- If there is still no match, then the simulator keeps jumping up until the top level module.
- If there is still no match from there, then the path is invalid and an elaboration error is produced.

If you use hierarchical names in testbenches only (as recommended) then pathname resolution is not a problem. The resolution is only an issue when you embed pathnames in hierarchical blocks, which we generally discourage (since it breaks reuse).

Generating Clocks

Behavioral Clocks



```
reg clk = 1;
always // simple
#(PERIOD/2) clk = ~clk;
```

Beware

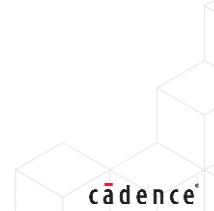
```
always @ (clk)
#5 clk = !clk; (doesn't oscillate)

always
clk <= #5 !clk; (zero delay loop)
```

```
reg clk = 1;
initial // delayed
 #(DELAY) forever
 #(PERIOD/2) clk = ~clk;
```

```
reg clk = 1;
initial // irregular
 #(DELAY) forever
begin
 #(( DUTY)*PERIOD) clk = 0;
 #((1-DUTY)*PERIOD) clk = 1;
end
```

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



The master clock generator is seldom part of the design and many people place it in the top-level wrapper, because it is also not really part of the stimulus generation or response checking operations. For a typical large and complex design, efficiency of the clock generator itself is not usually a concern.

In the “Beware” example section, the coding style to generate clock does not oscillate the clock and can lock the simulator.

Generating Stimulus

This section examines stimulus generation:

- The incremental approach
- Generating in-line stimulus
- Generating stimulus using loops
- Generating stimulus using tasks
- Generating random stimulus
- Applying Stimulus from a File
- Testing “corner” conditions
- Testing protocol interactions
- Vector capture and playback



This section examines stimulus generation. This introduction to test generation gives you a pretty good idea of what you can do with Verilog to test your design. The SystemVerilog language, which is a superset of Verilog, has much more powerful features directly targeting test generation. Industry advocates provide the Universal Verification Methodology (UVM), based on SystemVerilog, and plug-and-play verification IP components with which you can quickly construct a sophisticated verification environment.

The Incremental Approach

One of your most valuable test strategies is to the incremental approach:

- Build incrementally on what you know works:
 - First test for signs of life
 - Then, test interface and basic functions
 - Finally, test internal complex functionality
- For example, test of a storage device might:
 - Test reset (if that feature exists)
 - Test data lines individually or in combinations
 - Test address lines individually or in combination
 - Test functional “corners” i.e., back-to-back and overlapping operations
 - Test timing “corners” i.e., frequency, setup/hold, recovery/removal



A test sequence may in general incrementally contain elements from the following:

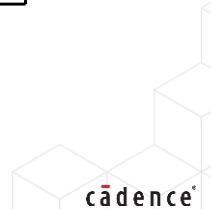
- Check-in tests, to verify the model before checking it into source control.
- Connectivity tests of the model, ASIC or other subunit instantiation.
- Functional tests of major widely-used functional features.
- Deterministic worst-case tests on data and control.
- Random worst-case tests on data and control.
- Tests of asynchronous interactions, including clock/data margining.
- Tests of inter-ASIC datapath and data processing algorithms.
- Hardware subsystems test using basic system-level operations and protocols.
- System boot.
- System multitasking and exception handling.
- Randomly sequenced full-system tests with heavy workload.

Generating In-Line Stimulus

The simplest approach is to use in-line stimulus generation:

- You only need to specify the variable value transitions.
- You can easily specify complex timing relationships.
- This testbench can become very large for complex tests.

```
module inline_tb;
  reg rd=0, wr=0;
  reg [4:0] addr;
  reg [7:0] dreg;
  wire [7:0] data=dreg;
  mem u1 (addr,data,rd,wr);
  initial
    begin
      #10 wr=1; addr=8'h00; dreg=8'h00;
      #10 wr=0;
      #10 rd=1; addr=8'h00; dreg=8'hzz;
      #10 rd=0;
      ...
    end
endmodule
```



The simplest way to generate a stimulus is with an in-line sequence of assignments.

This works reasonably well for an individual test with an irregularly-timed control protocol, as you specify exactly the transitions you need and exactly the time that you need each one.

This testbench can become very large for complex tests. When you see this type of stimulus, it is almost always machine-generated stimulus derived from dumped test vectors.

This example does one memory write operation followed by one memory read operation.

Generating Stimulus Using Loops

For repeated stimulus using a regularly-timed control protocol, you can “roll” your inline stimulus into loops. Advantages include:

- Compact easier to understand and debug stimulus generation.

```
module loop_tb;
  reg rd=0, wr=0;
  reg [4:0] addr;
  reg [7:0] dreg;
  wire [7:0] data=dreg;
  mem u1 (addr,data,rd,wr);
  initial
    begin: TEST
      integer i;
      for (i=0;i<=31;i=i+1)
        begin
          #10 wr=1; addr=i; dreg=i;
          #10 wr=0;
        end
      for (i=0;i<=31;i=i+1)
        begin
          #10 rd=1; addr=i; dreg=8'hzz;
          #10 rd=0;
        end
    end
endmodule
```

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



For repeated stimulus using a regularly-timed control protocol, you can “roll” your inline stimulus into loops. This produces stimulus generation code that is more compact and easier to understand and debug.

This example uses loops to first fill a memory with data and then to examine the data in memory.

Even very sophisticated testbenches use loops to generate stimulus, but they also take advantage of other techniques that we will examine next.

Generating Stimulus Using Tasks

Use a task if:

- You need the same sequence of control signals more than once.
- Or even if you just want to encapsulate them.

```
task write ( input [4:0] a,
             input [7:0] d );
begin
#10 wr=1; addr=a; dreg=d;
#10 wr=0;
end
endtask

task read ( input [4:0] a,
            output [7:0] d );
begin
#10 rd=1; addr=a; dreg=8'hZZ;
#10 rd=0; d=data;
end
endtask

initial begin: TEST
integer i;
for (i=0;i<=31;i=i+1)
// call tasks
end
```

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



Placing operations in tasks enhances readability, and if the operations are required in more than one place, it is essential for compact and concise code.

This illustration encapsulates the write and read operations in tasks, which procedural statements anywhere in the design can call upon.

Generating Random Stimulus

\$random [(seed)]

- Takes (and returns) a 32-bit seed
 - Seed the generator to produce repeatable sequences.
- Returns a random 32-bit signed integer
 - Verilog-1995: Algorithm is not standard and not portable.
 - Verilog-2001: Algorithm is standard and portable.

You can alter the random distribution:

- **\$dist_chi_square(seed,dgf)**
- **\$dist_erlang(seed,k,mean)**
- **\$dist_exponential(seed,mean)**
- **\$dist_normal(seed,mean,std)**
- **\$dist_poisson(seed,mean)**
- **\$dist_t(seed,dgf)**
- **\$dist_uniform(seed,start,end)**

```
module random_tb;
  reg rd=0, wr=0;
  reg [4:0] addr;
  reg [7:0] dreg,want,got;
  wire [7:0] data=dreg;
  mem u1 (addr,data,rd,wr);
  // assume task definitions
  initial begin: TEST
    integer i, seed;
    seed = 1;
    for (i=0;i<=31;i=i+1)
      write(i,$random(seed));
    seed = 1;
    for (i=0;i<=31;i=i+1) begin
      read(i,got);
      want = $random(seed);
      if (got !== want)
        // handle exception
    end
  end
endmodule
```

The operating environment for many devices includes some random inputs. A good test suite includes tests that mimic this randomness as much as practical. The random data might find errors in a design that more predictable data does not.

You can use the \$random system function to easily create random data patterns.

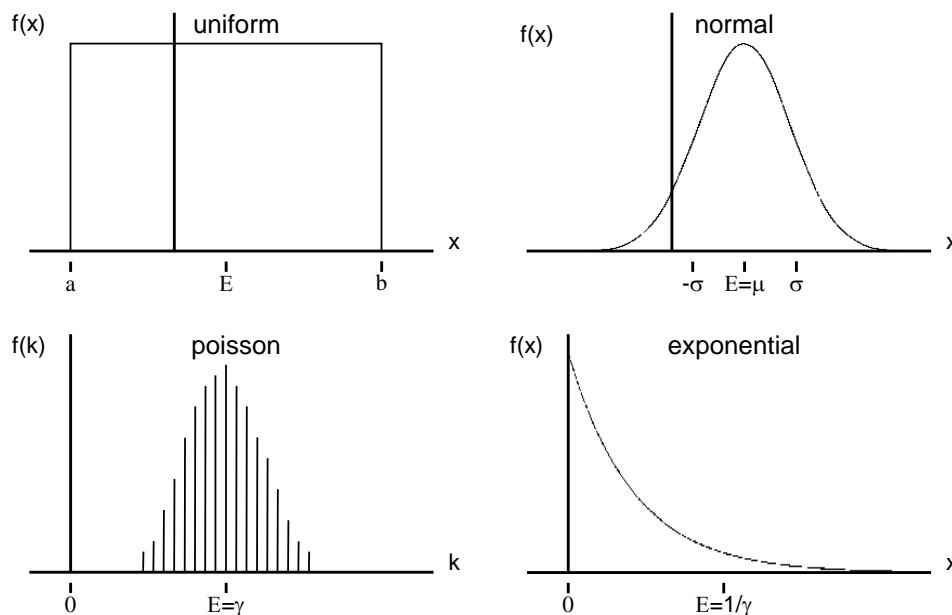
- The function takes and returns an integer seed, so the seed must be a variable and not just a value. You can provide an initial seed value to produce a repeatable sequence of pseudo-random numbers, and you can provide each function call instance with its own seed to make the function call instance independent of other function call instances.
- The function returns integer values. For random numbers wider than an integer, you can concatenate multiple \$random calls.

The \$random system function returns numbers uniformly distributed within the range of an integer. Realistically modeling the randomly distributed environment values usually requires a more complex distribution. Verilog provides a set of the most classic distributions that will probably be sufficient for your needs.

This example in a loop calls the write task and passes random data to write, then in a loop calls the read task to get the memory data and check it. Before each loop the example initializes the seed so that the random number sequences are the same for both loops.

Note: Simulators compliant with only the Verilog 1995 standard can produce different random number sequences. Simulators compliant with the Verilog 2001 standard will produce identical random number sequences.

Common Random Distributions



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



Verilog provides a selection of commonly used distribution functions. These are the four that you are most likely to use in a testbench:

- The uniform distribution equally distributes values between two limits. You use this distribution for a data item that is equally likely to be any value between the two limits.
- The normal distribution distributes values such that approximately 68% of the values are within the standard deviation of the mean. The normal distribution models biometric data of a large population.
- The poisson distribution is a discrete distribution that models the number of occurrences, referred to as “arrivals”, of an essentially infinite population of similar independent events per unit of distance, area, volume or time. You can use the poisson distribution to model the arrival of packets under steady-state conditions at a heavily-utilized router.
- The exponential distribution distributes values such that approximately 63% of the values are between 0 and the mean. Lower values are more likely than higher values. The exponential distribution models the interval between arrivals in a poisson process and can potentially model the size of packets arriving at a router, smaller packets being more frequent than larger packets.

Epsilon (E) is the *expected value*.

Mu (μ) is the *mean*.

Sigma (σ) is the *standard deviation*.

Lambda (γ) is the *arrival rate*

Testing Boundary Conditions

Test boundary conditions with worst-case tests that do the following:

- Test initial and terminal conditions.
- Focus on where the problems are.

Two Illustrations

```
reg [1:`WIDTH] memory [1:`SIZE];
...
for (i=1; i<=NUMBER; i=i+1)
    memory[i] = memory[i+1];
```

What is the architect's intent if NUMBER==0 ?

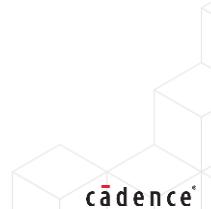
```
always @ (MEM_ACCESS)
  case ({READ, WRITE})
    1: memory[location] = ibus;
    2: obus = memory[location];
    3: ???
```

What is the architect's intent if NUMBER==`SIZE ?

```
endcase
```

What happens if READ and WRITE are both true?

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



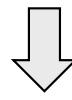
The illustrations indicate what is meant by a worst-case condition. Every design has its own set of worst-case conditions: A queue becomes full or empty; an arbiter receives simultaneous requests. The set of worst-case conditions can be very large, and at the beginning of the project, not well understood.

Example Worst-Case Test – FIFO Model

```

module fifo ( output [1:8] data_o, output full, empty,
              input  [1:8] data_i, input load, unload, rst, clk );
reg [1:8] busy, fifo[1:8];
integer i;
assign data_o = fifo[1], full = busy[8], empty = ~busy[1];
always @(posedge clk)
  if (rst)
    busy = 0;
  else
    case ({load, unload})
      2'b01: for (i=1; i<=8; i=i+1) // - UNLOAD ONLY -
        begin
          fifo[i] <= (i==8)? 8'bx : fifo[i+1];
          busy[i] <= (i==8)? 1'b0 : busy[i+1];
        end
      2'b10: for (i=8; i>=1; i=i-1) // - LOAD ONLY -
        begin
          fifo[i] <= busy[i]? fifo[i] : data_i;
          busy[i] <= (i==1)? 1'b1 : busy[i-1];
        end
      2'b11: for (i=1; i<=8; i=i+1) // - LOAD AND UNLOAD -
        begin
          fifo[i] <= (i==8)? data_i : busy[i+1]? fifo[i+1] : data_i ;
        end
    endcase
endmodule

```



1	2	3	4	5	6	7	8	b
2								u
3								s
4								y
5								b
6								i
7								t
8								s

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



The model treats the first and last locations differently than all other locations. Its treatment of the first and last locations is thus more susceptible to error, and the testbench must focus on testing model accuracy around those locations.

The model also becomes more complex in the area where it handles simultaneous read and write operations. This is an area that the model is more likely to inaccurate define, and the testbench is more likely to miss.

Example Worst-Case Test – FIFO Test Tasks

```

module test ( output reg [1:8] data_i, output reg load, unload, rst,
              input wire [1:8] data_o, input wire full, empty, clk );

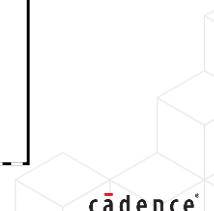
// - TASK TO CHECK FIFO DATA -
task checkit (input full_e, empty_e, input [1:8] data_e);
  if ( {full, empty} !== {full_e, empty_e}
    || !empty_e && data_o !== data_e ) begin
    $display("ERROR- FIFO TEST FAILED");
    $finish;
  end
endtask

// - TASK TO LOAD FIFO -
task loadit (input [1:8] data, input full_e, empty_e, input [1:8] data_e);
begin
  @(negedge clk) load = 1; data_i = data;
  @(negedge clk) load = 0; data_i = 8'hz;
  checkit (full_e, empty_e, data_e);
end
endtask

// - TASK TO UNLOAD FIFO -
task unloadit (input full_e, empty_e, input [1:8] data_e);
begin
  @(negedge clk) unload = 1;
  @(negedge clk) unload = 0;
  checkit (full_e, empty_e, data_e);
end
endtask

```

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



The testbench defines three tasks to off-load repetitive work from the main test block:

- The loadit task presents data to the FIFO and pulses the load signal.
- The unloadit task pulses the unload signal.
- The checkit task compares flags and data with expected values.

Example Worst-Case Test – FIFO Test Sequence

```

task resetit;
begin
    load = 0; unload = 0;
    @(negedge clk) rst = 1;
    @(negedge clk) rst = 0;
    checkit (0, 1, 8'hX);
end
endtask

initial begin: TST
    integer i;
    resetit;                                // Empty, not full
    loadit (1, 0, 0, 1);                     // Not empty, 1st data
    for (i=2; i<=7; i=i+1) loadit (i, 0, 0, 1); // Not yet full, still 1st data
    loadit (-1, 1, 0, 1);                   // Now full, still 1st data
    loadit (-1, 1, 0, 1);                   // Overflow ignored
    unloadit (0, 0, 2);                      // No longer full, 2nd data
    for (i=3; i<=8; i=i+1) unloadit (0, 0, i); // Not yet empty, most data
    unloadit (0, 1, 8'hX);                  // Now empty
    unloadit (0, 1, 8'hX);                  // Underflow ignored
    $display ("FIFO TEST PASSED");
    $finish;
end

endmodule

```

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



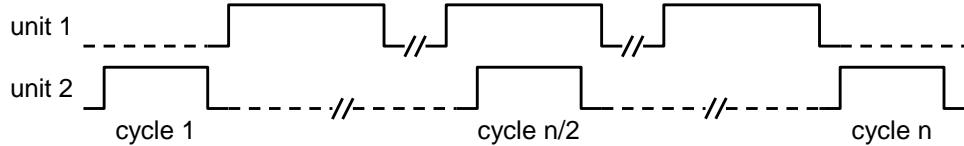
This sequence tests several FIFO boundary conditions. The test sequence verifies that the FIFO operates correctly, and also verifies that the FIFO does not operate incorrectly. The test sequence verifies:

- That the model sets the full flag only when the FIFO is full, and clears it only when the FIFO becomes not full.
- That the model sets the empty flag only when the FIFO is empty, and clears it only when the FIFO becomes not empty.
- That the model will not load data into a full FIFO, i.e., it does not change flags or data.
- That the model will not unload data from an empty FIFO, i.e., it does not change flags or data.

Testing Protocol Interactions

Test protocol interactions with timing offset (“sweep”) tests, that:

- Vary the relative arrival of events to test all relevant combinations.
- Test the interaction between two or more independent functions.
- Serve some of the same purposes as random test sequences.
 - But efficiently test only a preset set of possible interactions!



A sweep test sweeps one parameter across either a fixed point or another parameter, performing a regular search for combinations that the model may handle incorrectly.

A sweep test is ideal for test of a split-transaction bus, as it can verify that no phase of one transaction interferes with any phase of the other transaction.

Illustration of Sweep Test – Task Definitions

```

module top;
localparam PERIOD=20, LATENCY1=3;
reg clk, request1, request2, busy1, busy2, done1, done2;

always begin clk=1; #(PERIOD/2); clk=0; #(PERIOD/2); end

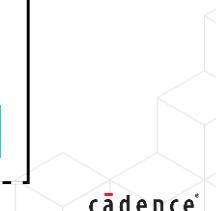
task start_device1; // MIMIC A DEVICE WITH SOME LATENCY
forever
begin
  wait (request1) @(posedge clk) busy1 = 1;
  repeat (LATENCY1) @(posedge clk); done1 = 1; busy1 = 0;
  @(posedge clk) done1 = 0;
end
endtask

task test_device1; // MAKE A REQUEST AND WAIT UNTIL DONE
begin
  @(posedge clk) request1 = 1;
  @(posedge clk) request1 = 0;
  wait (done1) @(posedge clk);
end
endtask

```

Tasks for device 2 are similar so not shown.

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



The testbench defines two tasks to support the sweep test:

- The start_device task mimics a device with some latency. It continually waits for a request, outputs a busy flag, waits the defined latency time, de-asserts the busy flag and pulses the done flag.
- The test_device task pulses the request flag, waits for the done flag, and exits.

Illustration of Sweep Test – Test Sequence

```

initial
  fork
    start_device1;
    start_device2;
  begin: TEST
    integer cycles1, cycles2, i;
    cycles1 = $time/PERIOD;           // Compute device 1 latency
    test_device1;
    cycles1 = $time/PERIOD - cycles1;
    cycles2 = $time/PERIOD;           // Compute device 2 latency
    test_device2;
    cycles2 = $time/PERIOD - cycles2;
    for (i=cycles1; i<=cycles2+2*cycles1; i=i+1) // Sweep 2 across 1
      fork
        begin repeat(cycles1+cycles2) @ (posedge clk); test_device1; end
        begin repeat(i) @ (posedge clk); test_device2; end
      join
    $finish;
  end
  join

endmodule

```

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

cadence®

The initial block forks simultaneous sequences to start both devices and to test them and their interaction with each other. The semi-intelligent test block determines the number of cycles needed by each device to complete its process, and then in a loop concurrently tests the devices. The loop starts device 1 at regular intervals and starts device 2 later for each iteration of the loop. The operation time of device 2 sweeps over the operation time of device 1, thereby testing any possible interference within this scenario.

Capturing and Playing Back Vectors

Why would you do vector capture and playback?

- To obtain and verify test vectors for off-line use, such as:
 - Fault detection analysis
 - ASIC foundry test machines

How would you do vector capture and playback?

- With system tasks and functions
 - Write textual data with a user-defined format
 - `$display`, `$write`, `$monitor`, `$strobe`
 - Write textual data with a predefined format
 - `$dump*`
 - Read textual data with a predefined format
 - `$readmem*`, `$getpattern`

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



You can use any of the file output system tasks to save signal values. You can then message the output as desired, read the vectors into a memory, and play them back with your own Verilog code.

The value change dump (VCD) facility is more efficient than the `$monitor` task, in performance and in storage space. With the VCD facility, you can save value changes of variables for any portion of the design hierarchy during any specified time interval. You can save results globally, without having to explicitly name all signals. Simulators produce a textual dump file that contains header information, variable definitions, and the value changes for all variables specified in the task calls.

The IEEE Standard 1364-2001 describes but does not require the `$getpattern` system task. Not all simulators support this system task.

Vector Capture and Playback Issues

You must resolve these vector capture and playback issues:

- Timing:
 - For asynchronous circuits, capture and play back any value change.
 - For synchronous circuits, capture and play back the last value change during each clock cycle:
 - Determine the capture and playback time within the clock cycle.
 - Ensure signal stability when captured, what about negative setup/hold?
- Bidirectional signals:
 - Capture a control signal to determine playback direction.
 - Refer to timing considerations presented above.
- Vector width and depth:
 - Exercising internal functions may require many signal transitions.
 - Vector capture may require many large pattern files.



You must carefully and methodically develop a customized vector capture and playback mechanism. Think through, step by step, what you want to do and how to write Verilog to implement your plan.

You can use the VCD facility to save only the signal state changes, thus conserving storage space. If you have a fully synchronous subdesign, you can ignore all changes except the last one before the active clock edge on any non-clock signal, but you must ensure that the changes meet setup and hold requirements when played back.

You must capture control signals for all bidirectional signals so your playback mechanism can drive stimulus and check response appropriately.

Because test at the system level typically requires many clocks to do something useful, many vectors captured at the subdesign level may contain only clock changes. You can prune these vectors if you are either absolutely sure that they do not contribute to subdesign state changes, or you regenerate those clocks during pattern playback.

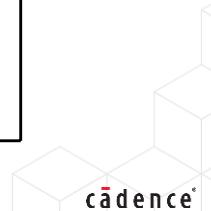
You should expect captured vector storage space to be an issue.

Capturing Vectors to a File

- You can capture the stimulus and response at the pins of your DUT.
- You can play these vectors back against the DUT using an efficient stripped-down test environment.
- You can provide these test vectors to the device manufacturer.
- You must resolve these issues:
 - Timing of the capture
 - Timing of the playback
 - Direction information

```
module capture_tb;
localparam PERIOD=10;
reg [7:0] stim_reg;
wire [7:0] resp_net;
DUT u1 (stim_reg, resp_net);
initial begin: TEST
  // stimulus/response behaviors
end
initial begin: CAPTURE
  integer stimfile, respfile;
  stimfile = $fopen("stim.txt");
  respfile = $fopen("resp.txt");
  if (!stimfile || !respfile)
    // handle exception, else...
  forever #(PERIOD) begin
    $fstrobeb(stimfile,stim_reg);
    $fstrobeb(respfile,resp_net);
  end
end
endmodule
```

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



You can capture the stimulus and response at the pins of your DUT, and later play these vectors back against the DUT using an efficient stripped-down test environment. You can also provide these test vectors to the device manufacturer.

It is very important to time the capture to record the steady-state value of the pins.

You may need to also record design-dependent direction information for bidirectional pins.

This example opens stimulus and response files and simply records hopefully steady-state pin data on a periodic basis.

Applying Vectors from a File

You can preload stimulus values into an array and then in a loop apply the array values.

Advantages include:

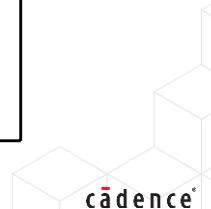
- Potential for run-time selection of tests from a test suite.

The file can contain:

- Manually or automatically generated test vectors.
- Captured test vectors.
- “Pseudocode” that your testbench executes to generate the stimulus.

```
module playback_tb;
localparam PERIOD=10;
reg [7:0] stim_reg;
wire [7:0] resp_net;
reg [7:0] stim_mem[0:whatever];
reg [7:0] resp_mem[0:whatever];
DUT u1 (stim_reg, resp_net);
initial begin: TEST
    integer i;
    $readmemb("stim.txt",stim_mem);
    $readmemb("resp.txt",resp_mem);
    for (i=0;i<=whatever;i=i+1)
        begin
            stim_reg = stim_mem[i];
            #(PERIOD);
            if (resp_net!==resp_mem[i])
                // handle exception here ...
        end
    end
endmodule
```

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



You can facilitate run-time selection of the test suite by applying stimulus from an array. You can place a forever loop in the testbench that loads and applies stimulus, and then waits for the user before continuing to the next loop iteration. The user can meanwhile change the file contents or use other implementation-specific features to replace the file.

As an alternative to directly applying file stimulus, you can incorporate a machine in your testbench to execute the pseudocode file to generate the stimulus.

This example loads stimulus vectors and expected response vectors from files, and then in a loop applies the stimulus and checks the response.

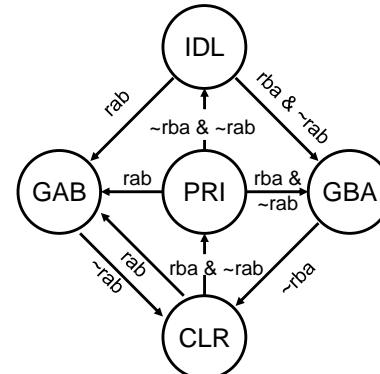
Example Vector Capture and Playback: Model

```

module arbx ( inout [3:0] busa, busb,
              output gab, gba,
              input rab, rba, rst,
              input clk );
localparam IDL=0, GAB=1, CLR=2, PRI=3, GBA=4;
reg [2:0] state;
assign busb=gab?busa:4'hz, busa=gba?busb:4'hz; // ASSIGN INOUT BUS
assign gab=(state==GAB), gba=(state==GBA); // ASSIGN OUTPUT GRANT

always @ (posedge clk)
if (rst)
  state <= IDL;
else
  case (state)
    IDL: state <= rab?GAB:rba?GBA:IDL;
    GAB: state <= rab?GAB:CLR;
    CLR: state <= rab?GAB:rba?PRI:CLR;
    PRI: state <= rab?GAB:rba?GBA:IDL;
    GBA: state <= rba?GBA:CLR;
    default state <= IDL;
  endcase
endmodule

```



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

cadence®

This model is a combined arbiter and 4-bit bus transceiver. Direction A to B has absolute priority, and if direction B to A has just relinquished the transceiver and requests to retake it, it must wait one additional clock to give direction A to B the opportunity to use the bus first.

Example Vector Capture and Playback: Testbench

```
`timescale 1 ns / 1 ns

module test
#( parameter PERIOD=10 )
( inout wire [3:0] busa, busb,
  input wire gab, gba,
  output reg rab, rba, rst,
  input wire clk );

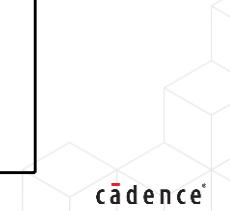
reg [3:0] busa_r, busb_r;
assign busa = busa_r, busb = busb_r;

`ifdef CAPTURE
  `include "capture.inc"
`endif

`ifdef PLAYBACK
  `include "playback.inc"
`endif

endmodule
```

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



The testbench declares output variables for the busa and busb inout ports and connects them to the ports. The Verilog language does not permit you to directly declare bidirectional ports to be variables.

The testbench directly declares output variables for the rab and rba and rst output ports.

Example Vector Capture and Playback: Capture

```

initial begin: Capture
    integer mcd;
    time t;
    // PLAYBACK REPLICATES RESET SO DO NOT CAPTURE THIS
    rab=0; rba=0; busa_r=4'hz; busb_r=4'hz; rst=1;
    @(negedge clk) rst = 0;
    @(negedge clk);
    // START MONITOR AFTER RESET
    t = $time;
    mcd = $fopen ("patterns.inc"); # 0{rab,gab_r,rba,gba_r,busa_r,busb_r}<=12'b0000zzzzzz;
    $timeformat (-9, 0, "", 3);
    $fmonitorb ( mcd, "#%", ($time-t),
        "{rab,gab_r,rba,gba_r,busa_r,busb_r}<=12'b",
        rab,gab_rba,gba,busa,busb, ";" );
    // ACTUAL TEST GOES HERE
    `include "stimulus.inc" // @(negedge clk) {rab, rba, busa_r, busb_r} = 10'h2zz; wait(gab);
    @(negedge clk);
    // CLEAN UP AND GO HOME
    $monitoroff;
    $fdisplay ( mcd, "#%t $finish;", ($time-t) );
    $finish;
end

```

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



The capture code uses the \$monitor system task to save a pattern vector at the end of each time slice in which any monitored signal changes. The capture code formats the \$monitor system task to save signal changes in syntactically correct Verilog so that the simulator can easily play them back. This inefficient method to capture and play back signal changes is impractical for simulations of a significant size. You would probably want to save signal changes to a waveform database and use the Programming Language Interface (PLI) to read and reapply them as you go.

Example contents of stimulus.inc file:

```

@(negedge clk) {rab, rba, busa_r, busb_r} = 10'h2zz; wait(gab)
@(negedge clk) {rab, rba, busa_r, busb_r} = 10'h25z;
@(negedge clk) {rab, rba, busa_r, busb_r} = 10'h2az;
@(negedge clk) {rab, rba, busa_r, busb_r} = 10'h1zz; wait(gba)
@(negedge clk) {rab, rba, busa_r, busb_r} = 10'h1z3;
@(negedge clk) {rab, rba, busa_r, busb_r} = 10'h1zc;
@(negedge clk) {rab, rba, busa_r, busb_r} = 10'h0zz; wait(!gba)

```

Module Summary

You should now be able to generate test stimulus.

This module discussed:

- Using hierarchical names
- Generating clocks
- Generating “in-line” stimulus
- Generating stimulus using loops
- Generating stimulus using tasks
- Generating random stimulus
- Testing “corner” conditions
- Testing protocol interactions
- Applying stimulus from a file



This module examined some common coding styles and methods applicable to testbench generation.

Module Review

1. The Verilog-1995 standard permitted use of hierarchical names to reference any named objects. What Verilog-2001 constructs are incompatible with access by hierarchical names?
2. Even if you enable a task from only one place in your testbench, why might you *still* want to keep that set of statements in a task rather than “in-line” it at the point of the call?
3. Which random distribution is that infamous “bell curve” your teachers used to map numerical test scores into letter grades?
4. What are some of the issues to consider when capturing test vectors to a file and then playing them back?

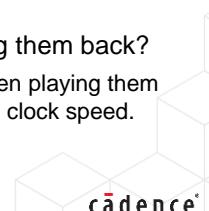


This page does not contain notes.

Module Review Solutions

1. The Verilog-1995 standard permitted use of hierarchical names to reference any named objects. What Verilog-2001 constructs are incompatible with access by hierarchical names?
 - Hierarchical names cannot access items of automatic tasks or automatic functions because those items exist only while the task or function is active. Although unnamed generate blocks have a machine-generated implementation-dependent name for display purposes, source code is not permitted to know what that name is so cannot access items within unnamed generate blocks.
2. Even if you enable a task from only one place in your testbench, why might you *still* want to keep that set of statements in a task rather than “in-line” it at the point of the call?
 - For the same reason that you use an HDL instead of schematic capture, representing operations as abstract tasks makes it easier for you to understand the overall problem.
3. Which random distribution is that infamous “bell curve” your teachers used to map numerical test scores into letter grades?
 - The “bell curve” is a normal distribution.
4. What are some of the issues to consider when capturing test vectors to a file and then playing them back?
 - Issues to consider must include the stability of the captured signals and the setup and hold times when playing them back, how to record and apply the directionality of bidirectional signals, and perhaps the difference in clock speed.

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



This page does not contain notes.



Lab

Lab 22-1 Verifying a Serial Interface Receiver

- For this lab, you will generate a simple test of the serial interface receiver.
- You will generate random data for four packets. You statically construct a 256-bit stream containing four valid packets that are surrounded by values that are something other than the header value.
- You will reset the receiver and present this stream to the receiver inputs. While adhering to the output protocol, you retrieve the receiver output data and verify that all packets are received and correct.

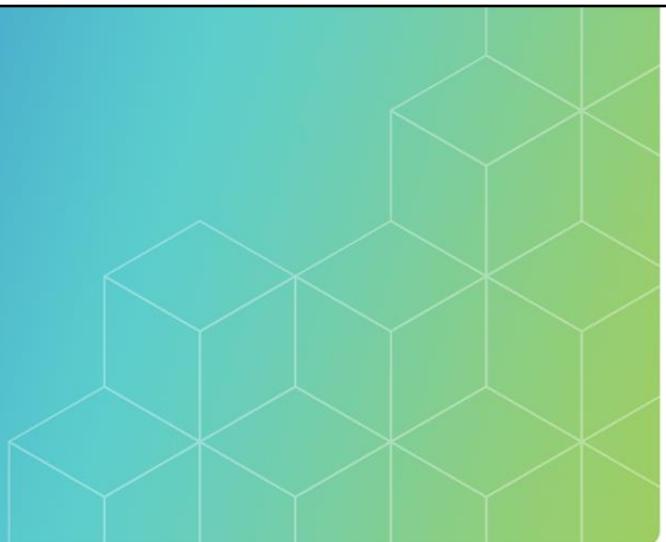
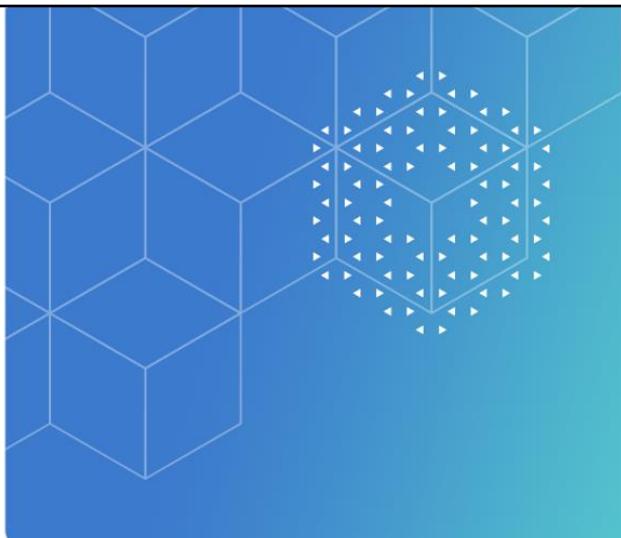


Your objective for this lab is to test a serial interface receiver.

The lab model is a serial interface receiver. The receiver interfaces between a serial input stream and a parallel output stream. A 24-bit packet represents the data. The packet consists of an 8-bit header with a value of 0xa5 followed by two 8-bit data bytes. The receiver initially shifts input data left into the header register until it detects the header value. For this reason, the header register must be initialized to a value opposite the leftmost header value bit. Upon detecting the packet header, the receiver clears the header register and shifts input data left into the body register while counting to 16. Upon the 16th count, the receiver moves the data to the output buffer, clears the counter, asserts the ready output, and again shifts input data left into the header register. The receiver can thus move a packet every 24 clocks.

The test environment reads the leftmost buffer byte while acknowledging the ready signal. The receiver shifts the output buffer left by one byte upon each such acknowledge. The receiver counts acknowledges and drops the ready signal upon the last acknowledge. The test environment can delay the last acknowledge up to and including the clock that loads the next data into the output buffer. Failure of the test environment to retrieve data within that interval results in lost data.

For this lab, you generate a simple test of the serial interface receiver. You generate random data for four packets. You statically construct a 256-bit stream containing four valid packets that are surrounded by values that are something other than the header value. You reset the receiver and present this stream to the receiver inputs. While adhering to the output protocol, you retrieve the receiver output data and verify that all packets are received and correct.



Module 23

Developing a Testbench

cadence®

This module presents design verification concepts, reinforced with small working examples and illustrations.

Module Objectives

In this module, you:

- Optimize the Testbench Development Effort
- Develop Reusable Testbenches

Topics

- Test Philosophy
- Test Configuration

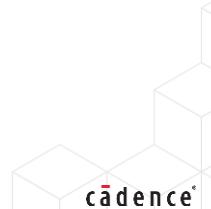


- Your first objective is to optimize the testbench development effort. The first part of this module supports that objective by exploring the philosophy of system-level test, including the test plan, completion criteria, and design data management.
- Your second objective is to develop reusable testbenches. The second part of this module supports that objective by presenting self-checking tests and various ways to configure the test environment.

Test Philosophy

This section discusses:

- The design verification challenge
- The design verification plan
- The design verification goal
- System-level test
- Completion criteria:
 - Coverage metrics
- Managing the design space
- Testing design regression
- Self-checking test



This section addresses several aspects of test philosophy.

The Design Verification Challenge

The design verification challenge is a resource issue.

Problem

- Technological advances drive system complexity.
- Technology alone cannot overcome the complexity problem.
- Verification is majority of total development effort... *and growing!*
- Integrity means design/implementation team is *NOT* verification team!
 - Although... an early implementer can be a verifier later
 - But must adhere to verification team's process and standards!

Solution

- The Design Verification Plan:
 - Know what you need and when you need it

We need a plan...

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



Design verification ensures that the design continues to comply with all system requirements throughout all levels of abstraction, from the system behavioral level to the block implementation level, and continuing through integration of the physical units.

You will need to integrate, structure, and coordinate your verification methods to minimize the probability that the development team misses errors, especially complex system-level design errors, until much later in the design process. Such large-scale redesign is much more costly than iteration confined to a single design phase.

The industry currently uses event-driven simulation as the most common design verification technology at the behavioral and the functional level. This technology is practical, well understood, and mature, but unable to keep up on its own with Moore's law: that design complexity grows at an exponential rate, approximately doubling every 12 to 18 months.

Faster technologies are available, for example, hardware acceleration, cycle-based simulation and emulation, but are less applicable at the higher levels of abstraction early in the design process.

The key to the design verification challenge is to not rely on technological advances alone, but to carefully plan and execute the overall design and design verification strategy.

The Design Verification Plan

The design verification plan is an important job.

There are two approaches:

- Just let it happen (it won't) – oops!
- Make it happen!
 - Test in accordance with the functional specification.
 - Paraphrase the spec; spec says "it does this", test that "it does this" ...
 - Remember the spec can be wrong too!
 - Schedule test delivery to coincide with module and model availability.
 - For example: stub (shell) / Bus-Functional Model (BFM) / fully functional model / hardware model.
 - Model availability is often an issue!

As time goes by...

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



A viable design verification plan must include the following elements:

- 1st – Use of the appropriate technology for each phase of the design process
 - This requires understanding the available technologies, the needs of the project, and the verification strategy. Depending upon the design phase, appropriate technologies can be anything from C-based stochastic analysis to a hardware prototype.
- 2nd – Creation of appropriate, easily-selectable simulation configurations
 - Configurations must be designed to focus verification efforts on key partitions of the design, while filling in the rest of the system context with behavioral models. The number and design of configurations also heavily depends upon the resources available, both of hardware and of personnel.
- 3rd – Development of formal, well-structured testbench templates
 - The use of testbench templates facilitates test generation, maintenance, and modification. Test procedures, test data, and test timing should be separated, and good programming practices, such as parameterization, should be used.
- 4th – Generation of appropriate tests in a timely manner

The Design Verification Goal

The design verification goal is progressive testing.

Design verification is a continuous parallel effort.

- At the behavioral level:
 - Verify implementation of architectural decisions
 - Test sub-block interface and interaction
 - Test timing to within cycle accuracy at sub-block interfaces
- At the functional level:
 - Verify implementation of behavioral models
 - Test complete design functionality
 - Test timing to within cycle accuracy at all levels of abstraction
- At the structural level:
 - Verify synthesis of functional implementation
 - Test subsystem interconnection and initialization
 - Test timing to high accuracy within structural level

We need a system...

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

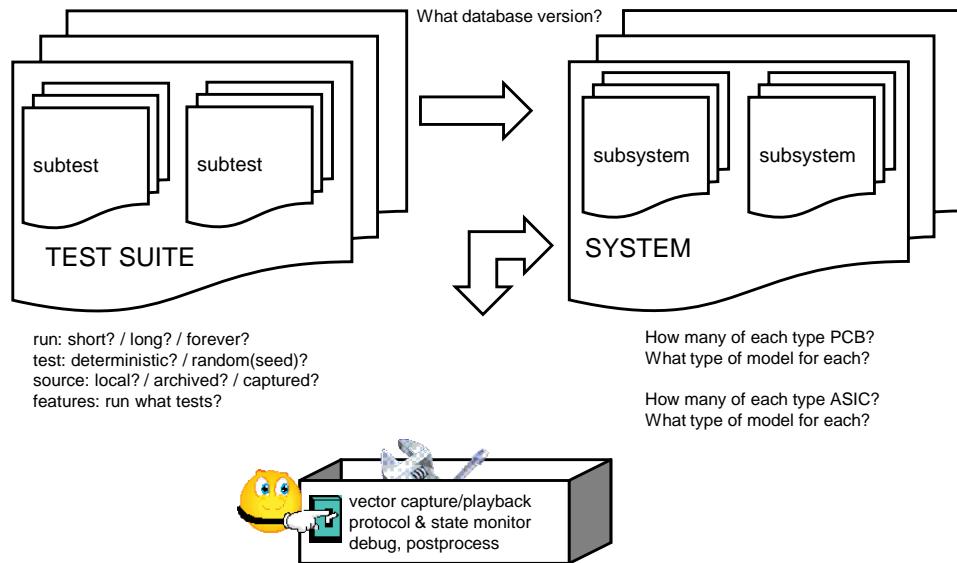


All design verification strategies have these two overall goals:

- The first goal for high-level system design verification is to validate the design intent before implementation is begun. This involves:
 - Verification of system functionality – that algorithm selection is correct.
 - Verification that the system design meets performance and cost requirements.
 - Verification that subsystem partitioning and packaging, into hardware and software, for example, or between physical components, is practical and effective.
- The second goal is to verify the design implementation. This involves:
 - Verification that all components, whether newly designed, reused, or standard, off-the-shelf, work as expected within the context of the new system-level design.
 - Verification of all internal functionality of newly designed subsystems.
 - Verification of hardware and software interaction.

System-Level Test and Configurability

System-level test is about configurability.



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



System-level test is about configurability. A simulation configuration groups specific models of specific subsystems together for a defined verification purpose. The simulation configuration may also constrain the type, detail, and length of tests, or you may wish to postpone such test selection until runtime.

You should include in your testbench:

- Tools for vector capture and playback.
- Monitors of bus protocol, arbiter fairness and state machine transitions.
- Debug tools.
- Postprocessors for the ASIC foundry.

System-Level Test and Team Communication

System-level test is about communication!

There are two approaches:

- System-level test:
 - Developed in parallel with hardware (tested “as a whole” from day 1)
 - Used (with reconfiguration) by everybody
 - Demands up-front planning and continued maintenance
 - Facilitates communication between designers and verifiers
 - Validates the verifier’s interpretation of the specifications
 - Enforces a common test framework, strategy, and style
 - Conserves resources due to less duplication of effort
- Design-unit test:
 - Owned by the designer/implementer
 - “Ad hoc”, quick and dirty, typically not communicated or maintained
 - Can be useful for preliminary “sanity” check

Ok, but when am I done?



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

System-level test is about communication. The vast majority of the design errors that persist beyond a single phase of the design cycle do so, not because the information required to prevent them is not known to the team at large, but because the information is not known to the persons responsible for the piece of the design containing the error. System-level test is a formidable ally to the design team in their constant war against miscommunication or lack of communication.

Completion Criteria

There are two approaches:

- Test until time runs out (it will) – oops!
- Determine priorities and metrics before time runs out – follow them!
 - What tests [must / should / could] be implemented?
 - Manage time / goals / expectations.
 - Manage “feature creep”.
 - How do you know when you are done?
 - How good is “good enough”?

Count the ways...

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



An important part of the design verifier’s job is to provide meaningful project status information to the management team. The management team needs a reasonably firm test plan, against which they can analyze the project status trend, to do project-related planning. Although the test plan should retain some flexibility throughout the project, the design verifier should also make clear to the management team the time and resources trade-offs associated with any modifications.

Coverage Metrics

The most widely used completion metric is coverage.

(What percent of your verification goal is complete?)

- Code coverage – Coverage of the design code
 - Anything the simulation tools can automatically instrument and measure
 - Code blocks, expression terms, net toggles, FSM states and transitions
- Functional coverage – Coverage of the design *functionality*
 - User defines the coverage items
 - Data oriented – Which specified values, ranges of values, transitions between values occurred
 - Control oriented – Which specified sequences of control signals occurred

Verilog has no special constructs supporting coverage!



The most widely used completion metric is coverage. Coverage is in two forms:

- Code coverage is a measure of how completely the testbench exercises the design code. Code coverage includes any code items that simulation tools can automatically instrument and measure. This might include for example code blocks, expression terms, net toggles, and FSM states and transitions. Code coverage does not say anything about design functionality.
- Functional coverage is a measure of how completely the testbench exercises the design functionality. You the user have to specify this functionality, so the coverage metric is only as valid as the test specification. Functional coverage is further divided into two forms:
 - Data-oriented functional coverage is a measure of which specified values, ranges of values, and transitions between values occurred.
 - Control-oriented functional coverage is a measure of which specified sequences of control signals occurred.

Verilog has no special constructs supporting coverage. For code coverage that the tool automatically instruments this is not an issue. For functional coverage you have to turn to high level verification languages, such as SystemVerilog.

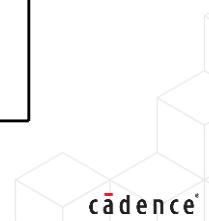
Example Code Coverage: Arbiter Model and Test

Code coverage provides an estimate of the upper limit of testbench quality.

```
module arb (
    output reg GNTA, GNTB,
    input wire REQA, REQB, RST, CLK
);

    always @ (posedge CLK)
        if (RST)
            {GNTA, GNTB} <= 0;
        else // UNIT A HAS PRIORITY
            case ({GNTA, GNTB})
                0: {GNTA, GNTB} <= {REQA, REQB && !REQA};
                1: {GNTA, GNTB} <= {REQA && !REQB, REQB};
                2: {GNTA, GNTB} <= {REQA, REQB && !REQA};
            endcase
endmodule
```

```
module arb_test;
    wire GNTA, GNTB;
    reg REQA, REQB, RST, CLK;
    arb u1 (GNTA, GNTB, REQA, REQB, RST, CLK);
    initial CLK=0; always #5 CLK=~CLK;
    initial begin
        @ (negedge CLK) REQA=0; REQB=0; RST=0;
        @ (negedge CLK) RST=1; // RESET
        @ (negedge CLK) RST=0;
        @ (negedge CLK) REQA=1; // REQ A
        @ (negedge CLK) REQA=0;
        @ (negedge CLK) REQB=1; // REQ B
        @ (negedge CLK) REQB=0;
        @ (negedge CLK) REQA=1; REQB=1;
        @ (negedge CLK) REQA=0;
        @ (negedge CLK) REQB=0;
        @ (negedge CLK) $finish;
    end
endmodule
```



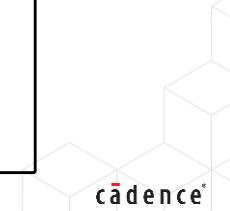
Code coverage provides an estimate of the upper limit of testbench quality. Block coverage, for example, indicates whether the covered code has been executed, and says nothing about how correct the code is. Code coverage is nonetheless very useful, for with fairly simple technology it can pinpoint an area of the design that the testbench has completely missed.

This arbiter simply responds with a grant to every request. The grant persists until the request drops. The arbiter gives priority to unit A if both units simultaneously make a request.

This short testbench resets the arbiter and then separately tests that it grants a unit A request and then a unit B request, and that upon simultaneous arrival of both requests, it grants the unit A request first.

Example Code Coverage – Block Coverage Report

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



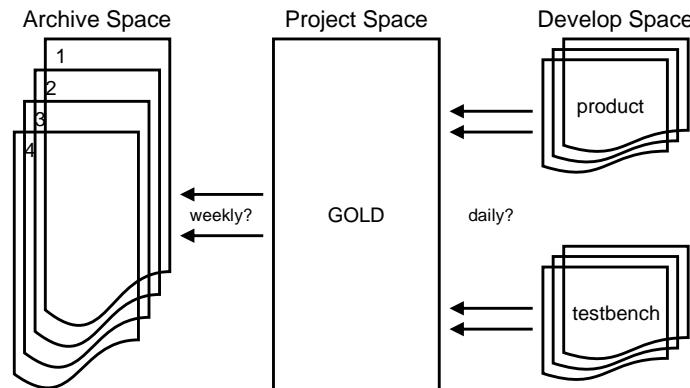
This is annotated block coverage reported by the Cadence Xcelium™ Comprehensive Coverage analysis tool. The author edited it slightly to reduce long file names and remove unneeded white space to fit it on the page. A code block is a block of code that executes together – either none of it executes or all of it executes. The tool measures block coverage instead of line or statement coverage because Verilog is a free-form language, so line counts and statement counts are meaningless. You can see that the testbench caused each block to execute.

You can easily understand the block coverage reports, so you should start your code coverage analysis with block coverage. Upon achieving 100% block coverage, as this testbench did, you should then analyze the expression coverage. Expression coverage would reveal that this testbench did not test that a grant that is asserted will stay asserted as long as a request stays asserted.

Managing Design Space

Problem: Multiple parallel design efforts negatively impact each other.

Solution: Isolate design efforts by dividing and controlling design space.



Use source control to help manage your design space.

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



Partition your design space into the following areas:

- 1st – The development space:
 - Copy elements to this private area for your individual design efforts;
 - Use the “golden” project space for all other design elements; and
 - Replace project space design elements only after thoroughly verifying them.
- 2nd – The project space:
 - Tag the in-progress elements to prevent duplication of design efforts;
 - Submit only thoroughly verified design elements; and
 - Continuously run regression tests.
- 3rd – The archive space:
 - Back up the project space here on a regular basis or upon completion of major features;
 - Continuously run the full test suite, incorporating random test sequences; and
 - Use this space for investigatory efforts such as timing analysis, fault analysis and so on...

Testing Design Regression

Characteristics

- Fully automated, usually chron-launched, shell scripts:
 - Control the what / where / when / how of system-level testing
 - Use good programming practices. Scripts are programs too!
 - Launch multiple simulations on multiple processors
 - Email a report with configurations, versions, errors, warnings

Concerns

- Implementers under schedule pressure check in minimally tested work.
- Implementers simultaneously check in tested, but incompatible, work.
- The regression cycle quickly exceeds the design modification cycle.
- Project management loses control of regression data. (What was *really* tested last week?)
- Regression testing *requires* testbenches to be self-checking!



You will want to continually in the background verify that addition of new features or modification of previous features does not cause test failure in the project space. This effort is termed “regression testing” or more confusingly just “regressions”. What this effort does is to detect the “regression” of the project to a less mature state.

Regression tests are usually fully automated chron-launched shell scripts that run multiple simulations on multiple processors, logging the tested configurations and versions, and emailing notification of any anomalies.

Regression testing of course requires that your testbenches be self-checking!

Self-Checking Test

There are two approaches:

- Visual compare (eyeballing) – after (or during!) simulation:
 - Ideal for lossy video processing (is it pretty enough?)
 - Very practical for simple functional tests of short duration
 - Generally not compatible with system-level test
- Automated compare (required for system-level test)
 - During simulation – Unattended run-time error handling
 - After simulation – Comparison of vector dump files may be sufficient

Examples

- Read-after-write or Cyclic Redundancy Character (CRC) checks
- Monitors of: Bus Protocols, Finite State Machine (FSM) states, timeouts
- Timing checks (as a form of assertion testing)
- Dump and compare of response vectors or memory contents

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



Testbench self-checking is in two forms:

- The most common form is to check data and perhaps even protocols on the fly in order to provide immediate diagnostic information pointing to the area and time of the problem.
- A second less-common form is to automate a post-processing comparison of dumped test vectors to those of a known-good design. This form may be useful if you want a very detailed check of functionality that is very unlikely to have changed.

Your testbench can be self-checking in many ways:

- You can read back data that you write to storage devices.
- You can compare check characters calculated on the transmission side and the reception side.
- You can code monitors for bus protocols, FSM state paths, and timeouts.
- You can imaginatively set up timing checks to check the timing relationship between two edges of the same or different signals.

Example Self-Checking Test

```

`timescale 1 ns / 1 ns
module mux_test;
localparam PERIOD=10;
localparam EXIT_ON_ERROR=1;
localparam STOP_ON_ERROR=0;
localparam MUX_TEST_WAIT=PERIOD-1;
wire y; reg a, b, s;
// NEW MODEL BEING VERIFIED
MUX2_1 mux ( y, s, b, a );
// "GOLDEN" MODEL THAT WE KNOW WORKS
wire results=y, expects=s?b:a;
initial
begin
  a=0; b=1; s=0; #(PERIOD);
  a=1; b=0; s=0; #(PERIOD);
  a=0; b=1; s=1; #(PERIOD);
  a=1; b=0; s=1; #(PERIOD);
  // A MISCOMPARE WON'T GET THIS FAR
  $display ("TEST PASSED");
  $finish;
end

```

```

// FOR EACH MUX TEST EVENT
always @*
// WAIT A PREDEFINED SETTLING TIME
# (MUX_TEST_WAIT)
// VERIFY THE RESULTS
if (results !== expects)
begin
  $display($time, "ERROR MESSAGE");
  if (EXIT_ON_ERROR) $finish;
  if (STOP_ON_ERROR) $stop;
end
endmodule

```

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



You first generate “expect” patterns using a model that you know works, such as:

- Your own knowledge of what the response should be at any given time.
- When modifying a model, the working model prior to modification.
- When generating a model of lower abstraction, the existing model of higher abstraction.

You then compare the model response to your previously generated “expect” patterns.

This example compares the results to “expect” patterns that a second model at a higher level of abstraction simultaneously generates. The known model of higher abstraction is a conditional expression.

A self-checking test may at first, before you fix it, falsely indicate miscomparisons:

- Perhaps due to different data types, especially if comparing data between two different simulators.
- Perhaps due to comparing data at different or unstable points in the clock cycle, especially when comparing a model with estimated timing to a simpler unit-delay model.
- Perhaps due to comparing data after different initialization sequences, as behavioral models often skip the prolonged initialization sequences the RTL and gate-level models must exercise.
- Perhaps due to comparing unknown or “don’t-care” data values.

Be careful to not confuse the logical equality (==) and case equality (==) operators. The logical equality operator produces an unknown result when comparing a high impedance or unknown state. The case equality operator always produces either a 1 or 0 result when comparing any two operands.

Example Self-Checking Test Alternative

```

`timescale 1 ns / 1 ns
module mux_test;
  localparam PERIOD=10;
  localparam EXIT_ON_ERROR=1;
  localparam STOP_ON_ERROR=0;
  wire y; reg a, b, s;

  // MODEL BEING VERIFIED
  MUX2_1 mux ( y, s, b, a );

  // VERIFY THE RESULTS
  task expect(input e);
    if (y !== e)
      begin
        $display($time,, "ERROR MESSAGE");
        if (EXIT_ON_ERROR) $finish;
        if (STOP_ON_ERROR) $stop;
      end
    endtask
  
```



```

// TEST THE MODEL
initial
begin
  {s,b,a}=3'b010; #(PERIOD) expect(1'b0);
  {s,b,a}=3'b001; #(PERIOD) expect(1'b1);
  {s,b,a}=3'b110; #(PERIOD) expect(1'b1);
  {s,b,a}=3'b101; #(PERIOD) expect(1'b0);
  // A MISCOMPARE WON'T GET THIS FAR
  $display ("TEST PASSED");
  $finish;
end
endmodule
  
```

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



This example utilizes the same model and test as previously, but instead of comparing the model results to those of a “golden” model, periodically calls a task to verify the model results.

Illustration of a Bus Arbitration Monitor

```

`ifdef PCI_MONITOR
task pci_mon;
    integer i;
    reg [1:0] cnt [3:0];
    fork
        for (i=0; i<=3; i=i+1) cnt[i]=0;
    forever
        @ (negedge FRAME_)
        @ (posedge CLK)
        fork
            begin: FAIR
                for (i=0; i<=3; i=i+1)
                    if (!REQ_[i] && GNT_[i])
                        if (cnt[i] == 3)
                            report_error(`ARB_ERROR, `ARB_FAIRNESS, i);
                    else
                        cnt[i] = cnt[i] + 1;
                    else
                        cnt[i] = 0;
                end // fair
            ...
            join
        endtask
    ...
initial if (`PCI_MONITOR+0) @ (posedge RST_) pci_mon;
`endif

```

Simultaneously start all PCI_MONITOR processes

Simultaneously start all processes that run for each frame

If desired start monitor task after reset

For each device:
if want and don't have bus
if everybody else had bus
report "no fair!"
else
keep waiting
else
this device happy

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



You can write a Verilog task to simultaneously monitor several bus protocol rules. Some examples of such Peripheral Component Interface (PCI) bus rules might be:

- That there are no bus contentions.
- That no master parks itself on the bus.
- That the bus does not become deadlocked.
- That a target lock is gained, held, and released in accordance with locking rules.

You can optionally configure the testbench to activate this “pci_mon” task upon initialization. The task can immediately fork independent, concurrent processes, only one of which is shown, that trigger on different sets of events. The process that activates upon the FRAME_ signal becoming true can again fork independent, concurrent processes, only one of which is shown. Finish this fork-join block before the arrival of the next frame.

The completed portion of the monitor task verifies that no request remains ungranted for longer than is required for all other devices to each acquire the bus once. You can modify the monitor to accommodate other priority schemes.

The monitor reports errors to a central error handler, using codes to represent the error types.

Test Configuration

This section discusses:

- Test configuration in source code
- Test configuration with run-time scripts
- Test configuration in microcode
- Test configuration with PLI



The following section discusses configuration of the test environment. It discusses test configuration in source code, with run-time scripts, in microcode, and with the Programming Language Interface (PLI).

Test Configuration Using Source Code Constructs

You can use source code constructs to statically configure your testbench:

- Select instances, tests, and tools to compile and invoke:
 - ``define, `ifdef, $test$plusargs, $value$plusargs`
- Specify configuration of system and testbench:
 - ``include` a file that you later link to one of a variety of design or test files
- Select level of abstraction on a per instance basis:
 - Use different cell names, for example, “asic_gate”, “asic_rtl”
 - Use Verilog-2001 configurations to get cells from different libraries



As you will not be synthesizing the testbench, you can use the full range of Verilog constructs, choosing those that are most efficient or most easily coded for your application.

Some of the most common ways of presenting tests to the system are to put them in source code, run-time scripts, or pattern memory or microcode, perhaps applied via the PLI. Some of the methods you can use to configure the system and testbench in source code are:

- You can use the ``define` and ``ifdef` compiler directives to specify portions of the code to be compiled, thus selecting representations of instances, tests to be applied, and debug tools to be compiled into the simulation database.
- You can use the ``include` directive to include portions of code. The ``include` directive can itself be selected by the ``ifdef` directive, and the final resolution of the included file can be changed by modifying file-system links at the operating system level.
- You can use slightly different module names for different representations of any instantiated module, or retrieve the modules from different libraries.

Illustration of Test Configuration in Source Code (1)

```
// sys_config.h
`define CPU_BASIC
//`define CPU_QUAD
`define IO_BASIC
//`define IO_FAST
```

```
// system.v
`include "sys_config.h"
module system;
`ifdef CPU_BASIC
    basic_cpu_pcb cpu(,,);
`endif
`ifdef CPU_QUAD
    quad_cpu_pcb cpu(,,);
`endif
`ifdef IO_BASIC
    basic_io_pcb io(,,);
`endif
`ifdef IO_FAST
    fast_io_pcb io(,,);
`endif
...
endmodule
```

```
// test_config.h
`define THIS_TEST
`define THAT_TEST
//`define SOME_TEST
```

```
// test.v
`include "test_config.h"
module test;
`ifdef THIS_TEST
    `include "this_test.v"
`endif
`ifdef THAT_TEST
    `include "that_test.v"
`endif
initial
begin
`ifdef THIS_TEST
    ;
`endif
`ifdef THAT_TEST
    that_test;
`endif
end
...
endmodule
```

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



This illustration shows how you can use text macros to configure the system under test. It uses text macros to select the system configuration and test configuration and launch the test tasks.

Illustration of Test Configuration in Source Code (2)

```
`ifdef RUN_FOREVER
    forever
`else
    `ifdef RUN_LONG
        repeat (`MEM_TEST_LONG)
    `else
        repeat (`MEM_TEST_SHORT)
    `endif
`endif

begin
`ifdef MEM_TEST_RANDOM
    tdata = $random(tseed);
    taddr = $random(tseed);
`else
    taddr = taddr + 1;
    tdata = taddr;
`endif
mem_write (taddr, tdata);
mem_read (taddr, tdata);
end
```

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

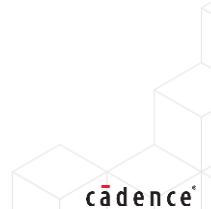


This illustration shows how you can use text macros to configure the testbench. Text macro variables select the duration of the test, and whether the memory access is random or determinate.

Test Configuration Using Run-Time Scripts

You can use run-time scripts to dynamically configure your testbench:

- Use platform utilities to construct a command line and run the simulator:
 - **awk, csh, make, perl, sed, sh**
- Use the scripting capabilities of the simulator interface:
 - **tcl**



You can use platform utilities to configure backplanes, check out the required components of a system configuration, build the simulator command line, and launch the simulator.

You can execute a file of run-time commands upon startup or at the command-line prompt. To test in this manner you must understand the progression of simulation time, and how the simulator performs interactive assignments to variables.

Example of Test Configuration with Run-Time Scripts (1)

```
// TAP controller bypass register
force TCK = 0;
force tc.next_state='RESET; release tc.next_state;
force TMS=1; repeat(5) begin #1 force TCK=1; #1 force TCK=0; end #11 $stop; .
force TMS=0; repeat(1) begin #1 force TCK=1; #1 force TCK=0; end #03 $stop; .
force TMS=1; repeat(1) begin #1 force TCK=1; #1 force TCK=0; end #03 $stop; .
force TMS=0; repeat(2) begin #1 force TCK=1; #1 force TCK=0; end #05 $stop; .
force TDI=0; repeat(1) begin #1 force TCK=1; #1 force TCK=0; end #03 $stop; .
if (TDO!==0) begin $display("DR ERROR- TDO=%b",TDO);$finish; end #00 $stop; .
force TDI=1; repeat(1) begin #1 force TCK=1; #1 force TCK=0; end #03 $stop; .
if (TDO!==1) begin $display("DR ERROR- TDO=%b",TDO);$finish; end #00 $stop; .
```

```
// TAP controller instruction register
force TCK = 0;
force tc.next_state='RESET; release tc.next_state;
force TMS=1; repeat(5) begin #1 force TCK=1; #1 force TCK=0; end #11 $stop; .
force TMS=0; repeat(1) begin #1 force TCK=1; #1 force TCK=0; end #03 $stop; .
force TMS=1; repeat(2) begin #1 force TCK=1; #1 force TCK=0; end #05 $stop; .
force TMS=0; repeat(2) begin #1 force TCK=1; #1 force TCK=0; end #05 $stop; .
if (TDO==1) begin $display("IR ERROR- TDO=%b",TDO);$finish; end #00 $stop; .
force TDI=1; repeat(1) begin #1 force TCK=1; #1 force TCK=0; end #03 $stop; .
if (TDO==0) begin $display("IR ERROR- TDO=%b",TDO);$finish; end #00 $stop; .
```



These examples test features of the Test Access Port (TAP) controller described by IEEE Standard 1149.1 as proposed by the Joint Test Action Group (JTAG). The two features tested are part of the universal feature set, and are valid for any TAP controller.

The TAP controller state machine is guaranteed to enter the reset state within five cycles of the Test Clock (TCK) pin if the Test Mode Select (TMS) pin is held high.

The first example steps the TAP controller state machine into the “shift data register” (SHIFT-DR) state and then shifts a zero followed by a one through the DR. The DR is selected by the Instruction Register (IR), which always selects the single-bit Bypass Register when reset, thus it is the Bypass Register that is tested.

The second example steps the TAP controller state machine into the “shift instruction register” (SHIFT-IR) state and then shifts a one into the multi-bit IR. This is sufficient to verify that the machine status captured in the IR contains 2'b01 in the two least significant bits, as required by the standard.

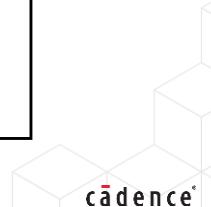
To run these examples “as is” the simulator must retain an interpretive engine. Purely compiled-code simulators cannot interpret Verilog code entered after the simulation starts.

Example of Test Configuration with Run-Time Scripts (2)

```
# TAP controller bypass register
proc clock {cnt} {
    for {set i 0} {$i<$cnt} {incr i} {
        run 1; force TCK 1;
        run 1; force TCK 0;
    }
}
force TCK 0
deposit top.tc.next_state 0
force TMS 1; clock 5 ; run 1
force TMS 0; clock 1 ; run 1
force TMS 1; clock 1 ; run 1
force TMS 0; clock 2 ; run 1
force TDI 0; clock 1 ; run 1
if {#TDO != 0} {
    puts "DR ERROR: TDO=#TDO"; finish
}
force TDI 1; clock 1 ; run 1
if {#TDO != 1} {
    puts "DR ERROR: TDO=#TDO"; finish
}
puts "DR PASS"; finish

# TAP controller instruction register
proc clock {cnt} {
    for {set i 0} {$i<$cnt} {incr i} {
        run 1; force TCK 1;
        run 1; force TCK 0;
    }
}
force TCK 0
deposit top.tc.next_state 0
force TMS 1; clock 5 ; run 1
force TMS 0; clock 1 ; run 1
force TMS 1; clock 2 ; run 1
force TMS 0; clock 2 ; run 1
if {#TDO != 1} {
    puts "IR ERROR: TDO=#TDO"; finish
}
force TDI 1; clock 1 ; run 1
if {#TDO != 0} {
    puts "IR ERROR: TDO=#TDO"; finish
}
puts "IR PASS"; finish
```

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



This illustrates exactly the same script-based test, but utilizes the Tool Command Language (Tcl), an interactive language that many compiled-code simulators and other Electronic Design Automation (EDA) tools support.

This example runs on an implementation that provides its own non-standard interactive Tcl commands, for example:

- deposit – To deposit a value that persists only until another event replaces it.
- finish – Similar to the \$finish system task.
- force – Similar to the force statement but forces a value and not a continuous assignment.
- run – To advance the simulation.
- The hash character (“#”) prefix to substitute the value of a simulation object.

Test Configuration Using Microcode

You can use microcode to configure your test session:

- Write the testbench or DUT to utilize a memory model.
 - Can involve the PLI to dynamically allocate memory.
- Compile and initialize the simulation only once.
- Configure the test session statically or dynamically.
 - Statically – Precompile the test selection.
 - Dynamically – Interactively (or by script) select the tests to run.
 - Dynamically – Download test vectors during run time.
- Execute the test residing in the memory model.



Microcode execution is somewhat similar to vector playback. Both methods load and store test data, usually into pattern memory, which you may statically or dynamically allocate. The testbench can directly apply playback vectors to simulation objects, such as nets and registers. The testbench can generate and apply test vectors by executing microcode in a machine that is either in the testbench or part of the design under test. A microcoded test is usually significantly more compact than the same test represented with raw vectors.

Example of Test Configuration Using Microcode

```

module test ( output reg RST_, input CLK, HALT );

task run (input [7:0] test);
  reg [8*8:1] testfile;
  begin
    testfile = { "CPU", 8'h30+test, ".pat" }; // GENERATE TEST FILE NAME
    $readmemb ( testfile, top.cp.mem ); // LOAD TEST MICROCODE
  end
endtask

always @(posedge HALT) begin
  $display ( "Halted at address = %h", top.cp.pc ); // LAST TEST INSTRUCTION
  $restart ( "cpu_test.dat" ); // RESTART SIMULATION
end

initial begin
  @(negedge CLK) RST_ = 0;
  @(negedge CLK) RST_ = 1;
  @(negedge CLK) $save ( "cpu_test.dat" ); // SAVE SIMULATION STATE
  $display ( "CPU HALT" ); // SIMULATION RESTARTS HERE
  $write ( "To run n'th test enter:" );
  $write ( "\ttb.run(n);\n" );
  $stop;
end

endmodule

```

For example:
"CPU0.pat"

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

cadence®

In this example, the testbench saves the entire simulation database immediately upon completing the initialization. The initialization sequence for a typical design could require significant simulation time that may not be necessary to repeat for each subtest.

This example generates a unique filename from the “run” task argument, and reads the vectors in that file into memory to exercise the CPU model.

The \$restart system task reloads the simulation database. The state of the simulation upon restarting is exactly the state saved by the \$save system task.

The IEEE Standard 1364-2001 Verilog describes but does not require the \$save and \$restart system tasks. Not all simulators support these tasks, and some offer this feature with some other proprietary interactive command.

To run these examples “as is”, the simulator must retain an interpretive engine. Purely compiled-code simulators cannot interpret Verilog code entered after the simulation starts.

Program Code Which Runs the Test

```

CPU0.pat
*****+
* Test program for the VeriRISC CPU system
*
* This diagnostic program tests the basic instruction set of the VeriRISC system. If the system executes each
* instruction correctly, then it should halt when the HLT instruction at address 17(hex) is executed.
* If the system halts at any other location, then an instruction did not execute properly. Refer to the
* comments in this file to see which instruction failed.
*****
//opcode_operand    // addr           assembly code
//-----//
@00_111_11100     // 00  BEGIN:   JMP TST_JMP
                   // 01      HLT   //JMP did not work at all
                   // 02      HLT   //JMP did not load PC, it skipped
101_11010         // 03  JMP_OK:  LDA DATA_1
                   // 04      SKZ
                   // 05      HLT   //SKZ or LDA did not work
101_11011         // 06      LDA DATA_2
001_00000         // 07      SKZ
111_10101         // 08      JMP SKZ_OK
                   // 09      HLT   //SKZ or LDA did not work
110_11100         // 0A  SKZ_OK: STO TEMP
                   // 0B      LDA DATA_1
                   // 0C      STO TEMP //store zero value in TEMP
101_11100         // 0D      LDA TEMP
001_00000         // 0E      SKZ   //check to see if STO worked
                   // 0F      HLT   //SSTO did not work
100_11011         // 10      XOR DATA_2
001_00000         // 11      SKZ   //check to see if XOR worked
111_10100         // 12      JMP XOR_OK
                   // 13      HLT   //XOR did not work at all
100_11011         // 14  XOR_OK: XOR DATA_2
001_00000         // 15      SKZ
                   // 16      HLT   //XOR did not switch all bits
                   // 17  END:    HLT   //CONGRATULATIONS - TEST1 PASSED!
                   // 18      JMP BEGIN //run test again
@1A_0000000        // 1A  DATA_1:   //constant 00(hex)
                   // 1B  DATA_2:   //constant FF(hex)
10101010          // 1C  TEMP:    //variable - starts with AA(hex)
@1E_111_00011      // 1E  TST_JMP: JMP JMP_OK
                   // 1F      HLT   //JMP is broken

```

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



This page does not contain notes.

Test Configuration Using the PLI

A peek at the PLI:

- What is it?
 - A library of *task/function* and access C routines
- How do I use it?
 - You customize the simulator to include your C routines
- What do I do with it?
 - Almost anything – You can access the simulation database at runtime
- Benefit
 - You can use C and C++ constructs in your testbench
- Concern
 - Initially steep learning curve



The Programming Language Interface (PLI) is an Application Program Interface (API) to your simulator environment. It contains utility and access routines that you call from your C programming language functions to interact with instantiated simulation objects. With the Verilog Procedural Interface (VPI), which evolved from the Open Verilog International (OVI) PLI version 2.0, you can also access behavioral constructs such as processes, blocks, and statements.

You compile your C routines with the PLI or VPI routine libraries and simulator core to create a customized version of the simulator. The customized simulator recognizes calls from your Verilog source to your C routines.

Some things you can do with the PLI are to:

- Determine what simulation objects of a specified type the design contains, where they are, and what their fanins and fanouts are.
- Calculate and annotate delays.
- Set values and monitor value changes and bus contention during simulation.
- Use dynamically allocated memory in your design.
- Access UNIX utilities, do file I/O and IPC, co-simulate, and import 3rd-party models.

Illustration of Test Configuration with the PLI

```

`timescale 1 ns / 1 ns
`define PERIOD 10
`define DELAY_MODE 0
module top;
reg [5:0] pattern_reg;
reg clk, EndOfData;
always begin clk=0; #( `PERIOD/2); clk=1; #( `PERIOD/2); end
initial begin
    $OpenStrobeFile(pattern_reg, "test.dat", EndOfData, `DELAY_MODE);
    forever @(posedge clk)
        if (!EndOfData)
            $ReadStrobeFile(pattern_reg);           → user-defined
                                                system task
        else
            begin
                $display("\n\n --- End of patterns --- \n\n");
                $stop;
            end
    end
endmodule

```

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



The testbench calls the user-defined \$OpenStrobeFile system task to open a channel to the “test.dat” pattern file, giving the application the pattern register, pattern file name, flag register, and delay mode. The PLI application uses the delay mode to signify whether it should update the pattern register immediately upon each \$ReadStrobeFile call, or after applying a pattern delay time.

The testbench then continues to update the pattern register upon each clock cycle, stopping when the “EndOfData” register asserts. The application asserts the EndOfData register when the testbench attempts to read pattern data past the end of the pattern file.

The readStrobeFile() C routine associated with the \$ReadStrobeFile user-defined system task reads and applies the next pattern and updates the state of the EndOfData register.

Module Summary

You should now be able to optimize testbench development and create reusable testbenches.

This module discussed:

- Test philosophy:
 - Verification challenges, plans, and goal
 - System-level test
 - Coverage metrics
 - Managing design space
 - Testing design regression
 - Self-checking test
- Test configuration:
 - Using source code constructs
 - Using run-time scripts
 - Using microcode
 - Using the PLI

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



You should now be able to optimize testbench development and create reusable testbenches.

This module presented design verification concepts, reinforced with small working examples and illustrations. The first part of this module explored the philosophy of system-level test, including the test plan, completion criteria, and design data management. The second part described self-checking test methods and various ways to configure the test environment.

Module Review

1. What are some characteristics of a system-level test?
2. Which are some characteristics of a regression test?
3. Suggest some ways to dynamically configure the test while the simulator is running.



This page does not contain notes.

Module Review Solutions

1. What are some characteristics of a system-level test?
 - System-level test requires up-front planning and some continued maintenance and restricts the style of the individual designer, but also facilitates communication among the development team and reduces duplication of verification efforts.
2. Which are some characteristics of a regression test?
 - A regression test suite is typically self-checking and launched by some automatic process that logs and reports the tested configuration and the results.
3. Suggest some ways to dynamically configure the test while the simulator is running.
 - You can write testbench code to dynamically react to keyboard or script input and to execute instructions from a file. The testbench code can be in Verilog or in the C programming language.



This page does not contain notes.



Lab

Lab 23-1 Testing a VeriRISC CPU Model

- For this lab, you interactively select and download test microcode and run it
- You generate a test that asks the user which program to run. You load that program into the CPU memory, reset the CPU, and let it run.
- You report the address where the program halts and whether that address is correct.

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



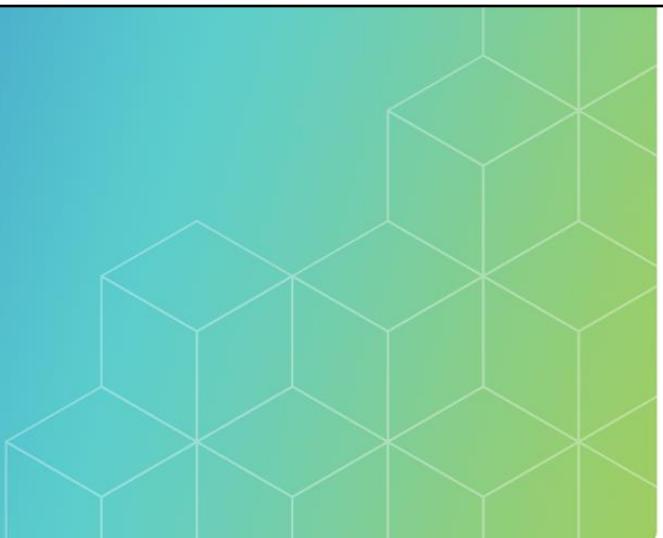
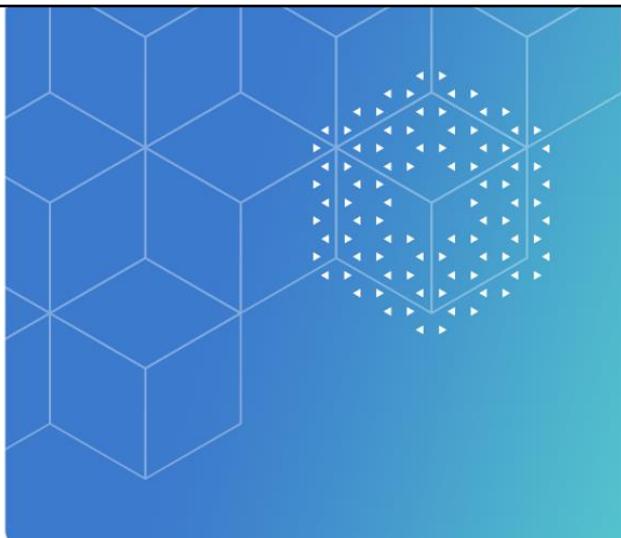
Your objective for this lab is to interactively select and download test microcode and run it.

The lab model is a VeriRISC CPU. The CPU utilizes a two-port memory so that it can fetch and execute an instruction on each clock cycle. The 8-bit CPU instruction consists of a 3-bit leftmost operation code encoding eight instructions, and a 5-bit rightmost operand addressing up to 32 words.

The lab provides three programs with incremental levels of complexity.

The CPU has only the HALT output, thus the test environment can detect program failure only by detecting that the CPU halted at an incorrect address.

For this lab, you generate a test that asks the user which program to run. You load that program into the CPU model memory, reset the CPU, and let it run. You report the address where the program halts and whether that address is correct.



Module 24

Example Verilog Testbench

cadence®

This module examines a testbench application that uses pseudocode.

Module Objective

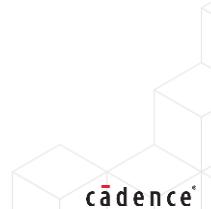
In this module, you:

- Create a testbench that executes pseudo-code

Topics

- A script driven testbench in Verilog-1995
- A script driven testbench in Veirlog-2001

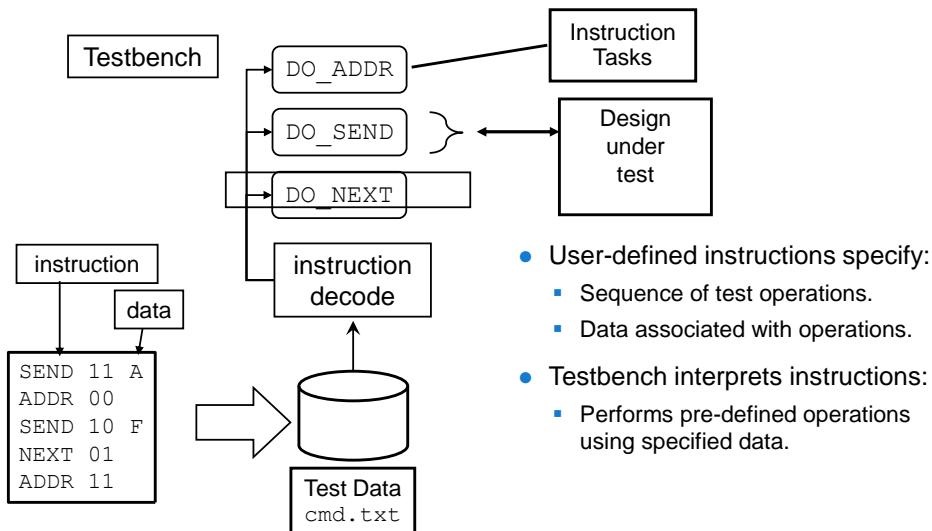
493 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to create a testbench that executes pseudo-code.

This module presents a testbench that executes pseudo-code, and then sidles off to take another look at the various simulation outputs.

Script-Driven Testbench



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



It is possible to define a set of instructions by which the operation of a testbench can be controlled. This collection of user-defined instructions is sometimes called pseudocode, and a testbench usually reads these instructions from a text file. This type of testbench is also known as a script-driven testbench.

The testbench is written to interpret the text of each instruction and then perform that operation. We will see an example of this on the following slides.

The benefits of a script-driven testbench are many, as in these examples:

- Higher level of abstraction for creating test sequences and data.
- Sequence of tests and data can be easily changed by editing the external file.
- A text file is human readable and is easier to understand than in-line, looped, or arrayed stimulus.
- Tests and testbench are modular – additional tests can be easily added.

One issue is that user-defined instructions are more likely to be written by hand, and therefore contain errors, than a file of stimulus data created by, for example, a graphics tool. Therefore, our testbench needs to carefully check the instruction data it is trying to read.

Script-Driven Testbench Verilog-1995

```

...
reg [15:0] cmdarr [0:6];
reg [15:0] cmd;
integer i;

parameter SEND = 4'h0, ADDR = 4'h1, NEXT = 4'h2;

initial begin
    $readmemh("data.txt", cmdarr);
    for (i=0;i<7;i=i+1)
        case (cmd[15:12])
            SEND:do_send(cmd[11:4], cmd[3:0]);
            ADDR:do_addr(cmd[11:4]);
            NEXT:do_next(cmd[11:4]);
            default:$display("unknown command %h",cmd);
        endcase
    end
...

```

Annotations pointing to specific parts of the code:

- Array for test data
- Instruction coding
- Read data into array
- Decode instruction
- Call tasks with specified data

data.txt

```

0_11_A //SEND 11 A
1_00_0 //ADDR 00
0_10_F //SEND 10 F
2_01_0 //NEXT 01
1_11_0 //ADDR 11
...

```

- You encode instructions in hex.
- \$readmemh reads instructions into array.
- Testbench decodes array instructions into control and data.
 - Testbench is simple.
 - Test data is not (machine code).



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

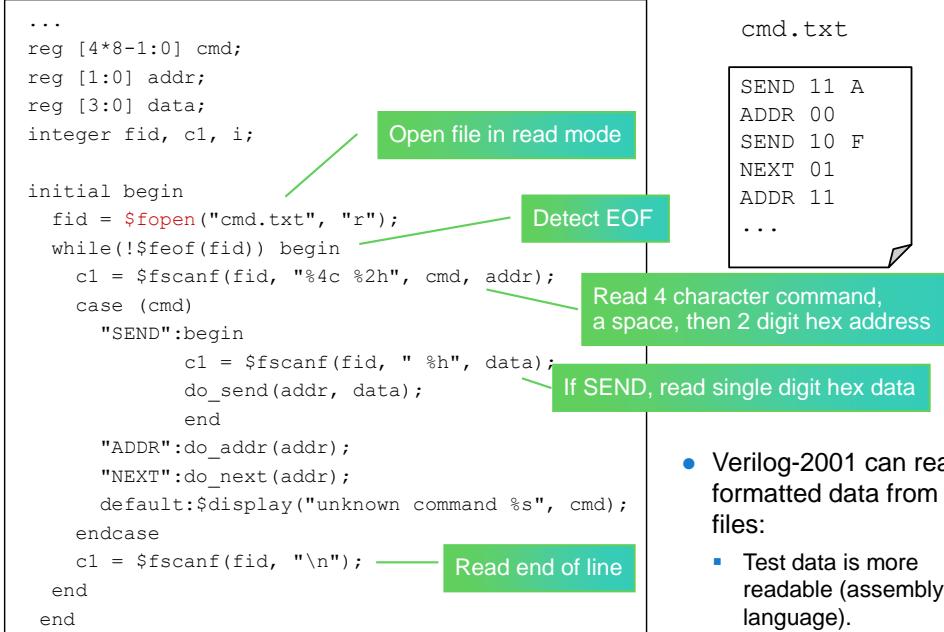
In this example, there are three forms of instruction allowed in the input text file: *ADDR XX*; *NEXT XX* and *SEND XX Y* where *XX* is a hex address and *Y* is a hex number.

In Verilog 1995, the usual way to read file data is to use *\$readmemb* or *\$readmemh* to read every line of test data into a suitably sized array. This requires you to encode test instructions and data into hex or binary numbers.

Once the test data is read into the array, you can then index each line of data ,decode the instruction bits and execute the required operations.

With this approach, the testbench is simple, but the test data is more difficult to read, write and maintain.

Script-Driven Testbench Verilog-2001



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

- Verilog-2001 can read formatted data from files:
 - Test data is more readable (assembly language).



Verilog 2001 enhances file IO with a series of C-like system tasks and functions. This allows formatted test data to be read. The example above works as follows:

- `$fopen` opens the `cmd.txt` file in read mode.
- `$feof` detects the End Of File (EOF)
- `$fscanf` reads formatted data from the file. The format string `"%4c %2h"` reads 4 characters followed by a space, then 2 hex digits. `$fscanf` assigns data matching the format specifiers to the `cmd` and `addr` variables. `cmd` is a concatenation of 4 bytes representing the ASCII encoding for the command string.
- The case statement decodes the command and calls the tasks with the required address data. If the command is `SEND`, we must read the data from the file. `$fscanf` reads a space, followed by a single hex digit which is assigned to `data`.
- The case default statement handles unknown commands.
- After the case, another `$fscanf` reads the end of line ("`\n`") character to advance the file read to the next line. Note that assignment variables in the `$fscanf` call are optional.

Module Summary

You can now create a testbench that is driven from a script and is also called a pseudocode testbench.

In this module, you learned how to write:

- A script-driven testbench using Verilog 1995 *readmemh* system task.
- A script-driven testbench using Verilog 2001 *fopen* system task.



This module examined a testbench application using pseudocode, and then sidled off to take another look at various simulation outputs.



Labs

Lab 24-1 Developing a Script-Driven Testbench Using Verilog 1995

- You write a testbench using the Verilog 95 \$readmemh system task.

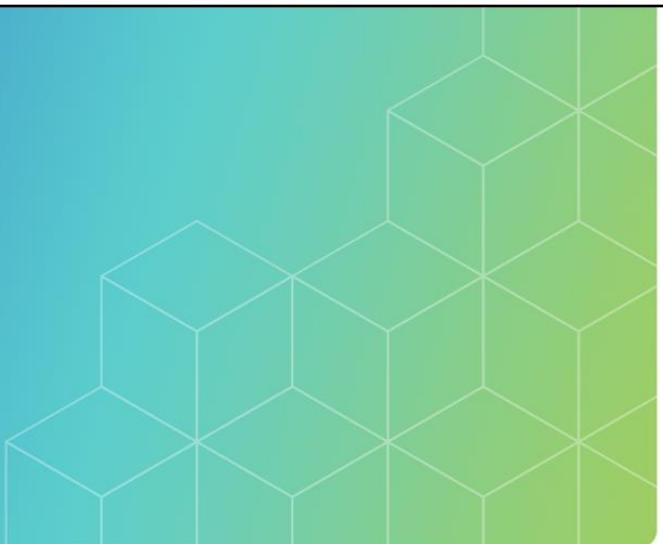
Lab 24-2 Developing a Script-Driven Testbench Using Verilog 2001

- You write another testbench using the Verilog 2001 \$fopen system task.



Try to write two testbenches. First, write a Verilog 1995 vreadmem_test.v that uses the system task \$readmemh for getting the required commands from a data.txt into the procedure call within the testbench.

For the second testbench, write a Verilog 2001 vfopen_test.v that uses the system task \$fopen to get the file containing the cmds to be opened by the procedure call within the testbench.



Module 25

Course Conclusions

cadence®

This page does not contain notes.

What This Training Covered

You can now appropriately and effectively utilize fundamental Verilog constructs for Design and Verification.

Topics

- Introduction
- Fundamentals
- Value set and data types
- Operators
- Procedural statements
- Compiler directives
- Simulation cycle
- Logic synthesis introduction
- Coding RTL for synthesis
- Avoiding simulation mismatches
- Designing finite-state machines
- Verification constructs
- Functions and Tasks
- System tasks and system Functions
- Testbench development and application

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



You can now start developing Verilog testbenches. Although Verilog is primarily a hardware design language, it does have nontrivial testbench capability and the year 2001 update significantly increased those capabilities. Verification personnel have developed very powerful testbenches over the more than two decade life of Verilog, starting long before it became a standard.

Any verification team you work with will almost undoubtedly have their own conventions you need to follow. What this module therefore presented was the “nuts and bolts”, i.e., the basics of Verilog design verification. You should consider this to be only the beginning of your learning.

Additional Resources

Standards

- [IEEE Std 1364™-2005 Verilog Hardware Description Language](#)
- [IEEE Std 1364.1™-2002 Verilog RTL Synthesis](#)

Cadence Documentation

- Verilog-XL Reference

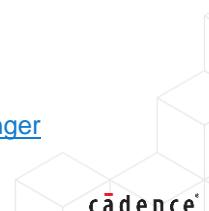
Internet Community

- [Accellera](#)
- [EDA.ORG](#)
- [FAQ: Comp.lang.verilog](#)
- [Verilog DOT COM](#)

Publications

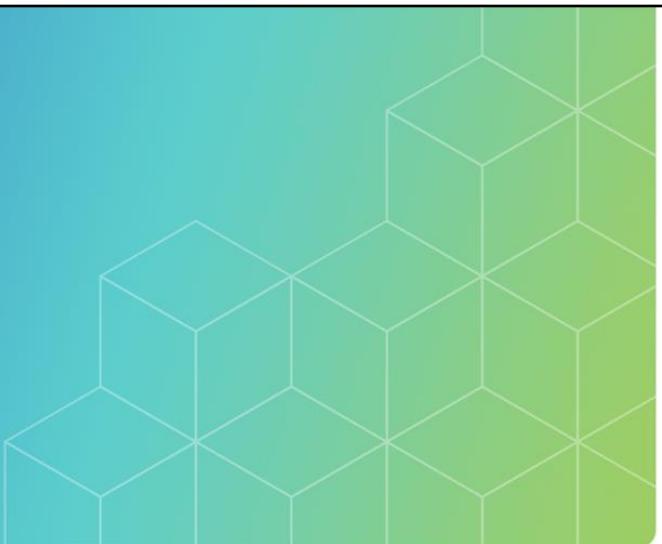
- Moorby, Philip R., and Donald E. Thomas. *The Verilog Hardware Description Language*. Springer Science+Business Media, 2002.

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



Here are some additional resources to help you with your Verilog projects:

- The IEEE Std 1364-2005 Verilog Hardware Description Language Reference Manual defines the Verilog language. This standard is unfortunately not free.
- The Cadence Verilog-XL Reference documents the Verilog language that it accepts. The Verilog language standard has diverged only slightly. This document is free to licensed users of the Cadence digital simulation tools.
- These are some of the most obvious internet resources. Many alternative internet resources are also available.
- The author lists this book in tribute to Verilog founder Philip Moorby. Many alternative books are also available.



Module 26

Next Steps

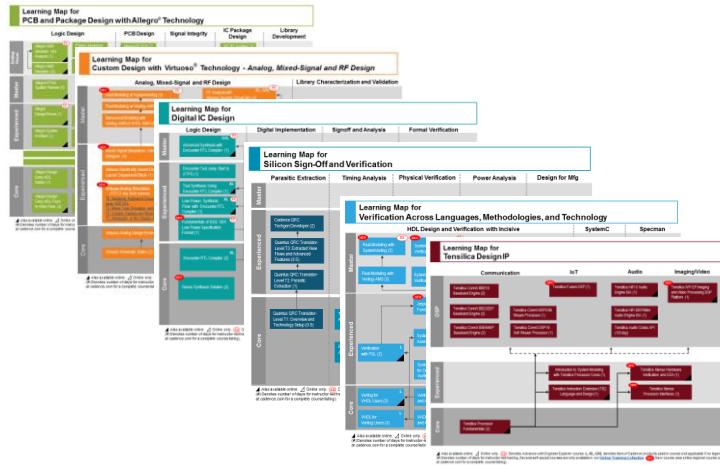
cadence®

This page does not contain notes.

Learning Maps

Cadence® Training Services learning maps provide a comprehensive visual overview of the learning opportunities for Cadence customers.

Click [here](#) to see all our courses in each technology area and the recommended order in which to take them.



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



Go here to view the learning maps:

http://www.cadence.com/Training/Pages/learning_maps.aspx

Cadence Learning and Support

The screenshot shows the Cadence Learning and Support website. At the top, there's a navigation bar with links for Cases, Tools, IP, Resources, Learning, Software, My Support, and Contribute Content. To the right of the navigation are a bell icon and a user profile icon. The main header features the Cadence logo and the text "LEARNING & SUPPORT". Below the header is a search bar with the placeholder "Start your search here..." and a magnifying glass icon. There are also buttons for "View History" and "Documents Liked". A large play button icon is overlaid in the center of the page. The background has a dark, futuristic circuit board design. At the bottom of the main content area, there's a row of six icons with labels: Installation & Licensing, Product Manuals, Training Courses, What's New, Troubleshooting Information, and Video Library. Below this row, a text box states: "Cadence Support now includes over 2000 product/language/methodology videos ("Training Bytes")!". The footer contains copyright information: "504 © Cadence Design Systems, Inc. All rights reserved." and the Cadence logo.

Click [here](#) to view the demo.

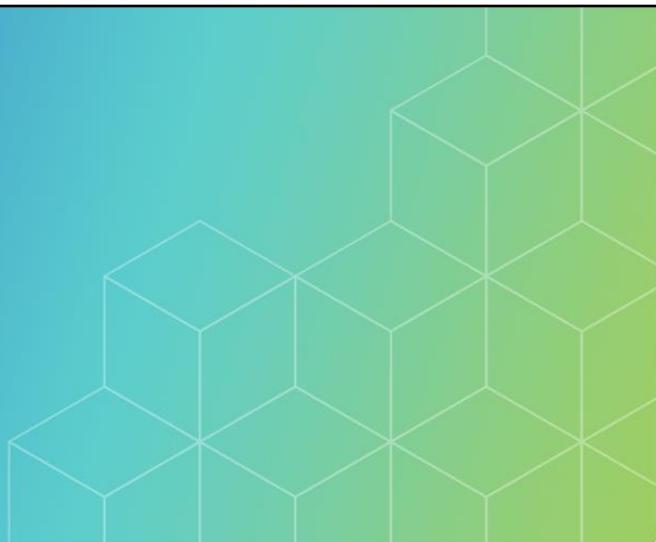
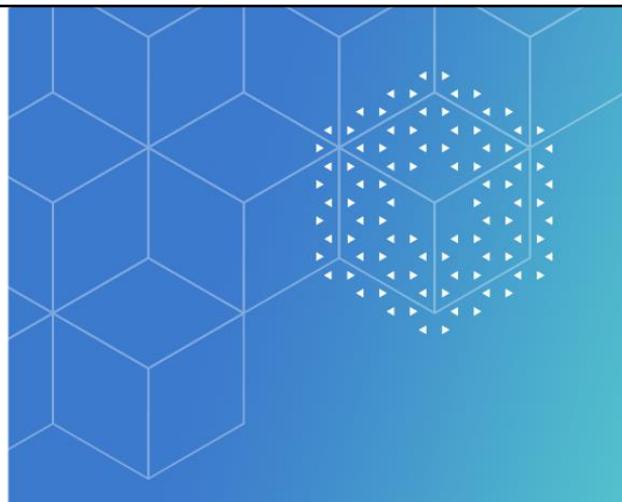
Wrap Up

- Complete Post Assessment, if provided
- Complete the Course Evaluation
- Get a Certificate of Course Completion

Thank you!



This page does not contain notes.



Appendix A

Configurations

cadence®

This module examines Verilog-2001 configurations. Verilog configurations provide a standard way to select design and testbench components from among multiple available representations.

Configurations Introduction

Why?

- As you develop and refine the design or testbench you accumulate multiple representations of the components:
 - Finer levels of abstraction
 - Before and after incorporating new features
 - Before and after incorporating change orders
- For performance reasons or merely for convenience you may want to “keep around” already-compiled versions of these components.

How?

- The Verilog-2001 update introduced the “library map file”
 - For compilation – Maps source locations to a symbolic library
 - For elaboration – Provides rules for resolving module instances



As you develop and refine the design or testbench, you accumulate multiple representations of the components, perhaps with more and less detail, before and after incorporating new features, or before and after incorporating change orders. For performance reasons or merely for convenience, you may want to “keep around” already-compiled versions of these components.

The Verilog-2001 update introduced the “library map file”, which for compilation, maps source locations to symbolic libraries, and for elaboration, provides rules for resolving module instances from those symbolic libraries. The standard calls this resolution process a “binding”. The mechanism for mapping symbolic libraries to file system locations is still vendor dependent. A library is a logical collection of design units and configurations. The standard refers to these design units and configurations as “cells”. A cell has no more than one representation in a given library, and you configure the design by providing rules to bind instances to different representations of the cell residing in different libraries.

The elaborator starts with one or more top-level modules not instantiated elsewhere, and for each top level module builds its hierarchy by binding instances to library cells as it works down each hierarchy.

Example Library Map File

The library map file:

- Maps each Verilog source path to a library name.
- Maps each cell instance to a library name.

include <i>pathname</i> ;	– Includes another library map file
library <i>libname pathname</i> [{ , <i>pathname</i> }]	– Declares a library
config <i>configname</i> ; ... endconfig	– States a configuration

```
include ${HOME}/libmap.txt;
library mylib myfile.v;
library techlib cell_lib/;
library worklib .../;
config top_rtl;
  design worklib.top;
  default liblist worklib;
  instance cpul use mylib.cpu:config;
  cell ADD use techlib.ADD;
endconfig
```

This example library map file includes a library map file from the home directory, maps three sets of Verilog source to libraries, and defines a configuration called *top_rtl* that binds the *cpul* instance to the *mylib.cpu* configuration and instances of the *ADD* cell to the *techlib* library.

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



For the compiler, the library map file maps each Verilog source to a library.

For the elaborator, the library map file maps each instance to a symbolic library. The instance mapping can be by default, by cell type, by individual instance, and by user-defined configurations.

For the elaborator, you can alternatively provide on the command line as you invoke the tool an ordered list of libraries for it to search to bind instances to cells.

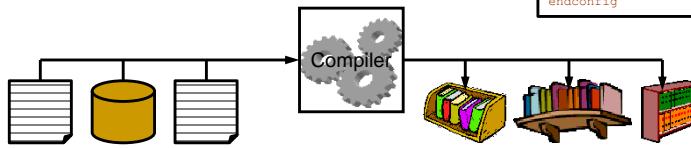
Declaring Libraries

Use the **library** keyword to declare a symbolic library and map source locations to the library:

```
library identifier pathname [ { , pathname } ]
[ -incdir pathname [ { , pathname } ] ;
```

- You can use .., ., and ... to represent directories and directory trees.
- You can use ? and * to represent any character, any string.
- File names matching multiple paths are resolved with this priority:
 1. Path names ending with an explicit filename
 2. Path names ending with a wildcarded filename
 3. Path names ending with a directory name

```
include ${HOME}/libmap.txt;
library mylib myfile.v;
library techlib cell.lib/;
library worklib .../i
config top_rtl;
  design worklib.top;
  default liblist worklib;
  instance cpu1 use mylib.cpu:config;
  cell ADD use techlib.ADD;
endconfig
```



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



A library is a logical collection of cells, each associated with a specific source. Tools do not have to pre-compile these sources, but many do.

All compliant compilers provide a tool-specific way to specify at least one library map file for a compilation session. The compiler reads the library map file, and using the list of pathnames associated with each library, matches each source file or directory of source files to a library. Tools that pre-compile the sources store the libraries in the file system in a vendor-dependent manner.

The standard permits vendors to also provide a tool-specific means to map source locations to libraries and permits configurations to utilize libraries not declared in the library map file.

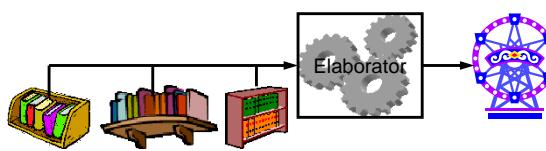
Stating Configurations

Use the **config** keyword to state a configuration:

```
config identifier ;
  design { [library.]cell } ;
  { rule }

endconfig
```

- A configuration has a name that can be the same as the design it configures.
- A configuration applies to one or more design units (usually just one).
 - You can omit the design library specification if it resides in same library as the config.
- Everything else is configuration rules (next page).



```
include ${HOME}/libmap.txt;
library mylib myfile.v;
library techlib cell_lib/;
library worklib .../;
config top_rtl;
  design worklib.top;
  default liblist worklib;
  instance cpul use mylib.cpu:config;
  cell ADD use techlib.ADD;
endconfig
```

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

cadence®

The library map file can also state one or more configurations. This is not strictly necessary, as the standard permits vendors to also provide a tool-specific means to provided an ordered list of libraries for the elaborator to search to bind instances to cells.

A configuration has a name and applies to one or more designs. If you provide configurations, you must provide a configuration for each top-level design, and as a convenience you may also provide configurations for sub-designs. A typical configuration usually applies to just one design. The configuration name can be the same as the design it applies to, but this is not required. If multiple configurations have the same name then they must reside in different libraries. The remainder of the configuration states rules for binding instances to cells of the design.

You provide to the elaborator the top-level units to elaborate, either as design names or as configuration names. You also provide either one or more library map files of declared configurations, or the before-mentioned tool-specific ordered list of libraries for the elaborator to search to bind instances to cells.

Configuration Rules and Precedence

1. For one instance of a cell, a specific cell or configuration.

```
instance top{.instance} use [library.]cell[:config]  
instance cpul use mylib.cpu:config;
```

2. For all instances of a cell type, a specific cell or configuration.

```
cell [library.]cell use [library.]cell[:config]  
cell ADD use techlib.ADD;
```

3. For one instance of a cell, a library list for an ordered search.

```
instance top{.instance} liblist [{library}]
```

4. For all instances of a cell type, a library list for an ordered search.

```
cell [library.]cell liblist [{library}]
```

5. A default library list for an ordered search.

```
default liblist [{library}]  
default liblist worklib;
```

6. The library declaration order.

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

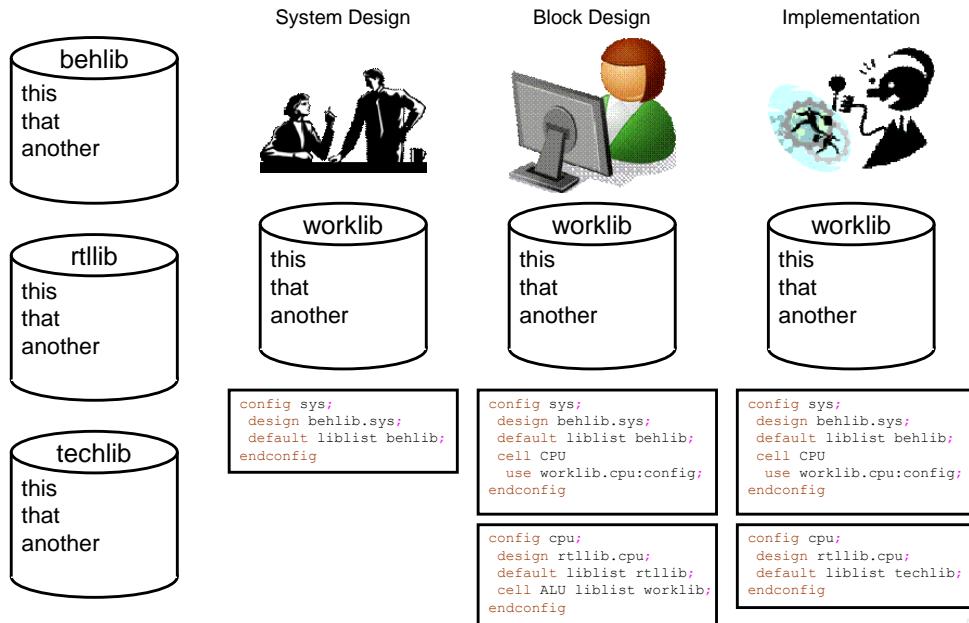


Configuration rules are simple:

- You can specify a default ordered library list;
- For all instances of a cell type, you can specify an ordered library list or a specific cell or configuration.
- For a specific instance of a cell, you can specify an ordered library list or a specific cell or configuration.

For each instance the elaborator uses the most specific of the applicable rules to bind the appropriate representation.

Example Configurations



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

cadence®

Verilog configurations are sufficiently flexible to support almost any convention you develop for your company or team. This illustration suggests one way in which you can use configurations.

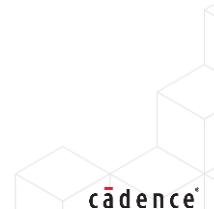
Imagine three project-wide roles for the libraries – to separate the behavioral, RTL and gate-level representations of design components. Imagine also three project-wide roles for people:

- The architects maintain local configurations that specify to get everything from the behavioral library.
- The implementers maintain local configurations that specify to get everything from the behavioral library except for their own block. For their own block they maintain a configuration that specifies to get everything from the RTL library except for the feature they are currently developing, which they instead get from their local library.
- The wizards in the back room want to test the post-synthesis netlist, so their block configuration specifies to get the block contents from the technology library.

Reference: Library Map File Syntax

<code>library_text</code>	<code>::=</code>	{ library_descriptions }
<code>library_descriptions</code>	<code>::=</code>	include_statement library_declaration config_statement
<code>include_statement</code>	<code>::=</code>	<code>include pathname ;</code>
<code>library_declaration</code>	<code>::=</code>	<code>library library_identifier pathname [{, pathname }]</code> [-incdir pathname { , pathname }] ;
<code>config_statement</code>	<code>::=</code>	<code>config config_identifier ;</code> <code>design { [library_identifier.]cell_identifier } ;</code> <code>{ config_rule_statement }</code> <code>endconfig</code>
<code>config_rule_statement</code>	<code>::=</code>	<code>default liblist_clause ;</code> cell_clause liblist_clause ; cell_clause use_clause ; inst_clause liblist_clause ; inst_clause use_clause ;
<code>liblist_clause</code>	<code>::=</code>	<code>liblist [{library_identifier}]</code>
<code>cell_clause</code>	<code>::=</code>	<code>cell [library_identifier.]cell_identifier</code>
<code>inst_clause</code>	<code>::=</code>	<code>instance topmodule_identifier{.instance_identifier}</code>
<code>use_clause</code>	<code>::=</code>	<code>use [library_identifier.]cell_identifier[:config]</code>

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



This page does not contain notes.

Appendix Review

1. What is the purpose of configurations?
2. What does a library map file contain?
3. What can configuration rules specify?
4. If a module and a configuration having the same name reside in the same library then how do you specify to use the configuration instead of the module?



This page does not contain notes.

Appendix Review Solutions

1. What is the purpose of configurations?
 - The purpose of configurations is to build a design and testbench by selecting components from among multiple representations.
2. What does a library map file contain?
 - A library map file contains optional include statements, at least one library declaration, and optional config statements.
3. What can configuration rules specify?
 - Configuration rules can specify:
 - A default ordered library list.
 - For all instances of a cell type, an ordered library list or a specific cell or configuration.
 - For a specific instance of a cell, an ordered library list or a specific cell or configuration.
4. If a module and a configuration having the same name reside in the same library then how do you specify to use the configuration instead of the module?
 - You differentiate between a module and a configuration of the same name by appending the :config suffix to mean a reference to the configuration.

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



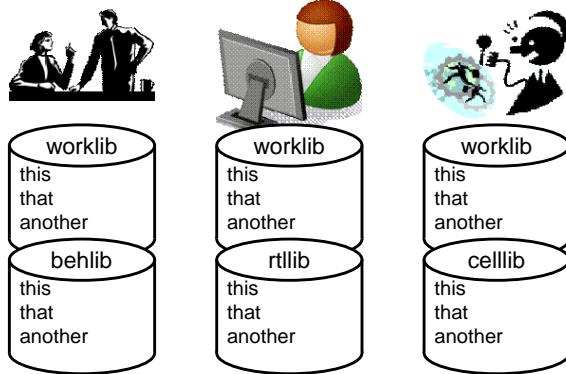
This page does not contain notes.



Lab

Lab A-1 Configuring a Simulation

- Use a library map file to define a configuration that includes at least one gate-level component.

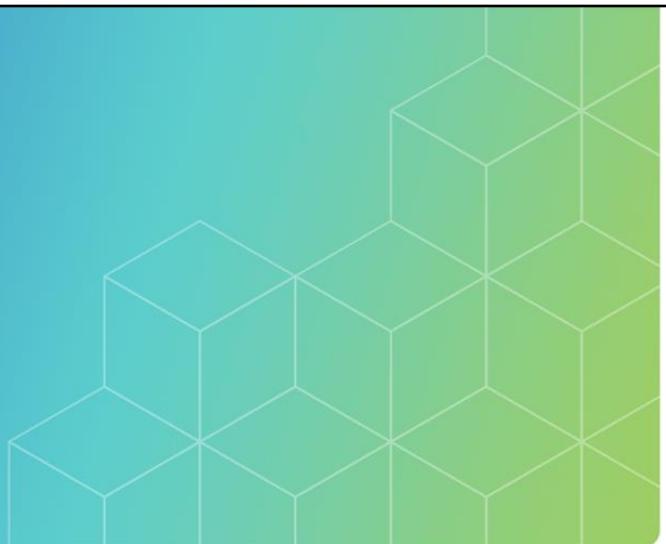


516 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to configure the design and test.

For this lab, you use a library map file to define a mixed-level configuration of components.



Appendix B

Modeling with Verilog Primitives and UDPs

cadence®

This module focuses on how to model component functionality with the built-in Verilog primitives.

Appendix Objective

In this appendix, you will:

- Use built-in primitives and continuous assignments to model logic and define combinational and sequential primitives (UDPs)

Topics

- Modeling drive and charge strength
- Modeling gate and net delays
- Choosing primitive types
- Built-in primitives
- User-defined primitives
- Example UDPs



This appendix explores the use of built-in primitives and continuous assignments to model detailed combinational behavior down to the switch level. It examines the built-in primitives, how to model primitive drive strength and capacitive net charge strength, and how to model propagation delays across primitives and nets.

Specifying Drive and Charge Strength

For most gates you can specify a drive strength:

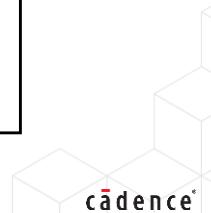
- Drive
 - **highz, weak, pull, strong, supply**
- Charge
 - **small, medium, large**
 - Apply only to **trireg** nets

Example

`(weak1, large0)`

<u>Level</u>	<u>Name</u>	<u>Mnemonic</u>
7	supply1	Su1
6	strong1	St1
5	pull1	Pu1
4	large1	La1
3	weak1	We1
2	medium1	Me1
1	small1	Sm1
0	highz1	Hi1
0	highz0	Hi0
1	small0	Sm0
2	medium0	Me0
3	weak0	We0
4	large0	La0
5	pull0	Pu0
6	strong0	St0
7	supply0	Su0

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



To facilitate your modeling of gate-level and switch-level circuits in the digital realm, you can specify delays for nets and primitives, drive strength for continuous assignments and primitives, and a charge strength for trireg nets.

Primitives and continuous assignments drive values with one of the supply, strong, pull, weak, or high-impedance strengths. They can, and often do, drive the 0 and 1 values with different strengths. In fact, it is illegal to drive both the 0 and the 1 values with the same high-impedance strength.

The trireg net can have a charge strength. A trireg charge strength together with its decay time models net capacitance.

Although the table lists discrete strength levels, for understanding the resolution of net values it is helpful to think of the driving value as occupying a range of strengths. This range can, in the case of an unknown value, overlap both the zero strengths and the one strengths.

Specifying Gate and Net Delays

Number of Values	#(delay)	#(rise, fall)	#(rise, fall, off)
Transition to 0	delay	fall	fall
Transition to 1	delay	rise	rise
Transition to Z	delay	min (rise, fall)	off
Transition to X	delay	min (rise, fall)	min (rise, fall, off)

```

and #(1,2) (out1, in1, in2, in3); // rise, fall
bufif0 #(3,4,5) (out2, in, ctrl); // rise, fall, turn-off
or #(3.2:4.0:6.3) (out, in1, in2); // min:typ:max
not #(1:2:3, 2:3:5) (o,in); // min:typ:max for rise, fall

```



The propagation delay of a gate or net is by default 0.

You can specify rise, fall, and turn-off delays:

- Rise delays apply to transitions ending in 1.
- Fall delays apply to transitions ending in 0.
- Turn-off delays apply to transitions ending in the high-impedance state.

Turn-off delay does not apply to primitive outputs that do not “turn off”.

You can specify single values, or a triplet of minimum, typical, and maximum values for each delay.

Any change in the inputs is reflected at the output only after the specified delay.

If you do not provide unique delays for transitions to the high-impedance or unknown states, then the worst case delays are used.

Choosing Primitive Types

n-input	n-output	3-state	Pull	MOS Switches	Bidirectional Switches
and nand	buf not	bufif0 notif0	pullup pulldown	nmos rnmos	tranif0 rtranif0
or nor		bufif1 notif1		pmos rpmos	tranif1 rtranif1
xor xnor				cmos r_cmos	tran rtran

- Most primitives can have an optional strength specification (more later).
- Most primitives can have an optional delay specification (more later).
- Individual primitive instances do not need an instance name.
- Primitives do not have named ports – upon instantiation, connections to output terminals precede connections to input terminals.



This table lists all the Verilog built-in primitives. Following pages describe each of these categories in more detail.

You can optionally specify a drive strength for an instance of any of these primitives except the switches. For gate-level modeling, specifying the gate drive strength more accurately models the value of a net driven by multiple devices having differently sized output transistors.

For the switch primitives, you instead select between a non-resistive switch and its resistive counterpart. The non-resistive switches generally pass the strength of the input value through to the output. The resistive switches generally reduce the strength one level from input to output. Exceptions to this general statement exist for both the non-resistive switches and the resistive switches.

You can optionally specify a delay for an instance of any of these primitives except the pullup or pulldown primitives and the uncontrolled bidirectional pass switches. The output of the pullup and pulldown primitives does not transition so cannot have a delay.

For individual primitive instances, you can omit the instance name. This is because primitives have no named internal items for you to access, so do not need a scope name.

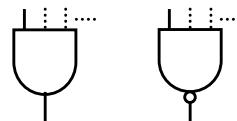
Primitives do not have named ports. Upon instantiation, you make ordered connections to output terminals followed by ordered connections to input terminals.

n-input Primitives

```
type [strength] [delay] [name] ( output, inputs... );
and (pull1,weak0) #(1,2) a1 (out, in1, in2);
```

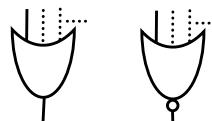
and / nand

and (o, i1, i2);



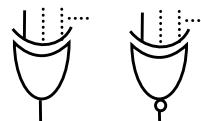
or / nor

or (o, i1, i2);



xor / xnor

xor (o, i1, i2);



The n-input gates have one output and one or more inputs and perform the expected logical function.

These gates by default strongly drive their outputs. You can optionally specify the supply, strong, pull, weak or high-impedance strengths. You specify the strengths in pairs, so if you need a non-default strength for one value then you need to also specify a strength for the other value. You can specify the high-impedance strength for only one of the values, as it makes no sense to instantiate a gate that does not drive either value.

The gates by default have no propagation delay. You can optionally specify either one or two propagation delays for the rise and fall transitions. If you specify one delay, then it applies to all transitions. If you specify two delays, then the minimum of the two delays applies to transitions to the unknown state.

n-output Primitives

```
type [strength] [delay] [name] ( outputs..., input );
buf (pull1,weak0) #(1,2) b1 (out1, out2, in);
```

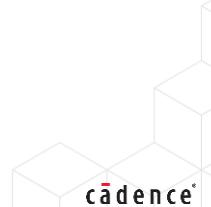
buf
buf (o, i);



not
not (o, i);



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



The n-output gates have one or more outputs and one input and perform the expected logical function.

These gates by default strongly drive their outputs. You can optionally specify the supply, strong, pull, weak or high-impedance strengths. You specify the strengths in pairs, so if you need a non-default strength for one value then you need to also specify a strength for the other value. You can specify the high-impedance strength for only one of the values, as it makes no sense to instantiate a gate that does not drive either value.

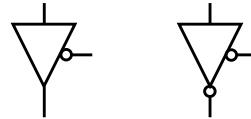
The gates by default have no propagation delay. You can optionally specify either one or two propagation delays for the rise and fall transitions. If you specify one delay, then it applies to all transitions. If you specify two delays, then the minimum of the two delays applies to transitions to the unknown state.

Having multiple outputs permits you to individually model the characteristics of the interconnect between each output and its load.

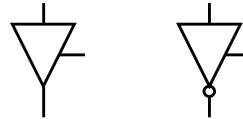
3-state Primitives

```
type [strength] [delay] [name] ( output, input, control );
bufif0 (pull, strong0) #(1,2,3) b1 (out, in, en_);
```

bufif0 / notif0
bufif0 (o, i, e_);



bufif1 / notif1
bufif1 (o, i, e_);



bufif0		CONTROL			
		0	1	Z	X
D	0	0	Z	L	
A	1	1	Z	H	
T	Z	X	Z	X	
A	X	X	Z	X	X

0 or Z but
definitely not 1

1 or Z but
definitely not 0

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



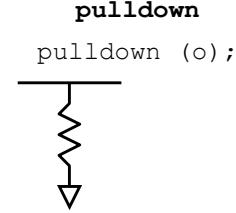
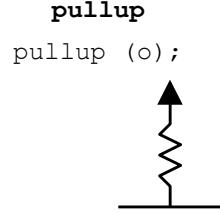
The 3-state gates have an extra enable input. When not enabled, the output of these gates float to the high-impedance state.

These gates by default strongly drive their outputs. You can optionally specify the supply, strong, pull, weak or high-impedance strengths. You specify the strengths in pairs, so if you need a non-default strength for one value then you need to also specify a strength for the other value. You can specify the high-impedance strength for only one of the values, as it makes no sense to instantiate a gate that does not drive either value.

The gates by default have no propagation delay. You can optionally specify either one, two or three propagation delays for the rise, fall and turn-off transitions. If you specify one delay, then it applies to all transitions. If you specify two delays, then the minimum of the two delays applies to transitions to the high-impedance or unknown states.

Pullup and Pulldown Primitives

```
type [strength] [name] ( output );  
pullup (weak1) p1 (out);
```



no delay!



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

The pull gates have one output and no inputs.

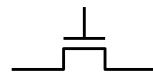
These gates by default drive their outputs with a pull strength. You can optionally specify the supply, strong, or weak strengths.

The gates have no propagation delay and you cannot specify any.

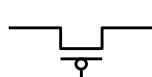
Unidirectional Switches

```
type [delay] [name] ( output, input, control );
type [delay] [name] ( output, input, ncontrol, pcontrol );
```

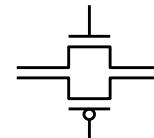
nmos / rnmos
nmos (o, i, e);



pmos / rpmos
pmos (o, i, e_);



cmos / rcmos
cmos (o, i, e, e_);



nmos	CONTROL			
	0	1	Z	X
D 0	Z	0	L	
A 1	Z	1	H	H
T Z	Z	Z	Z	Z
A X	Z	X	X	X

pmos	CONTROL			
	0	1	Z	X
D 0	0	Z	L	L
A 1	1	Z	H	H
T Z	Z	Z	Z	Z
A X	X	Z	X	X

- No strength specification!
- Non-resistive reduce supply to strong
- Resistive versions reduce strengths 2-6 by 1



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

The truth tables for the nmos and pmos switches are similar to those for the bufif1 and bufif0 with the exception that they also pass high impedance inputs through to the output.

You cannot specify a drive strength for the switch primitives. You can instantiate a non-resistive switch that passes value strengths from input to output except that it reduces supply strength to strong strength, or a resistive switch that reduces value strengths from input to output by one level except that it reduces supply strengths by two levels and does not reduce the weak capacitance or high-impedance strengths.

The gates by default have no propagation delay. You can optionally specify either one, two or three propagation delays for the rise, fall and turn-off transitions. If you specify one delay then it applies to all transitions. If you specify two delays then the minimum of the two delays applies to transitions to the high-impedance or unknown states. If you specify three delays then the minimum of the three delays applies to transitions to the unknown state.

A cmos gate acts exactly as if you connect a nmos gate and pmos gate in parallel.

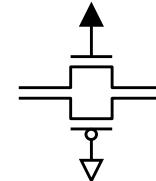
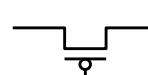
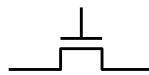
Bidirectional Switches

```

type [delay] [name] ( inout, inout, control );
type [name] ( inout, inout );

tranif1 / rtranif1      tranif0 / rtranif0      tran / rtran
tranif1 (w1, w2, e);    tranif0 (w1, w2, e_);   tran (w1, w2);

```



Delay applies only to *control* input
no strength specification!

- Non-resistive reduce *supply* to *strong*
- Resistive versions reduce strengths 2-6 by 1



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

The bidirectional switches are also known as “pass” gates, as they pass values from either inout to either inout. Both inout terminals must connect to nets.

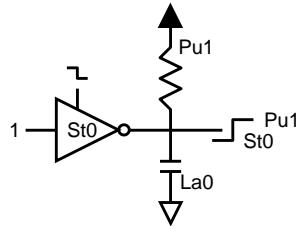
You cannot specify a drive strength for the switch primitives. You can instantiate a non-resistive switch that passes value strengths from input to output except that it reduces supply strength to strong strength, or a resistive switch that reduces value strengths from input to output by one level except that it reduces supply strengths by two levels and does not reduce the weak capacitance or high-impedance strengths.

The gates by default have no delay. You can optionally specify either one or two delays for the turn-on and turn-off transitions, but you cannot specify a delay for the data. If you specify one delay then it applies to all control-to-output transitions. If you specify two delays then the first is the turn-on delay and the second is the turn-off delay and the minimum of the two delays applies to control-to-output transitions to the high-impedance or unknown states.

Resolving Values of Unambiguous Strengths

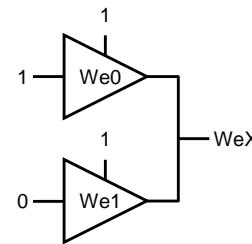
The one driver that is stronger than all the others "wins".

```
trireg (large) n;
pullup (n);
notif1 (n,1,en);
```



Resolution of equal strengths of opposite values produces a range of strengths.

```
bufif1 (weak0,weak1) (n,1,1);
bufif1 (weak0,weak1) (n,0,1);
```



strength0		strength1
7	6	5
5	4	3
4	3	2
3	2	1
2	1	0
1	0	0
0	0	1
0	1	2
1	2	3
2	3	4
3	4	5
4	5	6
5	6	7

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



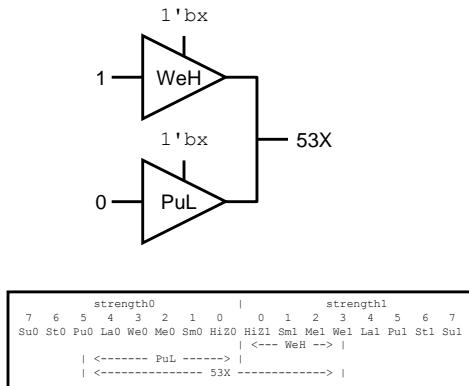
Where multiple sources each have a known value and a single strength, and one source is stronger than the others, that stronger source dominates the others and the result is the value and strength of the dominant source.

Two equal strongest drivers of opposite values produce a value that is unknown and having the range between the two strengths. For value resolution purposes a strength must always be either just one strength or just one continuous range of strengths.

Resolving Values of Ambiguous Strengths

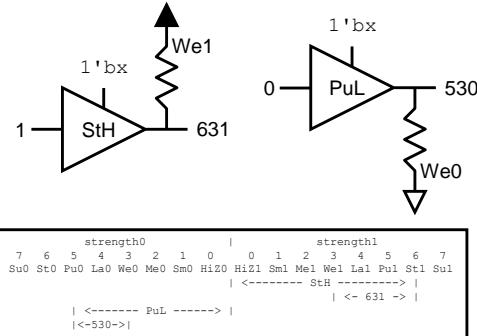
Resolution of multiple ambiguous strengths produces a range from the strongest strength0 to the strongest strength1.

```
bufif1 (weak0,weak1) (n,1,1'bx);
bufif1 (pull0,pull1) (n,0,1'bx);
```



Resolution of ambiguous strengths with unambiguous strengths removes the weaker ambiguous strength components.

```
bufif1 (n,1,1'bx);
pullup (weak1) (n);
bufif1 (pull0,pull1) (m,0,1'bx);
pulldown(weak0) (m)
```



Resolution of multiple ambiguous strengths produces a range from the strongest strength0 to the strongest strength1. Here we have a wire driven by a first buffer that because its control is unknown outputs a weak strength high and a second buffer that because its control is unknown outputs a pull strength low. The result is an unknown value between a pull strength0 and a weak strength1. The short notation for this is 53X – the strength0 followed by the strength1 followed by the value.

Resolution of ambiguous strengths with unambiguous strengths removes the weaker ambiguous strength components. Here we have a wire driven by a first buffer that because its control is unknown outputs a strong strength high, but because the net is also weakly pulled up, the net has no strength components below the weak strength. We also have a wire driven by a second buffer that because its control is unknown outputs a pull strength low, but because the net is also weakly pulled down, the net has no strength components below the weak strength.

If the strength is ambiguous but the value is not ambiguous, then the short notation is the strong strength followed by the weak strength followed by the value.

Reference: Drive Strengths and Charge Strengths

Type(s)	Optional Strength Spec	Comments
n-input, n-output, 3-state	(<i>strength0, strength1</i>) – or – (<i>strength1, strength0</i>)	strength0: supply0 strong0 pull0 weak0 highz0 strength1: supply1 strong1 pull1 weak1 highz1 Default is strong. One but not both can be highz.
pull	(<i>strength0, strength1</i>) – or – (<i>strength1, strength0</i>)	For pullup strength0 is ignored and can be omitted. For pulldown strength1 is ignored and can be omitted. Default is pull.
switch	–	Switches are not drivers.
continuous assign to scalar net	(<i>strength0, strength1</i>) – or – (<i>strength1, strength0</i>)	strength0: supply0 strong0 pull0 weak0 highz0 strength1: supply1 strong1 pull1 weak1 highz1 Default is strong. One but not both can be highz.
supply0, supply1	–	Strength is supply and cannot be changed.
tri0, tri1	–	Strength is pull and cannot be changed.
trireg net	(<i>charge-strength</i>)	charge strength: large medium small . Default is medium.

- It is the continuous assignment and not the net that gets the drive strength.
- A strength that appears in a declaration assignment is a drive strength.

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



You can specify a drive strength for a primitive instance or a continuous assignment to a scalar net or a scalar net declaration assignment:

- For the pullup and pulldown primitives, the default drive strength is the pull strength and you can specify the alternative supply, strong, weak or high-impedance strength. A strength you specify for a logic value it cannot drive is ignored.
- For the other nonswitch primitives and continuous assignments and net declaration assignments, the default drive strength is the strong strength and you can specify the alternative supply, pull, weak or high-impedance strength. You can specify a high-impedance strength for the 0 value or the 1 value but not for both the 0 and 1 values.
- You cannot specify a drive strength for the switch primitives.

You can specify a charge strength for a trireg net. The charge strength is the strength of the charge remaining on the net after all of its drivers turn off and before the charge decays. The default charge strength is the medium strength and you can specify the alternative large or small strength.

Reference: Gate and Net Delay

Type(s)	Optional Delay Spec	Comments
n-input, n-output	<code>#(delay) #(rise, fall)</code>	Can be just <i>dly</i> or <i>min:typ:max</i> Smallest delay to x
3-state, CMOS, MOS	<code>#(delay) #(rise, fall) #(rise, fall, turn-off)</code>	If 2 then smallest delay to z, x. If 3 then smallest delay to x.
pull	-	No delay specification
tran	-	No delay specification
tranif0, tranif1	<code>#(delay) #(on, off)</code>	Does not delay <i>data</i> . If 2 then smallest delay to x <i>from control</i>
trireg	<code>#(delay) #(rise, fall) #(rise, fall, decay)</code>	If 1 then all transitions. If 2 then smallest delay to x. If 3 then 3 rd is <i>decay</i> time to x.
other nets	<code>#(delay) #(rise, fall) #(rise, fall, turn-off)</code>	If 1 then all transitions. If 2 then smallest delay to z, x. If 3 then smallest delay to x.

```
bufif0 (pull1, strong0) #(1,2,3) b1 (out, in, en_);
```

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



You can specify a delay value as a single expression or as a colon-separated list of three expressions for the minimum, typical and maximum delays. Simulators provide a way to select among these three delays:

- For the n-input and n-output primitives you can provide one delay or separate rise and fall delays. If you provide separate rise and fall delays then transitions to the unknown state use the smallest of the two delays.
- For the 3-state, CMOS and MOS primitives and most nets you can provide one delay or separate rise and fall delays or separate rise, fall and turn-off delays. If you provide separate rise and fall delays then transitions to the high-impedance state and unknown state use the smallest of the two delays. If you provide separate rise, fall and turn-off delays then transitions to unknown state use the smallest of the three delays.
- The pullup, pulldown and uncontrolled bidirectional pass switches do not have delays.
- The controlled bidirectional pass switches do not have delays from the data input but do have turn-on and turn-off delays from the control input. If you provide separate turn-on and turn-off delays then transitions to unknown state from the control input use the smallest of the two delays.
- For the trireg net, the third delay, if provided, is the decay time from the point where the net is not driven to point at which its value becomes unknown. You cannot drive a trireg net to the high-impedance value. If you drive a high-impedance or unknown value onto a trireg net then an unknown state appears on the output after the smallest of the first two delays.

Built-In Primitives

The Verilog language has 26 built-in primitives.

Category	Primitive	Connections
n-input	and nand or nor xor xnor	(output, inputs...)
n-output	buf not	(outputs..., input)
3-state	bufif0 notif0 bufif1 notif1	(output, input, control)
Pull	pullup pulldown	(output)
Unidirectional Switches	pmos nmos rpmos rnmos cmos rcmos	(output, input, control) (output, input, n-control, p-control)
Bidirectional Switches	tran rtran tranif0 tranif1 rtranif0 rtranif1	(inout1, inout2) (inout1, inout2, control)

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



The and, nand, nor, or, xnor, and xor primitives can have multiple inputs. The output is always the first terminal in the list of connections.

The buf and not primitives can have multiple outputs. The input is always the last terminal in the list of connections.

The bufif0, bufif1, notif0, and notif1 primitives can drive a high-impedance value.

The pullup primitive drives a pull-strength logic 1 and the pulldown primitive drives a pull-strength logic 0.

The nmos, pmos, rnmos, and rpmos primitives are unidirectional switches that can pass a high-impedance value from input to output as well as drive a high-impedance output.

The cmos and rcmos primitives are unidirectional switches that can pass a high-impedance value from input to output as well as drive a high-impedance output. The cmos primitive is like paired nmos and pmos devices. The rcmos device is like paired rnmos and rpmos devices.

The tran and rtran primitives are bidirectional switches that can pass a high-impedance value from either inout to the other inout.

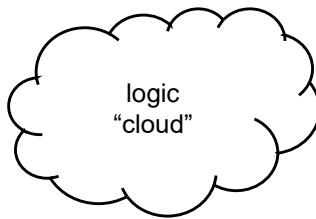
The tranif0, tranif1, rtranif0, and rtranif1 primitives are bidirectional switches that can pass a high-impedance value from either inout to the other inout, and also drive a high-impedance value.

The “resistive” versions of these primitives generally reduce the strength one level from input to output. They reduce supply strength by two levels and do not reduce the weak capacitance or high-impedance strengths.

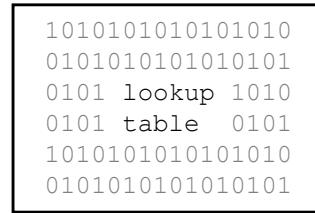
User-Defined Primitives (UDPs)

Define your own primitive to take advantage of UDP features:

- You can enhance simulation performance by replacing a “glitchy” circuit of several built-in primitives with a single UDP.
- You can more accurately model the functionality of hardware because you have exact control over the output response to input changes.



versus



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

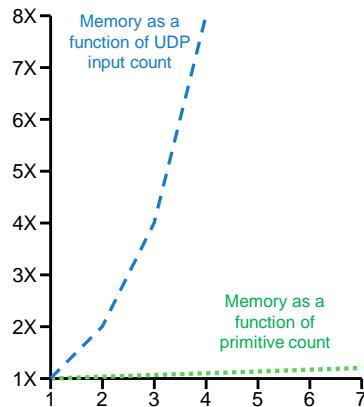


A combinational UDP can replace the logic of many primitives. This can reduce simulation time and memory requirements, which becomes more important if you instantiate the same logic numerous times.

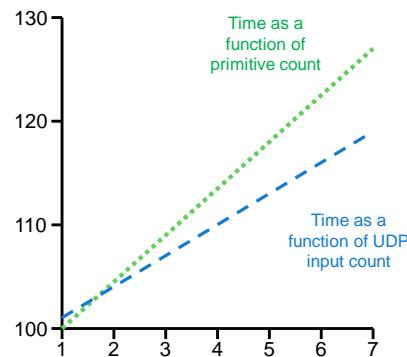
ASIC vendor simulation libraries typically utilize UDPs. This is okay because the simulation library itself will not be synthesized, and UDPs provide more accurate modeling of macro functionality than does behavioral RTL. Also, most of these libraries were at least initially prepared for use with the Verilog -XL simulator, which requires module and interconnect path timing to terminate at an acceleratable primitive.

UDPs and Platform Resources

Platform Memory Usage



Platform Time Usage



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



The UDP table size is exponentially proportional to the UDP input count.

The UDP table look-up time is constant regardless of the input count. The time to evaluate the inputs prior to look-up is linearly proportional to the input count, but increases more slowly than that of an equivalent circuit implemented with built-in primitives. This is because the simulator does not need to push otherwise wasted events through a cone of logic primitives.

Using a UDP is a trade of table memory for accuracy and performance. The memory disparity becomes less significant and the performance disparity becomes more significant when you instantiate very many UDPs of a few basic types that have perhaps up to five inputs. This is exactly what occurs in a postsynthesis netlist.

Defining a Primitive

This is what a user-defined primitive looks like:

- Define the UDP outside of any module description.
- Place the output first in the port list.
- Follow by up to ten inputs.
- For a sequential UDP:
 - Limited to 9 inputs
 - Declare the output type **reg**
 - Optionally initialize to 0, 1, or 1'bX

```
primitive name (output, inputs);
  output output;
  input input ...;
  reg output;
  initial output = value;
  table
    ...
  endtable
endprimitive
```



You declare a UDP in the definitions name space, that is, outside of and at the same level as other UDP declarations and module declarations.

Similarly to modules, you can use either the Verilog-1995 list of ports syntax or the Verilog-2001 list of port declarations syntax. Differently from modules, your one output port must appear first in the list, the list must not have inout ports, and you are limited to a maximum of ten input ports.

For a sequential UDP, that is, one that stores a value, the current value is considered one of the inputs, so you are limited to a maximum of nine input ports. You declare a register (reg) for the output of a sequential UDP, and you can optionally initialize the register to the 0, 1, or unknown value. You cannot initialize it to the high-impedance value and in fact a UDP cannot drive a high-impedance value.

Defining the Truth Table

This is what truth table entries for a sequential device may look like:

```
table
// d c : q : qnext
  0 r : ? : 0 ;
  1 r : ? : 1 ;
  ? n : ? : - ;
  * ? : ? : - ;
endtable
```

- Enter input states or transitions in the same order as the input port list.
- Then enter a colon, current state, colon, output state, and semicolon.
- The current state counts as one of the maximum of 10 inputs.

This is what truth table entries for a combination device may look like:

```
table
// s a b : y
  0 0 ? : 0 ;
  0 1 ? : 1 ;
  1 ? 0 : 0 ;
  1 ? 1 : 1 ;
endtable
```

- Enter input states in the same order as the input port list.
- Then enter a colon (:), the output state, and a semicolon (;).



Table row entries for a combinational device consist of input state values of 0, 1, X, or hook ("?") and output state values of 0, 1, or X. The hook character input symbol indicates an iteration over all possible values, which is essentially a don't-care situation.

The left table partially describes the logic of a multiplexor. For example, if the select input is 0 then the "a" input appears at the output. You will later see how to complete this table.

Table row entries for a sequential device may also contain input transitions from any of the input state values to any other of the input state values, and the output state value hyphen ("-"). The hyphen character output symbol indicates that this input transition does not change the output state.

The simulator does a separate table look-up for each transition of each changing input. For each table row you need to consider the effect of only a single input transition at a time.

The simulator acts upon the edge-sensitive entries first, then the level-sensitive entries, thus giving the level-sensitive entries higher priority.

Truth Table Symbols

Symbol	Values or Edges	Explanation
-		no change
?	0 or 1 or x	any value
b	0 or 1	any binary value
r	(01)	0->1 edge
f	(10)	1->0 edge
p	(01) or (0x) or (x1)	positive edge
n	(10) or (x0) or (1x)	negative edge
*	(??)	any edge



As the simulator does a separate table look-up for each transition of each changing input, you can think of the binary input states 0, 1, X as symbols themselves, for example the “0” input state really means either the 1-to-0 (10) or unknown-to-0 (X0) transition. For any input transition that the simulator does not find an output specification, the simulator places the unknown value on the output.

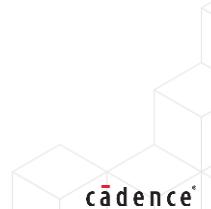
The hyphen (“-”) symbol applies only to the output and indicates that the input transition in this truth table row does not change the output.

The other symbols are each a short-cut for a set of input transitions. You can use the symbol or you can enter just exactly one input transition.

Example UDPs

You learn best by example, so here they are:

- Combinational example: 2-1 multiplexor
- Combinational example: full adder
- Level-sensitive sequential example: latch
 - Latch with low-active reset
- Edge-sensitive sequential example: D flip-flop
 - D flip-flop with low-active reset
 - D flip-flop using a notifier



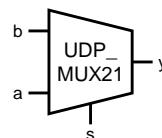
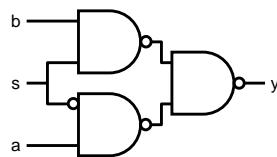
This section presents several examples of representative UDPs.

Combinational Example: 2-1 Multiplexor

- Define a UDP outside of any module.
- Place the output first in the port list.
- Make table entries in input port list order.
- Specify all combinations producing a known output.

Char	Iteration	Explanation
?	0 or 1 or x	any value

```
primitive UDP_MUX21 (y, s, a, b);
output y;
input s, a, b;
table
// s a b : 0
0 0 ? : 0; // select a
0 1 ? : 1;
1 ? 0 : 0; // select b
1 ? 1 : 1;
? 0 0 : 0; // pessimism
? 1 1 : 1; // pessimism
endtable
endprimitive
```



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



If the select input is 0, the output follows the a input regardless of the b input value.

If the select input is 1, the output follows the b input regardless of the a input value.

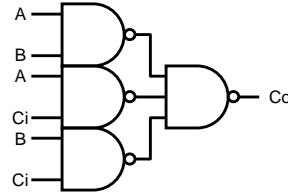
For every input combination for which you do not specify an output, the simulator makes the output unknown. You can almost always include table entries to reduce such pessimism. For this multiplexor, if the two data inputs have the same value, the output must assume that value regardless of the value of the select input. You cannot model this behavior with the built-in Verilog primitives.

The table entries to reduce pessimism somewhat overlap previous table entries. Determine for example which two table rows match a situation where all inputs are low and which two table rows match a situation where all inputs are high. Such overlap is common when reducing pessimism and is permitted if all the overlapping rows specify the same output value.

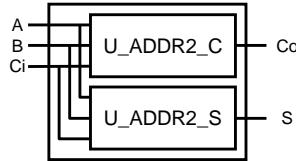
Combinational Example: Full Adder

You can implement a full adder with only two combinational UDPs.

```
// FULL ADDER CARRY-OUT TERM
primitive U_ADDR2_C (Co,A,B,Ci);
output Co;
input A, B, Ci;
table
// A B Ci : Co
1 1 ? : 1;
1 ? 1 : 1;
? 1 1 : 1;
0 0 ? : 0;
0 ? 0 : 0;
? 0 0 : 0;
endtable
endprimitive
```



```
// FULL ADDER SUM TERM
primitive U_ADDR2_S (S,A,B,Ci);
output S;
input A, B, Ci;
table
// A B Ci : S
0 0 0 : 0;
0 0 1 : 1;
0 1 0 : 1;
0 1 1 : 0;
1 0 0 : 1;
1 0 1 : 0;
1 1 0 : 0;
1 1 1 : 1;
endtable
endprimitive
```



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



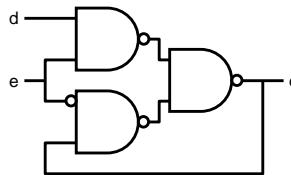
The carry output is high when the sum of the inputs is 2 or 3. The sum output is high when the sum of the inputs is 1 or 3.

You can implement the full adder with two combinational UDPs to replace five built-in Verilog primitives. This removes the internal nodes, thus reducing the simulation database size and platform memory requirements. It also reduces the work the simulator must do to propagate events, thus the time to simulate the design.

Level-Sensitive Sequential Example: Latch

- Declare the output type as **reg** if the UDP stores a value.
- You can use an **initial** statement to initialize the stored value.
- The current stored value is another table input.

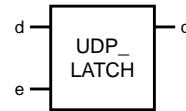
Char	Iteration	Explanation
-		No change
?	0 or 1 or x	Any value



```

primitive UDP_LATCH (q, d, e);
output q; reg q;
input d, e;
initial q = 1'b0;
table
// d e : q : qnext
0 1 : ? : 0 ; // enabled 0
1 1 : ? : 1 ; // enabled 1
? 0 : ? : - ; // disabled
0 ? : 0 : - ; // pessimism
1 ? : 1 : - ; // pessimism
endtable
endprimitive

```



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



A latch is logically a multiplexor with the output fed back to one of the inputs. If you model a latch with the built-in primitives then you waste computer and memory resources and your model is less than truly accurate and you may need to time propagation of the enable signal so that you do not lose data as you disable the latch.

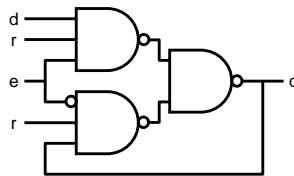
For this example:

- If the device you are to model has power-up reset, then you can model the reset with an initial statement.
- If the e input is 1, then the q output follows the d input regardless of the current state.
- If the e input is 0, then the q output maintains its current state regardless of the d input.
- If the e input is unknown, then the output still does not change if the d input is the same as the current state. Including such scenarios reduces pessimism. If you omit such scenarios, then the simulator must set the output to the unknown state.

Latch with Low-Active Reset

Easiest to consider reset as just another level-sensitive input.

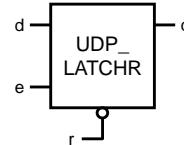
Char	Iteration	Explanation
-		No change
?	0 or 1 or x	Any value



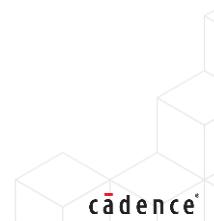
```

primitive UDP_LATCHR (q, d, e, r);
output q; reg q;
input d, e, r;
initial q = 1'b0;
table
// d e r : q : qnext
? ? 0 : ? : 0 ; // reset
0 1 1 : ? : 0 ; // enabled 0
1 1 1 : ? : 1 ; // enabled 1
? 0 1 : ? : - ; // disabled
0 ? ? : 0 : - ; // pessimism
1 ? 1 : 1 : - ; // pessimism
endtable
endprimitive

```



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



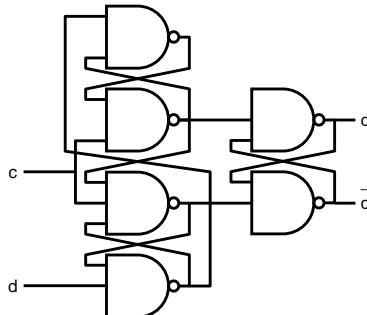
To add reset to the primitive latch we effectively gate off both multiplexor inputs. This is a situation where you must carefully read the vendor's documentation, as for some devices the reset has no effect while the latch is enabled.

The truth table for this latch has an added column and row for the reset, and also has a slight modification to reduce pessimism when the reset state is unknown.

Edge-Sensitive Sequential Example: D Flip-Flop

Specify all transitions producing a known output – one transition per row.

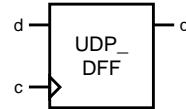
Char	Edges	Explanation
r	(01)	0->1 edge
p	(01), (0x), (x1)	positive edge
n	(10), (x0), (1x)	negative edge
*	(??)	any edge



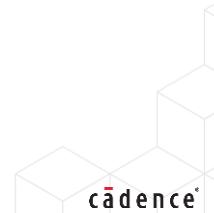
```

primitive UDP_DFF (q, d, c);
output q; reg q;
input d, c;
table
// d c : q : qnext
0 r : ? : 0 ; // clk rise
1 r : ? : 1 ; // clk rise
0 p : 0 : - ; // clk posedge
1 p : 1 : - ; // clk posedge
? n : ? : - ; // clk negedge
* ? : ? : - ; // data edge
endtable
endprimitive

```



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



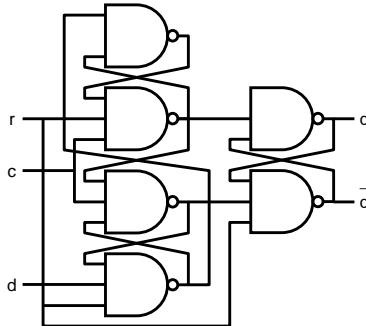
The simulator does a separate table look-up for each transition of each input. For each table row you need to consider the effect of only a single input transition:

- In a rising clock edge the output follows the data input.
- On any positive clock edge, if the data input matches the existing state, then the output does not change.
- For any negative clock edge the output does not change.
- For any data edge the output does not change.

D Flip-Flop with Low-Active Reset

Easiest to consider as transitions rather than levels.

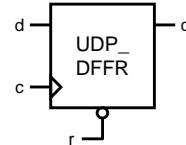
Char	Edges	Explanation
r	(01)	0->1 edge
p	(01), (0x), (x1)	positive edge
n	(10), (x0), (1x)	negative edge
*	(??)	any edge



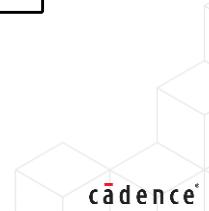
```

primitive UDP_DFFR (q,d,c,r);
output q; reg q;
input d, c, r;
table
// d c r : q : qnext
? ? 0 : ? : 0; // reset
? ? n : 0 : -; // rst negedge
? ? p : ? : -; // rst posedge
0 r ? : ? : 0; // clk rise
1 r 1 : ? : 1; // clk rise
0 p ? : 0 : -; // clk posedge
1 p 1 : 1 : -; // clk posedge
? n ? : ? : -; // clk negedge
* ? ? : ? : -; // data edge
endtable
endprimitive

```



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



To add reset to the primitive D flip-flop we effectively gate off all cross-coupled nands.

The truth table for this D flip-flop has an added column and three added rows for the reset, and also has a slight modification to reduce pessimism when the reset state is unknown but we are clocking in low data anyway.

D Flip-Flop Using a Notifier

The following example is of a positive edge-triggered D flip-flop with an asynchronous reset, complete with timing checks and path delays. The model instantiates a UDP which has an input for the value of a notifier reg.

```
`timescale 1 ns / 1 ns
module dffr_m (q, d, clk, rst);
  input d, clk, rst;
  output q;
  reg nt;
  UDP_DFFR u1 (q, d, clk, rst, nt);
  specify
    specparam tS = 2;
    (clk => q) = (2:3:4);
    $setup (d, posedge clk, tS, nt);
  endspecify
endmodule
```

```
primitive UDP_DFFR (q, d, c, r, n);
  output q; reg q;
  input d, c, r, n;
  table
    // d c r n : q : qnext
    ? ? 0 ? : ? : 0; // reset
    ? ? n ? : 0 : -; // rst negedge
    ? ? p ? : ? : -; // rst posedge
    0 r ? ? : ? : 0; // clk rise
    1 r 1 ? : ? : 1; // clk rise
    0 p ? ? : 0 : -; // clk posedge
    1 p 1 ? : 1 : -; // clk posedge
    ? n ? ? : ? : -; // clk negedge
    * ? ? ? : ? : -; // data edge
    ? ? ? * : ? : x; // notifier
  endtable
endprimitive
```

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



A specify block defines input-to-output module timing paths and assigns delays to them and checks the relationship between edges of input ports. A specify block can appear only within a module definition and applies to all instances of the module. You can later annotate the timing on a per-instance basis. This training elsewhere describes specify blocks and annotation in more detail.

This specify block:

- Declares a specify parameter and assigns a value to it.
- Declares a module timing path between the clock input and the data output and assigns to it a timing triple.
- Enables a setup timing check between the data input and the clock rising edge.

All timing checks examine the relationship between edges of one or two signals and report a failure to meet the specified limit. All timing checks also accept an optional argument that is the name of a register (reg) variable to toggle upon timing check failure. Model developers typically connect this variable to a UDP input to make the UDP output go to the unknown state upon failure of an associated timing check.

Appendix Summary

You can use built-in primitives and continuous assignments to precisely model logic:

- Built-in primitives include logic gates and pullup/pulldown and unidirectional switches and bidirectional switches.

You can specify drive strengths for primitives and continuous assignments, and charge strengths for capacitive nets:

- Verilog resolves the value and strength of a signal on a net.

You can specify propagation delays across primitives and nets, and decay time for capacitive nets:

- Generally up to 3 values for rise, fall, and turn-off, and these values can be minimum:typical:maximum triples.



This module explored the use of built-in primitives and continuous assignments to model detailed combinational behavior down to the switch level. It examined the built-in primitives, how to model primitive drive strength and capacitive net charge strength, and how to model propagation delays across primitives and nets.

Appendix Summary (continued)

You can extend the set of built-in primitives by defining your own:

- You can define combinational and sequential primitives.
- You define their behavior with a table.
- You instantiate them like a built-in primitive.

UDPs accurately, compactly and efficiently describe logic function:

- You can replace a circuit of built-in primitives by a single UDP.

UDPs cannot do some things that a built-in primitive can do:

- A UDP has a single output and at most 10 inputs.
- A UDP port cannot be bi-directional and cannot drive the high-impedance ("z") value.
- The logic synthesis standard does not accept UDPs.



This module explored user-defined primitives. It compared UDPs with the built-in Verilog primitives and explained why you would want to define your own UDP and how to define it. The module ended with several complete examples of representative UDPs.

Appendix Review

1. What Verilog element(s) can model capacitance?
2. What Verilog element(s) can model resistance?
3. Why do primitive instances not need instance names?
4. Why do primitive definitions not have port names?
5. What delay does the simulator use for transitions to X if you specify only rise, fall, and turnoff delays?



This page does not contain notes.

Appendix Review (continued)

6. Suggest some reasons to define your own primitives.
7. If multiple UDP inputs transition in the same simulation cycle, how many times does the simulator consult the look-up table?
8. What does the simulator do if some combination of input transition and states matches no table row?
9. What does the simulator do if some combination of input transition and states matches both a level-sensitive table construct and an edge-sensitive table construct?
10. What does the simulator do, if due to your use of wildcard symbols, some combination of input transition and states matches multiple level-sensitive table constructs, or multiple edge-sensitive table constructs, that specify different outputs?



This page does not contain notes.

Appendix Review Solutions

1. What Verilog element(s) can model capacitance?
 - Only a trireg net can have charge strength and decay time.
2. What Verilog element(s) can model resistance?
 - Only the rtran primitive models bidirectional uncontrolled resistance.
3. Why do primitive instances not need instance names?
 - Instance names provide a scope for accessing internal elements that primitives do not have.
4. Why do primitive definitions not have port names?
 - The number of primitive terminals is few and their order is fixed, thus accommodating named connections would provide little benefit.
5. What delay does the simulator use for transitions to X if you specify only rise, fall, and turnoff delays?
 - The simulator uses the worst-case (minimum) of the specified delays.
6. Suggest some reasons to define your own primitives.
 - The reasons are:
 - Precisely control output transitions in response to input transitions.
 - Replace a “glitchy” cloud of primitive logic with a few primitives having “clean” output transitions.
 - More efficiently utilize platform resources (cycles and memory).

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



This page does not contain notes.

Appendix Review Solutions (continued)

7. If multiple UDP inputs transition in the same simulation cycle, how many times does the simulator consult the look-up table?
 - The simulator consults the look-up table once for each transitioning input.
8. What does the simulator do if some combination of input transition and states matches no table row?
 - The simulator places the UDP output in the unknown ("X") state.
9. What does the simulator do if some combination of input transition and states matches both a level-sensitive table construct and an edge-sensitive table construct?
 - The simulator processes first the edge-sensitive construct, and then the level-sensitive construct, which overrides.
10. What does the simulator do, if due to your use of wildcard symbols, some combination of input transition and states matches multiple level-sensitive table constructs, or multiple edge-sensitive table constructs, that specify different outputs?
 - This situation is an error that should be detected during elaboration.



This page does not contain notes.

Appendix Exercise

- Given this behavioral model of one stage of a binary counter:
 - Convert the model to a structural representation that uses the Verilog built-in primitives. Assume that set and reset cannot both be simultaneously asserted.
 - Second convert the model to a user-defined primitive. Assume that set and reset cannot both be simultaneously asserted.

```
module countBit (
    output reg q;
    input clk, set_, rst_);
    always @(
        negedge clk
        or negedge set_
        or negedge rst_)
        if (~rst) q <= 0; else
        if (~set) q <= 1; else
            q <= ~q;
endmodule
```



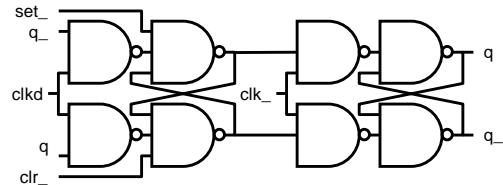
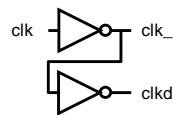
This page does not contain notes.

Appendix Exercise Solution

- Given this behavioral model of one stage of a binary counter:
- Convert the model to a structural representation that uses the Verilog built-in primitives. Assume that set and reset cannot both be simultaneously asserted.
- Second convert the model to a user-defined primitive. Assume that set and reset cannot both be simultaneously asserted.

```
module countBit (
  output reg q;
  input clk, set_, rst_);
  always @(
    negedge clk
    or negedge set_
    or negedge rst_)
    if (~rst) q <= 0; else
      if (~set) q <= 1; else
        q <= ~q;
endmodule
```

```
module countBit (
  output wire d,
  input clk, set_, rst_);
  not (clk_, clk), (clkd, clkd);
  nand (n1a, q_, clk), (n1b, q, clkd);
  nand (n2a, set_, n1a, n2b), (n2b, rst_, n1b, n2a);
  nand (n3a, n2a, clk), (n3b, n2b, clk);
  nand (q, set_, n3a, q_), (q, rst_, n3b, q);
endmodule
```



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



This page does not contain notes.



Labs

Lab B-1 Using Built-In Verilog Primitives with a Macro Library

- For this lab, you modify a provided macro library file to replace RTL descriptions with descriptions based upon the Verilog built-in primitives.

Lab B-2 Using User-Defined Verilog Primitives with a Macro Library

- For this lab, you further modify the macro library file to define a sequential flip-flop UDP and replace the primitive-based flip-flops with UDP-based flip-flops.

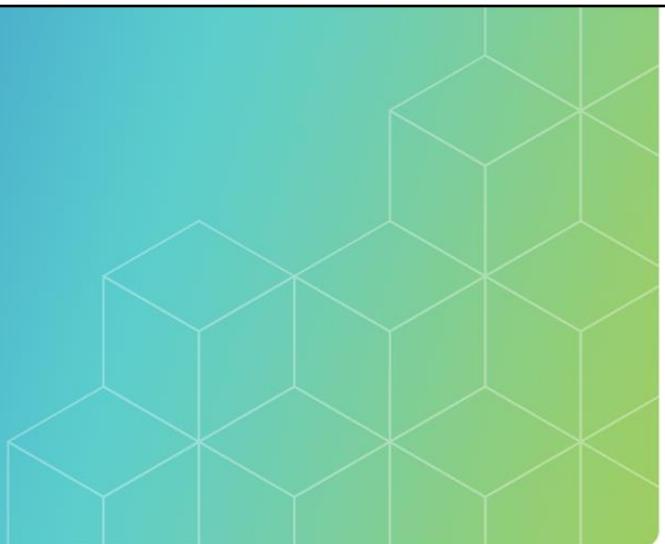
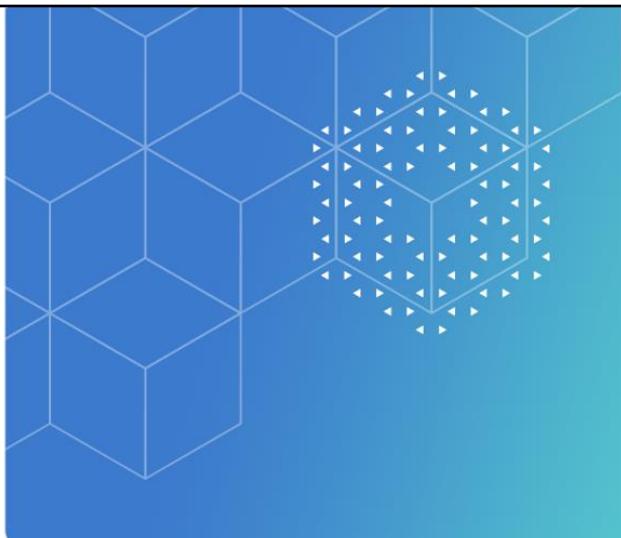


Your objective for the first lab is to use built-in primitives and continuous assignments to model logic.

For this lab, you use built-in primitives to define the functionality of a small set of ASIC macrocells.

Your objective for the second lab is to define combinational and sequential primitives.

For this lab, you use user-defined primitives to define the functionality of a small set of ASIC macrocells.



Appendix C

SDF Annotation Overview

cadence®

This page does not contain notes.

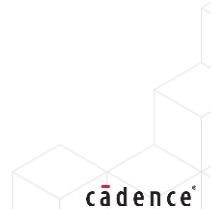
Appendix Objective

In this appendix, you will:

- Annotate timing information

Topics

- Briefly describe the purpose of Annotation
- Understand an SDF Timing Data File
- Work around SDF Annotation Issues
- Annotate SDF Timing Data



This module communicates to you the knowledge needed to annotate timing data to your Verilog designs and to modify the Standard Delay Format (SDF) file for debugging purposes.

This module briefly describes the purpose of annotation, explains some of the SDF keywords, discusses some issues you may encounter, and demonstrates the annotation process.

The IEEE standard 1497-2001 “SDF for the Electronic Design Process” defines the SDF version 4.0.

Timing Annotation Introduction

A silicon vendor simulation library contains estimated intrinsic timing.

For accurate timing simulation you need additional data:

- Drive strength
- Interconnect parasitics
- Total load
- Environmental factors
 - Process
 - Temperature
 - Voltage

You need to simulate fast clock with slow data, slow clock with fast data.

Most event simulators cannot directly do this.

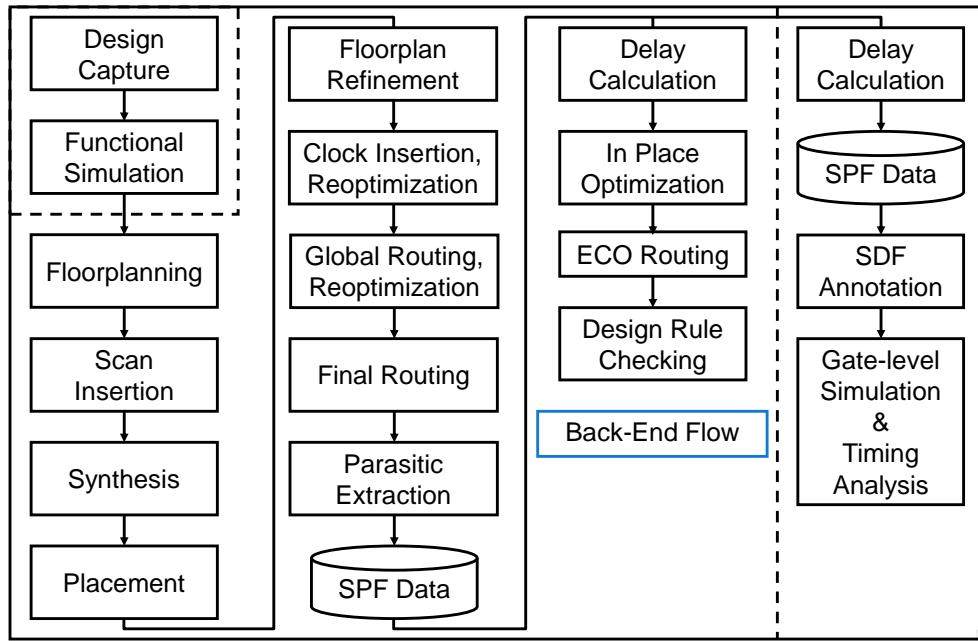
557 © Cadence Design Systems, Inc. All rights reserved.



A silicon vendor simulation library typically contains estimated intrinsic timing. For accurate timing simulation you need additional data, especially for the interconnect delay. Delay calculators provide this data using a standard delay format (SDF).

The hardware description languages have constructs to specify minimum, typical, and maximum timing information, but simulators can use only one of these timing sets during a simulation session. You can prepare and annotate a SDF file containing fast clock times and slow data times for setup checks, or slow clock times and fast data times for hold checks, and run the simulation with the new timing data. This will provide results that are more accurate, but still not as accurate as a static timing analysis tool.

Timing Data Flow



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

cadence®

The typical back-end flow now tightly integrates synthesis, layout, and route tools. These tools interchange data using proprietary database formats, and generate SDF data only for analysis tools outside of the flow.

A delay calculator requires a structural representation of the design. It can generate estimated delays based only on design connectivity and hierarchy, or detailed delays using parasitic information extracted by layout tools. The reduced parasitics may be in the Standard Parasitics Format (SPF).

Delay Calculators

Two major categories of delay calculators exist:

- Delay calculators embedded in the tools
 - Synthesis tools
 - Static timing analysis tools
- Custom delay calculators
 - User-defined or vendor-supplied
 - Directly annotate via API or generate SDF



You can write a custom delay calculator. You must choose an appropriate delay equation.

Most ASIC vendors provide a delay calculator based on their manufacturing process. They can write these in C and use the Application Programming Interface (API) to directly annotate the design during simulation. They can also write standalone programs that generate SDF for the built-in timing data annotator.

Understanding an SDF File

The SDF provides a tool-independent, uniform way to specify timing data.

This section presents an example SDF file and then explains the:

- SDF header data
- SDF cell timing data
 - SDF cell labels
 - SDF cell delays
 - SDF cell timing checks
 - SDF cell timing environment



This section presents an example SDF file and then explains each section of the file. It individually explains the label, delay, timing check, and timing environment clauses of the cell timing data. Some of these clauses have sub-clauses, thus forming a hierarchy of cell timing data.

Example SDF File

```
(DELAYFILE
  (SDFVERSION "4.0")
  (DESIGN "system")
  (DATE "Sun Feb 22 14:10:03 EST 2009")
  (VENDOR "Cadence")
  (PROGRAM "delay_calc")
  (VERSION "08.20-p001")
  (DIVIDER /) /*hierarchical divider */
  (VOLTAGE 4.5:5.0:5.5)
  (PROCESS "worst")
  (TIMESCALE 1ns) /* delay time units */
  (CELL (CELLTYPE "system") (INSTANCE block_1) /* top-level blocks */
    (DELAY (ABSOLUTE
      (INTERCONNECT D1/z P3/i (.155:::155) (.130:::130))))))
  (CELL (CELLTYPE "INV") (INSTANCE *) /* all instances of "INV" */
    (DELAY (INCREMENT
      (IOPATH i z (.345:::348) (.325:::329))))))
  (CELL (CELLTYPE "OR2") (INSTANCE B1/C1) /* this instance of "OR2" */
    (DELAY (ABSOLUTE
      (IOPATH i1 z (.300:::300) (.325:::325))
      (IOPATH i2 z (.300:::300) (.325:::325))))))
)
```

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



The SDF header specifies SDF file configuration information in proper order.

Cell references may be individual instances or type references.

Cell delays can be absolute or incremental, and can be conditional.

Cell delays may be single values or min:typ:max triples. If you omit a value from a delay triple, that delay will default to the delay specified in the cell timing model. The example illustrates this omission.

The SDF can also specify timing checks, which can be conditional, and timing constraints, which simulators ignore.

Understanding SDF Header Keywords

An SDF file starts with a header (most of which is merely documentation):

Keyword	Status	Format	Default
SDFVERSION	Required	qstring	none
DESIGN	Document	qstring	none
DATE	Document	qstring	none
VENDOR	Document	qstring	none
PROGRAM	Document	qstring	none
VERSION	Document	qstring	none
DIVIDER	Optional	hchar	“.”
VOLTAGE	Document	real rtriple	none
PROCESS	Document	qstring	none
TEMPERATURE	Document	real rtriple	none
TIMESCALE	Optional	number unit	1 ns

(DELAYFILE (SDFVERSION "4.0") (CELL ...) ...)

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



The SDFVERSION header keyword is required.

The DIVIDER defaults to the period (“.”) character and the TIMESCALE defaults to 1ns, so those header keywords are optional.

The remainder of the header is purely for documentation purposes.

Understanding SDF Cell Timing Keywords

A CELL keyword identifies a design subscope and the timing data to apply there:

Keyword	Status	Format	Default
CELL	Required		none
CELLTYPE	Required	qstring	none
INSTANCE	Required	see notes	see notes
LABEL	Optional	lbl_type	none
DELAY	Optional	deltype	none
TIMINGCHECK	Optional	tchk_def	none
TIMINGENV	Optional	te_def	none

```
( CELL
  ( CELLTYPE    type   )
  ( INSTANCE    scope  )
  ( LABEL      ...   )
  ( DELAY       ...   )
  ( TIMINGCHECK ...  )
  ( TIMINGENV   ...  )
  ...
  ...
```

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



A CELL keyword identifies a design subscope and the timing data to apply there.

The CELLTYPE clause is required. Annotation tools must by default report following cell instances that do not match the specified cell type.

The INSTANCE clause is required, and either an asterisk or hierarchical identifier may follow it:

- The hierarchical identifier is relative to the scope at which you instruct the annotator to perform the annotation. Legal identifier characters are alphanumerics, the dollar sign (\$), and the underscore (_). Any other identifier character will be individually escaped with a backslash (\) character. Note that this escape mechanism is not identical to the Verilog escape mechanism.
- The asterisk (*) wildcard character applies the timing data to all cells of the cell type.
- Lack of a hierarchical identifier, or existence of a wildcard, applies the timing data to the scope at which you instruct the annotator to perform the annotation.

Following slides describe the LABEL, DELAY, TIMINGCHECK, and TIMINGENV clauses.

Understanding SDF Cell Label Keywords

A LABEL keyword replaces or adds delay values to labels:

Keyword	Specifies
ABSOLUTE	Absolute (replaced) delay values
INCREMENT	Incremental (added) delay values

```
( LABEL
  ( ABSOLUTE  ( name delays )  ... )
  ( INCREMENT ( name delays )  ... )
  ...
)
```

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



A LABEL clause replaces or adds delay values to labels. A label is a Verilog specify block parameter (specparam).

- The ABSOLUTE sub-clause specifies a new value for the label.
- The INCREMENT sub-clause specifies a value to add to the previous value of the label.

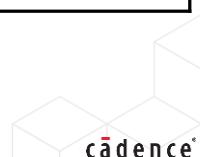
You can specify 1, 2, 3, 6, or 12 delay values. The delay values apply to the twelve edges of the four-value (01ZX) logic system, with varying degrees of granularity. Each delay value may be a single real number or a triple of real numbers specifying unique minimum, typical, and maximum values. You may omit up to two of the numbers in a triple.

Later slides provide examples of delay values.

Understanding SDF Cell Delay Keywords

A DELAY keyword applies cell and net delay and pulse reject and error values or ratios:

Keyword	Specifies
ABSOLUTE	Absolute delay replaces any existing delay
INCREMENT	Incremental delay adds to any existing delay
PATHPULSE	Path pulse controls reject and error values
PATHPULSEPERCENT	Path pulse controls reject and error ratios (percent)
DEVICE	Cell delay to a specific (or all) cell output(s) (or inout(s))
IOPATH	Cell delay from a specific cell input (or inout) to a specific cell output (or inout)
COND	Conditional cell delay
CONDENSE	Default cell delay (applies only to matching COND)
RETAIN	Time for IOPATH to maintain previous state
NETDELAY	Net delay from all sources of the net to all loads of the net
PORT	Net delay from all sources of the net to a specific input (or inout) port
INTERCONNECT	Net delay from a specific output (or inout) port to a specific input (or inout) port



A DELAY clause applies cell and net delay and pulse reject and error values or ratios.

- The ABSOLUTE sub-clause specifies a new value for the delay.
- The INCREMENT sub-clause specifies a value to add to the previous value of the delay.
- The PATHPULSE sub-clause specifies cell path pulse rejection, and optionally error, limits. The annotator by default sets the pulse rejection limit to the delay. A pulse having a duration lower than this limit is rejected. This is known as an “inertial” delay model. You can set the rejection limit lower, and you can optionally set an error limit that is between the rejection limit and the delay. A pulse having a duration of at least the rejection limit, but lower than the error limit, filters the transition to the unknown value at the output port. A rejection and error limit of zero passes all pulses regardless of width. This is known as a “transport” delay model.
- The PATHPULSEPERCENT sub-clause specifies cell path pulse rejection, and optionally error, limits as a percent of the delay. This is useful for providing estimated path pulse controls on a global basis that you can later override on a per-cell basis for individual cells.

You annotate cell delay with either the DEVICE or the IOPATH sub-clause:

- The DEVICE clause specifies the intrinsic delay of a cell, gate, or port instance.
- The IOPATH clause specifies the delay from an instance input to an instance output. You can make IOPATH delays conditional upon the state of some other input port, and for IOPATH delays you can specify a data retention time that is less than the delay. The output port transitions to the unknown state for the duration between the retention time and the delay time.

You annotate interconnect delay with either the NETDELAY, PORT or INTERCONNECT sub-clause:

- The NETDELAY clause specifies delay for all paths on the net. This clause is seldom used.
- The PORT clause specifies delay for all paths terminating at a specified cell input port.
- The INTERCONNECT clause specifies port-to-port interconnect delay.

Annotating Cell Delay

In the SDF file, annotate cell delay with:

- The DEVICE keyword to specify the intrinsic delay of a cell, gate, or port instance:
 - This specifies min:typ:max delay for rise, fall, and turn off edges:
`(DEVICE U1 (7:8:9) (1:2:3) (4:5:6))`
 - This specifies min:typ:max delay, reject, and error for all edges:
`(DEVICE U1 ((7:8:9) (1:2:3) (4:5:6)))`
- The IOPATH keyword to specify the delay from an instance input to an instance output:
 - This specifies min:typ:max delay for rise, fall, and turn off edges:
`(IOPATH INA OUTA (7:8:9) (1:2:3) (4:5:6))`
 - This specifies min:typ:max delay, reject, and error for all edges:
`(IOPATH INA OUTA ((7:8:9) (1:2:3) (4:5:6)))`



You annotate cell delay with either the DEVICE clause or the IOPATH clause:

- The DEVICE clause specifies the intrinsic delay of a cell, gate, or port instance.
 - The 1st short example specifies min:typ:max intrinsic delay for rise, fall, and turn off edges.
 - The 2nd short example specifies min:typ:max intrinsic delay and reject and error limits for all edges. Due to an additional set of parentheses, this is actually only one delay value, so applies to all edges. The three triples are the delay and the reject and error limits.
- The IOPATH clause specifies the delay from an instance input to an instance output.
 - The 3rd short example specifies min:typ:max I/O path delay for rise, fall, and turn off edges.
 - The 4th short example specifies min:typ:max I/O path delay and reject and error limits for all edges.

Example IOPATH Delay Specification Syntax

```
/* 1-delay expressions, with and without reject and error limits */

(IOPATH p_in p_out (delay_triple) )
(IOPATH p_in p_out ((delay_triple) (reject_triple) ))
(IOPATH p_in p_out ((delay_triple) ( ) (error_triple)))
(IOPATH p_in p_out ((delay_triple) (reject_triple) (error_triple)))
```

```
/* 2-delay expressions, with and without reject and error limits */

(IOPATH p_in p_out (rise_triple)
    (fall_triple) )
(IOPATH p_in p_out ((rise_triple) (reject_triple)
    ((fall_triple) (reject_triple) ))
(IOPATH p_in p_out ((rise_triple) ( ) (error_triple))
    ((fall_triple) ( ) (error_triple)))
(IOPATH p_in p_out ((rise_triple) (reject_triple) (error_triple))
    ((fall_triple) (reject_triple) (error_triple)))
```



Here is some example IOPATH delay specification syntax:

- The first is of one-delay expressions, with and without reject and error limits. As these are all just single delay values, they apply to all transitions. Note that if you want to annotate the error limit without annotating the reject limit, then you need to use a placeholder for the reject limit.
- The second is of two-delay expressions, again with and without reject and error limits. The rise delay applies to all potentially positive transitions and the fall delay applies to all potentially negative transitions. Other transitions use the worst-case delays. That is, transitions from unknown to high impedance use the maximum of the two delays and transitions from high impedance to unknown use the minimum of the two delays.

Annotating Net Delay

In the SDF file, annotate net delay with:

- The PORT keyword to specify the delay of all interconnections to an input port:
 - This specifies min:typ:max delay for rise, fall, and turn off edges:
`(PORT IN (7:8:9) (1:2:3) (4:5:6))`
 - This specifies min:typ:max delay, reject, and error for all edges:
`(PORT IN ((7:8:9) (1:2:3) (4:5:6)))`
- The INTERCONNECT keyword to specify the delay from an instance output to an instance input:
 - This specifies min:typ:max delay for rise, fall, and turn off edges:
`(INTERCONNECT a.OUT b.IN (7:8:9) (1:2:3) (4:5:6))`
 - This specifies min:typ:max delay, reject, and error for all edges:
`(INTERCONNECT a.OUT b.IN ((7:8:9) (1:2:3) (4:5:6)))`



You annotate interconnect delay with either the PORT clause or the INTERCONNECT clause:

- The PORT clause specifies the delay of all interconnections to an input port.
 - The 1st short example specifies min:typ:max delay for rise, fall, and turn off edges.
 - The 2nd short example specifies min:typ:max delay, and reject and error limits, for all edges. Due to an additional set of parentheses, this is actually only one delay value, so applies to all edges. The three triples are the delay and the reject and error limits.
- The INTERCONNECT clause specifies the delay from an instance output to an instance input.
 - The 3rd short example specifies min:typ:max delay for rise, fall, and turn off edges.
 - The 4th short example specifies min:typ:max delay, and reject and error limits, for all edges.

Example INTERCONNECT Delay Specification Syntax

```
/* 1-delay expressions, with and without reject and error limits */

(INTERCONNECT A.out B.in  (delay_triple) )
(INTERCONNECT A.out B.in ((delay_triple) (reject_triple) ))
(INTERCONNECT A.out B.in ((delay_triple) ()           (error_triple)))
(INTERCONNECT A.out B.in ((delay_triple) (reject_triple) (error_triple)))
```

```
2-delay expressions, with and without reject and error limits */

(INTERCONNECT A.out B.in  (rise_triple)
                  (fall_triple) )
(INTERCONNECT A.out B.in ((rise_triple) (reject_triple) )
                  ((fall_triple) (reject_triple) ))
(INTERCONNECT A.out B.in ((rise_triple) ()           (error_triple))
                  ((fall_triple) ()           (error_triple)))
(INTERCONNECT A.out B.in ((rise_triple) (reject_triple) (error_triple))
                  ((fall_triple) (reject_triple) (error_triple)))
```



Here is some example INTERCONNECT delay specification syntax:

- The first is of one-delay expressions, with and without reject and error limits. As these are all just single delay values, they apply to all transitions. Note that if you want to annotate the error limit without annotating the reject limit, then you need to use a placeholder for the reject limit.
- The second is of two-delay expressions, again with and without reject and error limits. The rise delay applies to all potentially positive transitions and the fall delay applies to all potentially negative transitions. Other transitions use the worst-case delays. That is, transitions from unknown to high impedance use the maximum of the two delays and transitions from high impedance to unknown use the minimum of the two delays.

Understanding SDF Cell Timing Check Keywords

A TIMINGCHECK keyword specifies minimum or maximum limits for the time between transitions of two signals or two transitions of the same signal:

Keyword	Stamp	Check	Value	Limit
SETUP	port 1	port 2	Positive	Min
HOLD	port 2	port 1	Positive	Min
SETUPHOLD	port 1,2	port 2,1	Signed	Min
RECOVERY	port 1	port 2	Positive	Min
REMOVAL	port 2	port 1	Positive	Min
RECREM	port 1,2	port 2,1	Signed	Min
SKEW	port 1,2	port 2,1	Signed	Max
BIDIRECTSKEW	port 1,2	port 2,1	Positive	Min
WIDTH	port	port	Positive	Min
PERIOD	port	port	Positive	Min
NOCHANGE	port 2,1	port 1,2	Signed	Min

A TIMINGCHECK clause specifies minimum or maximum limits for the time between transitions of two signals or two transitions of the same signal. All of these checks do approximately the same thing – they record the time of a transition, and then at a later transition of the same or a different signal, report any violation of the limit.

Only the SETUPHOLD, RECREM, SKEW, and NOCHANGE checks accept negative values. The SETUPHOLD check is a combination of SETUP and HOLD checks where one or the other check may use a negative value, provided that the sum of the two numbers is not less than zero. Likewise, the RECREM check is a combination of RECOVERY and REMOVAL checks.

The SKEW check has a maximum limit and all other checks have minimum limits.

Understanding SDF Cell Timing Environment Keywords

A TIMINGENV keyword associates constraint values with critical paths in the design and provides information about the timing environment in which the circuit will operate. Constructs in this subclause are used for forward annotation to design implementation tools.

Keyword	What It Specifies
PATHCONSTRAINT	Max path delay
PERIODCONSTRAINT	Max clock period
SUM	Max sum of multiple path delays
DIFF	Max difference between two path delays
SKEWCONSTRAINT	Max clock skew
ARRIVAL	Input port arrival time
DEPARTURE	Output port departure time
SLACK	Input port available slack
WAVEFORM	Clock waveform



A TIMINGENV clause associates constraint values with critical paths in the design and provides information about the timing environment in which the circuit will operate. Constructs in this subclause are used for forward annotation to design implementation tools. As they are not used for simulation, you will seldom see them.

Working Around Tool-Independent Annotation Issues

Here are some general rules that you should be aware of:

- Each SDF version reflects significant changes:
 - Cadence defined SDF version 1.0
 - OVI defined SDF versions 2.0, 2.1, 3.0
 - IEEE std. 1497-2001 defines SDF version 4.0
- The annotator must apply SDF data in file order:
 - Subsequent LABEL, TIMINGCHECK, TIMINGENV replace previous
 - Subsequent ABSOLUTE delays replace previous delays
 - Subsequent INCREMENT delays add to previous delays
- The annotator must apply cell path data only to existing constructs:
 - Edge-qualified SDF cell paths do not map to unqualified cell paths
 - Conditional SDF cell paths do not map to unconditional cell paths



The IEEE Std. 1497-2001 “SDF for the Electronic Design Process” defines the SDF version 4.0 and requires the SDF file to specify the SDF version. Open Verilog International (OVI) defined previous SDF versions and permits the annotator to by default assume the SDF version 1.0. Each SDF version contains significant differences that this training material does not re-document.

The annotator must apply SDF timing data in file order. Subsequent data targeting a previously annotated construct replaces the previously annotated data with one exception. This exception is that incremental delay data adds to the previously annotated data.

The annotator must apply cell path timing data only to existing constructs, and cannot only partially annotate the existing construct. You can annotate unqualified or unconditional data to qualified or conditional paths, but cannot annotate qualified or conditional data to unqualified or unconditional paths.

Working Around Tool-Specific Annotation Issues

Here are some tool-related issues that you should be aware of:

- The SDF escapes only individual identifier characters
- The SDF allows a port to be an internal node
- Annotators can convert INTERCONNECT delay to PORT delay
 - Annotators can ignore INTERCONNECT delay
- Annotators must attempt to apply DEVICE delays to cell timing paths
 - If unsuccessful, must apply to all primitives driving output port
- Simulation tools do not use TIMINGENV data
- Not all annotators use negative values
 - Shall substitute 0 in ABSOLUTE clauses
 - May substitute 0 in INCREMENT clauses
 - Simulators typically do not use negative delay



The SDF escapes individual illegal identifier characters with a preceding backslash (“\”) character. Other EDA tools escape the entire identifier. Verilog escapes otherwise illegal identifiers with a leading backslash character and a trailing whitespace (space, tab, or newline) character.

The SDF allows a port used for timing purposes to actually be an internal node that is not a connector at a design unit boundary. SDF files referencing such internal nodes might not be portable between EDA tools.

Annotators unable to annotate individual source-to-load interconnect pairs may choose to convert INTERCONNECT delay to PORT delay, and may also choose to simply ignore INTERCONNECT delay.

Annotators must attempt to apply DEVICE delays to cell timing paths from all input or inout ports to a specified or all output or inout port(s). If no such path exists for a given applicable output, the annotator must apply the delay to all primitives driving that output from any level in the cell hierarchy.

The TIMINGENV keyword provides timing environment and constraint data for forward annotation to design implementation tools. Simulators do not use those constraints.

Annotating SDF Data with \$sdf_annotation

You can annotate the Verilog portions of your design with **\$sdf_annotation**.

Only the SDF file name argument is required:

\$sdf_annotation (Default	Description
"sdf_file",	—	Required SDF file name
module_instance,	Current	Scope to annotate
"configuration_file",	—	Configuration file name
"log_file",	—	Log file name
"mtm_spec",	"TOOL_CONTROL"	Which value to annotate
"scale_factors",	"1.0:1.0:1.0"	Scale factors to apply
"scale_type"	"FROM_MTM"	What values to scale
) ;		
\$sdf_annotation ("timing.sdf", , , "sdf.log") ;		



You can annotate the Verilog portions of your design with the **\$sdf_annotation** built-in system task. Only the SDF file name argument is required. To provide later arguments, you need to at least provide commas as argument placeholders. Arguments you provide here override those in the configuration file.

- The **sdf_file** argument is the SDF file name as a character string literal or a register vector containing the equivalent ASCII string value.
- The **module_instance** optional argument specifies the scope in which to annotate the SDF information. The simulator by default annotates the scope of the **\$sdf_annotation** system task call.
- The **configuration_file** optional argument is the configuration file name. With this vendor-defined file you can presumably more precisely control the annotation. Neither the IEEE Std 1364-2001 Verilog nor the IEEE Std 1497-2001 SDF defines this file.
- The **log_file** optional argument is the log file name.
- The **scale_type** optional argument specifies which SDF file timing values to use as the basis for scaling and subsequent annotation. The “**FROM_MTM**” value means to use the minimum, typical and maximum values. You can alternatively select “**FROM_MINIMUM**”, “**FROM_TYPICAL**” or “**FROM_MAXIMUM**”.
- The **scale_factors** optional argument specifies the scale factors for generating new minimum, typical, and maximum timing values. The default is to not scale them.
- The **mtm_spec** optional argument specifies which new timing value to annotate. The default is to annotate the values selected upon simulator invocation, but you can alternatively annotate just the “**MINIMUM**”, “**TYPICAL**” or “**MAXIMUM**” values.

Appendix Summary

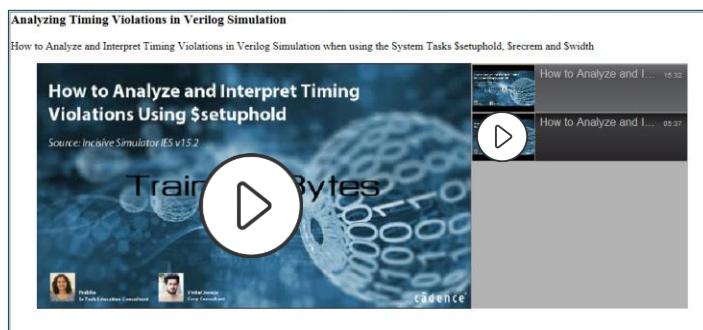
- The purpose of annotation is to enable simulation with:
 - Intrinsic timing estimates that are more accurate.
 - Interconnect delays.
- The SDF file contains a header and cell timing data for:
 - Labels, delays, timing checks and (rarely) timing environment.
- Annotation issues include those related to:
 - The SDF standard, the HDL, the annotation tool.
- Annotate using **\$sdf_annotation** with or without scaling.



This module communicated to you the knowledge you need to annotate timing data to your Verilog designs and to modify the SDF file for debugging purposes.

This module briefly described the purpose of annotation, explained some of the SDF keywords, discussed some issues you may encounter, and demonstrated the annotation process.

Videos on SDF Annotation in COS



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



There are 3 videos posted in support.cadence.com (COS) related to sdf annotation.

“How to Analyze and Interpret Timing Violations in Verilog Simulation when using System Tasks \$setuphold, \$recem and \$width.”

These videos explain simple real issues as examples using waveforms.

[How to Analyze and Interpret Timing Violations Using \\$setuphold](#)

[How to Analyze and Interpret Timing Violations Using \\$width](#)

[How to Analyze and Interpret Timing Violations Using \\$recem](#)



Lab

Lab C-1 Annotating an SDF with Timing

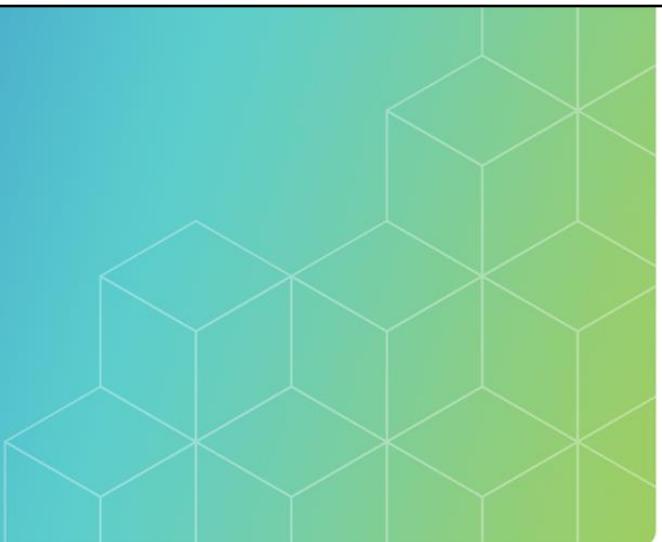
- For this lab, you annotate timing data to a design.
- The design is a serial interface receiver. It receives characters serially and transmits them in parallel.

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



Your objective for this lab is to annotate timing information.

For this lab, you annotate timing data to a design.



Appendix D

SystemVerilog Overview and UVM Methodology Introduction

cadence®

This page does not contain notes.

Appendix Objectives

In this appendix, you will:

- Introduce the scope, purpose and features of SystemVerilog
- Explain Metric Driven Verification
- Introduce Universal Verification Methodology (UVM)

Topics

- What is System Verilog?
- SystemVerilog extensions and segmentation
- Key features of the language for Design, Synthesis and Verification
- Metric Driven Verification and the advantages
- UVM introduction



This module explores features of SV and how they overcome the limitations of existing Verilog constructs.

Also the concepts of methodology, metric driven verification and UVM introduction are touched upon.

What Is SystemVerilog?

- An unified Hardware Design, Specification, and Verification Language:
 - Defines a concise unified language using a single simulation tool
- A set of Extensions to Verilog-2001, including:
 - Added data types and relaxation of rules on existing data types
 - Higher abstraction level modeling features
 - Language enhancements for synthesis
 - Interfaces to model communication between design blocks
 - Assertions, constrained randomization, functional coverage
 - Lightweight interface to C/C++ programs



SystemVerilog is the IEEE Std 1800-2005 “IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language”.

SystemVerilog includes the IEEE Std 1364-2001 “IEEE Standard Verilog® Hardware Description Language” by reference.

Several companies donated proprietary content to the Accellera EDA advocacy group, who integrated those Verilog extensions into SystemVerilog and championed the creation of the new IEEE standard.

SystemVerilog Language Extensions

SystemVerilog IEEE 1800											
Literals	Arrays	Classes	Scheduling	Processes	Random	Clocking	Coverage	Interfaces			
integer	packed	method	Observed	always_comb	rand/randc	##delay	covergroup	interface			
logic	unpacked	constructor	Re-active	always_latch	constraint		coverpoint	virtual			
time	slices	static	Re-inactive	always_ff	randomize()		cross	modport			
string	dynamic	this		join_any	solve before		wildcard	export			
pattern	associative	extends	Statements	always_latch	dist with	Assertions	sequence	import			
	queue	super	unique	always_ff	rand_mode	assert	bins	DPI			
Types	Declarations	cast	priority	join_any	constraint_mode	assume	illegal_bins	export			
logic		local	do while	join_none	\$urandom	cover		import			
bit	const		foreach	wait fork	\$urandom_range	property	Hierarchy	context			
byte	alias	Operators	return		randcase	sequence	package	pure			
shortint	type	assignment	break		randsequence	intersect	import				
int	var	wild equality	continue		static	first_match	nesting				
longint		tagged	final		automatic	throughout					
shortreal		overloading	iff		const ref	within					
void		streaming			void	bind					
Verilog-2001											
ANSI C style ports			standard file I/O			(* attributes *)			multi dimensional arrays		
generate			\$value\$plusargs			configurations			signed types		
localparam			'ifndef 'elsif 'line			memory part selects			automatic		
constant functions			@*			variable part select			** (power operator)		
Verilog-1995											
modules			\$finish \$fopen \$fclose			initial			+ = * /		
parameters			\$display \$fwrite			disable			%		
function/tasks			\$monitor			events			>> <<		
always @			'define 'ifdef 'else			wait # @			wire reg		
assign			'include 'timescale			fork-join			integer real		
						time			2D memory		
						packed arrays					

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

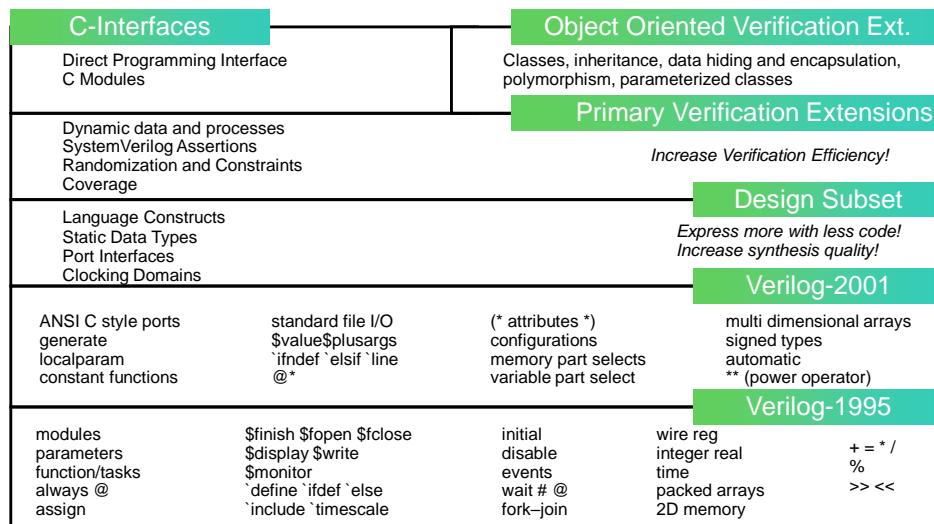


The IEEE Std 1364 standardized in 1995 what was previously the proprietary Verilog hardware description language.

The 2001 update to the Verilog standard added several convenience features, still primarily targeted at hardware design.

Accellera proposed an update to the Verilog standard that added many design convenience features and more than tripled the verification features. This slide illustrates just how much SystemVerilog adds to Verilog. To hopefully minimize confusion, Accellera requested and obtained the new IEEE standard number 1800 for these new features. The SystemVerilog standard incorporates the Verilog standard by reference – there is no such thing as “stand-alone” SystemVerilog.

SystemVerilog Segmentation



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



The SystemVerilog standard incorporates the Verilog standard by reference, rather than repeating it and thus potentially diverging from the Verilog standard.

You can easily segment SystemVerilog into just those parts you need to know for your product development role. The training materials focus first on the designer subset. This subset adds constructs with which you can more clearly and concisely code your RTL design.

In the design verification role, you can take advantage of most of the new verification features, without getting involved with ...

- The C++-like object oriented features; or
- Importing or exporting C routines.

Of course, for advanced verification, you will want to take advantage of the object oriented features, such as overloaded operators, classes, polymorphism, and more ...

... and you may even want to map SystemVerilog functions and tasks to C routines to reuse existing IP written in C.

Convenience and Synthesis Features

More concise RTL code

Enhancements to more clearly specify design intent

- Compact code is more readable and less prone to syntax errors

Fixes to make Verilog easier to use

- Alternatives to parallel/full case directives

Features in this category are synthesizable and expected to be used by designers for block-level design

```
initial
  begin : myBlock
  ...
  for (int i=0; i<12; ++i)
    intArray[i] += i*5;
  ...
end : myBlock
```

```
always_comb
  unique case (sel)
    2'b00: mux_out = in_a;
    2'b01: mux_out = in_b;
    2'b10: mux_out = in_c;
    default: mux_out = 8'hx;
  endcase
```



Several of the design features reduce the verbosity of your code and make it less likely for you to generate errors. The first example has a local variable declaration and an auto increment statement. The second example has a new kind of block that helps to guarantee that the synthesis tool creates the logic you intend it to create.

Data Types

Relaxation of Verilog data type rules

2-state types to describe designs using abstract modeling

User-defined types that can be defined once and used throughout the design

- Enumerated types for design modeling (FSM states, opcodes, etc.)

Supporting constructs for user-defined data types

- Packages to share declarations amongst several modules

```
module test;
    enum logic [1:0] {TRUE = 1, FALSE = 0, NOTSURE = X } answers;
    int error_count;           // 2-state int type init to 0 at time 0
    typedef enum logic[2:0] {IDLE, FIRST, SECOND, FINISH} mystates;
    mystates state, nstate;    // uses user-defined MyStates type
    ...
endmodule : test
```

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



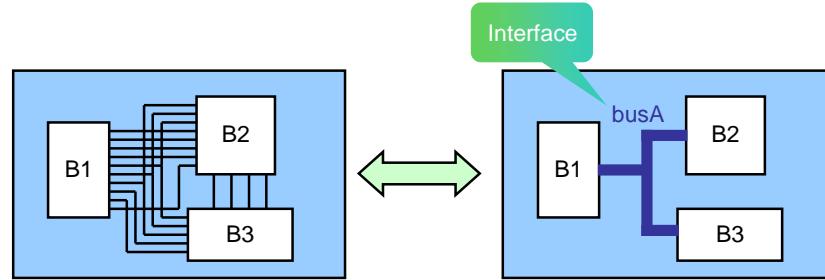
You will see how SystemVerilog:

- Relaxes the rules concerning assignments to variables.
- Provides 2-state types for higher testbench performance.
- Accepts user-defined types including enumerated types.
- Introduces a new package unit for capturing type declarations.

Interfaces and Connectivity

Automatic Port Connection

- Interfaces raise the level of abstraction and simplify design block communication by representing a number of signals as a single port.
- Allow module port directional information and tasks/functions to be defined inside the interface.



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



An interface is a new kind of design block that captures inter-module communication in one place. Rather than declare many ports and signals at each hierarchy level and connect them, you can declare the signals in one interface and declare ports of that interface type, which when connected to the interface, automatically make all those individual signal connections for you.

SystemVerilog Verification Features

Among the verification features that SystemVerilog offers are the following:

- Classes
- Enhanced Scheduling Semantics
- Constrained Randomization
- Interprocess Synchronization
- Clocking Blocks
- Program Blocks
- SystemVerilog Assertions (SVA)
- Data-Oriented Functional Coverage
- Direct Programming Interface (DPI)



This slide gives you an indication of the kinds of things that the second part of this training, “SystemVerilog for Verification”, covers. You use the abstract modeling features primarily for testbenches and for system-level design.

SystemVerilog Assertions (SVA)

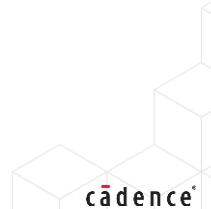
SystemVerilog Assertions (SVA)

- Used to specify and verify behavior of a design
- Can be added anywhere in a design hierarchy
- Ignored by synthesis tools
- Checked during simulation
 - Can also be used to provide functional coverage
 - Can also be used as an input to formal verification tools

```
// immediate assertion
always @(negedge clock)
    assert !(wr_en && rd_en)
        $display ("pass");
    else
begin
    $display("read/write fail");
    err_count++;
    -> rw_err_event;
end
```

```
// concurrent assertion
property abcd;
@(posedge clk)
a ##1 b |=> c ##1 d;
endproperty
```

```
A1: assert property (abcd);
```



The Cadence designed training, “SystemVerilog for Verification”, introduces SystemVerilog assertions. An assertion is a statement that some Boolean property about the design holds. Immediate assertions make that check when executed. Concurrent assertions exist as autonomous processes that can make checks spanning many cycles.

SystemVerilog Coverage

Coverage allows the user to tell how well a design has been tested.

There are two types of functional coverage in SystemVerilog:

- Data-oriented coverage
 - Coverage groups, coverage points and cross products
 - Checks combinations of data values have occurred
- Control-oriented coverage
 - Using SystemVerilog Assertions (SVA)
 - Checks whether sequences of behaviors have occurred

```
// SV coverage group:  
covergroup cg1 @(posedge clk);  
  Addr: coverpoint addr  
  { bins low   = { [0:'h0F], 19 };  
    bins mid[] = { 16, 17, 18 };  
    bins high  = { ['h14:'hFF] }; }  
  AddrXvalid : cross Addr, valid;  
endgroup : cg1
```

```
// SVA functional coverage  
property full2empty;  
  @(posedge clk)  
    fifo_full ##[1:$] fifo_empty;  
endproperty  
C1 : cover property (full2empty);
```



Coverage is a measure of how thoroughly the testbench exercises the design. SystemVerilog has data-oriented functional coverage and control-oriented functional coverage.

Object-Oriented Testbench Design

In a module-based testbench, modifying the functionality usually requires editing the code.

- Inflexible and difficult to maintain and reuse

An object-oriented (class-based) testbench can be modified without editing the original code.

With SV classes you can create a testbench from generic, reusable objects, and then at a later date:

- Add new declarations, constraints and functionality
- Extend, replace, override, disable or remove existing code
- Merge together and separate out objects at will

All this whilst leaving the original testbench unaltered.



This page does not contain notes.

The Need for Methodology

Why methodology?

- Formula for verification success
- Captures best-known practices with minimal improvisation
- Provides consistency and uniformity in the way verification components are developed and used/re-used

Benefits

- Improves productivity
- Encourages reuse
- Easy to maintain

Foresight....

- As the saying goes... "Hindsight is 20/20"
- A proven methodology helps solve unforeseen limitations by going where you have not been before



This page does not contain notes.

Verification Methodology Benefits

Productivity and Automation

- How much manual work can be saved?

Predictability

- Will I meet my deadlines on time?

Quality

- Will my design be free of bugs?

Ease of Use

- How easily can a Test Writer develop tests for an existing verification environment?
- One of the main motivations for moving to SystemVerilog.

Accommodating Unanticipated Requirements

- For example, a component developer cannot predict environment user requirements.

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



This page does not contain notes.

What Is UVM?

- The first standard, open, interoperable, and proven verification re-use methodology for SystemVerilog
- Delivers simulator, verification IP, and high-level language interoperability within and across companies
- Scalable to system-level verification
- Provides additional automation, which is key for enabling reuse and is missing from IEEE1800 SystemVerilog standard
- Developed, maintained, documented, and taught by multiple vendors
 - Does not lock users into a single vendor solution like existing class libraries

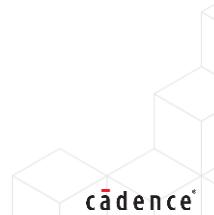


Universal Verification Methodology (UVM) is an Accellera standard, open-source class library for SystemVerilog verification, which not only supplies the basic building blocks for the metric driven verification, but also provides a standard methodology for building and executing verification environments.

UVM is covered in the Cadence SystemVerilog Advanced Verification with UVM training class.

Labs

There are no labs in this appendix.



This page does not contain notes.



cadence®

© Cadence Design Systems, Inc. All rights reserved worldwide. Cadence, the Cadence logo, and the other Cadence marks found at <https://www.cadence.com/go/trademarks> are trademarks or registered trademarks of Cadence Design Systems, Inc. Accellera and SystemC are trademarks of Accellera Systems Initiative Inc. All Arm products are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All MIPI specifications are registered trademarks or service marks owned by MIPI Alliance. All PCI-SIG specifications are registered trademarks or trademarks of PCI-SIG. All other trademarks are the property of their respective owners.