Cadence® Verilog® Language and Simulation

Version 3.4

Lecture Manual  February 18, 2002
# Table of Contents

## Cadence Verilog Language and Simulation

### Chapter 1  Getting Started

- Course Agenda ................................................................................................................. 1-3
- Course Schedule ............................................................................................................... 1-7
- Getting Help ................................................................................................................... 1-13
- Review ........................................................................................................................... 1-15

### Chapter 2  Verilog Applications

- What Is a Hardware Description Language? ................................................................... 2-3
- Why Use an HDL? ........................................................................................................... 2-5
- What is Verilog? .............................................................................................................. 2-7
  - Verilog History .......................................................................................................... 2-9
  - Verilog Applications ..................................................................................................... 2-11
- Levels of Abstraction ....................................................................................................... 2-13
  - Trade-Offs Among the Levels of Abstraction ........................................................... 2-15
  - Verilog-Supported Levels of Abstraction ................................................................ 2-17
  - The Behavioral Level ............................................................................................... 2-19
  - The RTL Level ......................................................................................................... 2-21
  - The Structural Level ............................................................................................... 2-23
  - One Language for All Levels ................................................................................... 2-25
- Summary ........................................................................................................................ 2-27
- Review ........................................................................................................................... 2-29

### Chapter 3  Introduction to Cadence Simulators

- Simulation Algorithms ..................................................................................................... 3-3
- The Time Wheel in Event-Based Simulation .................................................................. 3-5
- Event Simulation of a Verilog Model .............................................................................. 3-7
- The Cadence Verilog Simulators .................................................................................... 3-9
- Interpretive Versus Compiled-Code Simulation ............................................................ 3-11
  - Verilog-XL Can Interpret Behavioral Source .......................................................... 3-13
  - NC-Verilog Precompiles All Source Code ............................................................... 3-15
- Verilog Language Support ............................................................................................ 3-17
- Invoking Verilog-XL and NC-Verilog ............................................................................ 3-19
- Verilog-XL and NC-Verilog Differences ...................................................................... 3-21
- The Simvision Waveform Viewer ................................................................................. 3-25
Chapter 4  A Sample Design

Key Language Features ........................................................................................................ 4-3
The Verilog Module ............................................................................................................. 4-5
Module Ports ....................................................................................................................... 4-7
Module Instances ............................................................................................................... 4-9
A Simple and Complete Example ..................................................................................... 4-11
Device Under Test .............................................................................................................. 4-13
Testbench .......................................................................................................................... 4-15
The Value Change Dump (VCD) ....................................................................................... 4-27
Dumping Signals with $dumpvars ................................................................................. 4-29
Summary ............................................................................................................................ 4-31
Review ............................................................................................................................... 4-33
About Lab 1 ....................................................................................................................... 4-35

Chapter 5  Lexical Conventions in Verilog

White Space and Comments ............................................................................................. 5-3
Integer and Real Numbers ................................................................................................. 5-5
String Constants ................................................................................................................... 5-7
Identifiers ........................................................................................................................... 5-9
  Escaped Identifiers .......................................................................................................... 5-11
Special Language Tokens ................................................................................................. 5-13
  System Tasks and Functions ............................................................................................ 5-15
  Specifying Delay ............................................................................................................. 5-17
Compiler Directives .......................................................................................................... 5-19
  Substituting Text .............................................................................................................. 5-21
  Including Source Text .................................................................................................... 5-23
  Specifying the Timescale ............................................................................................... 5-25
Summary ............................................................................................................................ 5-29
Review ............................................................................................................................... 5-31
Chapter 6  Verilog Data Types and Logic System

The Verilog 4-Value Logic System ................................................................. 6-3
Major Data Type Classes .............................................................................. 6-5
   Nets .............................................................................................................. 6-7
   Registers .................................................................................................... 6-15
   Module Parameters .................................................................................... 6-21
Choosing the Correct Data Type ................................................................. 6-27
   Common Data Type Mistakes ................................................................... 6-29
String Literals ................................................................................................. 6-31
Verilog Memories ........................................................................................... 6-33
   Memory Addressing .................................................................................. 6-35
Summary ........................................................................................................... 6-37
Review .............................................................................................................. 6-39
About Lab 2 ...................................................................................................... 6-41

Chapter 7  Structural Modeling

Structural Modeling ....................................................................................... 7-3
Verilog Primitives .......................................................................................... 7-5
   Some Primitive Pins Are Expandable ..................................................... 7-7
   Conditional Primitives .......................................................................... 7-9
Primitive Instantiation .................................................................................. 7-13
Module Instantiation ...................................................................................... 7-15
Array of Instances ........................................................................................ 7-17
   Illegal Arrays of Instances .................................................................. 7-19
Logic Strength Modeling ............................................................................. 7-21
   The Signal Strength Value System ....................................................... 7-23
   How Verilog Resolves Signal Strengths ............................................... 7-25
Summary ......................................................................................................... 7-27
Review ............................................................................................................ 7-29
About Lab 3 .................................................................................................... 7-31

Chapter 8  Modeling Delay

Modeling Delay .............................................................................................. 8-3
   Lumped Delay ........................................................................................ 8-5
   Distributed Delay .................................................................................... 8-7
   Module Path Delay ................................................................................ 8-9
   Modeling Delay Accurately ................................................................. 8-11
# Chapter 9 Using Compiler Controls

Verilog Model Libraries ................................................................. 9-3  
  Modeling Verilog Libraries ..................................................... 9-5  
  Using Verilog Libraries ......................................................... 9-9  
Producing Case Insensitive Source ............................................. 9-19  
Compiler Directives ................................................................. 9-21  
Defining Text Macros ............................................................... 9-23  
  Pre-defined Text Macros....................................................... 9-25  
Selecting Simulation Delay Mode .............................................. 9-27  
  Selecting Delay Values ....................................................... 9-29  
  Selecting Delay Mode......................................................... 9-31  
  Inertial and Transport Delay Modes ................................. 9-33  
Verilog-XL Module Path Restrictions ....................................... 9-35  
  Verilog-XL Accelerated Objects ........................................ 9-37  
Protecting Source Code with Verilog-XL ................................. 9-39  
  Protecting All of Your Verilog Source ............................. 9-41  
  Protecting Selected Regions in a Source Description .... 9-43  
Coexecuting VHDL and Verilog with NC-Sim ......................... 9-45  
Summary ............................................................................ 9-47  
Review ............................................................................... 9-49  
About Lab 5 ...................................................................... 9-51  

# Chapter 10 Verilog Operators

Sizing and Signing .................................................................. 10-3
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenation Operator</td>
<td>10-5</td>
</tr>
<tr>
<td>Replication Operator</td>
<td>10-7</td>
</tr>
<tr>
<td>Negation Operators</td>
<td>10-9</td>
</tr>
<tr>
<td>Unary Reduction Operators</td>
<td>10-11</td>
</tr>
<tr>
<td>Arithmetic Operators</td>
<td>10-13</td>
</tr>
<tr>
<td>Shift Operators</td>
<td>10-15</td>
</tr>
<tr>
<td>Relational Operators</td>
<td>10-17</td>
</tr>
<tr>
<td>Equality Operators</td>
<td>10-19</td>
</tr>
<tr>
<td>Logical Equality Operator</td>
<td>10-21</td>
</tr>
<tr>
<td>Case Equality (Identity) Operator</td>
<td>10-23</td>
</tr>
<tr>
<td>Bit-Wise Operators</td>
<td>10-25</td>
</tr>
<tr>
<td>Logical Operators</td>
<td>10-27</td>
</tr>
<tr>
<td>Conditional Operator</td>
<td>10-29</td>
</tr>
<tr>
<td>Operator Precedence</td>
<td>10-31</td>
</tr>
<tr>
<td>Summary</td>
<td>10-33</td>
</tr>
<tr>
<td>Review</td>
<td>10-35</td>
</tr>
</tbody>
</table>

**Chapter 11 Behavioral Modeling**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavioral Modeling</td>
<td>11-3</td>
</tr>
<tr>
<td>Procedural Blocks</td>
<td>11-5</td>
</tr>
<tr>
<td>Sequential and Concurrent Blocks</td>
<td>11-7</td>
</tr>
<tr>
<td>Procedural Assignments</td>
<td>11-11</td>
</tr>
<tr>
<td>Blocking and Nonblocking Assignments</td>
<td>11-13</td>
</tr>
<tr>
<td>Procedural Timing Control</td>
<td>11-17</td>
</tr>
<tr>
<td>Simple Delay</td>
<td>11-19</td>
</tr>
<tr>
<td>Edge-Sensitive Control</td>
<td>11-21</td>
</tr>
<tr>
<td>Level-Sensitive Control</td>
<td>11-25</td>
</tr>
<tr>
<td>Missing Timing Controls</td>
<td>11-27</td>
</tr>
<tr>
<td>Timing Control Examples</td>
<td>11-29</td>
</tr>
<tr>
<td>Intra-Assignment Timing Control</td>
<td>11-33</td>
</tr>
<tr>
<td>Conditional Statements</td>
<td>11-37</td>
</tr>
<tr>
<td>if and if-else Statements</td>
<td>11-39</td>
</tr>
<tr>
<td>case Statements</td>
<td>11-41</td>
</tr>
<tr>
<td>Looping Statements</td>
<td>11-43</td>
</tr>
<tr>
<td>The for Loop</td>
<td>11-45</td>
</tr>
<tr>
<td>The forever Loop</td>
<td>11-47</td>
</tr>
<tr>
<td>The repeat Loop</td>
<td>11-49</td>
</tr>
<tr>
<td>The while Loop</td>
<td>11-51</td>
</tr>
<tr>
<td>Continuous Assignments</td>
<td>11-53</td>
</tr>
<tr>
<td>Summary</td>
<td>11-59</td>
</tr>
</tbody>
</table>
Chapter 12  Debugging with the Command-Line Interface

Overview ................................................................. 12-3
Entering the Interactive Mode .............................. 12-5
Command Overview .............................................. 12-9
Setting the Debug Scope ...................................... 12-17
Examining the Design .......................................... 12-21
Traversing the Design .......................................... 12-25
Reading Signal Values ......................................... 12-29
Investigating Signal Drivers ............................... 12-33
Setting Signal Values .......................................... 12-37
Setting Breakpoints ........................................... 12-41
Running the Simulation .................................... 12-45
Tracing Simulation Activity .............................. 12-49
Monitoring Signal Values ................................. 12-51
Capturing Waveform Data ................................. 12-55
Using the Command History List .................. 12-59
Executing a Command Script ......................... 12-63
Saving, Resetting, and Restarting Simulation ...... 12-67
Exiting the Simulator ........................................... 12-71
Other Miscellaneous Commands ................. 12-75
Summary ................................................................. 12-79
Review ................................................................. 12-81

Chapter 13  Debugging With the Graphical User Interface

Invoking the Graphical Environment ............... 13-3
SimControl ............................................................... 13-5
The Post-Processing Environment .................. 13-7
SimControl Menu and Tool Bars .................. 13-9
The SimControl Source Browser ................. 13-11
Selecting Objects ................................................ 13-13
Setting Breakpoints ........................................... 13-15
The Navigator ......................................................... 13-17
Watch Windows ...................................................... 13-19
The Signal Flow Browser ............................... 13-21
The Cycle View ....................................................... 13-25
Schematic Windows ........................................... 13-27
Chapter 17  High Level Verilog Constructs

Verilog Functions and Tasks ................................................................. 17-3
  Verilog Functions............................................................................. 17-5
  Verilog Tasks................................................................................. 17-15
Named Blocks .................................................................................. 17-25
Finite State Machines....................................................................... 17-27
  Explicit FSM ................................................................................. 17-29
  Implicit FSM ............................................................................... 17-31
Summary ......................................................................................... 17-33
Review ............................................................................................ 17-35
About Lab 10 .................................................................................... 17-37
About Lab 11 .................................................................................... 17-39

Chapter 18  User-Defined Primitives

What is a Verilog Primitive?............................................................... 18-3
What is a User-Defined Primitive?.................................................. 18-5
Why Define My Own Primitive?....................................................... 18-7
How Do I Define My Own Primitive?............................................... 18-9
  Defining the Truth Table ............................................................ 18-11
  Truth Table Symbols.................................................................... 18-13
Examples of UDPs.......................................................................... 18-15
  Combinational Example: 2-1 Multiplexer .................................. 18-17
  Combinational Example: Full Adder .......................................... 18-19
  Level-Sensitive Sequential Example: Latch ....................... 18-21
  Latch with Enable and Reset ..................................................... 18-23
  Edge-Sensitive Sequential Example: D Flip-Flop .................... 18-25
  D Flip-Flop With Synchronous Reset ....................................... 18-27
  D Flip-Flop Using a Notifier ...................................................... 18-29
Summary ......................................................................................... 18-31
Review ............................................................................................ 18-33
About Lab 12 .................................................................................... 18-35
Chapter 19  Modeling for Synthesis

Modeling Combinational Logic..................................................................................... 19-3
Continuous Assignments ......................................................................................... 19-5
Asynchronous Procedural Blocks ............................................................................. 19-7
Sensitivity Lists........................................................................................................ 19-9
Verilog Registers ................................................................................................... 19-11
Conditional Statements ......................................................................................... 19-15
Synthesis case Directives ..................................................................................... 19-25
Inferring Latches ..................................................................................................... 19-27
Functions ................................................................................................................ 19-31
Tasks ...................................................................................................................... 19-33
Complex Operators .............................................................................................. 19-35
Resource Sharing ................................................................................................. 19-37
Modeling Sequential Logic.......................................................................................... 19-39
Synchronous Procedural Blocks ............................................................................ 19-41
Verilog Registers ................................................................................................... 19-43
Blocking and Non-Blocking Assignments............................................................. 19-47
Modeling Resets ................................................................................................... 19-49
Finite State Machines............................................................................................. 19-51
Example Explicit FSM ........................................................................................... 19-53
Synthesis FSM Directives .................................................................................... 19-55
Additional Synthesis Directives ............................................................................ 19-59
Unsynthesizable Verilog Constructs ......................................................................... 19-61
Summary .................................................................................................................. 19-63
Review ....................................................................................................................... 19-65
About Lab 13 ............................................................................................................ 19-67
About Lab 14 ............................................................................................................ 19-69

Chapter 20  Modeling for Performance

Verilog-XL Simulation Algorithms ............................................................................. 20-3
Selecting Verilog-XL Optimizations ....................................................................... 20-5
Selecting Turbo Optimizations .............................................................................. 20-7
Selecting Twin Turbo Optimizations ..................................................................... 20-9
Combining Performance Options .......................................................................... 20-11
NC-Verilog Simulation Performance ....................................................................... 20-13
Optimizing XL Algorithm Performance .................................................................. 20-15
Items Accelerated by XL ....................................................................................... 20-17
Items not Accelerated by XL .................................................................................. 20-19
## Table of Contents

- Optimizing Behavioral Performance ................................................................. 20-23
  - Minimize File Output System Tasks ................................................................. 20-25
  - Keep Operations Vectored .................................................................................. 20-27
- Command-Line Simulation Control ............................................................................ 20-29
- Code Coverage ............................................................................................................. 20-33
  - Verilog-XL Statement Coverage ........................................................................... 20-35
  - NC-Sim Code Coverage ......................................................................................... 20-41
- Summary ...................................................................................................................... 20-47
- Review ......................................................................................................................... 20-49
- About Lab 15 ............................................................................................................... 20-51

### Chapter 21 Annotating SDF Timing

- Timing Annotation ......................................................................................................... 21-3
  - Timing Data Flow ...................................................................................................... 21-5
- Delay Calculators .......................................................................................................... 21-7
- Standard Delay Format ................................................................................................. 21-9
  - SDF Example ............................................................................................................ 21-11
- The SDF Annotator ....................................................................................................... 21-13
  - Running the SDF Annotator .................................................................................. 21-15
- Summary ...................................................................................................................... 21-17
- Review ......................................................................................................................... 21-19
- About Lab 16 ............................................................................................................... 21-21

### Chapter 22 Overview of the NC-Verilog Simulator

- Interleaved Native Compiled Code ............................................................................. 22-3
- Library-Based Design Data Management ..................................................................... 22-7
- Running the NC-Verilog Simulator ............................................................................... 22-9
  - The NC-Verilog Tool Flow .................................................................................... 22-11
- Summary ...................................................................................................................... 22-13
- Review ......................................................................................................................... 22-15

### Chapter 23 Getting Started with the NC-Verilog Simulator

- Creating the cds.lib File ............................................................................................... 23-3
- Creating the hdl.var File ............................................................................................. 23-5
  - Variables in the hdl.var File .................................................................................. 23-7
- Creating the Optional setup.loc File .......................................................................... 23-9
- Using the nchelp Utility .............................................................................................. 23-11
- Using the ncls Utility .................................................................................................. 23-13
### Chapter 24 Transitioning to the NC-Verilog Simulator

- Transitioning a Design to the NC-Verilog Simulator .................................................. 24-3
- Using the ncvilog Command ....................................................................................... 24-5
  - Introducing ncvilog............................................................................................... 24-7
  - Understanding ncvilog......................................................................................... 24-15
  - Simulating with ncvilog....................................................................................... 24-25
- Manually Transitioning a Design Using ncprep .......................................................... 24-39
  - What Does ncprep Do? ....................................................................................... 24-41
  - Should You Use ncprep? ..................................................................................... 24-43
  - How Does ncprep Work? ................................................................................... 24-45
  - Customizing the Environment ............................................................................ 24-47
- Manually Transitioning a Design ................................................................................. 24-49
  - Modifying the Source for Best Results............................................................... 24-51
  - Defining the Libraries and Variables .................................................................... 24-53
  - Invoking the NC-Verilog Tools ............................................................................. 24-55
- Making the Most of Incremental Compilation............................................................ 24-57
  - Incremental Compilation with ncvilog .............................................................. 24-59
  - Incremental Compilation in the Library-Based Mode .......................................... 24-61
  - Things to Be Aware Of ........................................................................................ 24-65
- Comparing ncvilog and the Library-Based Mode ....................................................... 24-67
- Summary .................................................................................................................... 24-69
- Review ....................................................................................................................... 24-71
- About Lab 18 ............................................................................................................. 24-73

### Chapter 25 Introduction to Cycle Simulation

- Introducing Cycle Simulation .................................................................................... 25-3
  - How Does it Work? ............................................................................................... 25-5
- Taking Advantage of Cycle Simulation ..................................................................... 25-7
- Adopting Cycle Simulation ...................................................................................... 25-9
- Handling Methodology Issues ............................................................................... 25-11
- Cycle Simulation in Your Design Flow .................................................................... 25-13
- Summary ................................................................................................................... 25-15
- Review ....................................................................................................................... 25-17
Chapter 26 Switch-Level Modeling

Verilog Switch Primitives................................................................. 26-3
Switch Instantiation ................................................................. 26-5
Switch Delay............................................................................. 26-7
Drive Strength.......................................................................... 26-9
Strength Reduction................................................................. 26-11
Switch-Level Networks ............................................................... 26-13
The Switch-XL Algorithm.......................................................... 26-15
Selecting the Switch-XL Algorithm ......................................... 26-17
The Switch-XL Strength Model.................................................. 26-19
Switch-XL Strength Reduction.................................................. 26-21
How Switch-XL Works .............................................................. 26-23
Timing Model Conversion......................................................... 26-25
Summary .................................................................................. 26-27
Review ...................................................................................... 26-29

Appendix A Glossary
Chapter 1: Getting Started

Objectives

In this section you will:

- Review the Verilog HDL and Cadence Verilog simulator topics
- Review the schedule
- Learn where to obtain additional help
Getting Started

This section presents the course agenda and schedule, and how to obtain help with Cadence software.
Course Agenda

During this course, you will gain knowledge on the Verilog HDL, including:

- Verilog applications
- Verilog language components
- Structural-level modeling and simulation
- Behavioral modeling and simulation
- Delay timing features and specifications
- Introduction to Verilog testbenches, including
  - Stimulus and control specification
  - Response generation and verification
- Tasks and functions
- User-defined primitives
- Verilog modeling style for synthesis
Course Agenda
Course Agenda

During the Verilog simulator portion of the course, you will gain knowledge on:

- Compiling and simulating your design
- Using source libraries
- Debugging with the Verilog-XL command-line interface
- Debugging with the NC-Verilog Tcl Interface
- Debugging with the Graphical User Interface (GUI)
- Delay calculation and annotation
- Modeling for simulation performance
- Compiling and simulating your design with the NC-Verilog simulator
- Transitioning your design environment to the NC-Verilog simulator
Course Schedule

Day 1
- Getting Started
- Verilog Applications
- Introduction to Cadence Simulators
- Sample Design
- Lexical Conventions
- Data Types

Day 2
- Structural Modeling
- Modeling Delay
- Verilog-XL Compilation and Libraries
- Verilog Operators
- Behavioral Modeling
Course Schedule

Day 3

- Debugging with the Command Line Interface
- Debugging with the Graphical User Interface
- Support for Verification
- Introduction to Verilog Testbenches
- Modeling Memories

Day 4

- High Level Constructs
- User Defined Primitives
- Modeling for Synthesis
- Modeling for Performance
- Annotating SDF Timing
Course Schedule (continued)
Course Schedule

Day 5

- Introduction to the NC-Verilog Simulator
- Using the NC-Verilog Simulator
- Transitioning to NC-Verilog

Lab17
Lab18
Getting Help

You can get help with Cadence software from the following sources:

- You can enter the `-help` command-line option when you invoke the tool. This displays brief help about available command-line options.
- The NC-Verilog simulator textual interface offers a `help` command. This displays brief help about interactive commands.
- The graphical interface offers a Help button and context-sensitive help.
- The Cadence online documentation library fully documents the software.

If you have a Cadence software support service agreement you can also:

- Browse the knowledge database: [http://www.cadence.com/sourcelink](http://www.cadence.com/sourcelink)
- Contact the Customer Response Center (CRC)
  - `support@cadence.com`
  - 1-877-237-4911
Getting Help

Online Help

Cadence online reference manuals and help files for each product install automatically when you install the product. All of these documents are part of the Cadence online documentation library. You can access this material online in two ways:

- You start the documentation server from a UNIX shell (enter cdsoct) and select from the Cadence reference manuals and help files installed with each product
- You can view relevant product information by selecting the Help button on the graphical user interface
- You can retrieve reference manuals from the installation in PDF format and print them using Adobe Acrobat Reader

Personal Contact

- If you cannot find the necessary information using any of the above methods, you can call the Customer Response Center at 1-877-237-4911 or send your request to support@cadence.com.
Review

1. What are the first three ways you should use to get help on understanding how to use a software tool command option?
Review

1. What are the first three ways you should use to get help on understanding how to use a software tool command option?

   Cadence recommends that you take the following actions, in this order, to get help:
   Use the help facility that comes with each tool. On the command line this often takes the form of a `-help` option or, if using a GUI, click on the Help button.
   Use the online documentation. For the Verilog-XL and NC-Verilog Help, enter: `cdsdoc`
   Call the Customer Response Center at 1-877-237-4911
Objectives

In this section you will learn:

- What is a Hardware Description Language (HDL)?
- Why you would use an HDL
- The Verilog HDL history and application
- What "levels of abstraction" means
Verilog Applications

This section defines what an HDL is, explains why you would use an HDL, and then segues from HDLs in general to Verilog in particular, its history, and what you might use it for. It then describes the various levels of model abstraction, and points out that you can use Verilog to model at all of these levels.
What Is a Hardware Description Language?

A Hardware Description Language (HDL) is a high-level programming language that offers special constructs with which you can model microelectronic circuits.

These special language constructs permit you to:

- Describe the operation of a circuit at various levels of abstraction
  - The behavior of a circuit
  - The function of a circuit
  - The structure of a circuit
- Describe the timing of a circuit
- Express the concurrency of circuit operation
What Is a Hardware Description Language?

HDLs share several features in common:

- An HDL usually contains both low-level and high-level constructs for modeling hardware at multiple levels of abstraction
- An HDL permits you to describe the timing characteristics and requirements of the hardware
- With an HDL you can express the concurrency of the hardware operation

Software programming languages typically have no concept of time. In hardware, there are delays associated with going from an input to an output. An HDL allows you to model these delays because it has a concept of time.
Why Use an HDL?

There are several benefits to using an HDL to describe your design:

- An HDL facilitates a top-down design methodology using synthesis
  - You can design at a high implementation-independent level
  - You can delay decisions on implementation details
  - You can easily explore design alternatives
  - You can solve architectural problems before implementation
  - You can automate mapping of your high-level description to a technology-specific implementation

- An HDL provides greater flexibility
  - You can re-use earlier design components
  - You can move your design between multiple vendors and tools

- An HDL permits you to take advantage of mature software design practices
  - You can more quickly capture your design intent
  - You can more easily manage your design data
Why Use an HDL?
What is Verilog?

Verilog is a Hardware Description Language:

- You can describe digital electronic systems at multiple levels of abstraction
  - Behavioral
  - Functional
  - Structural
- You can model the timing of the systems
- You can express the concurrency of the system operation
- You can test the systems
What is Verilog?
Verilog History

Engineers have simulated digital circuits for several decades.

- Initially to generate data for automated manufacturing test of digital circuits
- Later for design verification prior to implementation

This is a brief history of the Verilog HDL:

1980’s  Gateway Design Automation developed Verilog
1990    Cadence acquired Gateway
1991    Cadence released Verilog to the public domain.
        Open Verilog International (OVI) formed to:
        — Evolve and maintain Verilog
        — Promote the use of Verilog
1995    IEEE ratified the Verilog LRM (Std. 1364)
2001    IEEE updated the Verilog LRM
What is Verilog? (continued)

Verilog History

This course teaches to the IEEE Std. 1364-1995.
Verilog Applications

The Verilog HDL is used by:

- System architects doing high level system simulations
- Verification engineers writing advanced tests for all levels of simulation
- ASIC and FPGA designers writing RTL code for synthesis
- Library developers describing ASIC or FPGA cells, or higher level components
What is Verilog? (continued)

Verilog Applications
Levels of Abstraction

At each level of abstraction, you can describe a system as a group of hierarchical models in varying amount of detail. EDA tools facilitate this process.

- **Behavioral representation**
  - Verilog simulation, behavioral synthesis

- **Functional representation**
  - Verilog simulation, logic synthesis

- **Structural representation**
  - Verilog simulation
  - static functional analysis
  - static timing analysis
  - place & route

- **Physical representation**
  - Spice simulation
  - design rule checking
  - parasitic analysis
Levels of Abstraction
Trade-Offs Among the Levels of Abstraction

Each level of abstraction permits modeling at a higher or lower level of detail. More detail means more work for you and the simulator.
Levels of Abstraction (continued)

Trade-Offs Among the Levels of Abstraction
Verilog-Supported Levels of Abstraction

You can describe the operation of a circuit at various levels of abstraction. The Verilog HDL supports three main levels of abstraction:

- **Behavioral**
  - Describes a system by the flow of data between its functional blocks
  - Defines signal values when they change

- **Register Transfer Level (RTL) or Functional**
  - Describes a system by the flow of data and control signals between and within its functional blocks
  - Defines signal values with respect to a clock

- **Structural**
  - Describes a system by connecting predefined components
  - Uses technology-specific, low-level components when mapping from an RTL description to a gate-level netlist, such as during synthesis
Levels of Abstraction (continued)

Verilog-Supported Levels of Abstraction

Designers use all three levels of abstraction:

- Designers first model functional blocks at the behavioral level to promote design productivity and simulation performance
- Designers then fill in the functional details required by synthesis tools
- Synthesis library developers usually model cell components at the structural level

The Verilog HDL provides limited support for modeling at the very high (algorithmic) and very low (transistor) levels.
The Behavioral Level

The behavioral level describes the behavior of a design without implying any specific internal architecture:

- You use high level constructs, such as @, case, if, repeat, wait, while
- You can use any behavioral construct of the HDL in your testbench
- Synthesis tools accept only a limited subset of these behavioral constructs

This behavioral model defines the behavior of the design as seen at its ports:

```
module pipe ( out, in, clk );
output out; reg out;
input  in, clk;
    always @(in)
      @(posedge clk)
        out <= repeat(2) @(posedge clk) in;
endmodule
```
Levels of Abstraction (continued)

The Behavioral Level

This behavioral model defines the behavior of the design as seen at its ports. The behavior of the design is to wait for the input port value to change, sample the input port value on the next rising clock edge, and present that value to the output port on the third rising clock edge. This model does not define how the hardware would execute that behavior.
The RTL Level

The RTL (functional) level describes the design architecture in sufficient detail that a synthesis tool can construct the circuit.

This functional model defines three storage elements and their assignments:

```
module pipe (out, in, clk);
output out; reg out;
input  in, clk;
  reg one, two;
  always @(posedge clk) begin
    out <= two;
    two <= one;
    one <= in;
  end
endmodule
```
Levels of Abstraction (continued)

The RTL Level

The difference between a behavioral model and an RTL model is not always clear. Many people use the RTL level to mean the subset of the behavioral HDL constructs that the synthesis tool accepts. For an RTL model, you declare all registers and define their operation as a response to a clock edge.

Note that the RTL model provides sufficient architectural detail that the synthesis tool can construct a circuit.
The Structural Level

Synthesis tools produce a purely structural design description.

The structural level is also appropriate for small library components:

- You can use built-in Verilog primitives, such as the **and** gate
- You can describe your own User Defined Primitives (UDPs)

This structural model instantiates predefined library components:

```verilog
module pipe ( out, in, clk );
output out;
input in, clk;
    FD1 one_reg(.Q(one), .D(in ), .CP(clk));
    FD1 two_reg(.Q(two), .D(one), .CP(clk));
    FD1 out_reg(.Q(out), .D(two), .CP(clk));
endmodule
```
Levels of Abstraction (continued)

The Structural Level
One Language for All Levels

Designers usually mix levels of abstraction within a simulation:

- RTL and gate-level library components
- RTL functional submodule descriptions
- Structural system netlist
- Behavioral system testbench
Levels of Abstraction (continued)

One Language for All Levels
Summary

In this section you learned:

- What is a Hardware Description Language (HDL)?
- Why you would use an HDL
- The Verilog HDL history and application
- What "levels of abstraction" means
Summary
Review

1. What is a hardware description language (HDL)?
2. What is a primary advantage to using an HDL?
3. Who "owns" the Verilog HDL?
4. What levels of abstraction does the Verilog HDL most easily support?
5. True or false: The structural level of abstraction contains more design detail, which helps the simulator to simulate the design more quickly.
Review

1. What is a hardware description language (HDL)?

   An HDL is a programming-like language that describes hardware. It needs to describe the design structure, functionality, and timing, and needs to express the concurrency of design operation.

2. What is a primary advantage to using an HDL?

   An HDL allows a designer to capture the design intent at multiple levels of abstraction. Designer efficiency increases at higher levels of design abstraction.

3. Who "owns" the Verilog HDL?

   The Verilog HDL is IEEE standard 1364.

4. What levels of abstraction does the Verilog HDL most easily support?

   The Verilog HDL most easily supports the behavioral, functional (RTL), and structural levels of abstraction. It offers very little support for architectural analysis of a design, and no support for design at the physical level.

5. True or false: The structural level of abstraction contains more design detail, which helps the simulator to simulate the design more quickly.

   Simulation time is proportional to design detail. The higher levels of abstraction (with less detail) simulate more quickly.
Chapter 3: Introduction to Cadence Simulators

Objectives

In this section you will learn about:

- Logic simulation
- Running the Verilog-XL and NC-Verilog simulators
- Probing and displaying waveforms
Introduction to Cadence Simulators

This section describes event simulation in general, then introduces the Cadence Verilog-XL and NC-Verilog simulators; how you invoke them, what they do, what they support, and how they are different from each other. It then goes on to introduce the waveform display tool and explains how to save signal transition data for the display.
Simulation Algorithms

There are three broad categories of simulation algorithms:

- Time-based (used by SPICE simulators)
- Event-based (used by the Cadence Verilog-XL and NC-Verilog simulators)
- Cycle-based (used by the SpeedSim cycle-based simulator)
Simulation Algorithms

Time-based simulation algorithms evaluate the entire circuit on a periodic basis. These algorithms are suitable for simulation of analog circuits, but are inappropriate for simulation of digital circuits having very little activity at any given time step.

Event-based simulation algorithms process only the changes in circuit state. The simulation propagates values forward, through the circuit, in response to input pin events or autonomous event generators (such as clocks). These algorithms efficiently simulate digital circuits, especially circuits in which events do not propagate far.

Cycle-based simulation algorithms evaluate activated portions of the circuit when a trigger input changes. A trigger is any input that can immediately or eventually cause an output change. These algorithms efficiently simulate synchronous circuits, but are inappropriate for circuits with components that internally generate their own events, such as clocks, one-shots, and phase-locked loops.
The simulator creates the initial queues upon compiling the data structures.

The simulator processes all events on the current queue, then advances.

The simulator moves forward through time, never backward.

A simulation time queue represents concurrent hardware events.
The Time Wheel in Event-Based Simulation

The simulator starts at simulation time 0.

The simulator processes all events on the current time queue, then advances to the next queue.

While processing events in the current queue, the simulator can add events to the current and future queues.

The interval between time queues can be as low as the simulation time precision and as large as software and hardware limitations permit.

The total number of future time queues varies during simulation, and can be as low as 0 (at the end) and as numerous as software and hardware limitations permit.
Event Simulation of a Verilog Model

Event simulation of Verilog designs takes the following steps:

1. Compilation:
   - The simulator reads the design description, processes compiler directives, and builds a data structure that defines the design hierarchy.
   - This step is sometimes separated into two steps: compilation and elaboration.

2. Initialization:
   - The simulator initializes module parameters, sets other storage elements to the unknown (X) state, and sets undriven nets to the high-impedance (Z) state. When simulation commences at time zero, the simulator propagates these changes and executes the statements in each `initial` and `always` block up to a timing control.

3. Simulation:
   - The simulator processes events on the current time queue. This can add more events to the current and future time queues.
   - The simulator processes all events on the current time queue, then advances simulation time to the next time queue.
   - The simulator terminates when no future events exist.
Event Simulation of a Verilog Model
The Cadence Verilog Simulators

The Cadence Verilog-XL and NC-Verilog simulators are event-driven logic simulators that read Verilog HDL and simulate the description to reflect the behavior of real hardware.

The IEEE Std. 1364-1995 defines the event-driven scheduling semantics that both simulators comply with.
The Cadence Verilog Simulators

You can use the Cadence Verilog-XL and NC-Verilog simulators to:

- Determine feasibility of new design ideas
- Try out more than one approach to a design problem
- Verify functionality
- Identify design errors
Interpretive Versus Compiled-Code Simulation

This subsection discusses interpretive and compiled-code simulation:

- Verilog-XL can interpret behavioral statements during runtime
- NC-Verilog precompiles all source code before runtime
Interpretive Versus Compiled-Code Simulation
Verilog-XL Can Interpret Behavioral Source

Verilog-XL simulates Verilog source code as follows:

1. It reads your Verilog descriptions, checking for correct syntax and semantics, and implementing compiler directives.

2. It compiles the design into an intermediate format in memory, assembling all simulation objects into a hierarchy (the design data structure). Each element in the source code has some representation; you can access each element in the data structure.

3. It determines the simulation time precision, and constructs a time data structure (the time wheel) in memory for queueing events.

4. It reads, schedules, and eventually executes each statement.

Verilog-XL uses several acceleration algorithms to accelerate the simulation of various levels of abstraction.

Each time you rerun Verilog-XL, it must repeat all these steps.

Verilog-XL maintains an interpretive engine. When you enter interactive mode, you can enter Verilog behavioral statements at the keyboard or with a script. You cannot interactively create additional simulation objects.
Interpretive Versus Compiled Simulation (continued)

Verilog-XL can Interpret Behavioral Source

The Verilog-XL simulator was developed in parallel with the Verilog HDL, so it has become the defacto standard of how Verilog HDL should behave during simulation.

The Verilog-XL simulator simulates different levels of abstraction well, as it incorporates several acceleration algorithms. These include the Turbo algorithm, the XL algorithm, and the Switch-XL algorithm. This course will describe these algorithms in more detail later.
NC-Verilog Precompiles All Source Code

NC-Verilog fully precompiles Verilog source code into machine instructions:

1. **ncvlog** compiles your Verilog source files, checking for correct syntax and semantics, and implementing compiler directives. It generates intermediate data.

2. **ncelab** elaborates the design, constructing a design data structure, implementing compiler directives, and producing executable code. Elements in the source code may be optimized away unless you restrict the optimization. It generates intermediate data.

3. **ncsim** invokes a simulation kernel that loads your design data structure, constructs queues of events (the time wheel) and schedules and executes the machine executable code for each event. Some events may be collapsed (never scheduled) unless you restrict the optimization.

NC-Verilog uses a single simulation kernel for all compiled source code.

Each time you rerun NC-Verilog, it repeats only the steps needed to update your design.

NC-Verilog incorporates the Tool Command Language (Tcl). When you enter interactive mode, you can enter the standard Tcl commands and the NC-Verilog Tcl command extensions at the keyboard or with a script. You cannot interactively enter Verilog behavioral statements.
Interpretive Versus Compiled Simulation (continued)

NC-Verilog Precompiles All Source Code

The NC-Verilog simulator was developed more recently, but has been made to simulate Verilog descriptions as closely as possible to the way Verilog-XL does.

The NC-Verilog simulator only uses one simulation kernel to simulate all levels of abstraction. Therefore, there is no penalty for mixing levels of abstraction throughout your design. However, gate-level code may simulate less efficiently than higher levels of abstraction, due to the increased number of events generated.

The NC-Verilog simulator is capable of incrementally compiling source files, thereby reducing overall compilation time.

When you enter interactive mode, you can interact with your design and simulation using Tcl commands and a set of Tcl extensions specific to NC-Verilog. These are covered later in the course.
Verilog Language Support

The Verilog-XL and NC-Verilog simulators are intended to support the full Verilog language as described by the IEEE Std. 1364-1995.

You can design to the:

- OVI 2.0 Verilog Language Reference Manual, with these exceptions:
  - Attributes
  - Function output or inout arguments

Cadence intends the Verilog-XL simulator to remain dedicated to the 1995 standard. Please refer to the online document Cadence NC-Verilog Simulator Product Notes for up to date information regarding new capabilities of the NC-Verilog simulator.
Verilog Language Support

The following OVI 2.0 structures are not supported:

- Attributes
  Attributes are properties or characteristics of objects in a Verilog description.

- Function **output** or **inout** arguments
  OVI 2.0 allows **output** and **inout** argument values to be passed from functions.
Invoking Verilog-XL and NC-Verilog

Invoke the Cadence Verilog-XL or NC-Verilog simulators from a command shell:

```
verilog filenames and options
ncverilog filenames and options
```

For example:

```
verilog mux.v test.v -s +gui
ncverilog mux.v test.v -s +gui
```

Here are some commonly used options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c</td>
<td>Compile-only (do not simulate)</td>
</tr>
<tr>
<td>-f filename</td>
<td>Provide a file of command-line arguments</td>
</tr>
<tr>
<td>-q</td>
<td>Run in quiet mode</td>
</tr>
<tr>
<td>-s</td>
<td>Suspend simulation (wait for user input)</td>
</tr>
<tr>
<td>+gui</td>
<td>Use the GUI</td>
</tr>
</tbody>
</table>

Verilog-XL, by default, logs all terminal output to the `verilog.log` log file.

NC-Verilog, by default, logs all terminal output to the `ncverilog.log` log file.
Invoking Verilog-XL and NC-Verilog

Enter `verilog` or `ncverilog` in a command window to start the Cadence Verilog-XL or NC-Verilog simulators (respectively), followed by the names of your design files and any command-line options (in any order).

All minus (-) options are single-character options.

All plus (+) options are multi-character options.

The simulators accept user-defined plus options. You must be careful to correctly spell the plus options, as the simulator will simply assume that a misspelled option is one of your own.

Use the `-c` option to compile the design, to check syntax and semantics, but to not start simulation.

Use the `-f` option to provide a file of command-line arguments. This reduces repetitive typing of the entire command line during debug.

Use the `-q` option to run in quiet mode. This mode suppresses the copyright banner and some simulation statistics the simulator outputs upon terminating simulation.

Use the `-s` option to suspend simulation immediately after the simulator compiles and links your design. The simulator enters interactive mode, waiting for user input. To continue simulation, enter the period (.) character with Verilog-XL, or "run" with NC-Verilog.

Use the `+gui` option to start the simulator with the graphical user interface (GUI). You may want to also use the `-s` option when running with the GUI, as the simulator will otherwise start simulation without waiting for user input.
Verilog-XL and NC-Verilog Differences

The simulators differ in these most user-evident ways:

- The NC-Verilog simulator separates the compile, elaborate, and simulate steps, and performs only the steps necessary to update the design.
- The NC-Verilog simulator simulates much faster than the Verilog-XL simulator.
- The NC-Verilog simulator by default provides no interactive access to the design data structure.

Use the `+nclinedebug` option to preserve line debug capability.

Use the `+ncaccess+RWC` option to preserve read (R), write (W), and connectivity (C) access.

- The simulators provide different textual interfaces:
  - The Verilog-XL simulator interactively accepts Verilog behavioral statements.
  - The NC-Verilog simulator interactively accepts Tcl commands.
Verilog-XL and NC-Verilog Differences

The NC-Verilog simulator separates the compile, elaborate, and simulate steps, and performs only the steps necessary to update the design. It compiles only the modified files, files the modified files depend upon, and files dependent upon the modified file. It writes new native machine code only for behavioral statements in the modified files. It re-annotates timing only if the SDF file has been modified.

The NC-Verilog simulator, by default, runs in performance mode. As a result of performance optimizations, it by default provides no interactive access to the design data structure.

Use the `+nclinedebug` option to preserve line debug capability. With this option you can set breakpoints on lines of source code and step the simulator through lines of source code.

Use the `+ncaccess` option to set the default access capability for all objects. The default is no access. Use `+ncaccess+args` to turn on access and `+ncaccess-args` to turn off access. The `args` can be any combination of the letters R (read), W (write), and C (connectivity tracing).

NC-Verilog performance optimizations may unmask pre-existing race conditions in your Verilog code. Thus it is possible (though very unlikely) that your use of the `+ncaccess+` option could mask the race condition, making it very difficult to debug.
The SimVision Waveform Viewer

Start a SimVision Design Browser window by entering `simvision`.

Use a Design Browser window to select and send objects to a Waveform display.
The SimVision Waveform Viewer

A Design Browser window contains the following:

- The **Title Bar** informs you that this is a SimVision Design Browser window, and displays the window number. Design Browser windows are numbered incrementally.

- The **Menu Bar** offers full access to all SimVision Design Browser operations.

- The **Tool Bar** includes buttons to open a database, copy a selection to the clipboard, select all lower scopes, and send all selected objects to the target (or a new) Design Browser, Waveform, or Expression Calculator window.

- The **Scope Tree Pane** displays the design hierarchy with an appropriate icon for each type of scope (module, function, named block, task, etc.). You ascend and descend the hierarchy by clicking a node. You can right-select an icon to pop-up a menu of actions applicable to the node. You can search for scopes up or down the hierarchy, and filter scopes from the pane using the * and ? wildcards.

- The **Signals/Variables Pane** displays signals and variables of the selected scope with an appropriate icon for each type of signal (input, output, inout, internal, transaction, etc.). You can right-select an icon to pop-up a menu of actions applicable to the signal. You can search for signals up or down the hierarchy, and filter signals from the pane using the * and ? wildcards.

- At the bottom are a **Set As Target Window** button and **Message** and **Status** fields.
Start with a SimVision Waveform window by entering `simvision -waves`.
The Simvision Waveform Viewer

A Waveform window contains the following:

- **The Title Bar** informs you that this is a SimVision Waveform window, and displays the window number. Waveform windows are numbered incrementally.

- **The Menu Bar** offers full access to all SimVision Waveform View operations.

- **The Primary Tool Bar** includes buttons to open a database, move, copy, paste, and delete selected objects, group or bus (or ungroup or unbus) selected objects, add selected objects to this window, and send selected objects to the target Design Browser, Waveform, or Expression Calculator windows. The Tool Bar also contains buttons related to transaction analysis, that this course does not discuss.

- **The Secondary Tool Bar** contains features to navigate and manage the display panes.

- **The Time Status** pane displays the times of the baseline and primary cursors and the time between them. In this pane you can create cursors, select the primary cursor, and enable and disable marching waveforms.

- **The Times** pane displays the simulation timeline and the times of any cursors and markers.

- **The Names** pane displays the names of the displayed objects.

- **The Values** pane displays the values of the displayed objects.

- **The Waveforms** pane displays the waveforms of the displayed objects.
Start a Signalscan Waveform window by entering `signalscan`.
The Signalscan Waveform Display

The Signalscan window consists of the following:

- The **Title Bar** informs you that this is a Signalscan window, and displays which window, by number. If you open several windows, they are numbered incrementally.

- The **Menu Bar** lets you execute all of the basic commands.

- The **Tool Bar** includes buttons to copy, cut, paste, undo, delete, zoom, create markers, expand buses, launch the Design Browser, and format signal radices. You can customize the tool bar to meet your preferences.

- The **Multi-List** pane initially displays signal groups, but can also display user-defined markers and events.

- The **Waveforms** region displays waveforms that you have added.

- The **Names** pane displays signal names to the left of each waveform. You can drag and drop these signal names, and move the insertion marker by right-clicking in the pane.

- The **Time-Display** region displays the time values of the two cursors and the time between them.

**Note:** Use a Design Browser window to add signals to the Waveform window.
The Signalscan Waveform Display

Use the Design Browser window to select objects for the Waveform display.
The SignalScan Waveform Display

You can invoke the Design Browser window from the Waveform window with the "Windows — Design Browser" menu.

You can also invoke the Design Browser window from the Waveform tool bar with the "DesBrows:1" button.

Use the Design Browser to select objects for the Waveform display. You can also group objects and create busses in the Design Browser window.

The Design Browser window contains ten areas in addition to the title, menu and tool bars:

- The Current File field and drop-down list selects the design file to browse
- The Current Instance field and drop-down list selects the scope to browse
- The Bookmarks field and drop-down list selects bookmarked scopes
- The Groups field and drop-down list selects the group for grouping objects
- The Instances pane displays the modules instantiated in the current scope
- The Instances Filter field sets and displays the filter used for displaying instances
- The Variables pane displays the variables declared in the current scope
- The Variable Type buttons select the variable types to display
- The Variables Filter field sets and displays the filter used for displaying variables
- The To Add pane displays selected objects to be added to the Waveform display
The SHM Waveform Database

Waveform display tools read data from waveform databases.

The Simulation History Manager (SHM) is a proprietary waveform database.

Use the following system tasks to manipulate an SHM database:

<table>
<thead>
<tr>
<th>System task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$shm_open(&quot;filename&quot;);</td>
<td>Open database. Optionally provide name. Database name defaults to waves.shm</td>
</tr>
<tr>
<td>$shm_probe(signals);</td>
<td>Probe signals. Optionally specify signals. Signals default to ports at current scope.</td>
</tr>
<tr>
<td>$shm_save;</td>
<td>Save the SHM database</td>
</tr>
<tr>
<td>$shm_close;</td>
<td>Close the SHM database</td>
</tr>
</tbody>
</table>

For example:

```plaintext
initial

begin

$shm_open ( "waves.shm" ) ;
$shm_probe ( "AS" ) ;

end
```
The SHM Waveform Database

The Simulation History Manager (SHM) database is a record of the data signal changes that occur during the simulation of your design. It records changes only on the signals that you set probes on, and only during those simulation times for which the probes are active. Open the SHM database and probe signals at simulation time 0 to ensure a complete historical record of the specified signals.

Use the $shm_open system task to open a SHM database. You can provide arguments:

- `database_name` Database name defaults to `waves.shm`
- `is_sequence_time` Save multitransition data (0,1) defaults to 0 (not)
- `database_size` Database size (bytes) defaults to unlimited
- `is_compression` Database compression (0,1) defaults to 0 (none)

Use the $shm_probe system task to place SHM probes on signals. You can provide scope and node arguments (discussed in detail shortly). The signal transition data enters the database.

Use the $shm_save system task to flush buffered transition data to disk so that an application (such as a waveform display tool) has access to all the current data.

Use the $shm_close system task to close the SHM database. The simulator automatically closes the SHM database upon terminating simulation. You only need to close an SHM database if you wish to open a subsequent one. The $shm_open system task can open only one SHM database at a time.

Use the $shm_suspend system task to temporarily suspend dumping value change data.

Use the $shm_resume system task to resume suspended value change data dumping.
Probing Signals with $shmProbe

Provide $shmProbe with scope, node argument pairs.

You can default either or both arguments, for example:

- To probe the ports of the current scope
  
  $shmProbe();

- To probe the ports of the instances alu and adder
  
  $shmProbe(alu, adder);

- To probe all the nodes in the current scope
  
  $shmProbe("A");

- To probe all nodes in top.alu and below (including library cells), and the ports of the current scope and below (not including library cells)
  
  $shmProbe(top.alu, "AC", "S");
The SHM Waveform Database (continued)

Probing Signals with $shm_probe

Provide scope, node argument pairs to the $shm_probe system task. For example:

$shm_probe(scope0, node0, scope1, node1, ...);

The scope argument specifies the scope for the following node argument. The scope argument defaults to the current scope.

The node argument is a mnemonic specifying what signals you want to save at the specified scope. The node argument defaults to ports.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>----</td>
<td>Ports of the specified scope</td>
</tr>
<tr>
<td>&quot;S&quot;</td>
<td>Ports of the specified scope and below, excluding library cells</td>
</tr>
<tr>
<td>&quot;C&quot;</td>
<td>Ports of the specified scope and below, including library cells</td>
</tr>
<tr>
<td>&quot;A&quot;</td>
<td>All signals of the specified scope</td>
</tr>
<tr>
<td>&quot;AS&quot;</td>
<td>All signals of the specified scope and below, excluding library cells</td>
</tr>
<tr>
<td>&quot;AC&quot;</td>
<td>All signals of the specified scope and below, including library cells</td>
</tr>
</tbody>
</table>
Related Products

The following products are related to the Cadence Verilog simulators:

- The Cadence simulation analysis environment features the SimVision graphical interface tools
- The Cadence verification cockpit analysis environment upgrades the simulation analysis environment with HDL analysis and lint (HAL), and testbench development and transaction analysis tools
- The Cadence Verifault-XL simulator measures how well your testbench detects manufacturing faults
- The Cadence NC-Sim mixed language simulator upgrades the NC-Verilog simulator with coexecution of VHDL source
- Ambit BuildGates synthesis is a logic synthesis tool
- The FormalCheck model checker statically verifies that your functional design conforms to its specified properties
- The Cadence export model packager packages your Verilog, VHDL, or C model for protected distribution
Related Products

Cadence NC-VHDL simulator uses the same simulation kernel as NC-Verilog, allowing for very efficient coexecution of mixed-language designs.

Ambit BuildGates synthesis is a synthesis tool which parses your design, builds a generic design data structure, maps this structure to the cells available in the ASIC vendor’s technology library, and optimizes the design to minimize delay and footprint.

Cadence Verifault-XL simulator is a fault simulator. It grades the effectiveness of your test vectors to detect common faults in your design, such as stuck-at and bridge faults.
Summary

In this presentation you learned about:

- Logic simulation
- Running the Verilog-XL and NC-Verilog simulators
- Probing and displaying waveforms
Review

1. Is a simulator that propagates values forward, through the circuit, in response to input pin events, a time-based, event-based, or cycle-based simulator?

2. True or false: Events at any given simulation time can schedule additional events for the same simulation time.

3. Name some differences between the Verilog-XL and NC-Verilog simulators.

4. The $shm_probe system task with no arguments probes what signals?
Review

1. Is a simulator that propagates values forward, through the circuit, in response to input pin events, a time-based, event-based, or cycle-based simulator?

   Event-based simulation algorithms process only the changes in circuit state. The simulator propagates values forward, through the circuit, in response to input pin events or autonomous event generators (such as clocks).

2. True or false: Events at any given simulation time can schedule additional events for the same simulation time.

   An event at any given simulation time may schedule an event to occur at the same simulation time or at a later simulation time.

3. Name some differences between the Verilog-XL and NC-Verilog simulators.

   The NC-Verilog simulator separates the compile, elaborate, and simulate steps, and performs only the steps necessary to update the design; it simulates much faster than the Verilog-XL simulator, by default provides no interactive access to the design data structure, and interactively accepts Tcl commands rather than Verilog statements.

4. The $shm_probe system task with no arguments probes what signals?

   The $shm_probe system task with no arguments probes only the ports at the current scope. If you use this system task this way in a top-level testbench that does not declare ports, you will not capture any waveform data.
Chapter 4: A Sample Design

Objectives

In this section you will learn about:

- Key language features: modules; ports, instances
- The Value Change Dump (VCD) facility
A Sample Design

This section uses a small complete sample design to introduce the key Verilog language features, and the concept of a hierarchical design containing a design to be verified and a separate test fixture design. The section also introduces the standard Value Change Dump (VCD) tasks you can use to save waveform display data with any compliant simulator.
Key Language Features

This subsection discusses:

- The Verilog module
- Module ports
- Module instances
Key Language Features
The Verilog Module

The Verilog **module** is the basic building block of your design.

Start a Verilog module with the **module** keyword and end with **endmodule**.

```
module SN74LS74

endmodule

module DFF

endmodule

module ALU

endmodule
```
Key Language Features (continued)

The Verilog Module

The Verilog module is the basic design building block.

You model your design logic using one or more modules.

Every module description starts with the keyword **module**, has a name (SN74LS74, DFF, ALU, etc.), and ends with the keyword **endmodule**

Modules can represent:

- A physical block such as an IC or ASIC cell
- A logical block such as the ALU portion of a CPU design
- The Device Under Test (DUT)
- The testbench
- The complete system

Each Verilog module defines a new scope (level of hierarchy).
module DFF (q, qn, d, clk, rstn);
output q, qn;
input d, clk, rstn;
// DFF DEFINITION
endmodule

Note the module name, port list, and port declaration.
Key Language Features (continued)

Module Ports

Modules communicate with their environment through ports.
Module ports represent device pins and board connectors.
You list the module ports in parentheses after the module name.
You declare ports to be **input**, **output**, or **inout** (bidirectional).
Module Instances

You can create more complex modules by instantiating simpler modules:

```verilog
module REG4 (q,qn,data,clk,clrn);
output [3:0] q, qn;
input [3:0] data;
input clk, clrn;
    DFF d0 (q[0],qn[0],data[0],clk,clrn);
    DFF d1 (q[1],qn[1],data[1],clk,clrn);
    DFF d2 (q[2],qn[2],data[2],clk,clrn);
    DFF d3 (q[3],qn[3],data[3],clk,clrn);
endmodule
module DFF (q, qn, d, clk, rstn) ;

```

Module Instances

You can create a higher-level system or component by instantiating and connecting lower-level components.

The REG4 module instantiates the DFF module four times. Each DFF module has a unique instance name (d0, d1, d2, and d3). The instance name uniquely identifies the instance.

In this example, the order of port connections follows the order in which the submodule defines them. You will later see a way to make explicit connections without regard to order.

Each instance is a complete, independent, and concurrently active copy of the module.
A Simple and Complete Example

Testbench

Device Under Test

Stimulus and Control

Response Generation and Verification
A Simple and Complete Example

The example consists of a 2:1 multiplexer, which is the Device Under Test (DUT), and a testbench to provide a test stimulus and a verification mechanism.

We will describe the DUT at the gate level and describe the testbench behaviorally.
module MUX2_1 ( y, s, b, a );

// PORT DECLARATIONS
output y;
input s, b, a;
wire y, s, b, a;

// INTERNAL SIGNALS
wire sn, sb, sa;

// NETLIST
not (sn, s);
and (sb, b, s);
and (sa, a, sn);
or (y, sb, sa);
endmodule
A Simple and Complete Example (continued)

Device Under Test

The keywords `module` and `endmodule` begin and end the 2:1 multiplexer implementation.

Double slashes (`//`) start a comment that terminates at the end of the current line.

Signals flow into and out of a module through its ports:

- The input ports of the design are `a`, `b`, and `s`.
- The output port of the design is `y`.

This module instantiates the predefined `and`, `or`, and `not` Verilog primitives.

Another module can instantiate the multiplexer using only its module name and port declarations. No knowledge of underlying implementation is necessary. This is an important feature for the top-down design methodology, because a module implementation can change from behavioral level to gate level without affecting any of the higher-level modules using it.
Testbench

Testbench Template

module test;

    // DECLARE SIGNALS

    // INSTANTIATE MUX

    // APPLY STIMULUS

    // MONITOR SIGNALS

dendmodule

Why does this testbench have no ports?
A Simple and Complete Example (continued)

Testbench

This testbench declares no ports because it is a top-level module that no other module will instantiate.
Instantiating a Submodule

module test;

    // DECLARE SIGNALS
    reg a, b, s;
    wire y;

    // INSTANTIATE MUX
    MUX2_1 m1 ( y, s, b, a );

    // APPLY STIMULUS

    // MONITOR RESPONSE

endmodule
Testbench (continued)

You must first declare any signals that you want to be a register data type. The simulator assumes any undeclared signals that you use in an instantiation to be the default net type.

When you instantiate a module you must include:

- The name of the module
  You must use the name already given to the module.
  MUX2_1

- The name of this instance
  You can use any valid identifier name (This course discusses identifiers later).
  m1

- The port map
  This maps testbench signals to ports of the module instance.
  (y, s, b, a)
The simulator starts executing all procedure blocks at time 0.
The simulator executes all procedural blocks concurrently.
The simulator executes `initial` blocks once and `always` blocks continually.
Testbench (continued)

Your testbench will apply stimulus from within one or more procedural blocks.

The Verilog language offers two types of procedural blocks:

- The simulator executes **initial** blocks once
- The simulator executes **always** blocks continually, in a loop

This is how the simulator executes procedural blocks:

- The simulator starts executing procedural blocks at simulation time 0
  Controls within the procedural block can suspend its execution, waiting for time to elapse, conditions to become true, or events to occur
- The simulator executes procedural blocks concurrently
  This models the inherent concurrency of design hardware
- You cannot always know the order in which the simulator executes assignments in different concurrent procedural blocks
# Applying Stimulus

<table>
<thead>
<tr>
<th>Time</th>
<th>a</th>
<th>b</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

```// APPLY STIMULUS
initial begin
    a = 0; b = 1; s = 0;
    #5 a = 1; b = 0;
    #5 s = 1;
    #5 a = 0; b = 1;
    #5 $finish;
end```
Testbench (continued)

Signals of a register data type retain their value until reassigned.

The testbench declares $a$, $b$, and $s$ as register data type reg.

The testbench uses a procedural delay (#5) to separate stimulus value changes.

The testbench invokes the **$finish** system task to end simulation.
Checking the Response

Among the built-in system tasks and functions that Verilog provides are:

- The `$time` system function returns the current simulation time
- The `$monitor` system task displays the values of the listed arguments at the end of any time unit in which any of the arguments change value

```
$monitor ($time, o, in1, in2);
$monitor ($time,, out,, a,, b,, sel);
$monitor ($time,, "%b %h %d %o", sig1, sig2, sig3, sig4);
```
**Testbench** (continued)

The `$time` system function returns the current simulation time as a 64-bit unsigned integer.

The `$monitor` system task displays the values of the listed arguments at the end of any time unit in which any of the arguments change. It is not triggered by the change of `$time`. This task, by default, displays values in decimal. Similar tasks support the binary, octal, and hexadecimal default radixes.
module test;

  // DECLARE SIGNALS
  reg a, b, s;
  wire y;

  // INSTANTIATE MUX
  MUX2_1 m1 ( y, s, b, a );

  // APPLY STIMULUS
  initial begin
    a = 0; b = 1; s = 0;
    #5 a = 1; b = 0;
    #5 s = 1;
    #5 a = 0; b = 1;
    #5 $finish;
  end

  // MONITOR RESPONSE
  initial
    $monitor($time,,"y=%b a=%b b=%b s=%b",y,a,b,s);

endmodule
You normally apply stimulus at the behavioral level.

This example generates the following output:

0  y=0  a=0  b=1  s=0
5  y=1  a=1  b=0  s=0
10 y=0  a=1  b=0  s=1
15 y=1  a=0  b=1  s=1
The Value Change Dump (VCD)

Verilog provides a set of system tasks to record signal value changes in VCD format. The VCD format is ASCII data that most waveform display tools accept.

<table>
<thead>
<tr>
<th>System task</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$dumpfile(&quot;filename&quot;)</td>
<td>Open a VCD database for recording</td>
</tr>
<tr>
<td>$dumplimit (size);</td>
<td>Limit the size (in bytes) of the VCD database created</td>
</tr>
<tr>
<td>$dumpvars();</td>
<td>Select signals for recording</td>
</tr>
<tr>
<td>$dumpoff;</td>
<td>Stop recording</td>
</tr>
<tr>
<td>$dumpon;</td>
<td>Start recording again</td>
</tr>
<tr>
<td>$dumpall;</td>
<td>Record the values of all recorded signals</td>
</tr>
<tr>
<td>$dumpflush;</td>
<td>Flush all VCD data to disk</td>
</tr>
</tbody>
</table>
The Value Change Dump (VCD)

The VCD database is a record of the data signal changes that occur during the simulation of your design. It records changes only on the signals that you record.

Use the $dumpfile system task to open a VCD database. If you do not specify a filename argument, the simulator opens the default verilog.dump database. You can open only one database during simulation, and must open it before you invoke any other $dump tasks.

Use the $dumpvars system task to add signal names for the simulator to record. 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.

Use the $dumpall system task to create a checkpoint of the recorded signals. The VCD facility normally records individual signal values only as they change. You can use this task to update the VCD 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.
## Dumping Signals with $dumpvars

Supply *levels* and *scope* arguments to `$dumpvars`:

<table>
<thead>
<tr>
<th><code>$dumpvars</code>;</th>
<th>All signals in the hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$dumpvars(1,top);</code></td>
<td>All signals in scope <em>top</em></td>
</tr>
<tr>
<td><code>$dumpvars(2,top.u1);</code></td>
<td>All signals in scope <em>top.u1</em> and 1 level down</td>
</tr>
<tr>
<td><code>$dumpvars(3,top.u2);</code></td>
<td>All signals in scope <em>top.u2</em> and 2 levels down</td>
</tr>
<tr>
<td><code>$dumpvars(0,top.u3, top.u1.r0.q);</code></td>
<td>All signals in scope <em>top.u3</em> and below, and the signal <em>top.u1.r0.q</em></td>
</tr>
</tbody>
</table>

The following code could replace the `$monitor` command in your testbench:

```verbatim
initial
    begin
        $dumpfile("verilog.dump");
        $dumpvars (0, test);
    end
```
The VCD Database (continued)

Dumping Signals with $dumpvars

Syntax:

$dumpvars[<levels>, <scope>*];

A *levels* specification applies to subsequent scopes.

A *scope* refers to a signal, instance, or module in the hierarchy.

The *levels* defaults to 0 (all signals in the specified hierarchy).

The *scope* defaults to the current scope.
Summary

In this section you learned about:

- Key language features: modules; ports, instances
- The Value Change Dump (VCD) facility
Summary
Review

1. What is the basic Verilog building block?
2. How do modules communicate?
3. What are the two distinct components of a simulation?
4. How often does the simulator execute the `always` and `initial` procedural blocks?
5. How could you display signal values in ASCII to the screen as they change value?
Review

1. What is the basic Verilog building block?
   The basic Verilog building block is the module. Modules delineate hierarchy in Verilog. One module can instantiate another module, building up a more complex device.

2. How do modules communicate?
   Modules communicate with their environment through ports. Module ports represent device pins and board connectors.

3. What are the two distinct components of a simulation?
   The two distinct components in a simulation are the design component and the stimulus component. The design component is also called the Device Under Test (DUT). The stimulus component is also called the test harness, test fixture, or testbench. The stimulus component verifies the design component.

4. How often does the simulator execute the always and initial procedural blocks?
   The simulator executes each initial block once at the start of simulation, and executes each always block continually, in a loop, starting at the beginning of the simulation.

5. How could you display signal values in ASCII to the screen as they change value?
   You can use the $monitor system task to display signal values to the screen as they change value.
About Lab 1

The objective of this lab is to run a small simulation and verify the results with the waveform display tool.

In this lab, you will:

- Provide testbench stimulus for a mux
- Dump signals to a waveform database
- Use the waveform display tool to verify simulation results
About Lab 1
Chapter 5: Lexical Conventions in Verilog

Objectives

In this section you will learn about:

- White space and comments
- Integer and real numbers
- String constants
- Identifiers
- System tasks and functions
- Specifying delay
- Compiler directives
Lexical Conventions in Verilog

This section introduces the Verilog lexical conventions; white space, comments, identifiers, and literals. It goes on to introduce some of the more common system tasks and functions, and delay modeling, then ends with a discussion of compiler directives.
White Space and Comments

Verilog is a free-format language. The Verilog language ignores white space, except where needed to separate language tokens.

Begin a single-line comment with // and end it with the new-line character.

Begin a multi-line comment with /* and end it with */.

module MUX2_1 (out, sel, inb, ina);
// Port declarations
output out;
input sel, inb, ina;
/*
The netlist logic selects input "ina" when
sel is 0 and it selects "inb" when sel is 1.
*/
not (sel_, sel);
and (sela, ina, sel_),
(selb, inb, sel_);
or (out, sela, selb);
endmodule
White Space and Comments

White space includes space, tab, and newline characters. You can use whitespace to enhance the readability of your code.
The Verilog language permits integer and real variables and literals.

- You express real numbers in floating-point or scientific notation
- You express integer numbers with an optional size and radix

$$[[<\text{size}>]'<\text{radix}>]<\text{value}>$$

- **size** is the size in bits
- **radix** can be b (binary), o (octal), d (decimal), or h (hexadecimal)
- **value** is any legal number in the selected radix, including "x" and "z"

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>unsized decimal (extended to 32 bits with &quot;0&quot;)</td>
</tr>
<tr>
<td>'H83a</td>
<td>unsized hexadecimal (extended to 32 bits with &quot;0&quot;)</td>
</tr>
<tr>
<td>8'b1100_0001</td>
<td>8-bit binary</td>
</tr>
<tr>
<td>64'hff01</td>
<td>64-bit hexadecimal (extended to 64 bits with &quot;0&quot;)</td>
</tr>
<tr>
<td>9'017</td>
<td>9-bit octal</td>
</tr>
<tr>
<td>32'bz01x</td>
<td>extended to 32 bits with &quot;z&quot;</td>
</tr>
<tr>
<td>3'b1010_1101</td>
<td>3-bit number, truncated to 3'b101</td>
</tr>
<tr>
<td>6.3</td>
<td>decimal notation</td>
</tr>
<tr>
<td>32e-4</td>
<td>scientific notation for 0.0032</td>
</tr>
<tr>
<td>4.1E3</td>
<td>scientific notation for 4100</td>
</tr>
</tbody>
</table>
Integers and Real Numbers

Express real numbers in floating-point or scientific notation. Scientific notation represents real numbers as `<mantissa><e or E><exponent>`, for example `.314e-1`.

Express integer numbers as `[[<size>[<radix>]<value>]`. You can omit the `size` or both `size` and `radix`. The default size is 32 bits. The default radix is decimal. The `radix` and `value` are case-insensitive. You can insert underscore (`) characters anywhere in the value to promote readability.
String Constants

The Verilog language permits string constants, but has no explicit string data type:

- You can store ASCII character strings in a reg array

  This course discusses that application in a later section

- You will use string literals mostly as system task and function arguments
  - Enclose strings in double quotes on one line (do not embed newlines)
  - You can use escaped characters (such as \t and \n) in strings
  - You can use formatters (such as %b) to generate formatted output from a Verilog simulation

```verilog
$display("This is a string constant");
$write("This string has a \t tab and ends with a new line\n");
$monitor("This string formats a value: %b", monitored_object);
```
String Constants

Format Specifiers:

<table>
<thead>
<tr>
<th>%b</th>
<th>%c</th>
<th>%d</th>
<th>%h</th>
<th>%m</th>
<th>%o</th>
<th>%s</th>
<th>%t</th>
<th>%v</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>ASCII</td>
<td>decimal</td>
<td>hex</td>
<td>module</td>
<td>octal</td>
<td>string</td>
<td>time</td>
<td>strength</td>
</tr>
</tbody>
</table>

Escaped Characters:

\t tab
\n new line
\" double quote
\\ backslash
\<1-3 digit octal number> ASCII representation of above

You can use the %0d formatter to indicate that you want a decimal value with no leading blanks. The Verilog language permits no size formatters other than %0.

This course discusses escape characters and formatters in more detail later.
Identifiers

Identifiers are names you provide for Verilog objects within your design:

- Identifiers start with an alpha character (a-z, A-Z) or an underscore (_)
- Identifiers contain alphanumerics, dollar signs ($), and underscores
- Identifiers can be up to 1023 characters long
- Names of instances, modules, nets, ports, and variables are identifiers
- Verilog is case sensitive, sel and SEL are different identifiers

In this example, ina, inb, sel, sel_, sela, selb, out, and MUX2_1 are identifiers.

```verilog
module MUX2_1 (out, sel, inb, ina);
output out;
input sel, inb, ina;
    not (sel_, sel);
    and (sela, ina, sel_),
        (selb, inb, sel);
    or (out, sela, selb);
endmodule
```
Identifiers

These are legal identifiers:

  shift_reg1_q
  busa$index_3
  _bus8

These are illegal identifiers:

  34net // Begins with an illegal character
  $myreg // Begins with an illegal character
  a*3$reg // Contains an illegal character
  n@238 // Contains an illegal character

Verilog is case sensitive. All Verilog keywords are entirely lower-case. Your use of their upper-case look-alikes, for example "MÖDULE", as an identifier, is legal, but very confusing.
Escaped Identifiers

An escaped identifier eludes the Verilog identifier restrictions:

- Escaped identifiers start with backslash (\) and end with white space
- Escaped identifiers can contain any printable ASCII character
- The backslash and the white space are not part of the identifier

In this example, \2:1MUX and \~sel are escaped identifiers.

    module \2:1MUX (out,sel,inb,ina);
    output out;
    input sel,inb,ina;
    not (\~sel , sel);
    and (sela, ina, \~sel ),
    (selb, inb, sel );
    or (out, sela, selb);
endmodule

\(\text{Important}\)

Schematic capture and logic synthesis tools may create escaped identifiers. Support of escaped identifiers is not universal and has historically been problematic. You may want to avoid escaped identifiers wherever practical.
Identifiers (continued)

Escaped Identifiers

You create an escaped identifier by prefacing the identifier with a backslash (\). The backslash is not itself part of the identifier. You terminate the escaped identifier, or the escaped part of the identifier, with a whitespace character. The whitespace character is also not part of the identifier, and is not sufficient to separate the identifier from subsequent language tokens.

These are legal escaped identifiers:

\~#@sel
\busa+index
\{A,B\}
top.\3inst .net1 //escaped identifier in a hierarchical name

Important

You must end the escaped identifier with a whitespace character.

These statements are not syntactically valid:

$monitor("%d %b %b", $time, \busa+index, \busb+index);
always @(\~#@sel or inb or ina) out = \~#@sel ? inb : ina ;

Here are the corrections:

$monitor("%d %b %b", $time, \busa+index, \busb+index);
always @(\~#@sel or inb or ina) out = \~#@sel ? inb : ina ;
Special Language Tokens

This subsection discusses special language tokens:

- System tasks and functions
- Specifying delay
Special Language Tokens
# System Tasks and Functions

The Verilog language offers several system tasks and functions:

All system task and function names start with a dollar sign ($) character.

These are some of the more commonly used system tasks and functions:

<table>
<thead>
<tr>
<th>Task/Function</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$display</td>
<td>$display(&quot;out=&quot;,out);</td>
<td>Display argument values</td>
</tr>
<tr>
<td>$finish</td>
<td>$finish;</td>
<td>Terminate the simulation</td>
</tr>
<tr>
<td>$monitor</td>
<td>$monitor($time,sel,a,b,out);</td>
<td>Monitor signal changes</td>
</tr>
<tr>
<td>$stop</td>
<td>$stop;</td>
<td>Suspend the simulation</td>
</tr>
<tr>
<td>$time</td>
<td>time_var = $time;</td>
<td>Get the simulation time</td>
</tr>
</tbody>
</table>
Special Language Tokens (continued)

System Tasks and Functions

Use the $display system task to output argument values to the terminal and the log file. This system task by default outputs values in the decimal radix, but you can use formatters in a string argument to change the display radix.

Use the $finish system task to terminate the simulation, and optionally, depending upon the verbosity argument (0-2), report simulation statistics.

Use the $monitor system task to monitor changes of signal strength or value. This system task displays the signals values at the end of any simulation time step in which any of the signals transition.

Use the $stop system task to suspend simulation. The simulator returns to the interactive prompt and waits for user input.

Use the $time system function to return simulation time as a 64-bit quantity.
Specifying Delay

Use the pound sign (#) character to specify a procedural or primitive delay:

- You can use this simple delay specification in behavioral statements

```verilog
module MUX2_1 (out, sel, inb, ina);
output out; reg out;
input sel, inb, ina;
    always @(sel or inb or ina)
        #5 out = sel ? inb : ina;
endmodule
```

- You can attach this simple delay specification to primitives and nets

```verilog
module MUX2_1 (out, sel, inb, ina);
output out;
input sel, inb, ina;
    not #1 (sela, sel);
    buf #1 (selb, sel);
    and #2 (asel, ina, sela);
    and #2 (bsel, inb, selb);
    or #2 (out, asel, bsel);
endmodule
```
Special Language Tokens (continued)

Specifying Delay

You may see this simple delay notation, when applied to primitives, referred to as distributed delay, gate delay, lumped delay, intrinsic delay, and propagation delay.
Compiler Directives

Compiler directives direct subsequent compiler interpretation of source code:

- Compiler directives remain active from the file that defines them, potentially across other files, to the file that deactivates or overrides them
- All compiler directives start with an accent grave (``), also called a "back-tick"

These are some of the more commonly used compiler directives:

- `define Define a text replacement macro
- `ifdef Conditionally compile source code, dependent upon which text macros are defined
- `else
- `endif
- `include Include a file of source code
- `resetall Reset all compiler directives
- `timescale Establish a simulation timescale
- `undef Undefine a text replacement macro
Compiler Directives

Use the `define compiler directive to define a text replacement macro. This section explains this compiler directive in more detail later.

Use the `ifdef, `else, and `endif compiler directives to conditionally compile portions of source code, dependent upon which text replacement macros are defined.

Use the `include compiler directive to insert the contents of a source code file. This section explains this compiler directive in more detail later.

Use the `resetall compiler directive to reset all compiler directives to their reset state (if they have one). This directive does not reset text replacement macros.

Use the `timescale compiler directive to establish a simulation timescale. This section explains this compiler directive in more detail later.

Use the `undef compiler directive to undefine a single text replacement macro.

Use the `undefineall compiler directive with the Cadence NC-Verilog simulator to undefine all text macros at once. The IEEE Std. 1364-1995 does not describe the `undefineall directive. You should direct the compiler to use `undefineall only if running NC-Verilog:

```
`resetall
`ifdef INCA
  `undefineall
`endif
```
Substituting Text

The `define compiler directive provides a simple text-substitution facility.

```
`define <macro_name> <macro_text>
```

The compiler substitutes `macro_text` for `macro_name` at compile time.

You use text replacement macros to make your code more maintainable.

In this example, `not_delay`, `and_delay`, and `or_delay` are text substitution macros.

```
`define not_delay #1
`define and_delay #2
`define or_delay #2
module MUX2_1 (out, sel, inb, ina);
output out;
input sel, inb, ina;
    not `not_delay (sela, sel);
    buf `not_delay (selb, sel);
    and `and_delay (asel, ina, sela),
          (bsel, inb, selb);
    or `or_delay (out, asel, bsel);
endmodule
```
Substituting Text

You can use text replacement macros to make your code more maintainable. You can place such definitions that your design team will use in common into a header file that each source code file includes at its beginning. The header file would define global design parameters, like delays and widths of vectors, in a single place, so you can easily change them.

Use the `define compiler directive to define the text macro and optionally give it a non-null value.

```
`define <macro_name> <macro_text>
```

Use the `undef compiler directive to undefine a text macro.

```
`undef <macro_name>
```

Use the `define command-line option with the Cadence Verilog-XL and NC-Verilog simulators to define (or redefine) a text replacement macro.

```
+define+<macro_name>=<macro_text>
```

Use the `undefineall compiler directive with the Cadence NC-Verilog simulator to undefine all text macros at once.

```
`undefineall
```
Including Source Text

Use the `include` compiler directive to insert the contents of an entire file.

```
`include "<filename>"
```

Include commonly used global definitions, such as text replacement macros.

Include commonly used task definitions, to ensure consistent design reuse.

```
`resetall
`include "defines.inc"
module my_design;
  . . .
`include "this_task.inc"
`include "that_task.inc"
  . . .
```
Compiler Directives (continued)

Including Source Text

The `include` compiler directive inserts the entire contents of a file. You can provide either a full or relative path to the file. Ensure that you include the file contents in a valid context, for example you cannot include within a module description a file containing a `timescale` directive, and you must insert only within a module description a file containing elements of a module, such as block, object, and task definitions.

You would typically use this compiler directive to insert commonly used compiler directives, or portions of design descriptions common to several projects.

You can nest this compiler directive up to at least 16 levels.
Specifying the Timescale

Use the `timescale` compiler directive to specify the time unit and precision.

```
`timescale time_units / precision_units
```

The `time_units` declares the units of your time specifications.

The `precision_units` declares the precision of your time specifications.

Place the directive outside of a module definition.

Keep precision as close in scale to the time units as practical.

```
`timescale 1 ns / 100 ps
module MUX2_1 (out, sel, inb, ina);
  output out;
  input sel, inb, ina;
  not #1 (sela, sel);
  buf #1 (selb, sel);
  and #2 (asel, ina, sela),
     (bsel, inb, selb);
  or #2 (out, asel, bsel);
endmodule
```
Compiler Directives (continued)

Specifying the Timescale

The `time_units` and `precision_units` are each an integer value followed by a character string representing the magnitude and the unit, respectively:

- You can use integer values of 1, 10, or 100 for the time and precision values
- You can use seconds (s), milliseconds (ms), microseconds (us), nanoseconds (ns), picoseconds (ps), or femtoseconds (fs) for the time and precision units
- Your time units must be at least as large as your precision units

No standard default timescale exists. Simulators may assign a default timescale. To maintain timescale consistency between your files, you must have a timescale directive active for either all of your files or none of them. As with all compiler directives, a timescale directive is active from its first specification until specified again or reset.

Some simulators suffer a performance penalty if there is a large difference between the time units and the precision units, as the overall simulation time advances in increments of the precision units. For example, with `timescale 1 ns / 100 ps` the simulator takes 10 steps to advance 1 ns of simulation time, but with `timescale 1 ns / 1 ps` the simulator takes 1000 steps to advance 1 ns of simulation time. Simulators take these steps more or less efficiently.
Specifying the Timescale

The smallest precision of all the timescale directives determines the time unit of the overall simulation. For example, this simulation advances in 100fs steps.

```
`timescale 1ns/10ps
module1 ( . . . );
not #1.23 ( . . . ) // 1.23ns (12300 simulation time units)
...
endmodule

`timescale 100ns/1ns
module2 ( . . . );
not #1.23 ( . . . ) // 123ns (1230000 simulation time units)
...
endmodule

`timescale 1ps/100fs
module3 ( . . . );
not #1.23 ( . . . ) // rounded to 1.2ps (12 simulation time units)
...
endmodule
```
Specifying the Timescale (continued)

The smallest precision of all the timescale directives determines the time unit of the simulation, because the simulator must simulate the whole design accurately.

1. The first timescale directive sets the time units for *module1* to 1ns and precision to 10ps. The simulator must advance in steps no larger than 10ps.

2. The second timescale directive sets the time units for *module2* to 100ns and precision to 1ns. Because 1ns is greater than 10ps, the simulation time unit remains 10ps.

3. The third timescale directive sets the time units for *module3* to 1ps and precision to 100fs. Because 100fs is smaller than 10ps, the simulation time unit becomes 100fs.
Summary

In this section you learned about:

- White space and comments
- Integer and real numbers
- String constants
- Identifiers
- System tasks and functions
- Specifying delay
- Compiler directives
Summary
Review

1. Where does Verilog need white space?
2. What is the default size and base of integer literals?
3. What is a legal Verilog identifier?
4. With what system task can you immediately display a message to the screen?
5. What is the scope of a compiler directive?
Review

1. Where does Verilog need white space?

Verilog requires white space to clearly separate identifiers and some tokens, for example "initial begin", "posedge clock", or "(in1 or in2)". You can insert any amount of additional white space between the language tokens.

2. What is the default size and base of integer literals?

The simulator by default assumes that you provide 32-bit decimal literals. The simulator expands the literal to 32 bits if necessary, with zeroes if the leftmost bit is a 0 or 1, and with the value of the leftmost bit if it is not 0 or 1.

3. What is a legal Verilog identifier?

Unless escaped, a Verilog identifier must start with an alpha character or underscore, and contain only alphanumerics, dollar signs, and underscores. A Verilog identifier must be no more than 1023 characters long.

4. With what system task can you immediately display a message to the screen?

Use the $display system task to immediately display the formatted value of the task arguments to the standard output and log file.

5. What is the scope of a compiler directive?

A compiler directive is active during the compilation process from the time it is defined to the time it is reset or overridden, potentially across several files and modules. The order of compilation can change the scope of the effect of a compiler directive.
Chapter 6: Verilog Data Types and Logic System

Objectives

In this section you will learn about:

- The 4-value logic system
- The major data type classes
- Choosing the correct data type
- Using character strings
- Verilog memories
Verilog Data Types and Logic System

This section explains the 4-value Verilog logic system. It then describes module parameters and the major data type classes, and how to choose the most appropriate data type. It ends with a discussion of string literals and arrays.
The Verilog 4-Value Logic System

A Verilog logic value has one of four possible states:

- **0**, Zero, Low, False, Logic Low, Ground, VSS, Negative Assertion
- **1**, One, High, True, Logic High, Power, VDD, VCC, Positive Assertion
- **Z**, HiZ, High Impedance, 3-State, Float: Occurs at disabled driver or undriven node
- **X**, Unknown: Occurs at uninitialized storage elements or unresolvable logic conflicts.
The Verilog 4-Value Logic System

The Verilog unknown logic value "X" represents a situation where the simulator cannot predict the logic value the implemented node will attain. The node as implemented could attain a high, low, or intermediate state.

You can use the unknown logic value in a `casex` statement to indicate a "don’t-care" match item. This course discusses `casex` statements later.
Major Data Type Classes

The Verilog language contains three major data type classes:

- Nets
  - Represent physical connection between devices
- Registers
  - Represent abstract storage elements
- Module parameters
  - Configure module instances
Major Data Type Classes
Nets are continuously driven by the devices that drive them.
Continuous assignments, primitives, and registers can drive nets.
The simulator automatically propagates new driver values into the net.

In this example, Verilog primitives drive the nets \texttt{sel\_}, \texttt{sela}, \texttt{selb}, and \texttt{out}. The nets automatically and continuously reflect the values of the primitive outputs.

```verilog
module MUX2_1(out,sel,inb,ina);
output out;
input out,sel,inb,ina;
    not (sel_,sel);
    and (sela,ina,sel_);
    and (selb,inb,sel );
    or (out,sela,selb);
endmodule
```
Major Data Type Classes (continued)

Nets
Net Types

The Verilog language offers specific net types for specific technologies:

<table>
<thead>
<tr>
<th>Net Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wire, tri</td>
<td>For standard interconnection wires (default)</td>
</tr>
<tr>
<td>supply1, supply0</td>
<td>For power or ground rails</td>
</tr>
<tr>
<td>wor, trior</td>
<td>For multiple drivers that are Wire-OREd</td>
</tr>
<tr>
<td>wand, triand</td>
<td>For multiple drivers that are Wire-ANDed</td>
</tr>
<tr>
<td>trireg</td>
<td>For nets with capacitive storage</td>
</tr>
<tr>
<td>tri1, tri0</td>
<td>For nets that pull up or down when not driven</td>
</tr>
</tbody>
</table>

The simulator assumes an undeclared signal in an instance port map to be a single-bit net of type `wire`.

You can override this default net type with the following compiler directive:

```
default_nettype net_type
```
Nets (continued)

Verilog offers various net types for modeling design-specific and technology-specific functionality:

- **Net types tri and wire** are identical in functionality. Designers typically declare a bus net tri as a reminder that it is driven by multiple three-state drivers.

- **Net types triand and wand** are identical in functionality. These nets create a wired-AND configuration to resolve the conflict caused by multiple drivers. Likewise, the net types trior and wor create a wired-OR configuration.

- **The net type trireg** acts like the net type tri, but maintains its last value when all of its drivers go to the high impedance state. You can declare a trireg net to have a small, medium, or large capacitance value. The simulator uses the value to resolve the state of multiple connected trireg nets. You can specify a charge decay time when you declare the net.

- **The net type tri0** acts like the net type tri, but pulls resistively to 0 when all of its drivers go to the high impedance state. Likewise, the net type tri1 pulls resistively to 1.

Otherwise undeclared signals used in an instance port map default to a single-bit wire. You can override this using the `default_nettype net_type` compiler directive. With this directive, all such implicitly declared nets default to the net_type in the compiler directive.
Declaring Nets

Use this syntax to declare a net:

```
net_type [range] [delay] list_of_variables;
```

- Declare the net type
- You can optionally declare a [MSB:LSB] range
- You can optionally declare a net delay
- You can declare multiple nets of the same type, range, and delay

Here are some examples of net declarations:

```
wend w; // A scalar net of type "wend"
tri [15:0] busa; // A 16-bit three-state bus
wire [0:31] w1, w2; // Two 32-bit wires with msb = bit 0
wire #5 addr_sel; // A scalar wire with a delay
```
Nets (continued)

You can declare more than one net in the same declaration by using a list of names separated by commas.
Conflict Resolution

The Verilog simulator can resolve logic conflicts on multiply-driven nets.

<table>
<thead>
<tr>
<th>Wand/Triand</th>
<th>0 1 z x</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b x</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>0 1 1 x</td>
<td></td>
</tr>
<tr>
<td>1 0 1 1 x</td>
<td></td>
</tr>
<tr>
<td>0 1 z x</td>
<td></td>
</tr>
<tr>
<td>0 x x x</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Wire/Triand</th>
<th>0 1 z x</th>
</tr>
</thead>
<tbody>
<tr>
<td>b a</td>
<td>0 0 x 0 x</td>
</tr>
<tr>
<td>1 x 1 1 x</td>
<td></td>
</tr>
<tr>
<td>0 1 z x</td>
<td></td>
</tr>
<tr>
<td>0 x x x</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Wor/Trior</th>
<th>0 1 z x</th>
</tr>
</thead>
<tbody>
<tr>
<td>b a</td>
<td>0 0 1 0 x</td>
</tr>
<tr>
<td>1 1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>0 1 z x</td>
<td></td>
</tr>
<tr>
<td>0 x 1 x x</td>
<td></td>
</tr>
</tbody>
</table>
Nets (continued)

The Verilog language offers predefined logic conflict resolution features. You can, for example, use the wired-and net type to model multiple open-collector drivers, or the wired-or net type to model multiple ECL drivers.
Registers

A register maintains its value between discrete assignments.

You make assignments to registers in procedural code blocks.

Use registers to model synchronous hardware and apply testbench stimulus.

```
initial begin
    sel_reg = 0;
    ina_reg = 0;
    inb_reg = 1;
    if (out !== ina_reg) begin
        $display("ERROR");
        $finish;
    end
end
```
Major Data Type Classes (continued)

Registers

The Verilog register data type is an object that holds a value between assignments. You make a discrete assignment to it in a procedural block, and it holds the assigned value until you make another assignment to it. You use the register data type to model synchronous hardware and to apply stimulus in the testbench.
Register Types

The Verilog language offers four register data types:

- **integer**: Signed 32-bit integer variable
- **real**: Signed double precision floating-point variable
- **reg**: Unsigned storage of varying bit width
- **time**: Unsigned 64-bit integer variable

Verilog register data types do not imply structural storage elements as implemented in hardware!
**Registers** (continued)

The **integer** is a 32-bit signed quantity. You can use an **integer** variable as a loop index in your hardware model.

The **real** is a double-precision floating-point quantity. You would probably use a **real** variable only in a behavioral design or a testbench.

The **reg** is an unsigned quantity of any declared width. You would use a **reg** variable extensively in an RTL design.

The **time** is an unsigned 64-bit quantity. You would normally use a **time** variable in your testbench to store and manipulate simulation time values. You can also store and manipulate simulation time as **realtime** variable. The **realtime** data type is the same as the **real** data type.

The Verilog language requires you to declare as a register data type any variable you want to make an assignment to in a procedural block. This does not imply a storage element in the implemented hardware.
Declaring Registers

Use this syntax to declare a register:

```plaintext
integer list_of_register_variables;
real list_of_variables;
reg [range] list_of_register_variables;
time list_of_register_variables;
```

- Declare the register type
- You can optionally declare a [MSB:LSB] range for the `reg` type
- You can optionally declare an array of `integer`, `reg`, and `time` types
- You can declare multiple registers of the same type and range

Here are some examples of register declarations:

```plaintext
reg a; // A scalar reg
reg [3:0] v; // A 4-bit vector reg
reg [1:8] m,n; // Two 8-bit reg’s
```
Registers (continued)

You can declare more than one register in the same declaration by using a list of names separated by commas.
Module Parameters

Use module parameters to configure module instances:

```verilog
parameter list_of_assignments
```

- A parameter is an untyped constant
- You can use a parameter anywhere that you can use a constant or literal
- You can change the module parameters on an instance-by-instance basis

```verilog
module mod1(out,in1,in2);
  . . .
  parameter cycle = 20;
  parameter prop_del = 3;
  parameter setup = cycle/2 - prop_del;
  parameter width = 8;
  parameter x_word = 16'bx;
  parameter file = "/usr1/team/design/mem_file.dat";
  . . .
  wire [width-1:0] w1; // A wire declaration using parameter
  . . .
endmodule
```
Module Parameters

A module parameter is an untyped constant that assumes the data type required by its value. You can initialize a module parameter to any compile-time constant, and can change the type and value of a module parameter on an instance-by-instance basis. You would typically use a module parameter to define things such as delays and variable widths, that you would want to change on an instance-by-instance basis.

In the example, the \textit{file} parameter initializes to a register wide enough to store the 8-bit extended ASCII values for every character in the string assigned to it. If you set the \textit{file} parameter to a value of "ABC", it would be a 24-bit wide register set to the value 24’b01000001_01000010_01000011 (or 24’h414243). Defining the \textit{file} parameter this way allows you to open a file with $\texttt{fopen}(\textit{file});$, or print its string value with $\texttt{display}("%s", \textit{file});$. This section will further discuss string constants and literals later.
Overriding Module Parameters

Use the **defparam** statement to override a module parameter:

\[
\text{defparam parameter\_reference new\_parameter\_value}
\]

- With hierarchical references you can modify any module parameter from anywhere in the design

```verilog
module mod1(out, in1, in2);
  . . .
  parameter
    cycle = 8,
    real\_constant = 2.039,
    x\_word = 16'bx,
    file = "/usr1/team/design/mem_file.dat";
  . . .
endmodule

module test;
  . . .
  mod1 I1 (out, in1, in2);
  defparam
    I1\.cycle = 6,
    I1\.file = "./mem_file.dat";
  . . .
endmodule
```
Module Parameters (continued)

Use the `defparam` statement to override module parameters during compile time. With hierarchical references, you can change any parameter anywhere in the design from anywhere else in the design. Some simulators allow you to interactively modify a module parameter during runtime, but this change can no longer effect the design data structure.
You can override module parameters in the instantiation statement:

```
module_name [parameter_overrides] instance_name port_maps
```

- Override module parameters in the same order as they are declared

```
module mod1(out,in1,in2);
  ...
  parameter cycle = 8;
  parameter real_constant = 2.039;
  parameter x_word = 16'bx;
  parameter file = "/usr1/team/design/mem_file.dat";
  ...
endmodule

module top;
  ...
  mod1 #(6,3.1,16'bx, "/mem_file.dat") I1 (out,in1,in2);
  ...
endmodule
```

*How does Verilog know `#(6, 3.1, 16'bx, "/mem_file.dat")` is a parameter override and not a delay?*
Module Parameters (continued)

Use the pound sign (#) character to override module parameters within a module instance. You must override the module parameters in the same order as they are declared. You do not need to override all of the instance parameters, but you cannot skip over a parameter.

The following are legal module parameter overrides:

mod1 #(6) I1 (out,in1,in2);
mod1 #(6, 3.1) I1 (out,in1,in2);
mod1 #(6, 3.1, 16'b1, "./mem_file.dat") I1 (out,in1,in2);

The following are illegal module parameter overrides:

mod1 #(, 3.1) I1(out,in1,in2);
mod1 #(6, 3.1, , "./mem_file.dat") I1 (out,in1,in2);

The reason why there is no ambiguity in using the # character for module parameter override is that modules cannot have instance delays and primitives cannot have module parameters.
Choosing the Correct Data Type

Follow these rules when choosing between net and register data types:

- An **input** or **inout** port must be a net
- An **output** port can be a register data type
- A signal assigned a value in a procedural block must be a register data type
Choosing the Correct Data Type
Common Data Type Mistakes

These are some common data type usage errors:

- You make a procedural assignment to a net
  
  wire [7:0] databus:
  always @(read or addr) databus=read?mem[addr]:'bz;

  This is an illegal left-hand-side assignment

- You connect a register to an instance output
  
  reg myreg;
  and (myreg, net1, net2);

  This is an illegal output port specification

- You declare a module input port as a register
  
  input myinput;
  reg myinput;

  These are incompatible declarations
Choosing the Correct Data Type (continued)
String Literals

The Verilog language has no explicit string data type. However:

- You can store string literals in an appropriately-sized register
- You need 8 bits for each ASCII character

```verilog
parameter string_size = 10;
reg [string_size*8 - 1: 0] string_variable;
string_variable = "file.dat";

Note there is no terminating null character!
```

- The register value is concatenated 8-bit ASCII character values

```verilog
string_variable = 64'h66696c652e646174;
$display("\"%s\"", string_variable);
" file.dat"
```

- The simulator sets unassigned high-order bits to zero (null characters)
- The %s formatter displays leading null characters as blank characters
String Literals

The %s formatter displays leading null characters in the register as space characters.
   $display("\"%s\"", string_variable);
   " file.dat"

The %0s formatter does not display leading null characters in the register.
   $display("\"%0s\"", string_variable);
   "file.dat"

You can also display the register value in binary, octal, decimal, or hexadecimal.
   $display("%h", string_variable);
   000066696c652e646174

You can suppress display of the leading zeros.
   $display("%0h", string_variable);
   66696c652e646174
Verilog Memories

A Verilog memory is an array of reg vectors.

\[
\text{reg } [\text{MSB:LSB}] \text{ memory\_name } [\text{first\_addr:last\_addr}];
\]

You can use module parameters to configure the memory:

\[
\text{parameter wordsize } = 16;
\]
\[
\text{parameter memsize } = 1024;
\]
\[
\text{reg } [\text{wordsize-1:0}] \text{ mem } [0:\text{memsize-1}];
\]

You can also declare arrays of integer and time data types:

\[
\text{integer int\_array } [0:1]; // array of 2 integer variables
\]
\[
\text{time time\_array } [8:1]; // array of 8 time variables
\]
Verilog Memories

The Verilog language requires that range expressions in declarations be compile-time constant expressions. The compiler needs to know the size of `reg` data types and the depth of arrays before the simulation starts.
Memory Addressing

You can only access one word of memory at a time.

Access the word by indexing into the memory array.

```verilog
module mems;
reg [8:1] mema [0:255]; // memory "mema"
reg [8:1] mem_word; // temporary reg "mem_word"
.. .
initial begin

    // Display contents of the 6th memory address
    $displayb(mema[5]);

    // Display the MSB of the 6th memory word
    mem_word = mema[5];
    $displayb(mem_word[8]);
end

endmodule
```
Verilog Memories (continued)

Memory Addressing

Access a memory word by indexing within the range of declared addresses.

\texttt{mem\_name [addr\_expr]}

To access a bit or part select of a memory word, first store the word in a temporary variable.
Summary

In this section you learned about:

- The 4-value logic system
- The major data type classes
- Choosing the correct data type
- Using character strings
- Verilog memories
Summary
Review

1. What does the Verilog unknown signal value "X" mean?
2. What are the four Verilog register class variable types?
3. True or false: Each instance of a module definition can contain different constant values in its module parameters.
4. What port type can you directly declare to be a **reg** data type?
5. For what data types can you declare a one-dimensional array?
Review

1. What does the Verilog unknown signal value "X" mean?
   The Verilog unknown signal value "X" means that either an uninitialized register drives the output, or the simulator cannot resolve the conflict between multiple drivers driving the net to different values.

2. What are the four Verilog register class variable types?
   The four Verilog register types are integer, real, reg, and time. Some simulators let your store a value in a module parameter during runtime, but this is an anomaly. The trireg is a net with capacitive storage.

3. True or false: Each instance of a module definition can contain different constant values in its module parameters.
   You use a module parameter to parameterize a module definition. Each instance of the module definition can contain different constant values in its module parameters. You can set these different values as you instantiate the module, or with defparam.

4. What port type can you directly declare to be a reg data type?
   You can directly declare a output port to be a reg data type. Any other port type must be an explicit or implicit net so that the simulator can attempt to resolve any potential conflicts caused by multiple drivers of the signal.

5. For what data types can you declare a one-dimensional array?
   You can declare a one-dimensional array of integer, reg, or time. You cannot declare multi-dimensional arrays and cannot declare arrays of parameter, real, or any net types.
About Lab 2

The objective of this lab is to practice choosing the correct data types for your model.

In this lab, you will:

- Identify and correct several data type errors
Chapter 7: Structural Modeling

Objectives

In this section, you will learn about:

- Structural modeling
- Some Verilog primitives
- Instantiating modules and primitives
- Modeling primitive drive strength
Structural Modeling

This section defines structural modeling, describes the built-in Verilog primitives, and explains how to instantiate single (and arrays of) primitives and modules. The section then goes on to discuss signal strengths and how the simulator resolves the value of a net driven by more than one driver.
A structural model is a list of predefined components and their interconnections. A structural model can describe a schematic:

```verilog
module MUX4x1(Z,D0,D1,D2,D3,S0,S1);
  output Z;
  input D0,D1,D2,D3, S0,S1;
  and (T0, D0, S1_, S0_);
  and (T1, D1, S1_, S0);
  and (T2, D2, S1, S0_);
  and (T3, D3, S1, S0);
  not (S0_,S0), (S1_,S1);
  or (Z, T0, T1, T2, T3);
endmodule
```

```verilog
module srLatch (q, q_, s, r);
  output q, q_; input s, r;
  nor nr1 (q_, q , s);
  nor nr2 (q , q_, r);
endmodule
```
Structural Modeling

A structural model can describe a schematic. You create instances of predefined components and connect them to define a more complex design.

The multiplexor example:
- Is a combinational design
- Instantiates all the gates of a given type in one statement
- Does not use instance names
- Implicitly declares internal signals as scalar wires

The latch example:
- Is a sequential design
- Instantiates each gate in a separate statement
- Does use instance names
Verilog Primitives

The Verilog language offers 26 built-in primitives.

These 8 provide the basic logic functions:

<table>
<thead>
<tr>
<th>Primitive Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>Logical And</td>
</tr>
<tr>
<td>or</td>
<td>Logical Or</td>
</tr>
<tr>
<td>not</td>
<td>Inverter</td>
</tr>
<tr>
<td>buf</td>
<td>Buffer</td>
</tr>
<tr>
<td>xor</td>
<td>Logical Exclusive Or</td>
</tr>
<tr>
<td>nand</td>
<td>Logical And Inverted</td>
</tr>
<tr>
<td>nor</td>
<td>Logical Or Inverted</td>
</tr>
<tr>
<td>xnor</td>
<td>Logical Exclusive Or Inverted</td>
</tr>
</tbody>
</table>
Verilog Primitives

Verilog provides 26 built-in primitives that you can use to model functionality in detail. At the time that Verilog was first released, people typically modeled components for simulation at this level of detail. ASIC vendor simulation libraries still extensively use these primitives, both for the level of detail, and because historically the Verilog-XL simulator required that module and interconnect timing paths terminate at an acceleratable primitive.
Some Primitive Pins Are Expandable

The **and**, **nand**, **or**, **nor**, **xor**, and **xnor** primitives permit one or more inputs, and one output:

\[
\begin{align*}
\text{and} & \quad \text{(out, in)}; \\
\text{nand} & \quad \text{(out, in1, in2)}; \\
\text{or} & \quad \text{(out, in1, in2, in3)};
\end{align*}
\]

The **buf** and **not** primitives permit one input, and one or more outputs:

\[
\begin{align*}
\text{buf} & \quad \text{(out, in)}; \\
\text{not} & \quad \text{(out1, out2, in)}; \\
\text{buf} & \quad \text{(out1, out2, out3, in)};
\end{align*}
\]
Verilog Primitives (continued)

Some Primitive Pins Are Expandable

The **and**, **nand**, **or**, **nor**, **xor**, **xnor** gates have one output (first in the port list) and one or more inputs.

The **buf** and **not** gates have one or more outputs and one input (last in the port list).

This feature facilitates circuit redesign. For example, you might not need to add an additional primitive to accommodate an additional **enable** latch input port.
# Conditional Primitives

Verilog offers 4 conditional primitives.

Conditional primitives drive a high impedance state when disabled:

<table>
<thead>
<tr>
<th>Primitive Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>bufif1</td>
<td>Conditional buffer with logic 1 as enabling input</td>
</tr>
<tr>
<td>bufif0</td>
<td>Conditional buffer with logic 0 as enabling input</td>
</tr>
<tr>
<td>notif1</td>
<td>Conditional inverter with logic 1 as enabling input</td>
</tr>
<tr>
<td>notif0</td>
<td>Conditional inverter with logic 0 as enabling input</td>
</tr>
</tbody>
</table>
Verilog Primitives (continued)

Conditional Primitives
Conditional Primitives

<table>
<thead>
<tr>
<th></th>
<th>bufif1</th>
<th>bufif0</th>
<th>notif1</th>
<th>notif0</th>
</tr>
</thead>
<tbody>
<tr>
<td>in=0</td>
<td>Z</td>
<td>0</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>in=1</td>
<td>Z</td>
<td>1</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>in=z</td>
<td>Z</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>in=x</td>
<td>Z</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Conditional Primitives (continued)

Each conditional primitive has one output, one data input, and one control input (in that order).
Conditional primitives drive the high impedance state when disabled.

The symbols L and H represent known values with unknown strengths.
   The symbol L indicates a logic 0 value that might have only high-impedance strength.
   The symbol H indicates a logic 1 value that might have only high-impedance strength.
**Primitive Instantiation**

The syntax for a primitive instantiation is:

```
gate_type [drive_strength] [delay] instance [, instance ...];
```

where

```
instance := [instance_name] ( terminal [, terminal ...] )
```

- **You can omit the drive strength specification**
  ```
  not (strong1, weak0) n1 (inv, bit); // strength
  not n1 (inv, bit); // no strength
  ```

- **You can omit the delay specification**
  ```
  notif0 #3.1 n1 (out, in, cntrl); // delay
  notif0 n1 (out, in, cntrl); // no delay
  ```

- **You can omit the instance name**
  ```
  and a1 (out, in1, in2, in3, in4); // name
  and (out, in1, in2, in3, in4); // no name
  ```

- **You must specify outputs before inputs**
Primitive Instantiation

You can optionally assign a drive strength to a primitive instance. The simulator uses drive strength to resolve the value of a net having multiple drivers.

You can optionally assign a delay to a primitive instance. The delay is lumped at the output, so it is the same propagation delay for all inputs. If you do not assign a delay to a primitive instance, the delay defaults to 0.

You can optionally assign an instance name to a primitive instance.

You must specify all the outputs of the primitive before you specify its inputs. Note that only the buf and not primitives permit multiple outputs.

In addition to built-in primitives, you can create User Defined Primitives (UDPs). This course discusses UDPs in a later section.
Module Instantiation

The syntax for a module instantiation is:

\[
\text{module_type} \ [\text{parameter_assignments}] \ \text{instance} \ [\ , \ \text{instance} \ ...] ;
\]

where

\[
\text{instance} := \text{instance\_name} ( \ \text{connection} \ [\ , \ \text{connection} \ ...] \ )
\]

- You can omit the module parameter value assignments
- You must include an instance name
- You can choose explicit or implicit port mapping
  - You can connect a module port to an expression
  - You can leave a module port unconnected

submod m1 (Q1,QN1,J,K,R); // Port map implied by position
submod m2 (Q2,QN2,J,K,R1&2); // Port connected to expression
submod m3 (Q3, ,J,K); // Port left unconnected
submod m4 (.i2(K),.i1(J),.o1(Q4),.o2(QN4),.rs(R)); // Explicit
submod m5 (.i2(K),.i1(J),.o1(Q5),.o2(QN5),.rs(R1&R2)); // Expr
submod m6 (.i2(K),.i1(J),.o1(Q6),.o2(QN6)); // Unconnected port
Module Instantiation

Every module instance has an instance name.

You can imply the port map by listing the module port connections in the same order as the port list in the module definition. To do this, you must use a comma as a place holder for a port you want to leave unconnected.

If you make implicit port connections and leave a port unconnected (with an extra comma as a place holder), the simulator will not issue unconnected port warnings.

You can explicitly define the port map by individually specifying the connection to each port. For this method you simply do not list any port that you want to leave unconnected. The syntax for this connection method is:

```
.<port_name>(<port_connection>)
```

An unconnected input port assumes the high-impedance Z value.

If you use explicit port connections to a module instance, and leave some of the ports unlisted, the simulator will warn you about unconnected module instance ports, but will not tell you which ports are unconnected. You can avoid this warning by explicitly declaring unconnected ports:

```
.<port_name>()
```

You may want to explicitly list unconnected ports when using the explicit port connection methodology. This way, unconnected port warnings will indicate an instance with ports that should probably have connections, but do not.
Array of Instances

Your instance name can include a range.

No range specification:

```verbatim
module bus_driver (out, in, en);
output [3:0] out;
input [3:0] in;
input en;
    bufif0 u3 (out[3], in[3], en);
    bufif0 u2 (out[2], in[2], en);
    bufif0 u1 (out[1], in[1], en);
    bufif0 u0 (out[0], in[0], en);
endmodule
```

Range specification:

```verbatim
module bus_driver (out, in, en);
output [3:0] out;
input [3:0] in;
input en;
    bufif0 u[3:0] (out, in, en);
endmodule
```
Array of Instances

For terminal connections to an array of instances, a comparison is made between the bit length of each port expression in a declared instance array and the bit length of each single-instance port or terminal in an instantiated model or primitive.

For each port or terminal where the comparison shows the same bit lengths, the instance-array port expression is connected to each single-instance port.

If the bit lengths are different, then the bit length of each port expression in the declared instance array should be the product of the number of instances and the bit length of the corresponding single-instance port or terminal. In such a case, each instance gets a part-select of the port expression as specified in the range. The connections to the port of the instance start with the lowest bits of the port expressions (right-hand index) and continue to the highest bits (left-hand index).

The instance names generated by the range specification are exactly like the names used to reference bit-selects of a vector net.

This range of instances:

wire [2:0] out, in;
buf b[2:0] (out, in)

Generates the instances:

buf b[2] (out[2], in[2]);
buf b[1] (out[1], in[1]);
buf b[0] (out[0], in[0]);
Illegal Arrays of Instances

The following are illegal attempts to declare arrays of instances:

- You can associate an instance name with only one continuous range
  
  ```
  submod s[7:4] (node1, node2);
  submod s[3:0] (node3, node4);
  ```

- The vector port connector width must be a multiple of the instance port width
  
  ```
  module driver (o, i, e);
  output [3:0] o; input [3:0] i; input e;
  ...wire [10:0] buso, busi;
  driver d[2:0] (buso, busi, enb);
  ```

- Range indexes must be constants or constant expressions
  
  ```
  integer L1, L2, R;
  buffer b[L1+L2:R] (out, in);
  ```

- The `defparam` statement may not alter instantiation parameters
  
  ```
  defparam top.submod_array[0].WIDTH = 32;
  defparam top.submod_array[1].WIDTH = 16;
  ```
Array of Instances (continued)

Illegal Arrays of Instances

An array of instances has a continuous range. To declare an array of instances, one instance name is associated with only one range; you cannot specify the same instance name for another range.

If there are too many or too few bits to connect to all the instances, the simulator reports an error.

You must be able to represent an array of instances as a set of single instances and their port connections. You cannot use expressions that cannot be represented as separate part-select elements (for example, \( a+b \)). Range indexes must be local module expressions of constants or parameters that cannot exceed the range of the array.

For modules you instantiate in arrays, or their child modules, you may not use the `defparam` statement to alter parameters that appear within the constant expressions of a range or as part of the port declarations of the instantiated module.
Logic Strength Modeling

You can optionally assign drive strengths to some primitives:

- This can sometimes model circuit operation more accurately
- The simulator uses drive strengths to resolve the value of a net with multiple drivers
- You might model drive strengths for:
  - Open collector drivers
  - Three-state drivers
Logic Strength Modeling

Logic strength modeling is an important aspect of Verilog modeling. Typically only component modelers, such as ASIC or FPGA library developers, use this level of detail, but circuit designers who simulate with these detailed models must understand them.
The Signal Strength Value System

<table>
<thead>
<tr>
<th>Level</th>
<th>Type</th>
<th>%v formats</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supply</td>
<td>Drive</td>
<td>Su0, Su1</td>
<td>supply0, supply1</td>
</tr>
<tr>
<td>Strong</td>
<td>Drive (default)</td>
<td>St0, St1</td>
<td>strong0, strong1</td>
</tr>
<tr>
<td>Pull</td>
<td>Drive</td>
<td>Pu0, Pu1</td>
<td>pull0, pull1</td>
</tr>
<tr>
<td>Large</td>
<td>Capacitive</td>
<td>La0, La1</td>
<td>large</td>
</tr>
<tr>
<td>Weak</td>
<td>Drive</td>
<td>We0, We1</td>
<td>weak0, weak1</td>
</tr>
<tr>
<td>Medium</td>
<td>Capacitive</td>
<td>Me0, Me1</td>
<td>medium</td>
</tr>
<tr>
<td>Small</td>
<td>Capacitive</td>
<td>Sm0, Sm1</td>
<td>small</td>
</tr>
<tr>
<td>High Z</td>
<td>Impedance</td>
<td>Hi0, Hi1</td>
<td>highz0, highz1</td>
</tr>
</tbody>
</table>

A strength you assign to a primitive instance or net applies to any signals it drives:

    and (strong0, weak1) a1 (out, a, b);

You can display the strength and value of a net with the %v format specifier:

    $monitor ($time, "output = %v", out);
Logic Strength Modeling (continued)

The Signal Strength Value System

Primitive strength specification syntax:

\[<\text{drive_strength}> := (<\text{STRENGTH0}>,<\text{STRENGTH1}>) \mid (<\text{STRENGTH1}>,<\text{STRENGTH0}>)\]

You can assign either both strengths, or no strengths, to the 8 logical and 4 conditional primitives, and you can assign the strengths in either order.

- `nand (strong1,pull0) #(2:3:4) n1 (o,a,b);` // strength and delay
- `or (supply0,highz1) (out,in1,in2,in3);` // no instance name

You can assign either one strength or no strengths to the `pullup` and `pulldown` primitives.

- `pullup (weak1) (trisig);`

You can assign the capacitive strengths (`large, medium, small`) only to `trireg` nets and `tran` primitives.

- `trireg (small) t1;`
In Verilog, the highest strength overrides all other drivers:

<table>
<thead>
<tr>
<th>a’s output</th>
<th>b’s output</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>strong1</td>
<td>strong0</td>
<td>strongX</td>
</tr>
<tr>
<td>pull0</td>
<td>weak1</td>
<td>pull0</td>
</tr>
<tr>
<td>pull0</td>
<td>strong0</td>
<td>strong0</td>
</tr>
<tr>
<td>strong1</td>
<td>weak1</td>
<td>strong1</td>
</tr>
<tr>
<td>pull0</td>
<td>HiZ</td>
<td>pull0</td>
</tr>
<tr>
<td>HiZ</td>
<td>weak1</td>
<td>weak1</td>
</tr>
<tr>
<td>HiZ</td>
<td>HiZ</td>
<td>HiZ</td>
</tr>
</tbody>
</table>
Logic Strength Modeling (continued)

How Verilog Resolves Signal Strengths
Summary

In this section, you learned about:

- Structural modeling
- Some Verilog primitives
- Instantiating modules and primitives
- Modeling primitive drive strength
Summary
Review

1. Which built-in Verilog primitives accept a variable number of input connections?

2. Which built-in Verilog primitives can drive a high-impedance output, but cannot pass a high-impedance input to their output?

3. What does the #5 represent in this instantiation?

   ```verilog
dbl #5 bl (.o(out), .i(in));
```

4. True or false: You can spread an instance array over multiple statements.

5. When a wire connected to a pullup primitive is driven low by another primitive assigned the weak0 strength, what is the state of the wire?
Review

1. Which built-in Verilog primitives accept a variable number of input connections?
   The **and**, **or**, **nand**, **nor**, **xor**, and **xnor** primitives accept one or more input connections.

2. Which built-in Verilog primitives can drive a high-impedance output, but cannot pass a high-impedance input to their output?
   The **bufif0**, **bufif1**, **notif0**, and **notif1** primitives can drive a high-impedance output, but translate a high-impedance input to an unknown output value.

3. What does the #5 represent in this instantiation?
   ```
   bufr #5 b1 (.o(out), .in(in));
   ```
   The **bufr** type is not a built-in Verilog primitive, and primitives do not have port names, so this is a module instantiation and #5 is the redefinition of a module parameter.

4. True or false: You can spread an instance array over multiple statements.
   You must declare the full instance array in one statement.

5. When a wire connected to a **pullup** primitive is driven low by another primitive assigned the **weak0** strength, what is the state of the wire?
   The **pull1** strength (level 5) is stronger than the **weak0** strength (level3). In the Verilog signal strength system, the strongest signal overrides all other drivers.
About Lab 3

The objective of this lab is to use predefined library components to model a hierarchical 8-bit register.

In this lab, you will:

- Model an 8-bit register at the structural level
- Provide stimulus to test the register design

![Diagram of Register](image-url)
About Lab 3
Chapter 8: Modeling Delay

Objectives

In this section you will learn about:

- Modeling delay
- Specify blocks
- The standard delay format (SDF)
Modeling Delay

This section introduces and compares the different ways of modeling delay, then goes on to discuss the specification of propagation delay and timing checks in specify blocks, and ends with a brief discussion of delay specification in an SDF file.
Modeling Delay

You can model delay in three ways:

- You can lump the entire delay at the output of the last gate
- You can distribute the delay across the outputs of each gate
- You can specify pin-to-pin module path delays in a **specify** block

Let’s model these specifications:

- Propagation delay from A to O is 2
- Propagation delay from B to O is 3
- Propagation delay from C to O is 1
Modeling Delay

You can lump all delay onto the output port of the last gate:
- Most easy to implement
- Least accurate
- Simulates fastest

You can distribute delay across the outputs of all gates:
- More difficult to implement
- More accurate
- Simulates slower

You can model module path delay in a specify block
- Most accurate
- Simulates slowest

Synthesis tools totally ignore all delay specification in the Verilog source code. You provide timing constraints to the synthesis tool in a separate file, using the syntax required by the synthesis vendor.
Lumped Delay

You can lump the entire delay at the output of the last gate:

- Most easy to implement
- Least accurate
  - All pin-to-pin paths have the same delay

```verilog
`timescale 1 ns / 1 ns
module nor_or (O, A, B, C);
output O;
input A, B, C;
nor   (n, A, B);
or   #3 (O, C, n);
endmodule
```

Does this lumped delay accurately model the timing specifications?
Modeling Delay (continued)

Lumped Delay

You can model propagation delay across devices by lumping the delay onto the last gate in the model. This is an easy method of modeling delay, but is usually inaccurate, as it gives all pin-to-pin paths the same delay. If you use lumped delay, it is best to select the worst of the pin-to-pin delays.

In this example, all pin-to-pin paths have the same delay.
Distributed Delay

You can distribute the delay across the outputs of each gate:

- More difficult to implement
- More accurate
  - Pin-to-pin paths can have different delays

```
`timescale 1 ns / 1 ns
module nor_or(O, A, B, C);
output O;
input A, B, C;
  nor #2 (n, A, B);
  or #1 (O, C, n);
endmodule
```

Does this distributed delay accurately model the timing specifications?
Distributed Delay

You can model propagation delay across devices by distributing the delay among the gates in the model. Distributing delays on a real structural model can quickly become difficult. You still cannot represent different delays from different pins of the same primitive.

In this example, paths from A and B have the same delay.
You can specify pin-to-pin module path delays in a `specify` block:

```
`timescale 1 ns / 1 ns
module nor_or (O, A, B, C);
output O;
input A, B, C;
    nor (n, A, B);
    or (O, C, n);
    specify
        (A*>O)=2;
        (B*>O)=3;
        (C*>O)=1;
   endspecify
endmodule
```
Modeling Delay (continued)

Module Path Delay

You can model propagation delay across devices in a specify block. These delays, called *module path delays*, specify the delays from inputs to outputs of a module. With module path delays, you can model propagation delay accurately, and as the delays are separated from the device functionality, you can more easily change the delays later.

In this example, each pin-to-pin path has its own delay.
Modeling Delay Accurately

You can specify rise, fall, and turn-off delays for gates and nets.

```vhdl
and #(2,3) (out, in1, in2, in3); // rise, fall
bufif0 #(3,3,7) (out, in, ctrl); // rise, fall, turn-off
```

You can specify up to six delay values for module paths (0->1, 1->0, 0->Z, Z->1, 1->Z, Z->0).

```vhdl
(a -> b) = 3; // one delay value
(clk -> q) = (1, 2); // rise, fall
(en -> busa) = (5, 4, 7); // rise, fall, turn-off
(busa => busb) = (5, 12, 17, 10, 6, 22); // six delay values
```

You can specify minimum, typical, and maximum values (a timing triplet).

```vhdl
or #(3.2:4.0:6.3) o1(out, in1, in2); // min:typ:max
not #(1:2:3,2:3:5) (o,in); // min:typ:max for rise, fall
(b => y) = (2:3:4, 3:4:6, 4:5:8); // min:typ:max for rise, fall, and turnoff
```
Modeling Delay Accurately

A delay specification defines the propagation delay through a gate, net, or module. An input transition propagates to the output only after this delay. Delay is zero unless specified. Primitive turn-off delays apply only to the tristatable primitives.

- Rise delays are associated with transitions to 1: (0->1, Z->1, X->1)
- Fall delays are associated with transitions to 0: (1->0, Z->0, X->0)
- Turn-off delays are associated with transitions to Z: (0->Z, 1->Z, X->Z)
- Transitions to X occur as soon as possible and transitions from X take as long as possible
  - If you specify only one delay, all transitions use that delay
  - If you specify only rise and fall delays, 0->X and X->1 use the rise delay; 1->X and X->0 use the fall delay; while Z->X is potentially a rise or a fall, so uses the minimum of those delays; and X->Z uses the maximum
  - If you specify rise, fall, and turn-off delays, then a 1->X transition is potentially a rise or turn-off transition, and uses the minimum of these delays; an X->0 transition is potentially a fall transition, and uses that delay; an X->Z transition is a turn-off transition
  - If you specify six transitions, a 1->X is potentially a 1->0 or 1->Z, and uses the minimum of those two delays; an X->0 uses the maximum of a 1->0 or Z->0 delay; an X->Z uses the maximum of a 1->Z or 0->Z
  - The Cadence NC-Verilog simulator permits you to specify twelve delay values, which include transitions to and from X
The Specify Block

You describe module timing in a **specify** block:

- The **specify** block separates module timing from its functionality
- Start the block with the **specify** keyword and end it with **endspecify**
- Inside a **specify** block, you can:
  - Declare local timing constants
  - Define module timing paths and assign delays to them
  - Define pulse filtering controls for the module paths
  - Define timing checks
The Specify Block

A **specify** block defines the timing section of the module in a separate block. As a result, the functional verification becomes independent of the timing verification. The specify block can remain unchanged at different levels of abstraction.

A specify block is bounded by the keywords **specify** and **endspecify**, and must appear within a module definition.

Do not confuse specify parameters (keyword **specparam**, short for **specify parameter**) with module parameters (keyword **parameter**). You declare module parameters outside a specify block, and use them to configure instances of the module. You declare specparams inside a specify block, and use them as timing constants.

This subsection describes specify block parameters, module path delays, pulse filtering controls, and timing checks.
Specify Parameters

Inside a `specify` block you can declare local timing constants:

```verbatim
specparam identifier = value;
```

- Use specify parameters to simplify code maintenance
- Remember specify parameters are not module parameters!

This example uses specify parameters:

```verbatim`
timescale 1 ns / 1 ns
module nor_or(O, A, B, C);
output O;
input A, B, C;
    nor (n, A, B);
    or (O, C, n);
specify
    specparam tPAO=2, tPBO=3, tPCO=1;
    (A*>O)=tPAO;
    (B*>O)=tPBO;
    (C*>O)=tPCO;
endspecify
endmodule
```
The Specify Block (continued)

Specify Parameters

The `specparam` keyword declares specify parameters.

The `parameter` keyword declares module parameters.

The following summarizes the differences between the two parameter declarations:

- **Specify Parameters**
  - Declare with the `specparam` keyword
  - Declare and use *inside* specify blocks
  - All instances use the same definition
  - Use as a timing constant to simplify code maintenance
  - Cannot redefine for each instance

- **Module parameters**
  - Declare with the `parameter` keyword
  - Declare and use *outside* specify blocks
  - Each instance can have different values
  - Use to configure module instances
  - Redefine with the `defparam` keyword
Module Path Connections

You can define either full or parallel module timing paths:

- The `*>` token signifies a full connection
  - All inputs have a timing path to all outputs
- The `=>` token signifies a parallel connection
  - Each input has a timing path only to its corresponding output
  - Input and output groups must be the same size
- These tokens do not imply that a functional path exists!

### Full Module Path

- Inputs: a, b
- Outputs: w, y
- Connections: 4 paths
- Use `*>` to define path
  
  \[(a,b *\rightarrow w,y) = 15;\]

### Parallel Module Path

- Inputs: a1, a2
- Outputs: b1, b2
- Connections: 2 paths
- Use `=>` to define path
  
  \[(a1,a2 => b1,b2) = 15;\]
The Specify Block (continued)

Module Path Connections

This is a full path rise and fall delay specification from input \( r \) to outputs \( o1 \) and \( o2 \):
\[
(r \rightarrow o1, o2) = (1, 2);
\]

This is a full path delay specification from all inputs \( a \) to all outputs \( o \):
\[
(a[7:0] \rightarrow o[7:0]) = 6.3;
\]

This is a parallel path delay specification from inputs \( a \) and \( b \) to output \( out \). If the input and output group sizes do not match, then one group must have only one element:
\[
(a, b \rightarrow out) = 2.2;
\]

This is a parallel path delay specification from each bit of \( a \) to the corresponding bit of \( b \):
\[
(a[1:0] \rightarrow b[1:0]) = 3;
\]
State Dependent Path Delays

A state dependent path delay (SDPD) is a conditional delay

\[
\text{if} \quad \text{condition path}_-\text{declaration} \\
\text{ifnone path}_-\text{declaration}
\]

- The delay applies when a conditional expression is true or unknown
- The simulator uses the minimum delay if multiple SDPDs apply
- The simulator uses only the primitive delays (if any) if no SDPD applies

In this example, the delay from \( a \) or \( b \) to \( y \) is dependent on the state of the other input:

```
module XOR2 (y,a,b);
output y;
input a,b;
xor (y,a,b);
specify
  if ( a) (b=>y) = (5:6:7);
  if (!a) (b=>y) = (5:7:8);
  if ( b) (a=>y) = (4:5:7);
  if (!b) (a=>y) = (5:7:9);
endspecify
endmodule
```
The Specify Block (continued)

State Dependent Path Delays

Path delay can depend upon other input logic values.

You can model this situation with state dependent path delays (SDPDs).

The simulator applies the path delay if the expression is true or unknown (potentially true). The simulator uses the minimum delay if multiple SDPDs apply. Note that a path delay without a conditional expression always applies, but a state dependent path delay with shorter delay can override it.

If no path delay applies to an output transition, the simulator uses only the primitive distributed or lumped delays (if any exist and are enabled) for that output, otherwise applies zero delay to the output.

You can use the **ifnone** keyword to define a default path delay for an output.

```plaintext
ifnone path_declaration
```
Inertial and Transport Delay Modes

You can simulate module path delays in either inertial or transport delay mode:

- Inertial delay mode "swallows" input pulses of shorter duration than the delay
- Pure transport delay mode passes all input pulses, regardless of width

Inertial and Transport Delays

Inertial delay rejects all input pulses shorter than the delay

Transport delay passes all input pulses, regardless of their width

Note: The IEEE Std. 1364-1995 does not describe simulator timing mode control. Refer to the simulator vendor’s documentation for the command line options and/or compiler directives required to set this simulator timing mode.
Inertial and Transport Delay Modes

In this example, the intrinsic delay of the gate is 2 ns. If simulating inertial delays, a pulse of width 1 ns will not appear on the output.

The simulator always treats net and primitive delays as inertial. You can enable transport delay only for path delays, for example those delays that you specify in a `specify` block.

In pure transport delay mode, all pulses, regardless of their length, will appear on the output. You can change this behavior with the PATHPULSE$ `specparam` in the `specify` blocks.

With the Verilog-XL and NC-Verilog simulators, you can also control this behavior on a global basis with the `+pulse_e` and `+pulse_r` command line options.
Transport Delay Path Pulse Control

You can control transport delay module path pulse handling:

```
specparam PATHPULSE$ = (reject_value [, error_value]);
specparam PATHPULSE$path_source$path_destination = (reject_value [, error_value]);
```

- Use the special PATHPULSE$ specify block parameter
- You can specify one value (reject/pass) or two values (reject/error/pass)
- You can specify a default control and individual path controls

```
specify
  (in => out) = 5;
specparam
    PATHPULSE$ = (2,4);
endspecify
```
The Specify Block (continued)

Transport Delay Path Pulse Control

You can specify a default path pulse control for all outputs of the module:

```
specparam PATHPULSE$ = control_limits;
```

You can specify individual path pulse controls for any module path:

```
specparam PATHPULSE$<input$output = control_limits;
```

The simulator rejects input pulses shorter than the reject limit.

If you include an error limit, the simulator sets the output unknown (X) for input pulses that are shorter than the error limit, but at least as wide as the reject limit.

The example sets the module path delay to 5 ns, the reject limit to 2 ns, and the error limit to 4 ns. The simulator rejects any input pulse shorter than 2 ns. The simulator makes the path output unknown for an input pulse at least 2 ns wide but shorter than 4 ns. The simulator passes any pulse at least 4 ns wide.

Invoke the Verilog-XL and NC-Verilog simulators with the +pathpulse command line option to enable path pulse controls with the PATHPULSE$ specparam.
Timing Checks

Use timing checks to verify the timing constraints of the design.

Timing checks do the following:

- Record the simulation time of a data or reference event
- Wait for the second data or reference event
- Compare the elapsed time to the specified limit
- Report violations of the specified limit

The IEEE Std. 1364-1995 requires the following timing checks

<table>
<thead>
<tr>
<th>hold</th>
<th>period</th>
<th>setup</th>
<th>skew</th>
</tr>
</thead>
<tbody>
<tr>
<td>nochange</td>
<td>recovery</td>
<td>setuphold</td>
<td>width</td>
</tr>
</tbody>
</table>

The IEEE Std. 1364-2000 adds the following timing checks

| removal | recrem | timeskew | fullskew |
The Specify Block (continued)

Timing Checks

You can optionally have a timing check toggle a `reg` variable upon encountering a timing violation. You can use the `reg` transition to trigger the execution of behavioral code in your testbench, and to make the output of a user-defined primitive (UDP) go to the unknown (X) state. This course discusses UDPs in a later section.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$recovery</td>
<td>Minimum time from asynchronous control signal inactive edge to clock signal active edge</td>
</tr>
<tr>
<td>$removal</td>
<td>Minimum time from clock signal active edge to asynchronous control signal inactive edge</td>
</tr>
<tr>
<td>$recrem</td>
<td>Combined $recovery and $removal</td>
</tr>
</tbody>
</table>

The Cadence Verilog-XL and NC-Verilog simulators version 3.4 support the `$recrem` and `$removal` timing checks, but not the `$fullskew` and `$timeskew` timing checks.
Timing Checks

These are some of the more commonly used timing checks:

- The $setup$ system task checks data setup time before the clock
- The $hold$ system task checks data hold time after the clock
- The $setuphold$ system task combines the $setup$ and $hold$ tasks

```verilog
$setup(data, posedge clk, 4);
$hold(posedge clk, data, 3);
$setuphold(posedge clk, data, 4, 3);
```
Timing Checks (continued)

This is the syntax for the IEEE Std. 1364-1995 timing checks:

\[
\begin{align*}
\$hold & (reference\_event, data\_event, limit, notifier); \\
\$nochange & (reference\_event, data\_event, r\_offset, d\_offset, notifier);
\end{align*}
\]

\[
\begin{align*}
\$period & (reference\_event, limit, notifier);
\$recovery & (reference\_event, data\_event, limit, notifier);
\$setup & (data\_event, reference\_event, limit, notifier);
\$setup\_hold & (reference\_event, data\_event, s\_limit, h\_limit, notifier);
\$skew & (reference\_event, data\_event, limit, notifier);
\$width & (reference\_event, limit, threshold, notifier);
\end{align*}
\]

This is the syntax for two of the added IEEE Std. 1364-2000 timing checks:

\[
\begin{align*}
\$removal & (reference\_event, data\_event, limit, notifier);
\$recrem & (reference\_event, data\_event, rec\_limit, rem\_limit, notifier);
\end{align*}
\]

- The \textit{reference\_event} is a single edge transition
- The \textit{data\_event} is another single edge transition
- The \textit{limit} is the time limit used to detect timing violations
- The \textit{threshold} is an optional \texttt{$width$} argument — only pulses wider than \textit{threshold} can violate this check
- The \textit{notifier} is an optional Verilog \texttt{reg} that toggles upon a timing violation
Notifiers in Timing Checks

You can declare and use a notifier to identify and react to timing violations:

\$setuphold(ref\_event, data\_event, s\_limit, h\_limit, notifier);

- The notifier is an optional scalar `reg` argument
- The simulator toggles the notifier upon encountering a timing violation
- You can write a behavioral code block that triggers on the notifier transition
- You can define a user-defined primitive (UDP) to transition to the unknown (X) state when an input (the notifier) transitions
Timing Checks (continued)

When a timing check violation occurs, the simulator reports the violation to its display and to its log file, and toggles the notifier `reg`. You can create behavioral code or a user-defined primitive (UDP) that is sensitive to the notifier transition, and that sets the device output to the unknown (X) state. The simulator does not do this for you. The timing check does not by itself have any affect on simulation values.
Using a Notifier in Behavioral Code

`timescale 1 ns / 1 ns
module dff (q,d,clk,rst);
output q;
input d,clk,rst;
reg FLAG; // 1-bit notifier
.specify (clk => q) = (2:3:4); $setup(d,posedge clk,2,FLAG); $hold(posedge clk,d,2,FLAG);
.endspecify
endmodule

module test;
reg d,clk,rst;
dff d1 (q,d,clk,rst);
.. . // test
always @(d1.FLAG)
begin
  rst = 1; #10 rst = 0;
end
endmodule
Timing Checks (continued)

Initially, the value of notifier $FLAG$ is unknown (X). The simulator sets its value to 1 upon the first timing violation. Thereafter the simulator toggles its value upon each subsequent timing violation. It is only the change of the notifier value that is important, not the actual value.

In this example:

- The $d$ input changes 1 ns before the $clk$ rising edge, thus violating the 2 ns setup requirement
- The $d$ input changes 1 ns after the $clk$ rising edge, thus violating the 2 ns hold requirement
Conditional Timing Checks

You can control a timing check with a conditional expression:

- Use the &&& operator and an expression evaluating a single scalar signal
- Apply the conditional expression to the check event
- The simulator applies the timing check if the expression is true or unknown

module dff (q, qb, d, clk, rst);
output q, qb;
input d, clk, rst;
    // instantiate the dff UDP
    UDP_DFF(q_int, d, clk, rst);
buf (q, q_int);
not (qb, q_int);
    // create timing checks
specify
    $setup(d, posedge clk &&& ~rst, 2);
    $hold(posedge clk, d &&& ~rst, 2);
endspecify
endmodule
Timing Checks (continued)

Use conditional timing checks to reduce pessimism. The simulator performs the timing check only when the conditional expression is true or unknown. In this example, the simulator will not perform the setup and hold checks when the $rst$ signal is true (high).

The conditional expression can contain only a single scalar operand. You can perform the following operations on this operand:

- Negate with the bitwise negation operator (~)
- Compare to a scalar constant (0, 1, Z, or X) with an equality operator (== or !=)
- Compare to a scalar constant with an identity operator (=== or !==)

The simulator performs the timing check if the expression evaluates to 1, Z, or X.

As the conditional expression for a timing check cannot involve more than one signal, you may need to insert dummy logic to combine multiple signals into one scalar signal for use in the timing check.
Negative Timing Checks

You can perform negative timing checks:

- Use the `$setuphold` system task
- Negative setup time: You can set up the data after the clock event
- Negative hold time: You can release the data before the clock event
- Negative timing checks are necessary for high speed technologies

This example specifies a positive setup and negative hold check:

```vhdl
$setuphold(posedge clk, d, 6, -2);
```
Timing Checks (continued)

Use of negative timing checks may require cell modeling changes which this course does not discuss.
The SDF File

The Standard Delay Format (SDF) is a standard textual format for representing timing data. This data can include:

- Data identifying the design, instance, type, or library
- Scaling, environment, technology, and user-defined parameters
- Module path, device, interconnect, and port delays
- Incremental and absolute delays
- Conditional module path delays
- Hold, period, recovery, setup, skew, and width timing checks
- Timing environment constraints

Verilog simulators and timing analysis tools use SDF data.
The SDF File

The SDF allows timing data to be shared between tools. You can feed critical path information from synthesis tools forward to place and route tools. You can feed wire interconnect delay information from place and route tools back to simulation.

For deep-submicron ASIC technologies the interconnect delay dominates the silicon delay. The large majority of the information in a typical SDF file represents interconnect delay.
INTERCONNECT Delay

INTERCONNECT delay is the path delay in the device interconnect.

```
(DELAY (ABSOLUTE | INCREMENT (INTERCONNECT path1 path2 delay)))
```

This is an example of the SDF specification of an INTERCONNECT delay:

```
(CELL (INSTANCE top)
   (DELAY (ABSOLUTE
      (INTERCONNECT u1.u1.u2.o1 u1.u2.u1.i2 (5:6:7) (5.5:6:6.5) ))
   )
)
```
The SDF File (continued)

INTERCONNECT Delay

For the INTERCONNECT keyword:

- path1 is an output or inout, and may include an instance array index
- path2 is an input or inout, and may include an instance array index

The example specifies an interconnect delay from a device output to a device input. The example specifies minimum, typical, and maximum delays for rising and falling transitions.
IOPATH Delay

IOPATH delay is module path delay from an input port to an output port.

(DELAY (ABSOLUTE | INCREMENT (IOPATH path1 path2 delay)))

(DELAY (ABSOLUTE | INCREMENT (COND expr (IOPATH path1 path2 <delay>)))))

(DELAY (ABSOLUTE | INCREMENT (CONDELSE (IOPATH path1 path2 <delay>))))

This is an example of the SDF specification of an IOPATH delay:

(CELL (INSTANCE top.u1.u2)
  (DELAY (ABSOLUTE
    ( IOPATH in o1 (1:2:3) (1:3:4))
    ( COND en (IOPATH in o2 (2:4:5) (4:5:6) (4:5:7))
      ( IOPATH (posedge clk) o2 (2:3:4) (4:5:6)))
  )
)
The SDF File (continued)

IOPATH Delay

For the IOPATH keyword:

- \textit{path1} is an \textit{input} or \textit{inout}, and may include a \textit{posedge} or \textit{negedge} qualifier

- \textit{path2} is an \textit{output} or \textit{inout}, and may include an instance array index

For the COND keyword, \textit{expr} is a boolean expression of port states

The example specifies IOPATH delays from a device inputs to its outputs. The example specifies minimum, typical, and maximum delays for rising and falling transitions. For the conditional delay it also specifies a turn-off delay.

Some simulators annotate module path delays and timing checks only onto pre-existing specify-block targets. For this example, the \textbf{specify} block must contain:

```plaintext
specify
  (in *> o1) = (1:2:3, 1:3:4);
  if (en) (in *> o2) = (2:4:5, 4:5:6, 4:5:7);
  (posedge clk *> o2 = (2:3:4, 4:5:6);
endspecify
```
Summary

In this section you learned about:

- Modeling delay
- Specify blocks
- The standard delay format (SDF)
Review

1. Which type of delay specification is most accurate?
2. If you specify six delay values for transitions to 0, 1, and Z, what delay does the simulator use for transitions from Z to X?
3. What is the difference between a module `parameter` and a `specparam`?
4. Where would you specify timing checks for a Verilog simulation?
5. Where would you specify interconnect delay for a Verilog simulation?
Review

1. Which type of delay specification is most accurate?
   The module path delays are most accurate, as you can specify pin-to-pin path delays.

2. If you specify six delay values for transitions to 0, 1, and Z, what delay does the simulator use for transitions from Z to X?
   The simulator assigns the shortest applicable delays to transitions to X and the longest applicable delays to transitions from X, so the simulator will use the shorter of the (Z->0) and (Z->1) delays.

3. What is the difference between a module parameter and a specparam?
   The purpose of a module parameter is to accommodate instance-by-instance reconfiguration of a module, so the value of a module parameter may be different for each instance of the module. The purpose of a specparam is to define a constant for use in the specify block. This constant is the same for all instances of the module.

4. Where would you specify timing checks for a Verilog simulation?
   Specify timing checks in a specify block. You can modify and replace these checks by annotating with SDF information.

5. Where would you specify interconnect delay for a Verilog simulation?
   Specify interconnect delay in an SDF file.
About Lab 4

The objective of this lab is to use module path delays and timing checks to improve the timing accuracy of a model.

In this lab, you will:

- Add module path delays and timing checks to a flip-flop model
- Simulate the flip-flop using minimum, typical, and maximum timing specifications
About Lab 4
Chapter 9: Using Compiler Controls

Objectives

In this section you will learn about:

- Creating and using Verilog model libraries
- Selecting the simulation delay mode
- Protecting your Verilog source code
- Additional simulator-specific topics
Using Compiler Controls

This section presents Cadence Verilog-XL and NC-Verilog simulator compiler features related to its interpretation of source code, and simulation timing modes. It ends with a discussion of the simultaneous simulation of Verilog and VHDL source.
Verilog Model Libraries

There is a large base of vendor-supplied Verilog libraries:

- ASIC and FPGA vendors develop and provide technology-specific libraries
- Logic synthesis tools instantiate components from the libraries
- The simulator scans component libraries to define instantiated components

A composite library is used by several tools. It can contain information for:

- Logic synthesis (such as Ambit BuildGates)
- Logic simulation (such as Verilog-XL and NC-Verilog)
- Timing analysis (such as Pearl)
Verilog Model Libraries

Each library component contains both functionality and tool-specific timing and technology information.
Modeling Verilog Libraries

To create Verilog model libraries:

- Model each component (cell) as a separate module
- Place related modules in either one file or one directory

You can model the functionality of each cell at two levels:

- Structural
  - Verilog built-in primitives or User-Defined Primitives (UDPs)
  - Use for combinational logic and simple sequential logic
- Behavioral
  - Procedural blocks and assignment statements
  - Use for large or complex components such as RAM or ROM
Verilog Model Libraries (continued)

Modeling Verilog Libraries

If you place the modules in a library directory, you must name each file the same as the module it contains. There are no specific filename extension requirements for Verilog source, but you would typically give the file name an extension such as:

<table>
<thead>
<tr>
<th>Extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.v</td>
<td>Generic Verilog source</td>
</tr>
<tr>
<td>.vg</td>
<td>Verilog gate level source</td>
</tr>
<tr>
<td>.vr</td>
<td>Verilog RTL level source</td>
</tr>
<tr>
<td>.vb</td>
<td>Verilog behavioral level source</td>
</tr>
<tr>
<td>.vt</td>
<td>Verilog testbench source</td>
</tr>
</tbody>
</table>
Modeling Verilog Libraries

The simulator needs to know which modules are library (leaf) cells:

- Only library cells contain logic in a hierarchical gate-level design
- Delay calculation and annotation applies only to library cells
- You typically do not (and sometimes cannot) debug library cells

You need to define a library cell for the simulator:

- Enclose the cell between the `celldefine and `endcelldefine directives
- Define the logic function of the cell
- Specify the cell timing in a specify block
Enclose the library cell definition between the `celldefine` and `endcelldefine` directives.

Insert the `timescale` directive before the cell module definition to define the time units and time precision for the cell.

Model the cell definition in two parts:

- Describe the functionality of the cell
- Describe the timing behavior of the cell

```verilog
`celldefine
`timescale 1ns / 100ps
module full_adder(cout, sum, a_in, b_in, c_in);
input a_in, b_in, c_in;
output cout, sum;
  // Functional description
  . . .
  // Timing information
  . . .
endmodule
`endcelldefine
```
Using Verilog Libraries

To use Verilog model libraries with the Cadence Verilog-XL and NC-Verilog simulators:

- To use a library file, include the \(-v\) command-line option
  \[-v <library_file_name>\]

- To use a library directory, include the \(-y\) and \(+libext+\) command-line options
  \[-y <library_directory_name> +libext+<library_file_extension>\]

**Note:** With the \(-y\) option, the simulator does **not** assume the \(.v\) extension!
Verilog Model Libraries (continued)

Using Verilog Libraries

When you use a library directory, you must use the \texttt{+libext+} option to specify the file extensions, for example:

\begin{verbatim}
+libext+.v
\end{verbatim}

You can specify an ordered list of file extensions for the \texttt{+libext+} option, for example:

\begin{verbatim}
+libext+.v+.vg+.vb++
\end{verbatim}

The simulators would first look for a file with a \texttt{.v} extension, then a \texttt{.vg} extension, then a \texttt{.vb} extension, and finally a file with no extension (the consecutive "++" characters indicate a null extension).

As the simulators actually search for file suffixes, rather than extensions, you do not need to use the dot (\texttt{.}) character. You could, for example, search for files ending in \texttt{_v} or \texttt{src}.

The simulators scan the library directories and files to resolve instantiated modules. They compile only the modules that they need, not the entire contents of the library. If you inadvertently enter the library file name on the command line without the \texttt{-v} option, the simulators will not treat the file as a library, but will instead compile all the module definitions it contains, even those it does not need.
### Scanning Library Files

Each `-v` option specifies one library file.

```verilog
verilog test.v design.v -v library_file_1.v -v library_file_2.v

// library_file_1.v
`timescale 1ns/10ps
`celldefine

module and2 (. . .);
  . .
endmodule

module mux21 (. . .);
  . .
endmodule

module dffrb (. . .);
  . .
endmodule

`endcelldefine
```
Using Verilog Libraries (continued)

The simulator scans the library files to resolve a reference to a not yet defined module or UDP. The simulator only compiles the definitions that it needs. If the library file contains more than one description of the same cell, the simulator compiles only the first definition that it encounters.
Scanning Library Directories

Each \(-y\) option specifies one library directory.

The \(+\text{libext}\) option specifies the valid file extensions.

```
verilog test.v design.v -y library_directory +libext+.v

// library_directory/and2.v
`timescale 1ns/10ps
`celldefine
module and2 (. . .);
  . . .
endmodule
`endcelldefine

// library_directory/mux21.v
`timescale 1ns/10ps
`celldefine
module mux21 (. . .);
  . . .
endmodule
`endcelldefine
```
Using Verilog Libraries (continued)

The simulator scans the files in a library directory to resolve a reference to a not yet defined module or UDP. The simulator only compiles the cell definitions that it needs.
The `uselib Compiler Directive

The `uselib compiler directive:

`uselib file=file_name_path dir=directory_name_path
libext=file_extension

- Defines where the simulator searches for subsequent module definitions
- Is active until the compiler encounters another `uselib or a `resetall
- Overrides any command-line options that configure the library search

If the simulator cannot find the component in the `uselib location, it does **not** revert to the libraries specified on the command line with the \-v or \-y options.

**Note:** With the `uselib directive, the simulator **does** assume the .v extension!
Using Verilog Libraries (continued)

The `uselib` compiler directive specifies where the simulator searches for the definitions of undefined modules and UDPs instantiated in a design description.

Each `uselib` directive explicitly defines the libraries in which to find any subsequent undefined instances, until the compiler encounters another `uselib` directive, which completely redefines the search. Without `uselib` compiler directives, the simulator searches libraries specified with the `-v` and `-y` command line options.

Use an empty `uselib` directive to deactivate the preceding `uselib` directive.

You can use the `define` compiler directive and/or the `+define+` command-line option to define the paths with which the `uselib` compiler directive configures the search. This enables you to use brief macro names in the `uselib` compiler directives anywhere in the design description, thus localizing the search path definitions, simplifying future changes.

```
`define TTL_LIB dir=/libs/TTL/ libext=.v
`define TTL_UDP file=/libs/TTL/udp.lib
`uselib `TTL_LIB `TTL_UDP
```
Example of the `uselib Compiler Directive

A single `uselib directive can specify more than one library search location.

In this example, cells in the TTL library directory can instantiate UDPs defined in the udp.lib library file:

```verilog
module adder (c_out, sum, a, b, c_in);
output c_out, sum;
input a, b, c_in;

`uselib dir=/libs/FAST_LIB/
SN7486 u1 (half_sum, a, b);
`uselib dir=/libs/TTL/ libext=.v file=/libs/TTL/udp.lib
SN7408 u2 (half_c, a, b);
SN7408 u3 (tmp, c_in, half_sum);
SN7486 u4 (sum, c_in, half_sum);
SN7432 u5 (c_out, tmp, half_c);
`uselib
endmodule
```
Using Verilog Libraries (continued)

The first `uselib` directive directs the compiler to obtain all component definitions from the FAST_LIB library directory. To resolve the definition of `u1`, the compiler must find a file named `SN7486.v` in the FAST_LIB library directory.

The second `uselib` directive directs the compiler to obtain all component definitions from the TTL library directory or `udp.lib` library file. To resolve the definition of `u4`, the compiler must again find a file named `SN7486.v`, but this time in the TTL library directory.

The TTL library components presumably reference UDPS defined in the `udp.lib` library file.

The third (null) `uselib` directive directs the compiler look to the command line to configure the library search. You can alternatively use the `resetall` directive, which resets all directives to their reset state (if they have one).
Producing Case Insensitive Source

Verilog is a case-sensitive language:

- Use lowercase for Verilog keywords
- Use any case for identifiers
- Use the \(-u\) command-line option to compile your design in case-insensitive mode
  — The simulator maps all identifiers to upper case, in effect making the simulation case-insensitive
- Use the \(-c\) option if you want to compile your design, but not simulate it
- Use the \(-d\) option to decompile and output the case-insensitive design
Producing Case Insensitive Source

Verilog is a case-sensitive language:

- All Verilog keywords are in lowercase, for example, the keyword `input`, not `INPUT`
- Verilog identifiers (for example `sel & SEL`) that do not match in case are different names

Use the `-u` command-line option to run your simulation in case-insensitive mode. The simulator converts identifiers to uppercase. Keywords remain in lowercase.

*What happens if you simulate this model in case-insensitive mode?*

```verilog
module MUX2_1 (out, sel, inb, ina);
output out;
input sel, inb, ina;
  not (SEL, sel);
  and (sela, ina, SEL);
  and (selb, inb, sel);
  or (out, sela, selb);
endmodule
```

In this example, signals `sel` and `SEL` are different signals. If you use the `-u` option, they become the same signal, and the simulation fails. This command-line option helps uncover errors that would occur if you used the model with case-insensitive tools.
Compiler Directives

Compiler directives direct the compiler’s interpretation of subsequent source.
Although Verilog describes several, compilers can handle them differently:

- The `resetall` directive resets compiler directives to their default state
  — Cadence Verilog simulators do not reset text macros
- The `undef` directive undefines a single text replacement macro
  — The NC-Verilog simulator supports `undefineall` to undefine all macros
- The `include` compiler directive includes another source file
  — You can use the `+incdir` command-line option to specify additional directories in which to find the included files

The simulator first looks for the included file in the current directory, then searches the directories in the order that you list them with `+incdir` on the command line.

`+incdir+directory1+directory2+...directoryN`
## Compiler Directives

A compiler directive is active from the point that the compiler encounters it in the source stream until another compiler directive deactivates or overrides it. This means that a compiler directive can be active across multiple source files.

The IEEE Std. 1364-1995 requires only the following compiler directives:

<table>
<thead>
<tr>
<th><code>celldefine</code></th>
<th><code>endif</code></th>
<th><code>timescale</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>default_nettype</code></td>
<td><code>ifdef</code></td>
<td><code>unconnected_drive</code></td>
</tr>
<tr>
<td><code>define</code></td>
<td><code>include</code></td>
<td><code>undef</code></td>
</tr>
<tr>
<td><code>else</code></td>
<td><code>nounconnected_drive</code></td>
<td></td>
</tr>
<tr>
<td><code>endcelldefine</code></td>
<td><code>resetall</code></td>
<td></td>
</tr>
</tbody>
</table>

The IEEE Std. 1364-1995 describes, but does not require, the following compiler directives:

<table>
<thead>
<tr>
<th><code>default_decay_time</code></th>
<th><code>delay_mode_path</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>default_trireg_strength</code></td>
<td><code>delay_mode_unit</code></td>
</tr>
<tr>
<td><code>delay_mode_distributed</code></td>
<td><code>delay_mode_zero</code></td>
</tr>
</tbody>
</table>
Defining Text Macros

You can use compiler text replacement macros to configure your design.

You can use the `define` command-line argument to define the text macros.

Syntax:

```
#define MACRO_NAME
#define MACRO_NAME = MACRO_TEXT
```

Example:

```
`define gate and
module test;
    reg a, b;
    `gate (c, a, b);
initial begin
    a = 0; b = 1;
    $monitor ($time,, c, a, b);
    #1 $finish;
end
endmodule
```

Session:

```
verilog test.v `define+gate=or
```
Defining Text Macros

There is no defined limit to the size or number of text replacement macros you can define.

In the example, the source code defines the gate text replacement macro and assigns to it the string value "and". The +define+ command-line option defines the macro and assigns to it the string value "or". The command-line option overrides the source code definition; the compiler will warn you that it is ignoring the source code definition.

You can use a text replacement macro to conditionalize compilation of source code. For this purpose you do not need to assign a value to the macro.

```
`ifdef DEBUG
   $display("DEBUG is turned ON.");
`endif
```
Pre-defined Text Macros

The Cadence Verilog-XL and NC-Verilog simulators pre-define text macros that you can use to conditionally compile source code:

- For Verilog-XL and NC-Verilog:
  - verilog

- For NC-Verilog only:
  - INCA
Defining Text Macros

Pre-defined Text Macros

You can use the `verilog` text macro to conditionalize compilation of Cadence-specific constructs, and the `INCA` text macro to conditionalize compilation of NC-Verilog specific constructs. The NC-Verilog `undefineall` compiler directive does not undefine these pre-defined text macros (although the `undef` directive does).

```
`ifdef verilog
  $display("Running a Cadence Verilog simulator.");
`ifdef INCA
  `ifdef INCA
    $display("Running Cadence NC-Verilog.");
  `endif
`endif
```

```
Selecting Simulation Delay Mode

This subsection discusses selection of simulation delay mode:

- Selecting the minimum, typical, or maximum delays
- Selecting only the distributed delays, or only the module path delays
- Selecting inertial or transport delay for module paths
Selecting Simulation Delay Mode
Selecting Delay Values

The simulator by default simulates with typical timing values.

You can choose to simulate with minimum, typical, or maximum timing values.

Select the timing values with these command line options:

- +mindelays
- +typdelays
- +maxdelays
Selecting Simulation Delay Mode

Selecting Delay Values

For the \texttt{+mindelays}, \texttt{+typdelays}, and \texttt{+maxdelays} timing delay options to have effect, you must specify timing values in the \texttt{min:typ:max} format. This delay specification format is called a timing triplet.
## Selecting Delay Mode

You can further specify the simulation timing mode, either globally using command line options or on a section-by-section basis using compiler directives.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+delay_mode_path</td>
<td><code>delay_mode_path</code></td>
<td>Utilizes only specify block timing. Ignores all distributed primitive and wire delay.</td>
</tr>
<tr>
<td>+delay_mode_distributed</td>
<td><code>delay_mode_distributed</code></td>
<td>Utilizes only distributed primitive and wire delay. Ignores specify block timing.</td>
</tr>
<tr>
<td>+delay_mode_unit</td>
<td><code>delay_mode_unit</code></td>
<td>Reduces non-zero distributed timing to 1 unit. Ignores specify block timing.</td>
</tr>
<tr>
<td>+delay_mode_zero</td>
<td><code>delay_mode_zero</code></td>
<td>Reduces non-zero distributed timing to 0 units. Ignores specify block timing.</td>
</tr>
</tbody>
</table>
Selecting Simulation Delay Mode

Selecting Delay Mode

The simulator by default first uses distributed primitive and net delays (if any), then extends the delay time (if necessary) to meet the module path delay specifications in the `specify` block.

If you specify the path delay mode (`+delay_mode_path` command-line option or `delay_mode_path` compiler directive), the simulator uses only module path delays.

If you specify the distributed delay mode (`+delay_mode_distributed` command-line option or `delay_mode_distributed` compiler directive), the simulator uses only distributed primitive and net delays.

If you specify the unit delay mode (`+delay_mode_unit` command-line option or `delay_mode_unit` compiler directive), the simulator uses only distributed primitive and net delays, and reduces each of those non-zero delays to one (of the simulator time precision units).

If you specify the zero delay mode (`+delay_mode_zero` command-line option or `delay_mode_zero` compiler directive), the simulator ignores all timing information.

Any of these timing modes can cause the simulator to ignore delay in combinational feedback paths, for example in a clock oscillator, thus causing the simulator to hang. For such constructs you may elect to use the local compiler directives, rather than the global command-line options.

These timing mode command-line options and compiler directives do not affect procedural timing controls.
Inertial and Transport Delay Modes

The simulator by default simulates delays as inertial. Inertial delay "swallows" input pulses shorter than the propagation delay.

Use the +transport_path_delays command-line option to simulate module path delays in transport mode. Transport delay passes all input pulses, regardless of width, to the output. You cannot apply this mode to primitive and net delays.

verilog source.v +transport_path_delays

**Inertial and Transport Delays**

- **Inertial delay** rejects all input pulses shorter than the delay.

- **Transport delay** passes all input pulses, regardless of their width.

```
Inertial delay rejects all input pulses shorter than the delay

Transport delay passes all input pulses, regardless of their width
```
Selecting Simulation Delay Mode (continued)

Inertial and Transport Delay Modes

Inertial delay mode does not pass input pulses shorter than the propagation delay. This delay mode characterizes switching delay. The simulator always simulates primitive and net delay in inertial delay mode, and by default also simulates path delay in inertial delay mode.

Transport delay mode passes all input pulses, regardless of width, to the output. This delay mode characterizes transmission lines. Use the +transport_path_delays command-line option to simulate module path delays in transport delay mode. Use the +transport_int_delays command-line option to simulate interconnect delays in transport delay mode.

In pure transport delay mode, the simulator passes all input pulses, regardless of width, to the output. You can control module path pulse handling with the PATHPULSE$ specparam in the specify blocks. Use the +pathpulse command-line option to enable this module path pulse control.

You can use the +pulse_e/n and +pulse_r/m command line options to specify global module path pulse control, and the +pulse_int_e/n and +pulse_int_r/m command line options to specify global interconnect pulse control.
Verilog-XL Module Path Restrictions

For the Verilog-XL simulator, module path delays must end at the output terminal of a single accelerated primitive.

Illegal

![Behavioral Code Block](A,B *> O) = 2;

Legal

![Behavioral Code Block](A,B *> O) = 2;

Illegal

![Behavioral Code Block](A,B *> O) = 2;

Legal

![Behavioral Code Block](A,B *> O) = 2;
Verilog-XL Module Path Restrictions

For the Verilog-XL simulator, the destination of a module path must be the output of a single acceleratable primitive. Verilog-XL issues an error message if it is not:

Error! Path delay output is not accelerated due to . . . [Verilog-PDOMBA<number>]

Error! Multiple path delays defined to node . . .
Path delay outputs must have only one driver within the module [Verilog-PDOMOD]

For the Verilog-XL simulator, you can buffer a module output that is not acceleratable, for example, that is driven by a procedural block. Insert a **buf** primitive at each module output that does not need to pass a Z value. Insert a **mos** primitive at each module output that does need to pass a Z value.
Verilog-XL Accelerated Objects

Verilog-XL accelerates nets and primitives, with the following exceptions:

- Nonaccelerated nets
  - Net declared with the `vectored` keyword
  - Net with a non-zero delay
  - Net that is or has been forced during the simulation session
  - Net that has a continuous assignment made to it (unless `+caxl` is used)

- Nonaccelerated primitives
  - Bidirectional primitives `tran tranif0 tranif1 rtran rtranif0 rtranif1`
  - `buf` and `not` gate with more than one output
  - Primitive with an input connected to a bit or part-select of a vector register
  - Primitive with an expression on an input (unless `+caxl` is used)
  - Primitive with a non-constant delay expression
  - Primitive with delay in excess of 65535 distinct delays
Verilog-XL Module Path Restrictions (continued)

Verilog-XL Accelerated Objects

The `$shownonxl()` system task locates non-XL structures. It takes a module instance as its argument, and scans that instance. With no argument, it scans the entire design hierarchy.

Verilog-XL does not accelerate a forced net. After you `force` a net, Verilog-XL no longer accelerates it, even after you `release` it. Use `$showforces` to show currently forced nets.

The Verilog-XL XL algorithm does not accelerated bidirectional primitives. You can accelerate bidirectional primitives with the Switch-XL algorithm.

The XL algorithm supports no more than 65,535 distinct gate delays. After the simulator reaches this limit during compilation, it does not accelerate subsequent gates having distinct delays. The delays `(3, 4)` and `(3, 2)` are distinct, for example, but `#3` and `(3,3)` are not.
Protecting Source Code with Verilog-XL

You can protect proprietary information in a design:

- You can protect Verilog source descriptions in two ways:
  - Automatically protect all compiled source with the `+autoprotect` option
  - Protect selected modules or regions between the `protect` and `endprotect` compiler directives with the `+protect` option
- The protection mechanism creates a new file containing protected code
Protecting Source Code with Verilog-XL

When the simulator protects source code, it encloses the protected source code between the compiler directives `protected` and `endprotected`.

When you simulate protected code, these compiler directives direct the simulator to not allow the user access to the protected areas.

The Cadence NC-Verilog simulator reads protected code. It does not generate protected code.

You can use the Cadence export model packager to protect your proprietary models. The Cadence export model packager generates OMI-compliant models usable by all simulators that support the OMI standard.

**Note:** Verilog-XL ignores the `include` compiler directive when generating protected code. The protected code will not contain the included source. You can work around this by first editing your source file to directly include the source that the compiler directive would have included.
Protecting All of Your Verilog Source

Use the +autoprotect command-line option to protect all of your Verilog source.

```
verilog AND2.v +autoprotect
```

This creates a new source file with only the module name readable.

**Original source:**

```
module AND2(a, b, c);
output a;
input b, c;
   and al(a, b, c);
specify
   (b => a) = (1.2,2.0);
   (c => a) = (1.4,1.8);
endspecify
endmodule
```

**Protected source:**

```
module AND2 `protected
8Q@RMSQH5DT^<oAXO3N^VVhaD89ldTiYhe
@n==0Pi@Q\JifJ[Z<ai766?ReA0RPaQ94H
GHN60=Y[KQgBdW4dO3662XYUXK=;CM=EVC
cZo2@65`Lm<XL:0VR[CAUbShNPHne2
IVFe3ZFa;pB5fO6kpT_<h$
 `endprotected endmodule
```
Protecting Source Code with Verilog-XL (continued)

Protecting All of Your Verilog Source

With +autoprotect you can protect numerous modules without editing each module description.

Verilog-XL by default appends "p" to the design file name. With this command line, Verilog-XL creates the design.vp protected file.

`verilog design.v +autoprotect`

You can specify the extension with the +autoprotect command-line option. With this command line, Verilog-XL creates the design.v.prot protected file.

`verilog design.v +autoprotect.prot`
Protecting Selected Regions in a Source Description

Use the `+protect` command-line option and the `protect` and `endprotect` compiler directives to protect selected regions of your Verilog source.

```
verilog AND2.v +protect
```

This creates a new source file with only selected regions protected.

**Original source:**

```verilog
module AND2(a, b, c);
output a;
input b, c;
`protect
    and a1(a, b, c);
specify
    (b => a) = (1.2,2.0,2.8);
    (c => a) = (1.4,1.8,2.6);
endspecify
`endprotect
endmodule
```

**Protected source:**

```verilog
module AND2(a, b, c);
output a;
input b, c;
`protected
    `endprotected
endmodule
```
Protecting Selected Regions in a Source Description

With the `+protect` command-line option and the `protect` and `endprotect` compiler directives, you can make portions of the description, such as the port lists, visible. The Verilog-XL simulator protects the region between `protect` and `endprotect` and leaves the rest unchanged.

Verilog-XL by default appends "p" to the design file name. With this command line, Verilog-XL creates the `design.vp` protected file.

```
verilog design.v +protect
```

You can specify the extension with the `+protect` command-line option. With this command line, Verilog-XL creates the `design.v.prot` protected file.

```
verilog design.v +protect.prot
```
Coexecuting VHDL and Verilog with NC-Sim

You can include VHDL models in Verilog designs, and coexecute them with the NC-Sim simulator. Coexecution uses a single process, and is more efficient than cosimulation.

- Precompile the VHDL models with `ncvhdll`
- Run `ncshell` to generate Verilog wrappers for each VHDL model
  
  You can instantiate a VHDL object directly (without a shell) that does not do port type conversion, if you use its defined name and port order

- Assure that an NC-Sim coexecution simulator license is available

- Run the NC-Sim simulation, in either "library-based" or "Verilog-XL" mode

Here is an example wrapper:

```verilog
module MULTIPLIER (IN1, IN2, OUT1, CLK)
(* integer foreign = "VHDL(event) WORKLIB.MULTIPLIER"; *)
input [15:0] IN1, IN2;
input CLK;
output [31:0] OUT1;
endmodule
```
Coexecuting VHDL and Verilog with NC-Sim

You can also use the ncsHELL utility to generate shells to import LMSFI, FMI, or Swift models into a VHDL design.

The NC-Sim option is licensed separately from the NC-Verilog or NC-VHDL options. If the license is available and if the design incorporates a VHDL model, the simulator automatically and efficiently coexecutes the design partitions in a single execution kernel.

You can run the NC-Sim simulator in either the classical "library-based" mode (invoke ncvlog, ncelab, and ncsim), or the "Verilog-XL" mode (invoke neverilog).

There are some considerations in coexecution:

- While you can scope and probe inside instantiated models, you must follow the access rules for coexecution and for VHDL.
- Port order, size, type, and index direction of the shell must match the VHDL module.
- You cannot pass a Verilog tran gate signal to a VHDL inout port.
- You cannot use save, restart, or reset when coexecuting.
- You can reference hierarchical names in your Verilog design hierarchy (make Verilog out-of-module references or OOMRs). These references can pass through (but not target) VHDL objects. If any VHDL object name in the path uses upper-case or illegal characters, or is a Verilog keyword, you must use the nmp utility to determine its valid generated Verilog name. Similarly, you must remember to change VHDL instance indices \((n)\) into Verilog indices \([n]\).
Summary

In this section you learned about:

- Creating and using Verilog model libraries
- Selecting the simulation delay mode
- Protecting your Verilog source code
- Additional simulator-specific topics
Summary
Review

1. With what command-line options do you specify library names?

2. If the simulator cannot find the definition of an instance in the library specified with the `uselib` compiler directive, where else does it look?

3. With what simulator option do you map all identifiers to upper case?

4. True or false: The `resetall` compiler directive resets all compiler directives to their default state and undefines all defined text replacement macros.

5. The simulator by default uses which timing values?

6. The simulator by default uses what timing mode?
Review

1. What command-line options do you use to specify library names?
   You use the \(-v\) option, and/or the \(-y\) and \(+libext+\) options.

2. If the simulator cannot find the definition of an instance in the library specified with the `uselib` compiler directive, where else does it look?
   The simulator does not look anywhere else.

3. With what simulator option do you map all identifiers to upper case?
   Use the \(-u\) command-line option to map all identifiers to upper case, thus effectively making the simulator case-insensitive with respect to identifiers.

4. True or false: The `resetall` compiler directive resets all compiler directives to their default state and undefines all defined text replacement macros.
   The IEEE Std. 1364-1995 does not specify what should happen to text macros when you issue the `resetall` compiler directive. Cadence Verilog simulators leave them unchanged. To reset a text macro, you undefine it with the `undef` compiler directive.

5. The simulator by default uses which timing values?
   The simulator by default uses the typical timing values only.

6. The simulator by default uses what timing mode?
   The simulator by default first uses distributed primitive and net delays (if any), then extends the delay time (if necessary) to meet the module path delay specifications in the `specify` block.
About Lab 5

This lab is in two parts.

The objective of the first part is to use command-line options and compiler directives to specify component libraries.

In this part, you will:

- Use command-line options to specify library files and directories
- Use compiler directives to specify library files and directories

The objective of the second part is to use command-line options to specify the simulation timing values and mode.

In this part, you will:

- Use command-line options to select minimum, typical, or maximum delays
- Use command-line options to select path, distributed, or unit delays
About Lab 5
Chapter 10: Verilog Operators

Objectives

In this section you will learn about:

- Sizing and signing of expressions
- The Verilog operators and their precedence
- How to use the Verilog operators
Verilog Operators

This section explains the sizing and signing of operands, lists the Verilog operator order of precedence, and thoroughly reviews the Verilog operators.
Sizing and Signing

Verilog simulators follow these sizing and signing rules:

- They left-extend unsized literals to 32 bits with:
  - 0 if the leftmost bit of the literal is 0 or 1
  - Z or X (respectively) if the leftmost bit of the literal is Z or X
- They zero-extend operands to matching sizes before operating on them
- They truncate or zero-extend high-order RHS bits to meet the LHS size
- They perform bitwise assignments of signed values to unsigned registers

```verilog
module sign_size;
reg [3:0] a, b;
reg [7:0] c;
    initial begin
        a = -1; // 1111
        b =  8; // 1000
        c =  8; // 00001000
        c = c + a; // 00010111
        b = b + a; // 0111
    end
endmodule
```
Sizing and Signing
The concatenation operator (`{}`) concatenates sized expressions to form a new vector quantity.

```
module concat;
    reg [7:0] a, b, c, d, y;
    initial begin
        a = 8'b00000011;
        b = 8'b00000100;
        c = 8'b00011000;
        d = 8'b11100000;
        y = {a[1:0], b[2], c[4:3], d[7:5]};
        $displayb(y); // 11111111
    end
endmodule
```
Concatenation Operator

You can concatenate an unlimited number of operands. Place the comma-separated list of sized operands inside curly braces. Failure to size an operand will result in an error message.

This is wrong, not all operands are sized:
\[ n[7:0] = \{ 3'b011, \ b0 \}; \]

This is the corrected version:
\[ n[7:0] = \{ 3'b011, 5'b0 \}; \]
Replication Operator

The replication operator (\{n\}) replicates an expression a fixed number of times to form a new vector quantity.

```
module replicate;
    reg [3:0] a;
    reg [7:0] y;
    initial begin
        a = 4'b1001;
        y = {{4{a[3]}},a};
        $displayb(y); // 11111001
    end
endmodule
```

The number of replications must be a constant expression.
Replication Operator

Replication is simply the concatenation of a sized expression a fixed number of times.

These are some examples of common failures to size all of the operands:

\[
\begin{align*}
    a[7:0] & = \{4\{b10\}\}; \\
    b[7:0] & = \{2\{5\}\};
\end{align*}
\]
Negation Operators

The logical negation operator (\(!\)) produces a 0, 1, or X scalar value.

The bitwise negation operator (\(\sim\)) inverts each individual bit of the operand.

```verilog
module negation;
    initial begin
        $displayb ( !4'b0100 ); // 0
        $displayb ( !4'b0000 ); // 1
        $displayb ( !4'b00z0 ); // x
        $displayb ( !4'b000x ); // x
        $displayb ( ~4'b01zx ); // 10xx
    end
endmodule
```
Negation Operators

The logical negation operator reduces an operand to its logical inverse. For example, if an operand contains all zeroes, it is false (logic 0), so its inverse is true (logic 1).
Unary Reduction Operators

The unary reduction operators\((&, |, ^, ^\sim)\) produce a 0, 1, or X scalar value. They operate across all bits of a vector.

```verilog
module reduction;
    initial begin
        $displayb ( &4'b1110 ); // 0
        $displayb ( &4'b1111 ); // 1
        $displayb ( &4'b111z ); // x
        $displayb ( &4'b111x ); // x
        $displayb ( ^4'b1110 ); // 0
        $displayb ( ^4'b1111 ); // 1
        $displayb ( ^4'b111z ); // x
        $displayb ( ^4'b111x ); // x
        $displayb ( ^~4'b1111 ); // 1
        $displayb ( ^~4'b1110 ); // 0
        $displayb ( ^~4'b111z ); // x
        $displayb ( ^~4'b111x ); // x
    end
endmodule
```
Unary Reduction Operators

Unary reduction operators operate on all bits of a single operand to produce a single-bit result.

As Verilog is a free-form language, you must group (with parentheses) a reduction AND or OR operator that immediately follows a bitwise AND or OR operator (respectively).

<table>
<thead>
<tr>
<th>Ambiguous Syntax</th>
<th>Unambiguous Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a &amp; &amp; b )</td>
<td>( a &amp; (&amp;b) )</td>
</tr>
<tr>
<td>( a \mid \mid b )</td>
<td>( a \mid (\mid b) )</td>
</tr>
</tbody>
</table>
The arithmetic operators (*, /, %, +, -) produce numerical or unknown results:

- The (+, -) operators take precedence when used as unary operators
- Integer division discards any remainder
- A modulus operation retains the sign of the first operand
- An unknown operand produces an unknown result
- Assignment of a signed value to an unsigned register is 2’s-complement

module arithmetic;
  initial begin
    $display ( -3 * 5 ); // -15
    $display ( -3 / 2 ); // -1
    $display ( -3 % 2 ); // -1
    $display ( -3 + 2 ); // -1
    $display ( 2 - 3 ); // -1
    $displayh (32'hfffffffffd / 2); // 7fffffffe
    $displayb (2 * 1'bx); // xx...
  end
endmodule
Arithmetic Operators

The **integer** register data type is signed.

The **reg** and **time** register data types are unsigned.

Operations on signed and unsigned data types produce different results. For example, division of a negative integer by 2 retains the sign bit, while division of the same bit pattern in a 32-bit **reg** does not.
Shift Operators

The shift operators (\texttt{<<, >>}) shift the left operand left or right the number of times given by the right operand.

The simulator treats the right operand as unsigned (-2 is \texttt{+4294967293}).

If the right operand is unknown, the simulator sets the result unknown.

```verilog
module shift;
    initial begin
        $displayb ( 8'b00011000 << 2    ); // 01100000
        $displayb ( 8'b00011000 >> 2    ); // 00000110
        $displayb ( 8'b00011000 >> -2   ); // 00000000
        $displayb ( 8'b00011000 >> 1'bx ); // xxxxxxxx
    end
endmodule
```
Shift Operators

The left shift operator (<<) shifts the left operand left by the number of positions specified by the right operand.

The right shift operator (>>) shifts the left operand right by the number of positions specified by the right operand.

The simulator truncates the RHS high-order bits or zero-extends the RHS low-order bits as needed to fit the width of the LHS.

The IEEE Std. 1364-1995 states: "The right operand is always treated as an unsigned number". Simulators should, for example, treat -1 as if it were +4294967294.
Relational Operators

The relational operators (<, <=, >=, >) produce 0, 1, or X scalar values.

The relational operators all have the same precedence.

An unknown operand may produce an unknown result.

    module relational;
    initial begin
        $displayb ( 4'b1010 < 4'b0110 ); // 0
        $displayb ( 4'b0010 <= 4'b0010 ); // 1
        $displayb ( 4'b1010 < 4'b0x10 ); // x
        $displayb ( 4'b0010 <= 4'b0x10 ); // x
        $displayb ( 4'b1010 >= 4'b1x10 ); // x
        $displayb ( 4'b1x10 > 4'b1x10 ); // x
        $displayb ( 4'b1z10 > 4'b1z10 ); // x
    end
endmodule
Relational Operators

The IEEE Std. 1364-1995 states: "If ... the relationship is ambiguous, then the result shall be a 1-bit unknown value (X)". Some simulators set the result to unknown if either operand contains Z, or X, even though the relationship is not ambiguous.
Equality Operators

The logical equality operator (==) does not perform a definitive match for Z or X. The case equality operator (===) does perform a definitive match for Z or X.

All equality operators have the same precedence.

```
a = 2'b1x; b = 2'b1x;
if (a == b)
    $display("a equal to b");
else
    $display("a not equal to b");

a not equal to b
```

```
a = 2'b1x; b = 2'b1x;
if (a === b)
    $display("a identical to b");
else
    $display("a not identical to b");

a identical to b
```
Equality Operators

The difference between the logical and case equality operators is the handling of the Z and X values.

The logical equality operator cannot definitively match a Z or X value in either operand.

(2'b0x == 2'b1x) is false (0); the operands are not equal
(2'b1x == 2'b1x) is unknown (X); the operands could be equal

The case equality (identity) operator can definitively match a Z or X value in both operands.

(2'b0x === 2'b1x) is false (0), the operands are not identical
(2'b1x === 2'b1x) is true (1), the operands are identical
Logical Equality Operator

The logical equality (==) and inequality (!=) operators produce 0, 1, or X scalar values.

```verilog
module equality;
    initial begin
        $displayb ( 4'b0011 == 4'b1010 ); // 0
        $displayb ( 4'b0011 != 4'b1x1x0 ); // 1
        $displayb ( 4'b1010 == 4'b1x1x0 ); // x
        $displayb ( 4'b1x1x0 == 4'b1x1x0 ); // x
        $displayb ( 4'b1z1x0 == 4'b1z1x0 ); // x
    end
endmodule
```
Equality Operators (continued)

Logical Equality Operator

An expression with the logical equality operator evaluates to:

- True (1) if the LHS and RHS have equal known values
  — The value is known only if each bit is either 0 or 1
- False (0) if the LHS and RHS are not equal
- Unknown (X) if the LHS and RHS could be equal
  — The value is unknown if either the LHS bit or RHS bit is unknown (X) or high-impedance (Z)

The logical inequality operator (!=) is the inverse of the logical equality (==) operator.
Case Equality (Identity) Operator

The case equality (===) and inequality (!==) operators produce only 0 and 1 scalar values.

```verilog
module identity();
    initial begin
        $displayb ( 4'b01zx === 4'b01zx ); // 1
        $displayb ( 4'b01zx !== 4'b01zx ); // 0
        $displayb ( 4'b01zx === 4'b00zx ); // 0
        $displayb ( 4'b01zx !== 4'b11zx ); // 1
    end
endmodule
```
Equality Operators (continued)

Case Equality (Identity) Operator

The case equality operators function similarly to the case statement item matching operation. They perform a definitive match of bit positions that are Z or X. The case equality operators are also called the identity operators because they test to see if the operands are identical.

An expression with the case equality operator evaluates to:

- True (1) if every bit of the LHS is identical to its corresponding RHS bit
- False (0) if any bit of the LHS is not identical to its corresponding RHS bit

The case inequality operator (!==) is the inverse of the case equality (==) operator.
The bit-wise operators (&, |, ^, ^~) operate on each individual bit of a vector. Unknown bits of an operand do not necessarily produce unknown results.

```
module bit_wise;
    initial begin
        $displayb ( 4'b01zx & 4'b0000 ); // 0000
        $displayb ( 4'b01zx & 4'b1100 ); // 0100
        $displayb ( 4'b01zx & 4'b1111 ); // 01xx
        $displayb ( 4'b01zx | 4'b1111 ); // 1111
        $displayb ( 4'b01zx | 4'b0011 ); // 0111
        $displayb ( 4'b01zx | 4'b0000 ); // 01xx
        $displayb ( 4'b01zx ^ 4'b1111 ); // 10xx
        $displayb ( 4'b01zx ^~ 4'b0000 ); // 10xx
    end
endmodule
```
Bit-Wise Operators

Bit-wise binary operators perform bit-wise operations on two operands. The simulator first zero-extends the shorter operand to the width of the wider operand. Then the simulator performs the operation between the corresponding bits of each operand.
Logical Operators

The logical operators (&&, ||) produce a 0, 1, or X scalar value:

- An operand is logically false if all of its bits are 0
- An operand is logically true if any of its bits are 1

```verilog
module logical;
    initial begin
        $displayb ( 2'b00 && 2'b10 ); // 0
        $displayb ( 2'b01 && 2'b10 ); // 1
        $displayb ( 2'b0z && 2'b10 ); // x
        $displayb ( 2'b0x && 2'b10 ); // x
        $displayb ( 2'b1x && 2'b1z ); // 1
        $displayb ( 2'b00 || 2'b00 ); // 0
        $displayb ( 2'b01 || 2'b00 ); // 1
        $displayb ( 2'b0z || 2'b00 ); // x
        $displayb ( 2'b0x || 2'b00 ); // x
        $displayb ( 2'b0x || 2'b0z ); // x
    end
endmodule
```
Logical Operators

Logical binary operators operate on logic values. If an operand contains all zeroes, it is false (logic 0). If it contains any ones, it is true (logic 1). If it contains unknowns and no ones, its logical value is unknown.
Conditional Operator

The conditional operator (?:) selects from two operands, based on a third.

```
conditional_expression ? true_expression : false_expression
```

```verilog
module driver(O,I,E);
output O; input I,E;
    assign O = E ? I : 'bz;
endmodule

module mux41(O,S,A,B,C,D);
output O;
input A,B,C,D; input [1:0] S;
    assign O = (S == 2'h0) ? A :
      (S == 2'h1) ? B :
      (S == 2'h2) ? C : D;
endmodule
```
Conditional Operator

The conditional operator is also called the *ternary* operator. That is because it takes three operands. Each conditional operator must have all three operands.

You can think of a conditional operator as somewhat similar to the **if-else** statement:

```
if (conditional_expression) LHS = true_expression; else LHS = false_expression;
```

An unknown conditional expression can still produce a known value if all possible selections are identical.
# Operator Precedence

This table shows the Verilog operators, from higher to lower precedence. Note that AND operators always have higher precedence than OR operators of the same type. This section gave an explanation and example of each operator.

<table>
<thead>
<tr>
<th>Type of Operators</th>
<th>Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenate &amp; replicate</td>
<td>{ }</td>
</tr>
<tr>
<td></td>
<td>{{ }}</td>
</tr>
<tr>
<td>Unary</td>
<td>! ~ &amp; ^ ^~</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>* / %</td>
</tr>
<tr>
<td></td>
<td>+ -</td>
</tr>
<tr>
<td>Logical shift</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>Relational</td>
<td>&lt; &lt;= &gt; &gt;=</td>
</tr>
<tr>
<td>Equality</td>
<td>== != === !==</td>
</tr>
<tr>
<td>Binary bit-wise</td>
<td>&amp; ^ ^~</td>
</tr>
<tr>
<td>Binary logical</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>Conditional</td>
<td>? :</td>
</tr>
</tbody>
</table>
Operator Precedence

The behavior of the exclusive OR (^) and exclusive NOR (^~) operators is such that their precedence with respect to adjacent AND (&) and OR (|) operators is irrelevant.
Summary

In this section you learned about:

■ Sizing and signing of expressions
■ The Verilog operators and their precedence
■ How to use the Verilog operators
Summary
Review

1. True or false: You can replicate an expression a variable number of times.
2. Which operators produce signed results?
3. The conditional operator takes how many operands?
4. Which operators always produce fully known results?
5. What is the order of precedence (highest to lowest) for the ~, &, | and ?: operators?
Review

1. True or false: You can replicate an expression a variable number of times.
   You replicate an expression a fixed number of times. The compiler must know the size of the result.

2. Which operators produce signed results?
   Arithmetic operators take signed operands and produce signed results. All other operators treat the operands as bit vectors and ignore the operand sign.

3. The conditional operator takes how many operands?
   The conditional operator is a ternary operator. It takes three operands. It selects from two operands based upon the logic value of a third operand.

4. Which operators always produce fully known results?
   Only the case equality (identity) operator always produces a known result.

5. What is the order of precedence (highest to lowest) for the ~, &, | and ?:: operators?
   The concatenation and replication operators have highest precedence, followed by the negation operators. The AND operators have higher precedence than OR operators. The conditional operator has lowest precedence of all operators.
Chapter 11: Behavioral Modeling

Objectives

In this section you will learn about behavioral modeling:

- Procedural blocks
- Procedural assignments
- Procedural and intra-assignment delays
- Branching and looping statements
- Continuous assignments
Behavioral Modeling

This section discusses the Verilog behavioral modeling constructs. Most of this section deals with procedural blocks: concurrent and sequential statements, blocking and nonblocking procedural assignments, procedural and intra-assignment timing controls, and conditional and looping statements. The section ends with a discussion of continuous assignments outside of procedural blocks.
Behavioral Modeling

With behavioral modeling you describe the system at a high level of abstraction. You describe system behavior with a set of concurrently active procedural blocks.

At every active edge of clk
If rstn is not true {
    set q to the value of data  
    set qn to inverse of data  
}

At every assertive edge of rstn {
    set q to 0  
    set qn to 1  
}
Behavioral Modeling

With behavioral modeling you describe the system at a high level of abstraction. At this level of abstraction, implementation is not as important as the overall functionality of the system.

You describe system behavior with a set of concurrently active procedural blocks that together describe the operation of the system.

The Verilog language offers several constructs for modeling system behavior. Among these are the case, if, for, forever, wait, and while statements.
The basis for behavioral modeling is the procedural block:

- Procedural blocks are of two types:
  - *initial* blocks
  - *always* blocks

- Procedural blocks have the following components:
  - Timing controls
  - High-level constructs (conditional statements, loops)
  - Procedural assignment statements
Procedural Blocks

Procedural blocks are of two types:

- The simulator executes an initial block once
- The simulator executes an always block continually, in a loop

Procedural blocks have the following components:

- Timing controls control the execution of statements in the block
- Behavioral constructs describe the behavior of simulation objects
- Procedural assignment statements describe the flow of data
Sequential and Concurrent Blocks

Block statements group two or more statements together:

- The keywords **begin** and **end** group sequential statements
- The keywords **fork** and **join** group concurrent statements
Procedural Blocks (continued)

Sequential and Concurrent Blocks

Use the block statement keywords to group two or more statements together so that they syntactically act as one statement.

Enclose sequential block statements between the keywords **begin** and **end**. The simulator executes these statements in sequence.

Enclose concurrent block statements between the keywords **fork** and **join**. The simulator executes these statements in parallel. You cannot know the actual order in which the simulator executes concurrent statements scheduled for the same simulation time.

Use the **initial** and **fork-join** block statements only in purely behavioral code or in a testbench. Use the **always** and **begin-end** block statements to describe the operation of concurrent hardware for a logic synthesis tool.
Sequential and Concurrent Blocks

This is the difference between a sequential and a concurrent block:

- The simulator executes statements in a sequential block in sequence
  - It finishes the current statement, then begins the next
  - You always know the order in which it actually executes the statements
  - The simulator exits the block after finishing the last statement

- The simulator executes statements in a concurrent block in parallel
  - It starts executing all statements simultaneously
  - You cannot know the order in which it actually executes statements scheduled for the same simulation time
  - The simulator exits the block after finishing the latest statement

These two blocks perform the same operations.

```
begin    fork
    #5 a = 1;  #5 a = 1;
    #5 a = 2;  #15 a = 3;
    #5 a = 3;  #10 a = 2;
end      join
```
The two blocks function identically. The assignments in the **fork-join** block are deliberately out of order to illustrate that order does not matter. The **fork-join** construct provides you with the flexibility to simplify your behavioral code or testbench, but is not synthesizable.
Procedural Assignments

Procedural assignments are assignments inside procedural blocks.

Make procedural assignments only to register data types.

This example contains procedural assignments.

```vhdl
module adder (cout, sum, a, b, cin);
  output cout, sum;
  input a, b, cin;
  reg cout, sum;

  always @(a or b or cin)
  begin
    sum  = a ^ b ^ cin;
    cout = a & b | a & !b & cin | !a & b & cin;
  end

endmodule
```
Procedural Assignments

The right-hand side of a procedural assignment can be any valid expression.

The left-hand side of a procedural assignment must be a register data type. The simulator will report an error if you attempt to make a procedural assignment to a net.
Blocking and Nonblocking Assignments

The simulator completes a blocking assignment (=) in one pass:

- It evaluates the RHS expression and stores the value in the LHS register
- This makes modeling concurrency somewhat difficult

// THIS DOES NOT WORK!
always @(posedge clk)
dout = dout^reg1;

// TRY THIS INSTEAD!
reg temp;
always @(posedge clk)
begin
    temp = dout;
dout = dout^reg1;
    reg1 = temp^din;
end
Procedural Assignments (continued)

Blocking and Nonblocking Assignments

The simulator completes a blocking assignment in one pass. It evaluates the RHS expression and stores the value in the LHS register. This makes modeling concurrency somewhat difficult, as you must ensure that upon each clock edge, the simulator uses the current value of each register before updating it. You can do this by carefully ordering statement execution and using temporary variables where necessary. To order statement execution, you need to make all assignments to a network of registers in a single procedural block.
The simulator completes a nonblocking assignment ($<=$) in two passes:

- It evaluates the RHS expression and *schedules* the LHS assignment
- It updates the LHS only after evaluating all the RHS expressions in the design
- This greatly simplifies modeling concurrency

```verilog
// THIS WORKS FINE!

always @(posedge clk)
  dout <= dout^reg1;

always @(posedge clk)
  reg1 <= dout^din;
```
Blocking and Nonblocking Assignments (continued)

The simulator completes a nonblocking assignment in two passes. It evaluates the RHS expression and schedules the LHS assignment. It updates the LHS only after evaluating all the RHS expressions in the design. This greatly simplifies modeling concurrency, as the simulator will use the current value of each register before updating it.
Procedural Timing Control

You can place timing controls inside procedural blocks:

- The simple "pound" (#) delay
- The edge-sensitive (@) control
- The level-sensitive (wait) control
Procedural Timing Control

The simple "pound" (#) delay suspends subsequent statement execution for the specified time.

The edge-sensitive (@) control suspends subsequent statement execution until any of the specified list of events occurs. You can list virtual events and signal transitions. You can further qualify the signal transitions with the posedge or negedge keywords.

The level-sensitive (wait) control suspends subsequent statement execution unless or until the expression is true. If the expression is already true when the simulator encounters this control, it immediately proceeds to execute the next sequential statement.
Simple Delay

The simple "pound" (#) delay suspends subsequent statement execution for the specified time.

Use simple delays to delay stimulus in a testbench, or to approximate propagation delay in behavioral and RTL models.

This example models 2 unit delay from a to y and 3 unit delay from b to y.

```verilog
module mux21 (y, s, b, a);
output y; reg y;
input s, b, a;
    always @(s or b or a)
      if (s)
        #3 y = b;
      else
        #2 y = a;
endmodule
```
Procedural Timing Control (continued)

Simple Delay

Parameterization of behavioral delays is common. This defines the delay in a single easily-modified place at the head of the file.

```verilog
module mux21 (y, s, b, a);
output y; reg y;
input s, b, a;
parameter tPay=2, tPby=3;
    always @(s or b or a)
        if (s)
            #tPby y = b;
        else
            #tPay y = a;
endmodule
```
The edge-sensitive (@) control suspends subsequent statement execution until any of the specified list of events occurs. You can list named events and signal transitions. You can further qualify the signal transitions with the `posedge` or `negedge` keywords.

Use the edge-sensitive control for combinational and sequential logic in behavioral and RTL models.

This example has a full sensitivity list for the combinational logic and a clock-qualified sensitivity list for the sequential logic.

```verbatim
module counter (cnt, data, ld, en, clk, rst);
output [7:0] cnt; reg [7:0] cnt;
input [7:0] data; input ld, en, clk, rst;
reg [7:0] ncnt;
always @(data or ld or en or rst)
    if (rst) ncnt = 0;
    else if (ld) ncnt = data;
    else if (en) ncnt = cnt+1;
    else ncnt = cnt;
always @(posedge clk)
    cnt <= ncnt;
endmodule
```
Edge-Sensitive Control

Do not confuse the `or` keyword in the sensitivity list with the bit-wise OR operator (`|`) or the logical OR operator (`||`). The `or` keyword simply separates the listed events, and does not perform any operation on them.
Edge-Sensitive Control

Use the `event` keyword to declare a named event.

Use named events for edge-sensitive timing control in a purely behavioral model or in a testbench. Logic synthesis tools do not accept the `event` keyword.

This example illustrates the use of named events to control block timing. A process not described here triggers the named events with the `->` operator.

```verbatim
module arbiter;
  event reqa, reqb;
  event acka, ackb;
  reg runa, runb;
  initial {runa,runb}=0;
  always @reqa begin
    wait (!runb) runa = 1;
    if (!runb) @acka;
    runa = 0;
  end
  always @reqb begin
    wait (!runa) runb = 1;
    if (!runa) @ackb;
    runb = 0;
  end
endmodule
```
Edge-Sensitive Control (continued)

A named event:

- Is declared with the `event` keyword
- Has no connectivity, duration, or value
- Is triggered with the `->` operator in a procedural block
- When triggered allows processes waiting for it to proceed
- Is not synthesizable
Level-Sensitive Control

The level-sensitive (**wait**) control suspends subsequent statement execution unless, or until, the expression is true. If the expression is already true when the simulator encounters this control, it immediately proceeds to execute the next sequential statement.

Use **wait** for level-sensitive timing control in a purely behavioral model or in a testbench. Logic synthesis tools do not accept the **wait** keyword.

This example waits asynchronously for either **ack** or **rst** to be true (logic 1).

```verilog
module behavioral (req,run,ack,rst);
output req; reg req;
input run,ack,rst;
    always @(posedge run) begin
        req = 1;
        wait (ack || rst)
            req = 0;
    end
endmodule
```
Procedural Timing Control (continued)

Level-Sensitive Control
Missing Timing Controls

A zero-delay behavioral loop is a common coding error.

module comparator(out, in1, in2);
output [1:0] out;
input [7:0] in1, in2;
reg [1:0] out;

always
  if (in1 == in2)
    out = 2'b00;
  else if (in1 > in2)
    out = 2'b10;
  else
    out = 2'b11;

initial #1 $finish; // Never executes
endmodule

How would you fix this zero-delay loop?
Procedural Timing Control (continued)

Missing Timing Controls

Simulation time advances when all the events in the event queue finish. In a zero-delay loop, the simulator continuously adds events to the same queue. As the queue never empties, the simulator appears to hang.

In this example, simulation events occur, but simulation time does not advance. This can happen when there is an **always** block or **forever** loop with no timing controls.

If your simulation appears to hang for a long time and you do not know why, try entering `Control-c` to see if you can interrupt the simulation and get an interactive prompt. If you do get a prompt, try single stepping the simulation to see if it is continuously executing the same set of statements over and over without advancing simulation time. If so, you’ve found a zero-delay behavioral loop. This course discusses zero-delay loops further in a section about design debugging.
Timing Control Examples

This unsynthesizable model uses the `wait` construct.

```verilog
module example (run, set, clk);
output run; reg run;
input set, clk;
    always begin
        wait (set)
            @(posedge clk)
                #3 run = 1;
            @(posedge clk)
                #3 run = 0;
        wait (!set);
    end
endmodule
```
Procedural Timing Control (continued)

Timing Control Examples

The example uses the `wait` construct. Synthesis tools do not accept the `wait` keyword and ignore all `#` delays and `specify` blocks.

The model is subject to ambiguity if the `set` signal changes at the same simulation time as the active edge of the `clk` signal.
Timing Control Examples

This synthesizable model uses synchronous procedural blocks.

```verilog
// Detect positive edge of set
always @(posedge set or posedge det)
  if (det) det <= 0; else det <= !run;

// Save new set until next active clock
always @(posedge clk or posedge det)
  if (det) new <= 1; else new <= 0;

// Pulse run for one clock
always @(posedge clk)
  run <= #3 new && !run;
```
Timing Control Examples (continued)
Intra-Assignment Timing Control

Procedural timing controls delay subsequent statement execution.

\[
\begin{align*}
delay\_control & \enspace statement\_or\_null \\
eVENT\_control & \enspace statement\_or\_null \\
wait & \enspace ( \enspace expression \enspace ) \enspace statement\_or\_null \\
@\(\text{posedge} \enspace clk\) & \enspace q = d;
\end{align*}
\]

Intra-assignment timing controls delay LHS assignment but not RHS evaluation.

\[
\begin{align*}
lvalue & = \text{delay\_control} \enspace expression \\
lvalue & = \text{event\_control} \enspace expression \\
q & = @\(\text{posedge} \enspace clk\) \enspace d;
\end{align*}
\]

This implies temporary storage of the RHS value, similarly to:

\[
\begin{align*}
\begin{align*}
\text{begin} \\
\quad \text{temp} & = d; \\
\quad @\(\text{posedge} \enspace clk\) \enspace q = \text{temp};
\end{align*}
\end{align*}
\]
\text{end}
Intra-Assignment Timing Control

The simulator immediately evaluates the RHS expression. If the assignment is blocking, the simulator schedules a return to complete the statement execution. If the assignment is nonblocking, the simulator schedules the assignment, but immediately proceeds to execute the subsequent sequential statement.
Intra-Assignment Timing Control

The simulator completes a blocking assignment (=) in one pass:

- It evaluates the RHS expression and stores the value in the LHS register.
- This makes modeling concurrency somewhat difficult.

```verbatim
// THIS DOES NOT WORK!   // TRY THIS INSTEAD!
always @(posedge clk)
dout = dout^reg1;
always @(posedge clk)
reg1 = dout^din;
```

```verbatim
always @(posedge clk)
dout = #1 dout^reg1;
always @(posedge clk)
reg1 = #1 dout^din;
```
Intra-Assignment Timing Control  (continued)

The simulator completes a blocking assignment in one pass. It evaluates the RHS expression and stores the value in the LHS register. This makes modeling concurrency somewhat difficult, as you must ensure that upon each clock edge, the simulator uses the current value of each register before updating it. You can do this with an intra-assignment timing control, which delays the LHS update, but not the RHS evaluation.
Conditional Statements

This subsection discusses conditional statements:

- The `if` and `if-else` statements
- The `case` statement
Conditional Statements
if and if-else Statements

The if and if-else statements may be the most common branching statements.

    if ( expression ) statement_or_null [ else statement_or_null ]

You can nest if-else statements in two ways:

Example 1

    if (sel==3)
        y = d;
    else
        if (sel==2)
            y = c;
        else
            if (sel==1)
                y = b;
            else
                if (sel==0)
                    y = a;

Example 2

    if (sel[1])
        if (sel[0])
            y = d;
        else
            y = c;
    else
        if (sel[0])
            y = b;
        else
            y = a;
Conditional Statements (continued)

if and if-else Statements

In nested if sequences, every else is associated with the closest previous if at the same nesting level. Not every if will necessarily be associated with an else, though:

```plaintext
if (cond1) begin
  if (cond2)
    do_c1_c2;
  end
else
  do_not_c1;
```
The **case** statement is like an **if-else** statement that uses identity operators.

```
case ( expression ) case_item ... endcase
```

The **case** statement may sometimes be more readable.

```
if-else statement case statement
if (sel===3)
  y = d;
else
  if (sel===2)
    y = c;
  else
    if (sel===1)
      y = b;
    else
      y = a;
endcase
```

**Note:** The Verilog **case** statement automatically breaks after the first match.
Conditional Statements (continued)

case Statements

The **case** statement is a special multiway conditional statement that tests whether the expression matches one of a number of other expressions and branches accordingly:

- The **case** statement does a bit-by-bit comparison for an exact match (including x and z)
- The simulator executes the optional **default** item only if none of the other items match the case expression

**Important**

It is a good programming practice to always use the **default** statement to detect unexpected values.

The **casex** and **casez** variations allow designation of don’t-care bit positions:

- You can designate don’t-care bit positions in either the case expression, or the case match item, or both
- In **casex** statements, the simulator does not compare a bit position having a z or x value
- In **casez** statements, the simulator does not compare a bit position having a z value
- For the comparison, the simulator considers the ? and z characters interchangeable
Looping Statements

This subsection discusses:

- The **for** loop
- The **forever** loop
- The **repeat** loop
- The **while** loop
Looping Statements
The for Loop

This is the for loop formal syntax:

```
for ( assignment ; expression ; assignment ) statement
```

Here are some examples of for loops:

```
// X detection
for (index = 0; index < size; index = index + 1)
  if (val[index] === 1'bX)
    $display ("found an X");

// Memory initialization
for (i = size; i != 0; i = i - 1)
  memory[i-1] = 0;

// Factorial sequence
factorial = 1;
for (j = num; j != 0; j = j - 1)
  factorial = factorial * j;
```
Looping Statements (continued)

The for Loop

Syntax:

\[
\text{for ( assignment ; expression ; assignment ) statement}
\]

1. The simulator performs the initial \textit{assignment}
2. The simulator evaluates the \textit{expression} and exits the loop if it is not true
3. The simulator executes the \textit{statement}
4. The simulator executes the second \textit{assignment} and loops back to (2)
The forever Loop

This is the **forever** loop formal syntax:

```
  forever statement
```

Here is an example of a **forever** loop:

```
// Clock
reg clk;
initial begin
  clk = 0;
  forever begin
    #10 clk = 1;
    #10 clk = 0;
  end
end
```
The forever Loop

A **forever** loop executes a statement (or block of statements) until the simulation ends.

A **forever** loop should be the last item in a sequential **begin-end** block, as the simulator cannot execute any statements following it.

Synthesis tools do not accept **forever** loops. Use a **forever** loop only in a purely behavioral design or in a testbench.
The repeat Loop

This is the repeat loop formal syntax:

    repeat ( expression ) statement

Here is an example of a repeat loop:

// Parameterizable shift and add multiplier
module multiplier (result, op_a, op_b);
parameter size = 8;
output [2*size:1] result;
input [ size:1] op_a, op_b;
reg [2*size:1] result;
    reg [2*size:1] shift_opa;
    reg [ size:1] shift_opb;
always @(op_a or op_b) begin
    result = 0;
    shift_opa = op_a; // Zero extend left
    shift_opb = op_b;
    repeat (size) begin
        if (shift_opb[1]) result = result + shift_opa;
        shift_opa = shift_opa << 1; // Shift left
        shift_opb = shift_opb >> 1; // Shift right
    end
end
endmodule
Looping Statements (continued)

The repeat Loop

A repeat loop executes a block of statements a fixed number of times. The example sizes the variable shift_opa large enough to hold values that are shifted left.
The while Loop

This is the **while** loop formal syntax:

```plaintext
while ( expression ) statement
```

Here is an example of a **while** loop:

```plaintext
// Count the number of ones
task count_ones;
input [31:0] invar;
output [31:0] number;
begint
  number = 0;
  while (invar) begin
    if (invar[0])
      number = number + 1;
    invar = invar >> 1;
  end
end
endtask
```
Looping Statements (continued)

The while Loop

A while loop executes a statement (or block of statements) as long as its expression is known and nonzero.

If the expression is initially false, the statements are not executed.
Continuous Assignments

You can model combinational logic with continuous assignments:

assign [drive_strength] [delay] list_of_assignments

- You can use continuous assignments instead of primitives
- Continuous assignments are the lowest level behavioral modeling construct
- Use continuous assignments *outside* of a procedural block
- Continuous assignments cannot contain timing controls
- You can make an explicit or implicit continuous assignment

```vhdl
wire ab;
assign ab = a & b; // explicit
wire in_ = ~in; // implicit
```
Continuous Assignments

The continuous assignment is the lowest form of behavioral construct. The continuous assignment creates a continuous relationship between a RHS expression and a LHS net. You can use a continuous assignment to replace a network of primitives, and as with primitives, you can assign drive strength and delay to the continuous assignment. As you make continuous assignments outside of any procedural blocks, they cannot have timing controls.
Continuous Assignments

This example makes continuous assignments in the net declaration and with the `assign` keyword:

```vhdl
module assigns (o1, o2, eq, AND, OR, even, odd, one, SUM, COUT, a, b, in, A, B, CIN);
output [7:0] o1, o2;
output [31:0] SUM;
output eq, AND, OR, even, odd, one, COUT;
input a, b, CIN;
input [7:0] in;
input [31:0] A, B;
wire [7:0] #3 o2; // no assignment yet, but a delay
tri AND = a&b, OR = a|b; // two assignments
wire #5 eq = (a == b); // implicit, with delay
wire (weak1, strong0) [7:0] #(3,5,2) o1 = in; // strength and delays
assign o2[7:4] = in[3:0], o2[3:0] = in[7:4]; // part-select
tri #5 even = ^in, odd = ~^in; // delay, two assignments
wire one = 1'b1; // constant assignment
assign {COUT, SUM} = A + B + CIN ; // assignment to a concatenation
endmodule
```

**With what delay does the simulator transition vector net o1 from 'h55 to 'hAA?**
Continuous Assignments (continued)

The simulator uses the final value (not the transition) of the least significant bit of a vector net, to determine the delay with which to transition the net. If the final value is 0, 1, or Z, it uses the falling, rising, and turnoff delays, respectively. If the final value is unknown (X), the simulator uses the least delay.

The example above illustrates the flexibility and simplicity of continuous assignments. You can make continuous assignments:

- To any net type
- With the `assign` keyword or when you declare the net
- With a propagation delay
- With charge and drive strengths
- From any expression, including constant and conditional
- From the return values of user-defined functions
- To bit- or part-selects of vectored nets
- To several nets at once in a concatenation
Continuous Assignments

These examples make continuous assignments to conditional expressions:

```verilog
module cond_assigns1 (mux1, sel, a, b, c, d);
    output mux1; input [1:0] sel; input a, b, c, d;
    assign mux1 =
        (sel[1]==0) ? (sel[0]==0) ? a : b
        : (sel[0]==0) ? c : d;
endmodule

module cond_assigns2 (mux2, sel, a, b, c, d);
    output mux2; input [1:0] sel; input a, b, c, d;
    tri1 mux2;
    assign mux2 = sel == 0 ? a : 'bz;
    assign mux2 = sel == 1 ? b : 'bz;
    assign mux2 = sel == 2 ? c : 'bz;
endmodule

What is the value of mux2 when sel==3 ?
```
Continuous Assignments (continued)

This example illustrates:

- A continuous assignment to MUX1 with nested conditional expressions
- Multiple continuous assignments to MUX2

In this example, no more than one continuous assignment will drive MUX2 at any time. As MUX2 is a tri1 net, when no continuous assignment drives it, it will pull resistively to the logic 1 value.
Summary

In this section you learned about behavioral modeling:

- Procedural blocks
- Procedural assignments
- Procedural and intra-assignment delays
- Branching and looping statements
- Continuous assignments
Summary
Review

1. What is the difference between sequential and concurrent statement blocks?

2. Which procedural timing control does not necessarily suspend statement execution?

3. True or false: You can place an **always** loop inside a task definition.

4. What type of logic can you model with continuous assignments?
Review

1. What is the difference between sequential and concurrent statement blocks?

   The simulator executes statements in a sequential block in sequence. It finishes the current statement, then begins the next, and exits the block after finishing the last statement. You always know the order in which it actually executes the statements.

   The simulator executes statements in a concurrent block in parallel. It starts executing all statements simultaneously, and exits the block after finishing the latest statement. You cannot know the order in which it actually executes statements scheduled for the same simulation time.

2. Which procedural timing control does not necessarily suspend statement execution?

   The level-sensitive (wait) control suspends subsequent statement execution unless, or until, the expression is true. If the expression is already true when the simulator encounters this control, it immediately proceeds to execute the next sequential statement.

3. True or false: You can place an always loop inside a task definition.

   The always keyword creates a continuous process loop. You cannot use this keyword in any behavioral statement. Use the forever keyword to create a behavioral statement loop.

4. What type of logic can you model with continuous assignments?

   You can model only combinational logic with continuous assignments. You can specify charge or drive strengths, and delays.
About Lab 6

The objective of this lab is to use continuous assignments to model a multiplexer that is scalable through parameter redefinition.

In this lab, you will:

- Model a multiplexor design with a continuous assignment
- Use parameters and parameter redefinition to scale your design
About Lab 6
Chapter 12: Debugging with the Command-Line Interface

Objective

In this section, you will learn about debugging with the Verilog-XL and NC-Verilog command-line interfaces:

- Entering the interactive simulation mode
- Examining and traversing the design
- Reading and writing simulation objects
- Setting breakpoints and running the simulation
- Tracing events and monitoring signals
- Capturing waveform data
- Using the command history list
- Executing commands from a script
- Saving, resetting, and restarting the simulation
Debugging with the Command-Line Interface

This Cadence Verilog-XL and NC-Verilog simulator-specific section visits each step of the debug process, from entering the interactive mode and setting the debug scope, through traversing and examining the design, writing, reading, and monitoring signals, to saving, resetting, and restarting the simulation, and exiting the simulation. For each activity, this section presents the appropriate interactive command, explains how to use it, and describes the results of using it.

This section separately describes the Verilog-XL and NC-Verilog interactive commands. As some students may want to refer only to the Verilog-XL commands, or only to the NC-Verilog commands, this section duplicates extraneous information applicable to both simulators.
Overview

You can interact with the simulators through a textual command interface:

- Verilog-XL has a command-line interface (CLI)
  - You can enter Verilog procedural statements

- NC-Verilog has the Tool Command Language (Tcl)
  - You can enter standard and extended Tcl commands

- Both simulators also include the SimVision GUI
  - SimVision automatically issues interactive commands in response to your tool button click or menu selection

Note: The remainder of this section clearly separates the Verilog-XL and NC-Verilog commands. Some of the commands are similar, but with subtly different effects.
Overview

A Verilog-XL CLI command is either a Verilog procedural statement or a block of statements. Verilog procedural statements include procedural assignments, loops, conditional statements, and task and function invocations. You can enter more than one Verilog statement on a line, but you must terminate each statement with a semicolon, as in your source code.

Verilog-XL has additional system tasks for source-level debugging that are not part of the IEEE Std. 1364. You can place them in a source file or enter them at the keyboard.

An NC-Verilog Tcl command consists of one or more words — the name of the command followed by arguments to the command. Words are separated by spaces or tabs. You can enter more than one command on the command line, using a semicolon to separate the commands.

You can obtain information about the standard Tcl commands from the internet or in any technical bookstore or library.

NC-Verilog-specific extensions to the standard Tcl command set allow you to debug your design from the `ncsim>` command prompt.

This section briefly describes several NC-Verilog-specific Tcl extensions. The Cadence online document *Cadence NC-Verilog Simulator Help* describes them in detail.

You can enter CLI and Tcl commands either interactively at the command line, or from a script or keyfile.
Entering the Interactive Mode

Verilog-XL

Enter the interactive mode by interrupting the simulation:

- Invoke the simulator with the \(-s\) command-line option
- Enter a Control-c asynchronous interrupt
- Execute the \$stop system task
- Activate a breakpoint

When you interrupt the simulator, it enters interactive mode and prompts you:

\[ C1 > \]

While the simulation is suspended, all signals retain their current state.

You can enter interactive commands at the prompt, and then resume simulation.
Entering the Interactive Mode

Verilog-XL

The simulator allows you to interrupt simulation at discrete points in time to interact with the design under test. You can enter interactive mode in one of four ways:

- If you invoke the simulator with the `-s` command-line option, the simulator compiles and links the design, and enters interactive mode, waiting for you to continue the simulation.

- You can enter `Control-c` asynchronously at the keyboard to interrupt simulation.

- The simulator enters interactive mode upon executing the `$stop` system task.

- The simulator enters interactive mode upon activating a breakpoint.

When the simulator enters interactive mode it prompts you for command input. You can enter any procedural statement, and some additional commands that apply only to the debug environment. You cannot interactively modify the design or testbench hierarchy, change the timescale, or create new procedural blocks.
Entering the Interactive Mode

NC-Verilog

Enter the interactive mode by interrupting the simulation:

- Invoke the simulator with the \(-s\) command-line option
- Enter a Control-c asynchronous interrupt
- Execute the \(\$\text{stop}\) system task
- Activate a breakpoint

When you interrupt the simulator, it enters interactive mode and prompts you:

\texttt{ncsim>}

While the simulation is suspended, all signals retain their current state.

You can enter interactive commands at the prompt, and then resume simulation.
Entering the Interactive Mode (continued)

NC-Verilog

The simulator allows you to interrupt simulation at discrete points in time to interact with the design under test. You can enter interactive mode in one of four ways:

- If you invoke the simulator with the `-s` command-line option, the simulator compiles and elaborates the design, and enters interactive mode, waiting for you to continue the simulation
- You can enter `Control-c` asynchronously at the keyboard to interrupt simulation
- The simulator enters interactive mode upon executing the `$stop` system task
- The simulator enters interactive mode upon activating a breakpoint

When the simulator enters interactive mode it prompts you for command input. You can enter standard Tcl commands, NC-Verilog extensions to the Tcl, or shell commands to be passed to the operating system.
Command Overview

Verilog-XL

You can use these standard Verilog keywords and system tasks in your debug process:

\[
\begin{align*}
$display & $[inc]save * $monitor & $reset * $showscopes * $strobe \\
$finish & $input * $[real,s]time & $restart * $showvars * $write \\
force & $list * & release & $scope * $stop
\end{align*}
\]

Note: The IEEE Std. 1364-1995 describes, but does not require, the keywords and system tasks marked here with an asterisk (*). However, most Verilog simulators will offer these features.
Command Overview

Verilog-XL

You can use these system tasks to perform the following kinds of operations:

- **$display, $write** — Display the formatted value of the system task arguments
- **$finish** — Finish (terminate) the simulation
- **force** — Force a signal to a continuous assignment
- **$input** — Input (read) a command script of Verilog procedural statements
- **$list** — List the source code at the current (or a specific) scope
- **$monitor** — Display the formatted value of the arguments whenever one changes
- **release** — Release a forced signal
- **$reset** — Reset the simulation to the initial time 0 state
- **$restart** — Restart a simulation from a saved (checkpointed) state
- **$save, $incsave** — Save (checkpoint) the simulation data structure, fully or incrementally
- **$scope** — Set the current debug scope
- **$showscopes** — Show the scopes rooted at the current debug scope
- **$showvars** — Show detailed information concerning variables at the current scope
- **$stop** — Stop (suspend) the simulation
- **$strobe** — Display the formatted value of the task arguments at the end of the time instant
- **$time, $realtime, $stime** — Get simulation time as a time, real, or integer data type
## Command Overview

### Verilog-XL

The simulator offers several additional debugging commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Command</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{db_break}{\text{after</td>
<td>before}}\text{time}$</td>
<td>$[\text{db_}]\text{cleartrace}$</td>
</tr>
<tr>
<td>$\text{db_break}{\text{once}}\text{atline}$</td>
<td>$\text{db_help}$</td>
<td>$\text{deposit}$</td>
</tr>
<tr>
<td>$\text{db_break}{\text{once}}\text{onnegedge}$</td>
<td>$[\text{db_}]\text{setfocus}$</td>
<td>$\text{history}$</td>
</tr>
<tr>
<td>$\text{db_break}{\text{once}}\text{onposedge}$</td>
<td>$[\text{db_}]\text{settrace}$</td>
<td>$\text{list_forces}$</td>
</tr>
<tr>
<td>$\text{db_break}{\text{once}}\text{when}$</td>
<td>$\text{db_showbreak}$</td>
<td>$\text{showallinstances}$</td>
</tr>
<tr>
<td>$\text{db_[\text{delete</td>
<td>enable</td>
<td>disable}]_break}$</td>
</tr>
<tr>
<td>$\text{db_[\text{delete</td>
<td>enable</td>
<td>disable}]_focus}$</td>
</tr>
</tbody>
</table>

**Note:** For help on debugging commands, enter `?` or `$\text{db\_help}$` from the interactive mode, or refer to the Cadence online documents *Verilog-XL Reference* and *Verilog-XL User Guide*. 
Command Overview (continued)

Verilog-XL

You can use these system tasks to perform the following kinds of operations:

- `$db_break{before|after}time` — Break before or after simulation time
- `$db_break[once]atline` — Break before line or after primitive
- `$db_break[once]onnegedge, $db_break[once]onposedge` — Break on a signal edge
- `$db_break[once]when` — Break when a variable changes value
- `$db_disablebreak, $db_enablebreak, $db_deletebreak` — Operate on a break
- `$db_disablefocus, $db_enablefocus, $db_deletefocus` — Operate on a focus
- `$db_setfocus` — Specify the scope foci to focus on for `$db_` commands
- `$[db_]settrace, $[db_]cleartrace` — Trace or discontinue tracing behavioral statements and unidirectional primitives
- `$db_showbreak` — Show the list of breaks
- `$db_showfocus` — Show the list of foci
- `$db_step` — Step behavioral statements and unidirectional primitives in the foci
- `$db_steptime` — Advance simulation by the specified number of time units
- `$cputime` — Return simulation CPU time in 10’ths of seconds
- `$deposit` — Deposit a value onto a net
- `$history` — Display the command history list
# Command Overview

## NC-Verilog

You can use these standard Tcl commands in your debug process:

<table>
<thead>
<tr>
<th>alias *</th>
<th>concat</th>
<th>foreach</th>
<th>history *</th>
<th>puts</th>
<th>switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>break</td>
<td>eval</td>
<td>format</td>
<td>if</td>
<td>return</td>
<td>unset</td>
</tr>
<tr>
<td>case</td>
<td>exit *</td>
<td>gets</td>
<td>incr</td>
<td>set</td>
<td>while</td>
</tr>
<tr>
<td>close</td>
<td>for</td>
<td>help *</td>
<td>open</td>
<td>source *</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** The Cadence online document *Cadence NC-Verilog Simulator Help* and the `help` command partially document the standard Tcl commands marked here with an asterisk (*).
Command Overview (continued)

NC-Verilog

alias — Create a command alias
exit — Exit the Tcl
help — Display brief help on Tcl commands
history — Manipulate the command history list
source — Source a Tcl command script
## Command Overview

**NC-Verilog**

The simulator offers several additional Tcl commands; among them are:

<table>
<thead>
<tr>
<th>call</th>
<th>drivers</th>
<th>probe</th>
<th>run</th>
<th>strobe</th>
<th>where</th>
</tr>
</thead>
<tbody>
<tr>
<td>coverage</td>
<td>finish</td>
<td>process</td>
<td>save</td>
<td>task</td>
<td></td>
</tr>
<tr>
<td>database</td>
<td>force</td>
<td>release</td>
<td>scope</td>
<td>time</td>
<td></td>
</tr>
<tr>
<td>deposit</td>
<td>input</td>
<td>reset</td>
<td>status</td>
<td>value</td>
<td></td>
</tr>
<tr>
<td>describe</td>
<td>omi</td>
<td>restart</td>
<td>stop</td>
<td>version</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** For brief help on debugging commands, enter `help` or `help <command>` from the interactive mode. For a detailed description, refer to the Cadence online document *Cadence NC-Verilog Simulator Help.*
Command Overview (continued)

NC-Verilog

You can use these commands to perform the following kinds of operations:

- **call** — Invoke a user-defined system task (PLI or VPI application)
- **coverage** — Report on statement and FSM coverage
- **database** — Manipulate an EVCD, SHM, or VCD waveform database
- **deposit** — Deposit a value onto a simulation object
- **describe** — Describe one or more simulation objects
- **drivers** — Show detailed driver information about one or more simulation objects
- **finish** — Finish the simulation, optionally displaying simulation statistics
- **force** — Force an object to take and maintain a value
- **input** — Input a script of Tcl commands
- **omi** — Communicate with OMI managers
- **probe** — Place a probe on one or more simulation objects
- **process** — Report on scheduled or currently executing processes
- **release** — Release a forced value on one or more simulation objects
- **reset** — Reset the simulation to time 0
- **restart** — Restart simulation from a previously saved snapshot
- **run** — Run, step and trace through a simulation
Setting the Debug Scope

Verilog-XL

Use \texttt{$\texttt{showscopes}[(n)]}$ to show scopes defined at the current level.

Provide an optional nonzero argument to see all downward scopes.

\begin{verbatim}
C1 > $showscopes;
Directory of scopes at current scope level:
    module (register), instance (r1)
    module (clock), instance (c1)
    task (expect)

Current scope is (register_test)
Highest level modules:
    register_test
\end{verbatim}

Use \texttt{$\texttt{scope}$} to set the debug scope.

\begin{verbatim}
C2 > $scope(r1);
\end{verbatim}

Use the colon (:) key to display the current simulation location.

\begin{verbatim}
C3 > :
Line 17, file "libdir/clock.v", module (register_test.c1)
Scope is (register_test.r1)
\end{verbatim}
Setting the Debug Scope

Verilog-XL

When you first enter the interactive mode, the current debug scope is at the top level.

Use the $scope system task to change the debug scope to the specified scope. You can use full or relative hierarchical scope names.

Use the $showscopes system task to see all scopes at the current debug scope. Provide an optional nonzero argument to see all scopes in the hierarchy rooted at the current debug scope.

You can directly reference objects defined in the current module without providing their full hierarchical names.
Setting the Debug Scope

NC-Verilog

Use `scope -show` to show scopes defined at the current level.

```
ncsim> scope -show
Directory of scopes at current scope level:
    module (register), instance (r1)
    module (clock), instance (c1)
    task (expect)

Current scope is (register_test)
Highest level modules:
    register_test
```

Use `scope -set` to set the debug scope.

```
ncsim> scope r1
```

Use `where` to display the current simulation location.

```
ncsim> where
Line 30, file "./libdir/cells_lib.v", scope (register_test.r1.d7)
Scope is (register_test.r1)
```
Setting the Debug Scope (continued)

NC-Verilog

When you first enter the interactive mode, the current debug scope is at the top level or at the source code scope where the simulator encountered the $finish or $stop statement.

Use the scope command to set the current scope, display all or some of the source code at the current scope, or to display information about a specific scope.

Use the scope -set command (the default) to change the debug scope to the specified scope. You can use full or relative hierarchical scope names.

You can directly reference objects defined in the current module without providing their full hierarchical names.
Examing the Design

Verilog-XL

Use $list to decompile the debug (or specified) scope.

Use the `-d` command-line option to decompile the entire design.

```verilog
C4 > $list(register_test.cl);
// libdir/clock.v
7 `timescale 1ns / 1ns
7 module clock(clk);
8 output
8     clk; // = StX
10 parameter
10     PERIOD = 20; // = 32'h14, 20
12 reg
12     start; // = 1'hx, x
14     nand #(PERIOD / 2)
14         (clk, clk, start);
17* initial
18     begin
19         start = 0;
20         #(PERIOD / 2)
21         start = 1;
22     end
24 endmodule
```
Examining the Design

Verilog-XL

Use the $list system task to display the source code at the current (or a specified) scope. The line-numbered output references the source file name, lists the current value for each object (and strength for nets), and places an asterisk next to line numbers that are currently active.
Examining the Design

NC-Verilog

Use `scope -list` to list the debug (or specified) scope source.

```verilog
ncsim> scope -list 7 - register_test.cl
7: module clock (clk);
8: output clk;
9:
10: parameter PERIOD=20; // Can be customized
11:
12: reg start; // Start clk from x to 1
13:
14: nand #(PERIOD/2) (clk, clk, start); // Toggles after (PERIOD/2)
15:
16:
17: initial
18: begin
19: start = 0; // clk transitions X to 1 at time 0 + (PERIOD/2)
20: #(PERIOD/2)
21: start = 1; // clk transitions 1 to 0 at time (PERIOD/2) + (PERIOD/2)
22: end
23:
24: endmodule
```
Examining the Design (continued)
Debug system tasks accept hierarchical pathname arguments.

The following command sets are equivalent:

```verbatim
$scope(register_test.r1.d7);
$list;

$scope(register_test.r1);
$list(d7);

$scope(register_test);
$list(r1.d7);
```
Traversing the Design
Traversing the Design

NC-Verilog

Debug commands accept hierarchical pathname arguments.

The following command sets are equivalent:

```plaintext
scope -set register_test.r1.d7
scope -list

scope -set register_test.r1
scope -list d7

scope -set register_test
scope -list r1.d7
```
**Reading Signal Values**

**Verilog-XL**

Use `$display`, `$strobe`, and `$write` to display information or values.

```
C5 > $display("%0d NS", $time," out=%b data=%h load=%b", out, data, load);
  30 NS out=x000z000 data=xx load=x
```

Use the `$stime`, `$time`, and `$realtime` system functions to obtain simulation time.

```
C6 > $strobe("%0d NS", $time," out=%b data=%h load=%b", out, data, load);
C7 > #1 $stop; .
  31 NS out=x000z000 data=xx load=x
C7 : $stop at simulation time 31
```
Reading Signal Values

Verilog-XL

Use the $display, $strobe, and $write system tasks for general purpose display of design values. The $display system task immediately displays the formatted values of its arguments. The $strobe system task works similarly, but waits until the end of the current simulation time instant. The $write system task works identically to $display, but does not automatically output a newline character.

Use the $time, $realtime, and $stime system functions to obtain the current simulation time, as a time, real, and integer data type (respectively).
Reading Signal Values

NC-Verilog

Use the standard Tcl `format` command to format a string, and `puts` to write it.

```
puts "[time] [format "out=%s data=%s load=%s" [value %b out] [value %h data] [value %b load]]"
30 NS out=8'bx000z000 data=8'hff load=1'b1
```

Use `value`, `describe`, and `scope -describe` to display information or values.

```
ncsim> value %b out
8'bx000z000

ncsim> describe out[3]
out.........wire [7:0]
  out[3] (wire/tri) = HiZ
```

Use `time` to display the current simulation time.

```
ncsim> time -delta
10 NS + 0
```
Reading Signal Values (continued)

NC-Verilog

Use the **value** command to display the value of one or more simulation objects. You can set the default display format in the `vlog_format` variable. If not set, it defaults to `%h`. You can shortcut the `value` command with "#", i.e. replace `[value out]` with `#out`.

Use the **describe** command to obtain a brief description of simulation object(s).

Use the **time** command to display the current simulation time. The example displays simulation time using the units given by the `$display_unit` Tcl variable, which defaults to **auto** (the largest base that displays a time no smaller than 1) and includes the delta cycle count (the number of times zero-delay events have been propagated).

**Note:** To obtain the value of an object, include it in a breakpoint condition, or probe it, you must have set read access for the object during elaboration. You can do this globally with the `+ncaccess+r` command-line option.
Investigating Signal Drivers

Verilog-XL

Use $showvars or $showvariables to obtain detailed information about a net or register.

C8 > $showvars(register_test.r1.rst_);
rst_ (register_test.r1) wire = St0
   St0 <- (register_test.r1): port 5

Use $countdrivers to count all drivers of a single bit of a net.

C9 > $display("register_test.r1.d7.rst_ has %0d drivers",
    > $countdrivers(register_test.r1.d7.rst_));
register_test.r1.d7.rst_ has 0 drivers
Investigating Signal Drivers

Verilog-XL

Use the `$showvars` system task to list all the contributors to the value of one or more signals. You can provide multiple signal and/or scope arguments. The system task by default reports the following information on all signals in the current scope:

<table>
<thead>
<tr>
<th>Name of the variable</th>
<th>Current value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope of the variable</td>
<td>Whether the variable is forced</td>
</tr>
<tr>
<td>Type of variable</td>
<td>Decompilation of drivers, with output values</td>
</tr>
<tr>
<td>Future value scheduled</td>
<td>Future values of drivers, if scheduled</td>
</tr>
</tbody>
</table>

Use the `$showvariables` system task to list all the contributors to the value of one or more signals. Without arguments, this system task acts like `$showvars`. You can provide a (0-7) integer argument:

- If bit 0 is 1 it displays information for the hierarchy rooted at the current debug scope
- If bit 1 is 1 it suppresses information on the drivers of the signals
- If bit 2 is 1 it reports only on variables with unknown values
Investigating Signal Drivers

NC-Verilog

Use **drivers** or **scope -drivers** to display detailed information about contributors to the value of nets or registers.

```
ncsim> drivers register_test.r1.rst_
register_test.r1.rst_...input (wire/tri) = St0
    St0 <- (register_test.r1) input port 5, bit 0
(./register_test.v:18)
```
Investigating Signal Drivers (continued)

NC-Verilog

Use the **drivers** command to list all the contributors to the value of one or more signals. Use the **drivers -future** command to list scheduled values. You can provide multiple signal arguments. The **scope -drivers** command reports the following information on all signals in the current scope:

<table>
<thead>
<tr>
<th>Name of the variable</th>
<th>Current value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope of the variable</td>
<td>Whether the variable is forced</td>
</tr>
<tr>
<td>Type of variable</td>
<td>Decompilation of drivers, with output values</td>
</tr>
<tr>
<td>Future value scheduled</td>
<td>Future values of drivers, if scheduled</td>
</tr>
</tbody>
</table>

**Note:** To show drivers, you must have set connectivity access during elaboration. You can do this globally with the `+ncaccess+c` command-line option.
Setting Signal Values

Verilog-XL

Use **force** to force a signal to a continuous assignment.

C10 > force register_test.r1.rts_ = register_test.r1.rst_;

Use **release** to release a **force**.

C11 > release register_test.r1.rts_;

**Note:** Use **$list_forces** to list all currently active forces.

Use **$deposit** to deposit a value on a net or register. The signal retains the deposited value until its next scheduled change. A deposit to a currently forced register has no effect. A deposit to a currently forced net takes effect when you release the net.

C12 > $displayb(register_test.r1.rts_);

C13 > $deposit(register_test.r1.rts_, 0);
C14 > 12
0
Setting Signal Values

Verilog-XL

You might want to interactively patch the design to initialize all or portions of a circuit to a known state, set the circuit to a different state, override incorrect values and continue simulation, or break feedback loops by setting them to a known state.

Use the **force** statement to temporarily force a signal to a continuous assignment. The forced values propagate normally through the circuit. You can force only whole unarrayed registers and nets, and bit or part selects only of expanded vector nets. A **force** overrides all other drivers, and stays in effect until replaced with another **force** or canceled by a **release**.

Use the **release** statement to cancel forces on one or more signals. The object value returns to the value of its drivers.

Use the **$list_forces** system task to list the currently active force statements.

Use the **$deposit** system task to give a whole net or register a value that will then propagate forward. You can deposit 0, 1, Z and X states. The signal retains the deposited value until its next scheduled change.

You can also interactively make procedural assignment statements.
Setting Signal Values

NC-Verilog

Use **force** to force a signal to a value.

```
ncsim> force register_test.r1.rts_ = 1'b0
```

Use **release** to release a **force**.

```
ncsim> release register_test.r1.rts_ -keepvalue
```

**Note:** Use **scope -drivers** to list all drivers, including currently active forces, on signals at the current scope. Deposited values are not drivers.

Use **deposit** to deposit a value on a net or register. The signal retains the deposited value until its next scheduled change. A deposit to a currently forced register has no effect. A deposit to a currently forced net takes effect when you release the net.

```
ncsim> value %b register_test.r1.rts_ 1'b0
ncsim> deposit register_test.r1.rts_ 0
ncsim> !-2
value %b register_test.r1.rts_ 1'b0
```
Setting Signal Values (continued)

NC-Verilog

You might want to interactively patch the design to initialize all or portions of a circuit to a known state, set the circuit to a different state, override incorrect values and continue simulation, or break feedback loops by setting them to a known state.

Use the `force` command to temporarily force a signal to a value. The forced value propagates normally through the circuit. You can force only whole unarrayed registers and nets, and bit or part selects only of expanded vector nets. The value assigned must be a Verilog literal. A `force` overrides all other drivers, and stays in effect until replaced with another `force` or canceled by a `release`. This command behaves similarly to a Verilog `force` statement.

Use the `release` command to cancel forces on one or more signals. The object value returns to the value of its drivers. This command behaves identically to a Verilog `release` statement.

Use the `deposit` command to set the value of a specified wire, register, memory, or memory element. The value assigned must be a Verilog literal. The deposited value propagates normally through the circuit.

To patch a signal, you must have set write access to the object during elaboration. You can do this globally with the `+ncaccess+w` command-line option.
Setting Breakpoints

Verilog-XL

Use `$stop` or choose from several source-level debugging system tasks to set and control breakpoints during your interactive session. Enter a period (.) to resume simulation.

```
C1 > #10$stop; break once after 10 more time units
C2 > $db_breakonnegedge(rst_); Set break (1) on neg edge register_test.rst_.
C3 > .
C1: $stop at simulation time 10
C3 > .
Break (1) occurred on neg edge register_test.rst_.
C3 > force register_test.r1.rts_ = 0;
C4 > $db_breakonposedge(rst_); Set break (2) on pos edge register_test.rst_.
C5 > .
Break (2) occurred on pos edge register_test.rst_.
C5 > force register_test.r1.rts_ = 1;
C6 > $db_disablebreak(1); disable breakpoint 1
Disabled break (1) on neg edge register_test.rst_.
```
Setting Breakpoints

Verilog-XL

You can set simple breakpoints using the delay specification in conjunction with $\text{stop}$. You can set complicated breakpoints using the behavioral constructs of the Verilog language. The simulator will even interactively accept some compiler directives. Until you complete the interactive command, the simulator responds with $>$ instead of $\text{Cn}>$. After setting the breakpoint, the simulator does not automatically continue execution. Enter a period (.) to resume simulation.

Upon resuming simulation, the simulator executes all pending interactive commands concurrently. If you want the simulator to execute a set of commands sequentially, you must group them in a \texttt{begin-end} block.
Setting Breakpoints

NC-Verilog

Use `stop` to create, disable, enable, delete, or show breakpoints of various kinds during your interactive session. Enter `run` to resume simulation.

```
ncsim> run 10
Ran until 10 NS + 0
ncsim> stop -object rst_ -if {#rst_==0}
Created stop 1
ncsim> run
20 NS + 0 (stop 1: register_test.rst_ = 0)
./register_test.v:63 {rst_, load, data} =
ncsim> force register_test.r1.rts_ = 0
ncsim> stop -object rst_ -if {#rst_==1}
Created stop 2
ncsim> run
40 NS + 0 (stop 2: register_test.rst_ = 1)
./register_test.v:64 {rst_, load, data} =
ncsim> force register_test.r1.rts_ = 1
ncsim> stop -disable 1
```
Setting Breakpoints (continued)

NC-Verilog

Use the `stop` command to create, delete, disable, or enable a breakpoint, or to show information about it. You can set breakpoints on conditions, lines, objects, and times.

Until you complete the interactive command, the simulator responds with `>` instead of `ncsim>`. After setting the breakpoint, the simulator does not automatically continue execution. Enter `run` to resume simulation.

Conditional breakpoints need to access object values. Invoke the simulator with the `+ncaccess+r` option to enable read access to simulation objects on a global basis.

Line breakpoints need to access source code lines. Invoke the simulator with the `+nclinedebug` option to enable access to source code lines.
Running the Simulation

Verilog-XL

Use the period (.) key to resume simulation.

Use the semicolon (;) key to step one statement.

Use the comma (,) key to step one statement and trace events.

Use $db_step$ to step one (or a specified number of) statements, stepping over any part of the design not in the focus.

Use $db_steptime$ to step the simulation a specified amount of time.

```cpp
C7 > $db_steptime(20);
Stepped to time 60.
C8 > $db_step(1);
Stepped to line 64, scope register_test, file register_test.v, time 60.
```
Running the Simulation

Verilog-XL

Use the $db_step system task to step the simulation one (or a specified number of) statements. This system task steps over scopes not in the current focus. Use the $db_setfocus, $db_disablefocus, $db_enablefocus, and $db_deletefocus system tasks to establish the focus of your debug activity.

The current focus by default contains all scopes in the design. Until you set a focus, $db_step acts identically to the comma (,). If you have turned tracing on with the $settrace system task or the -t command-line option, then the semicolon also performs a trace step.
Running the Simulation

NC-Verilog

Use `run -cycle` to run to the next (or a specified number of) delta cycles.

Use `run -phase` to run to the next phase of the delta cycle.

Use `run -next` to step one statement, stepping over subprogram calls.

Use `run -return` to run until the current subprogram returns.

Use `run -step` to step one statement, stepping into subprogram calls.

Use `run -timepoint` (the default) to step a specified relative (default) or absolute time.

```
ncsim> run -timepoint 20
Ran until 60 NS + 0
ncsim> run -step
Stepped to 60 NS + 0
```
Running the Simulation (continued)

NC-Verilog

Use the run command to start simulation or resume a previously halted simulation. Simulation proceeds according to the specified arguments. The timepoint may be relative (default) or absolute time. Use the run command without arguments to run simulation until it completes, or until a breakpoint or error occurs.

A Verilog subprogram is a Verilog function or task.

A Verilog process is a behavior that can be scheduled. These include procedural blocks, continuous assignments, nonblocking assignments, and $monitor statements.

The simulator may proceed through several delta cycles to process all simulation events scheduled for a given simulation time instant. A delta cycle consists of two phases: A signal evaluation phase; and a process execution phase. Either phase can generate new events to be handled by the other phase during the current simulation time instant.
Tracing Simulation Activity

Verilog-XL

Use $settrace and $cleartrace to turn tracing on and off.

Use $db_settrace and $db_cleartrace to turn tracing on and off for the area of the design in the focus.

Use the -t command line option to turn on tracing for the simulation session.

C9 > $settrace;
C10 > $db_step(4);
SIMULATION TIME IS 60
C10: $db_step(4);
L18 "register_test.v": wire clk >>> FROMXL NET = St0
L64 "register_test.v": @(negedge clk) >>> CONTINUE
L64 "register_test.v": expect('h0);
L40 "register_test.v": if(out !== expects) >>> TRUE
L41 "register_test.v": begin
Stepped to line 42, scope register_test.expect, file register_test.v, time 60.

Note: There are no comparable NC-Verilog tracing commands. You could perform limited tracing with the stop-delta, run-phase and process-all commands. The SimVision GUI versions 3.2 and higher provide some NC-Verilog simulation tracing capability.
Tracing Simulation Activity

Verilog-XL

The trace displays the line number and file name of the source statement for each event.

You can trace simulation events to debug race conditions. As tracing produces voluminous output, you would want to trace only a short interval of simulation time around the race condition.
Monitoring Signal Values

Verilog-XL

Use $monitor, $monitoroff, and $monitoron to create, disable, and enable signal monitoring.

```
C11 > $monitorb("%0d",$time,,rst,,load,,data,,out);
C12 > $db_steptime(40);
```

```
60 1 1 11111111 x000zx00
73 1 1 11111111 x111z111
80 1 1 00000000 x111z111
95 1 1 00000000 0000z000
```
Monitoring Signal Values

Verilog-XL

Use the $monitor system task to monitor signal value changes. The $monitor system task displays the formatted values of all of its arguments at the end of any simulation time instant in which any of them changed. You can have at most one $monitor system task active at a time. A new $monitor system task overrides the previous one.
Monitoring Signal Values

NC-Verilog

Use **probe -screen** to probe signal output to the screen.

```bash
ncsim> probe -screen -format "%b %b %b %b" rst_ load data out
Created probe 1
ncsim> run 40
1'b1 1'b1 8'b1111111 8'bx0000x00
1'b1 1'b1 8'b1111111 8'b111z111
1'b1 1'b1 8'b00000000 8'b111z111
1'b1 1'b1 8'b00000000 8'b0000z000
```

Use **strobe** to display object values at time intervals or when they change.

```bash
ncsim> strobe -condition {
  [value clk] = 1'b1
} {load %b} data
  Setting up strobe condition - '[value clk] == 1'b1'
ncsim> run 20
Time | load | data |
-----------------------------
110 NS | 1'b1 | 8'h0f |
Ran until 120 NS + 0
```
Monitoring Signal Values (continued)

NC-Verilog

Use **probe -screen** to place a probe on a simulation object and send the value change data to the screen, rather than to a waveform database. You can use the **-redirect** option to send the output to a file.

Use the **strobe** command to display object values at intervals. You can have only one **strobe** active at a time. The **strobe** command is a pre-defined Tcl procedure that sets a breakpoint that when activated executes a **value** command for the objects specified. You can set the breakpoint on a condition, object, or time. You can set the default display format in the **strobeFmt** variable. If not set, it defaults to %oh.
Capturing Waveform Data

Verilog-XL

Use $shm_open, $shm_suspend, $shm_resume, and $shm_close to open, suspend, resume, or close an SHM database.

This example opens an SHM database directory waves.shm.

C13 > $shm_open("waves.shm");

Use $shm_probe to probe signal change data to the SHM database.

This example probes all nodes from r1 down to (and including) library cells.

C14 > $shm_probe(r1, "AC");

Use $dumpfile, $dumpoff, and $dumpon to open, suspend, or resume a VCD database.

This example opens a VCD database file verilog.dump.

C13 > $dumpfile("verilog.dump");

Use $dumpvars to probe signal change data to the VCD database.

This example probes all nodes from r1 down to (and including) library cells.

C14 > $dumpvars(0,r1);
Capturing Waveform Data

Verilog-XL

Use the $shm_open, $shm_suspend, $shm_resume, and $shm_close system tasks to open, suspend, resume, or close an SHM database. You can only have one SHM database open at a time.

Use the $shm_probe system task to probe signal change data to the SHM database. The system task accepts optional scope, and node specifier, argument pairs. Each node specifier applies to at most one scope argument. You can omit either argument. By default the system task probes ports of the current scope. You can provide the following node specifiers:

- "S" — All ports down to but excluding library cells
- "C" — All ports down to and including library cells
- "AS" — All nodes down to but excluding library cells
- "AC" — All nodes down to and including library cells

Use $dumpfile, $dumpoff, and $dumpon to open, suspend, or resume a VCD database. You can open only one VCD database during simulation, so no system task exists to close it.

Use the $dumpvars system task to dump signal change data to the VCD database. The system task accepts an optional level argument, and optional scope (or signal) arguments. You can omit either argument. The default level (0) indicates the entire hierarchy, and the default scope is the top level of the design. Level argument 1 dumps only the specified scope, 2 includes one level down, and so on.
Capturing Waveform Data

NC-Verilog

Use **database** to open, disable, enable, close, or show information about an EVCD, SHM or VCD database.

This example opens the SHM database `waves` in directory `waves.shm`.

```plaintext
ncsim> database -open waves -into waves.shm
```

Use **probe** to create, disable, enable, delete, or show information about a probe.

This example creates a probe named `peek` that probes all nodes from `r1` down to (and including) leaf cells into the default SHM database, and if invoked in the GUI, adds the signals to the waveform display (and opens it if not open).

```plaintext
ncsim> probe -create -name peek -shm r1 -all -depth all -waveform
```
Capturing Waveform Data (continued)

NC-Verilog

Use the `database` command to open an EVCD, SHM or VCD database, or to close, disable, enable, or show information about open databases. You cannot open an NC-Sim version 3.4 EVCD database for Verilog using the `database` Tcl command. Open an NC-Sim version 3.4 EVCD database for Verilog using the `$dumpports` system task in your Verilog source code.

Use the `probe` command to create a probe, or to delete, disable, enable, or show information about probes. The simulator saves value change data to the database upon each signal transition. You cannot create an NC-Sim version 3.4 EVCD probe for Verilog using the `probe` Tcl command. Create an NC-Sim version 3.4 EVCD probe for Verilog using the `$dumpports` system task in your Verilog source code.

You cannot disable VCD or EVCD probes. You can delete VCD and EVCD probes only immediately after you create them (if simulation time has not yet advanced).
Using the Command History List

Verilog-XL

Use $\texttt{history}$ to list previously executed interactive commands.

Enter the command number to re-execute a command.

Enter \textit{-command\_number} to deactivate a command.

C9 > $\texttt{history}$
SIMULATION TIME IS 60
C9: $\texttt{history}$

Command history:
C1 $\texttt{shm\_open("waves.shm")}$
C2 $\texttt{shm\_probe(r1, "AC")}$
C3 $\texttt{monitorb(\"%0d\", $time, , rst_, , load, , data, , out)}$
C4 #10
   $\texttt{stop}$
C5 $\texttt{db\_breakonceonnegedge(rst_)}$
C6 force register\_test.r1.rts\_ = 0;
C7 $\texttt{db\_breakonceonposedge(rst_)}$
C8 force register\_test.r1.rts\_ = 1;
C9* $\texttt{history}$

\textbf{Note:} The asterisk (*) denotes currently active commands.
Using the Command History List
Using the Command History List

NC-Verilog

Use **history** to list previously executed interactive commands.

Enter **!command_number** or **history redo command_number** to re-execute a command.

ncsim> history
1 database -open waves -into waves.shm
2 probe -create -name peek -shm r1 -all -depth all -waveform
3 probe -screen -format "%b %b %b %b" rst_ load data out
4 run -timepoint 10
5 stop -object rst_ -if {#rst_ == 0} -delbreak 1
6 run
7 force register_test.r1.rts_ 0
8 stop -object rst_ -if {#rst_ == 1} -delbreak 1
9 run
10 force register_test.r1.rts_ 1
11 history
ncsim> history redo 8
Created stop 3
Using the Command History List (continued)

NC-Verilog

Use the `history` command to modify the number of commands the history mechanism will keep, to redo a specific previous command, and to substitute for parts of a previous command before re-executing it. The `history` command is a standard Tcl command.
Executing a Command Script

Verilog-XL

Use $input or the -i command-line option to execute a script file.

The simulator executes the command script upon entering interactive mode.

Assuming the command script contains:

```
$display("Executed script at %0d",$time);
```

If you invoke the simulator with the -i and -s options, it enters interactive mode, and echoes and executes the script commands. If you input the command script with $input and run the simulation, the simulator echoes and executes the script at the next interactive prompt.

```
verilog -f run.f -i script.txt -q -s
C1 > $display("Executed script at %0d",$time);
  Executed script at 0
C2 > begin $input("script.txt"); #10 $stop; end .
C2: $stop at simulation time 10
C3 > $display("Executed script at %0d",$time);
  Executed script at 10
```
Executing a Command Script

Verilog-XL

The simulator by default writes interactive commands to the `verilog.key` keyfile.

You can use the `-k` command-line option to rename the keyfile.

You can edit the keyfile or create your own command script file.

You can use `$input` or the `-i` command-line option to read and execute a file containing interactive commands, such as the keyfile.

The simulator reads and executes the command script upon entering interactive mode.

**Note:** Group statements in interactive mode with `begin-end` to achieve sequential ordering. The simulator normally executes interactive statements concurrently upon resuming simulation.
Executing a Command Script

NC-Verilog

Use `input` or the `+ncinput+` command-line option to execute a script file.

The simulator executes the command script upon entering interactive mode.

Assuming the command script contains:

```plaintext
puts [format "Executed script at %s" [time -nounit]]
```

If you invoke the simulator with the `-s` and `+ncinput+` options, it enters interactive mode, and echoes and executes the script commands. If you input the command script with `input` and run the simulation, the simulator echoes and executes the script at the next interactive prompt.

```plaintext
ncverilog -f run.f -q -s +ncinput+script.tcl
ncsim> puts [format "Executed script at %s" [time -nounit]]
Executed script at 0
ncsim> input script.tcl; run 10
Ran until 10 NS + 0
ncsim> puts [format "Executed script at %s" [time -nounit]]
Executed script at 10
```
Executing a Command Script (continued)

NC-Verilog

The simulator by default writes interactive commands to the *ncsim.key* keyfile.

You can use the `-k` command-line option to rename the keyfile.

You can use the `save -environment` command to interactively save command scripts.

You can edit the keyfile or create your own command script file.

You can use the `input` command or the `+ncinput+` command-line option to read and execute a file containing interactive commands, such as the keyfile.

The simulator reads and executes each command at an interactive prompt.

*Note:* You can also use the standard Tcl `source` command. This course emphasizes the `input` command extension, as it executes as if you had entered the commands at the keyboard: it executes each command individually at an interactive prompt, it displays the result of each command as executed, and it does not exit upon encountering an error.
Saving, Resetting, and Restarting Simulation

Verilog-XL

Use $\texttt{save}$ and $\texttt{incsave}$ to save the entire simulation data structure.

\begin{verbatim}
C1 > $db_breakonposedge(rst_); // Reset the design
C2 > $save("save.dat"); // save the initialized design
C3> forever #100000 // After every 100000 ns
    > $incsave("incsave.dat"); // Incrementally save the simulation
\end{verbatim}

Use $\texttt{restart}$ or the $\texttt{-r}$ command-line option to restart the simulation from the time of the save.

This saves you from having to recompile and resimulate up to that point.

\begin{verbatim}
C1 > $restart("incsave.dat");
\end{verbatim}

Use $\texttt{reset}$ to reset the simulation to the time 0 state.
Saving, Resetting, and Restarting Simulation

Verilog-XL

You can save the simulation data structure into a file and restart the simulation later. Use this feature to check-point a long simulation, so that you can protect your simulation from machine failures, or perform additional tests on an already initialized design.

Use the $save system task to save the simulation data structure. Provide a filename argument in which to store the simulation data structure.

Use the $incsave system task to perform subsequent incremental saves. Incremental saves use less file space.

Use the $restart system task or −r command line option to restart the simulation with a previously saved data structure. Provide as an argument the name of the file in which you stored the simulation data structure. With the −r command line option you do not specify the design source files.
**Saving, Resetting, and Restarting Simulation**

**NC-Verilog**

Use **save** to save a simulation snapshot.

```ncsim> stop -object rst_ -if {#rst_ == 1} -delbreak 1
Created stop 1
ncsim> run
40 NS + 0 (stop 1: register_test.rst_ = 1)
./register_test.v:64 {rst_, load, data} = 10'b1_0_11111111;
@ (negedge clk) expect ('h00);
ncsim> save save_dat
Saved snapshot worklib.save_dat:v
ncsim> stop -time 100000 -execute {save incsave_dat}
Created stop 2```

Use **restart** or the **-r** command-line option to reload a simulation snapshot.

```ncsim> restart save_dat```

Use **reset** to reload the time zero snapshot.

**Note:** The NC-Verilog simulator stores elaborated design data and the initial time 0 snapshot in a derived-objects library. The **save** and **restart** commands need to save and restore only the design state, not the entire simulation data structure.
NC-Verilog

You can save the simulation state into a snapshot and restart the simulation later. Use this feature to check-point a long simulation, so that you can protect your simulation from machine failures, or perform additional tests on an already initialized design.

Use the `save` command to save the current snapshot. Provide a snapshot name argument.

Use the `restart` command or `-r` command-line option to restart the simulation with a previously saved snapshot. Provide as an argument the name of the saved snapshot.

Note that with the `-r` command line option you do not specify the design files.

The simulator cannot execute `save` when it is in the middle of executing a procedural statement. Use `run -clean` to reach the next "clean" point in simulation, then issue the `save` command. You can then reload the snapshot at a later time to continue simulation.

The state of the Tcl environment (including probes) is not saved with the snapshot. To save the Tcl environment, you must perform a separate `save -environment`. To restore the Tcl environment upon `restart`, you must `source` the saved environment file.
**Exiting the Simulator**

**Verilog-XL**

In batch mode, the simulator exits upon executing `$finish` in the source code.

In interactive mode, the simulator exits if:

- You enter `$finish[(n)]` from the interactive prompt
- You enter `Control-d` from the interactive prompt
- The simulator executes `$finish[(n)]` in the source code

```
C6 > $finish(1);
C6: $finish at simulation time 80
```

The optional command argument controls the display of run-time statistics:

- 0 — Display no statistics
- 1 — Display simulation time (default)
- 2 — Display simulation time and memory and CPU usage
Exiting the Simulator

Verilog-XL

Occasionally, a user may inadvertently code a compute-intensive loop that does not advance simulation time. Such a loop will eventually use up all of the platform memory if allowed to continue.

The simulator may not respond to the Control-c interrupt while executing a compute-intensive loop that does not advance simulation time. You may need to kill the process.

You can kill a UNIX process by sending it the KILL signal. You can usually do this interactively by entering Control-\. You can do it from a Bourne or C shell by entering:

```
kill –KILL pid
```

The pid is the process ID of the simulator process. You can find it by using the ps command and looking for the pid associated with the simulator.

You can kill an NT process with the Task Manager.

At the interactive prompt, use the $finish system task to exit simulation and return control to the operating system. You can optionally display simulation time and memory/CPU usage statistics, depending upon the task argument.
Exiting the Simulator

NC-Verilog

In batch mode, the simulator exits upon executing $\text{finish}$ in the source code.

In interactive mode, the simulator exits if:

- You enter $\text{finish}[n]$ or $\text{exit}$ from the interactive prompt
- You enter Control-d twice consecutively from the interactive prompt
- The simulator does not exit upon executing $\text{finish}[(n)]$ in the source code; instead it returns to the interactive prompt

The optional command argument controls the display of run-time statistics:

- 0 — Display no statistics (default for $\text{finish}$)
- 1 — Display simulation time (default for $\text{finish}$)
- 2 — Display simulation time and memory and CPU usage
Exiting the Simulator (continued)

NC-Verilog

Occasionally, a user may inadvertently code a compute-intensive loop that does not advance simulation time. Such a loop will eventually use up all of the platform memory if allowed to continue.

The simulator will eventually respond to repeated Control-c interrupts (5-6) while executing a compute-intensive loop that does not advance simulation time.

At the interactive prompt, use the finish command to exit simulation and return control to the operating system. You can optionally display simulation time and memory/CPU usage statistics, depending upon the optional command argument.
Other Miscellaneous Commands

Verilog-XL

- **$cputime** — Get CPU time for the simulation

  ```
  C2 > $display("Simulation took %g/10 seconds", $cputime);
  Simulation took 6/10 seconds
  ```

- **$listcounts** — report statement coverage of the current (or a specific) scope.

  ```
  C25 > $listcounts(r1);
  ```
Other Miscellaneous Commands

Verilog-XL

Use the $cputime system function to get the CPU time of the simulation process in 10’ths of seconds.

Use the $listcounts system task to report statement coverage of the current (or a specific) scope. You need to invoke the simulator with the +listcounts option to enable this feature. The report lists each statement in the scope with the number of time the simulator executed it.
Other Miscellaneous Commands

NC-Verilog

- **alias** — Set, unset, or show a simulator command alias
  
  ncsim> alias -set go {run 10 ns; value sum}

- **call** — Invoke a user-defined system task or function
  
  ncsim> call {$myprinter} {"set value to "} 8'hc

- **process** — Identify processes that are executing or are currently scheduled to execute
  
  ncsim> process -all 10

- **Use status** to display the CPU time in simulation.
  
  ncsim> status
  Memory Usage - text: 3656824, static: 541600, dynamic: 835136, total: 5053560
  CPU Usage - 12.21 seconds (user = 6.3)
  Simulation Time - 18721 NS + 0

- **task** — Invoke a Verilog task
  
  ncsim> task my_pci_protocol_monitor

- **version** — Display the version of the simulator
Other Miscellaneous Commands (continued)

NC-Verilog

Use the alias command to set, clear, or display command aliases. The alias command is a standard Tcl command.

Use the call command to invoke a user-defined system task or function (Verilog) or a C-Interface Function (VHDL). Insert the -systf or -predefined option to differentiate between identically named functions when coexecuting a Verilog and a VHDL simulation. Note that you cannot use the call command to invoke Verilog tasks and functions (see task). You use it only to invoke external user-defined system tasks or functions that use the PLI or VPI.

Use the process command to identify processes that are executing or are currently scheduled to execute. A Verilog process is a procedural block, continuous assignment, nonblocking assignment, or $monitor statement.

Use the status command to display memory usage, CPU usage, and simulation time.

Use the task command to call a Verilog HDL task. Note that you cannot use the task command to invoke external user-defined system tasks or functions that use the PLI or VPI (see call). To pass argument values to the task, deposit values on its input and inout ports before calling it. For example, if task test is defined with input N, you could call it as follows:

csim> deposit test.N 1; task test

Use the version command to display the simulator version.
Summary

In this presentation, you learned about:

- Entering the interactive simulation mode
- Examining and traversing the design
- Reading and writing simulation objects
- Setting breakpoints and running the simulation
- Tracing events and monitoring signals
- Capturing waveform data
- Using the command history list
- Executing commands from a script
- Saving, resetting, and restarting the simulation
Review

1. How do you enter the interactive mode?
2. How do you set the current debug scope?
3. How do you obtain the current value of a signal?
4. How do you set the value of a signal?
5. How do you capture SHM waveform data?
Review

1. How do you enter the interactive mode?
   
   You enter the interactive mode with the \texttt{-s} command-line option, the \texttt{Control-c} keyboard interrupt, a \texttt{$stop} in your source code, or upon activating a breakpoint.

2. How do you set the current debug scope?
   
   Set the current debug scope with the Verilog-XL \texttt{$scope} system task or the NC-Verilog \texttt{scope -set} Tcl command.

3. How do you obtain the current value of a signal?
   
   Obtain the current value of a signal with the Verilog-XL \texttt{$display} system task or the NC-Verilog \texttt{value} Tcl command.

4. How do you set the value of a signal?
   
   Set the value of a signal with the Verilog-XL \texttt{$deposit} system task or \texttt{force} statement, or the NC-Verilog \texttt{deposit} or \texttt{force} Tcl commands. You can \texttt{deposit} only to whole nets and registers, You can also \texttt{force} bit and part selects of expanded vector nets. A \texttt{force} overrides a \texttt{deposit}.

5. How do you capture SHM waveform data?
   
   Open an SHM database with the Verilog-XL \texttt{$shm_open} system task or the NC-Verilog \texttt{database -open} Tcl command. Probe signals to the database with the Verilog-XL \texttt{$shm_probe} system task, or the NC-Verilog \texttt{probe -create -shm} Tcl command.
Chapter 13: Debugging With the Graphical User Interface

Objective

In this section, you will learn about the SimVision graphical simulation analysis environment:

- SimControl
- Navigator
- Watch windows
- Signal Flow Browser (SFB)
- Cycle View
- Schematic windows
- Waveform View
Debugging with the Graphical User Interface

This Cadence Verilog-XL and NC-Verilog simulator-specific section presents a high-level overview of the graphical Cadence simulation analysis environment (SimVision). It presents each component of the environment and its main features. As the GUI is designed to be self-evident and intuitively friendly to use, the section does not document the GUI in detail. In general, almost every textual command you can enter interactively is available through a GUI menu item, and many commands are associated with GUI tool buttons and context-sensitive pop-up menus. The GUI issues these commands for you when you select the menu item or tool button.
Invoking the Graphical Environment

Invoke the graphical simulation analysis environment with the +gui option.

```
verilog +gui -s command_line_options design_files
```

SimControl is your main control window. It provides access to the other windows.

- **SimControl** is your main control window. It provides access to the other windows.
- **Navigator** traverses the design hierarchy.
- **Watch windows** display sets of signals and their values.
- **Signal Flow Browser** traces signal flow upstream to its source.
- **Cycle View** facilitates single stepping and tracing simulation activity (NC-Verilog-only).
- **Schematic windows** display sets of signals and their values and connectivity.
- **Waveform View** displays waveforms. You can also invoke simvision on the command line.

**Note:** You should also use the -s option when you use the +gui option with Verilog-XL. Without the -s option, Verilog-XL will immediately commence simulation without waiting for user input.
Invoking the Graphical Environment

The SimVision Graphical Environment consists of:

- **SimControl** — Your main graphical interface with the simulator
  Use SimControl to advance and interrupt simulation, break into and alter simulation, control scope and so on.

- **Navigator** — A graphical tool with which you navigate the design hierarchy and retrieve information about objects

- **Watch windows** — Windows that display sets of signals and their values in a tabular view

- **Signal Flow Browser** — A graphical tool with which you trace signal flow backward through a design to its sources

- **Cycle View** — A graphical tool that lets you step through a simulation cycle, stopping at each time point, delta cycle, simulation phase, or scheduled process

- **Schematic windows** — Windows that display sets of signals and their values and connectivity in a schematic view

- **Waveform View** — The SimVision graphical waveform viewer

For more information please refer to the Cadence online document *SimVision Analysis Environment User Guide*. 
Debugging With the Graphical User Interface

SimControl

Source

Browser

Scopes

Compiling source file "register_test.v"
Compiling source file "register.v"
Scanning library file "libdir/cells_lib.v"
Scanning library directory "libdir"

Warning! Implicit wire has no fanin "register.v", 18: rts_

Highest level modules:

register_test

Type ? for help

Messages
SimControl

The SimControl main window contains six areas:

- The Menu Bar at the top of the main window contains eight pull-down menus
- The Tool Bar contains fixed menu buttons for frequently used menu items:
  - Run/Stop, Single-Step or Step-Over the simulator
  - Set Scope, Scope Up, Show Execution, and Edit Source
  - Set Breakpoint, Set Force, and Show Value
  - Navigator, Watch View, Signal Flow Browser, Schematic Window, and Waveform View
- The Source Browser displays the source code for the module at the current debug scope
- The Scopes region displays scopes and navigates between them
  - Use the Scope field to enter a debug scope or select from recently visited scopes
  - Use the Subscopes or Files/Units field to select a subscope or related File or Unit
    For Verilog, you can elect to select a design file related to the current debug scope
    For VHDL, you can elect to select a design unit related to the current debug scope
- The I/O Region is your textual interface with the interactive simulation
- The Message Region displays the status of the simulator, and informative messages
The Post-Processing Environment

Use the SimVision post-processing mode to debug simulation data off-line.

The source code snippet and the user interface image are shown to illustrate the debugging process in SimVision.
SimControl (continued)

The Post-Processing Environment

Invoke the simulator with the +ppe option or select the SimControl File — Enter Post Processing menus to enter the post-processing mode.

In the post-processing mode, you release the simulator license, and work with design debug data you previously stored in a waveform database.

Open your previously-generated waveform database with the File — Waveform Database — Open... menus.

In post-processing mode, you can jump, scan, step, or play forward or backward, in time. You can read whatever signal values and connectivity information you have saved during a previous simulation.
**SimControl Menu and Tool Bars**

Use the SimControl pull-down menus and fixed menu buttons to:

- Edit source files, save and restart the simulation, open and close databases
- Run, step, stop, and reset the simulation
- Set time, object, and line breakpoints, force values, probe signals
- Show breakpoints, probes, databases, scopes, the execution point
- Select scopes and subscopes to view and signals and ports to access
- Invoke the Navigator, Watch, SFB, Schematic, and Waveform tools
- Manage user-defined buttons, set the display radix and user preferences
- Access related online documentation

<table>
<thead>
<tr>
<th>File</th>
<th>Run</th>
<th>Set</th>
<th>Show</th>
<th>Select</th>
<th>Windows</th>
<th>Options</th>
<th>Help</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Scope</td>
<td></td>
<td>Objects</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time: 0

<table>
<thead>
<tr>
<th>File</th>
<th>Set</th>
<th>Show</th>
<th>Select</th>
<th>Windows</th>
<th>Options</th>
<th>Help</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Db:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time: 0
SimControl (continued)

SimControl Menu and Tool Bars

With the File menu you can edit source files, save and restart the simulation, open and close SHM databases, execute command scripts, save and restore the graphical environment, reinvoke the simulation, and enter post-processing mode.

With the Run menu you can continue the simulation, or single-step the simulation, stepping into or over subprogram calls.

With the Set menu you can set time, object, and line breakpoints, force values, probe signals, monitor signals, open a database, and set the debug scope.

With the Show menu you can show breakpoints, forces, probes, databases, scopes, the execution point (time and scope), signal values, describe the contents of the current debug scope, and display the command history.

With the Select menu you can select subscopes or any subclass of signals up to the entire contents of the current debug scope.

With the Windows menu you can invoke the Navigator, Waveform, Signal Flow Browser, Watch Objects, and Schematic Views windows.

With the Options menu you can create, edit, reorder, import, and export user-defined tool buttons, set the default value display radix, and set user preferences.

With the Help menu you can access related online documentation.

SimVision also offers tool buttons for the more popular menu selections.
The SimControl Source Browser

The Source Browser displays the source at the current debug scope.

You can select any object in the Source Browser and perform operations on it.

```verilog
module register_test;
  wire [7:0] out;
  reg [7:0] data;
  reg rst_;
  reg locd;
  integer fail;

  // Instantiate submodules
  register r1 (out, data, clk, locd, rst_);
  clock cl (clk);
```
The SimControl Source Browser

The Source Browser displays the source code for the current debug scope.

When you change scope, the Source Browser updates to show the source code for whatever module is the current scope.

You can select any object (signal, scope, or line number) in the Source Browser.

Right-select a signal, scope, or line number in the Source Browser to see a pop-up menu of commands that affect that object or line.
Selecting Objects

In SimVision, selection and pop-up menus make the mouse a powerful tool:

- Object selection is exported to all SimVision windows
- With the Control key you can toggle selection of multiple objects
- Right-select an object to see a context-sensitive pop-up menu
- You can drag-and-drop selected objects between windows
Selecting Objects

When you select one or more objects in a SimVision window, you simultaneously select them in all other SimVision windows where those objects exist. SimVision displays the full hierarchical names of the selected objects in command forms and message windows.

When you hold the Control key and select on an object with the left mouse button, you toggle the selection of that object without affecting the selection of other objects.

When you select an object with the right mouse button, SimVision pops-up a menu with a set of actions that you can apply to the object. In the Source Browser you can also select a line number. The commands that apply to a particular object may differ between SimVision tools.

When you select an object with the middle mouse button, you can drag that object to another location, including a different window.
Setting Breakpoints

With these forms you can set, enable, disable, list, and delete breakpoints. You can set a breakpoint for simulation time, a change of value, a condition, or a line of code. The breakpoint capabilities vary between simulators.

Set—Breakpoint

Show—Breakpoints...
Setting Breakpoints

SimControl offers four methods for setting breakpoints:

- Select the Set — Breakpoint menus
- Select the Show — Breakpoints... menus and select the Set... button
- Select the Set Breakpoint button to set a breakpoint on an object
- Right-select a signal name and select Set Break from the pop-up menu

SimControl offers three kinds of breakpoints:

- Time — Break at a specified absolute or relative simulation time
- Object — Break when a specified signal changes value
  — Verilog-XL can also break on a specific edge
  — NC-Verilog can also break when a conditional expression of signal values becomes true
- Line — Break at a specified source code line
Use the Navigator to view the design hierarchy, and objects at the current scope.
The Navigator

You can invoke the Navigator with the **Windows — Navigator...** menu.

You can also invoke the Navigator with the **Navigator** button.

The Tool Bar of the Navigator is a subset of the Tool Bar in the main window.

Use the Navigator to view your current design hierarchy and the objects (and their values) at the current scope. The Navigator contains a hierarchy browser and a scope viewer.

The Scope Tree window displays component instances in the current scope. As you expand the display to include lower levels of your design, the Navigator displays the components of your design in a tree structure. Each node of the tree is a component within your design hierarchy. Browse the hierarchy by double-clicking the left mouse button on a node to expand to the next (or collapse to the last) level. As you traverse the design, the Navigator displays the objects of the selected node and their values in the Objects List window.

The Objects List window lists the objects contained in the scope, their values, and a description of each object. Use the Current Scope combo box to move between scopes and show objects and values quickly, without changing the view in the Scope Tree window.
Watch Windows

Use a Watch window to display selected signals and their values. The simulator by default updates these values when interrupted. You can set an option to continuously update displayed signal values.
Watch Windows

You can invoke a Watch window with the Windows — Watch Objects — View menu.

You can also invoke a Watch window with the Watch Objects button.

Use the Options menu to customize the display of signal names and values.

Use a Watch window to visually monitor the values of selected signals, to set breakpoints, and to force signals to specific values.

You can select the signals to watch before opening a Watch window, by selecting the signals in any SimVision window, or by using the SimControl Select menu to select all signals, all ports, or all ports of all scopes of the current debug scope.

You can select the signals to watch after opening a Watch window. Select the signal in any other SimVision window and use the Watch View Edit — Add Objects menu or Add Object button to add the selected signal(s), or use the middle mouse button to drag-and-drop the object(s) into the Watch window.

Advance simulation to the next signal transition or breakpoint with the Find Next Edge button.

Lock the window to prevent new updates with the Lock Window button.

You cannot add new signals or advance time in a locked window. When you unlock it, it immediately updates to reflect the current values and the current time.

In the post-processing mode, you can edit the Time field to view values at the new time.
The Signal Flow Browser

Use the Signal Flow Browser to trace drivers of a signal and contributing signals.

![Signal Flow Browser](image)
The Signal Flow Browser

You can invoke the Signal Flow Browser with the **Windows — Signal Flow Browser**... menu.

You can also invoke the Signal Flow Browser with the **Signal Flow Browser** button.

To display the drivers and contributors in a stack view, use the **Stack View** button.

To display the drivers and contributors in a tree view, use the **Tree View** button.

To replace the currently traced signal with another selected signal, use the **Trace Back** button.

Use the Signal Flow Browser to iteratively trace the drivers of a signal and signals contributing to the drivers.

You can select the signal to trace before opening the Signal Flow Browser, by selecting the signal in the Source Browser.

You can select the signal to trace after opening the Signal Flow Browser, by entering a hierarchical signal name in the **Trace** field, or by selecting a signal in any other SimVision window and either using the **Trace Back** button or using the middle mouse button to drag-and-drop the signal into the Signal Flow Browser.
Signal Flow Browser Operations

You can perform the following operations from the Signal Flow Browser:

- You can display the value of a signal in your choice of four radixes.
  Select the signal and choose the **Radix** menu.

- You can show the drivers of a signal.
  Select the signal and choose the **Trace — Show Drivers** menu, or just double-click the signal.

- You can view details about a signal’s driver or input.
  Select the driver and choose the **View — Driver Info...** menu.

- You can show the signals contributing to a driver.
  Select the driver and choose the **Trace — Show Inputs** menu, or just double-click the driver.

- You can trace a driver across a module boundary to show its contributor.
  Select the driver and choose the **Trace — Descend** menu.
Signal Flow Browser (continued)

If you had the following module instantiation in a Driver frame:

```register r1(.r(reg_out) ...)
```

You can select `reg_out` and use the **Trace — Descend** menu to show that `r` is the contributing signal.
The Cycle View

Use the Cycle View to step through a simulation cycle, stopping at each time point, delta cycle, simulation phase, or scheduled process.

Note: The NC-Verilog simulator supports this feature.
The Cycle View

You can invoke the Cycle View with the **Windows — Cycle View...** menu.

You can also invoke the Cycle View with the **Cycle View** button.

With the **Options** menu you can set the number of behavioral blocks to display.

With the tool buttons, you can run the simulation to the next time point, delta cycle, phase, or process, invoke the Navigator, and print the Cycle View window.

Use the Cycle View to step through a simulation cycle, stopping at each time point, delta cycle, simulation phase, or scheduled process:

- A timepoint defaults to 10 ns
- A delta cycle is a single simulation cycle consisting of two phases
- A phase is either signal evaluation or process execution
- A process is a VHDL **process** or a Verilog **always** or **initial** block, or some other scheduled behavior

Use the SimControl tool buttons to single-step through or over subprogram calls.
Schematic Windows

Use a Schematic window to display selected portions of the design.
Schematic Windows

You can invoke a Schematic window with the **Windows — Schematic Views — View** menu.

You can also invoke a Schematic window with the **Schematic Window** button.

Use a Schematic window to visually analyze the values, sources, and sinks of selected signals, to set breakpoints, and to force signals to specific values.

You can select the signals to analyze before opening a Waveform window, by selecting the signals in any SimVision window, or by using the SimControl **Select** menu to select all signals, all ports, or all ports of all scopes of the current debug scope.

You can select the signals to analyze after opening a Schematic window. Select the signal in any other SimVision window and use the Schematic Window **Edit — Add Objects** menu or **Add Objects** button to add the selected signal(s), or use the middle mouse button to drag-and-drop the object(s) into the Schematic Window.

With the provided tool buttons, you can also set a breakpoint or force an object to a value, zoom the display in, out, or to fit the current window size, ascend the simulation hierarchy, fill, collapse, or delete any displayed module, search for simulation objects, invoke the Navigator, Watch, Signal Flow Browser, or Waveform windows, and print the schematic display. You can lock or unlock the window, run the simulation to the next signal change, and trace logic that drives or loads the displayed objects. You can set the trace to increment by single objects or by entire scopes.
Waveform View

Use a Waveform window to display probed signal value change data.
Waveform View

You can invoke the Waveform View with the **Windows — Waveform...** menu.

You can also invoke the Waveform View with the **Waveform View** button.

Use a Waveform window to display probed signal value change data.

You can select the signals to probe and display before opening a Waveform window, by selecting the signals in any SimVision window, or by using the SimControl **Select** menu to select all signals, all ports, or all ports of all scopes of the current debug scope.

You can add selected signals to the already open Waveform window by again clicking the **Waveform View** button.

With the primary tool bar you can open a database, move, copy, paste, and delete selected objects, group or bus (or ungroup or unbus) selected objects, add selected objects to this window, and send selected objects to the target Design Browser, Waveform, or Expression Calculator windows. The Tool Bar also contains buttons related to transaction analysis, that this course does not discuss.

With the secondary tool bar you can search for signals up or down the display, move the primary cursor to the previous or next marker, selected signals edge, or expression, enter the simulation time range for display, or simply zoom the display in, out, full, or between the baseline and primary cursors.
# Waveform View Windows

The SimVision Waveform view provides access to these windows:

<table>
<thead>
<tr>
<th>Window</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Browser</td>
<td>Browse the design and select objects</td>
</tr>
<tr>
<td>Waveform</td>
<td>Analyze probed signals value change data</td>
</tr>
<tr>
<td>Expression Calculator</td>
<td>Create and edit expressions of signals</td>
</tr>
<tr>
<td>Measurement</td>
<td>Report signal statistics</td>
</tr>
<tr>
<td>Transaction Explorer</td>
<td>Analyze recorded transaction occurrence data</td>
</tr>
<tr>
<td>Utilities</td>
<td>Manage SimVision Waves objects</td>
</tr>
<tr>
<td>Design Search</td>
<td>Search for design objects</td>
</tr>
<tr>
<td>Console</td>
<td>Issue SimVision Waves Tcl commands</td>
</tr>
</tbody>
</table>

**Note:** The Utilities window contains tabs to manage markers, groups, expressions, cursors, databases, mnemonic maps, and the Transaction Explorer.
Waveform View

Waveform View Windows

Use a **Design Browser** window to browse the design and send selected objects to the Waveform or Expression Calculator windows.

Use a **Waveform** window to analyze probed signals value change data.

Use an **Expression Calculator** window to create and edit expressions of signals.

Use a **Measurement** window to display statistics of signals between the baseline and the primary cursor. Most of these statistics apply only to analog signals.

Use a **Transaction Explorer** window to analyze recorded transaction occurrence data. The transaction recorder is part of the Cadence *verification cockpit* product. Please refer to the *Transaction Explorer User Guide* for more information.

Use a **Utilities** window to manage markers, groups, expressions, cursors, databases, mnemonic maps, and the Transaction Explorer.

Use a **Design Search** window to perform an advanced search through the databases for design objects, and send the objects to the Design Browser or Waveform windows. With this window you can set the databases, hierarchy, and object types to retrieve, and use the * and ? wildcards.

Use a **Console** window to interact with SimVision Waves using Tcl commands.
Summary

In this section you learned about the SimVision graphical simulation analysis environment:

- SimControl
- Navigator
- Watch windows
- Signal Flow Browser (SFB)
- Cycle View
- Schematic windows
- Waveform View
Review

1. What are the main components of the SimVision GUI?
2. With what component can you display the design hierarchy?
3. With what component can you display and monitor a set of signals?
4. Can you set a breakpoint on an object from a Watch window?
Review

1. What are the main components of the SimVision GUI?
   The seven main window-based components of the SimVision GUI are:
   - SimControl — For controlling the simulation session
   - Navigator — For navigating the design hierarchy
   - Watch windows — For monitoring signal values
   - Signal Flow Browser — For tracing signal paths
   - Cycle View — For stepping through the simulation
   - Schematic windows — For viewing your design components graphically
   - Waveform View — For displaying waveforms

2. With what component can you display the design hierarchy?
   You can display and traverse the design hierarchy with the Navigator. The Navigator displays scopes and objects and the object values. You can display the source code at a scope with the Source Browser, part of the SimControl main window.

3. With what component can you display and monitor a set of signals?
   You can display and monitor a set of signals in your design with a Watch window.

4. Can you set a breakpoint on an object from a Watch window?
   You CAN set a breakpoint on an object from a Watch window.
About Lab 7

The objective of this lab is to use the graphical simulation analysis environment to debug a model of an 8-bit register.

In this lab, you will:

- Run the simulator with the SimVision graphical simulation analysis environment
- Use components of the graphical environment to debug a design
About Lab 7
Chapter 14: Support for Verification

Objectives
In this section you will learn about:

- Accessing simulation time
- Displaying signal values
- Monitoring signal values
Support for Verification

This section introduces the system tasks you can use to access and display simulation time and object values, write output to a file, and load data from a file into a Verilog memory.
Accessing Simulation Time

Use $\text{time}$, $\text{realtime}$, or $\text{stime}$ to obtain the current simulation time:

- $\text{time}$ returns time as a 64-bit unsigned integer
- $\text{stime}$ returns time as a 32-bit unsigned integer
- $\text{realtime}$ returns time as a real number

Each of these system functions returns a value that is scaled to the time unit of the module that invoked it.
Accessing Simulation Time

The $time system function returns simulation time scaled and rounded to the calling module timescale, as a 64-bit double-precision unsigned integer time data type.

The $realtime system function returns simulation time scaled to the calling module timescale, as a 64-bit double-precision floating-point real data type.

The $stime system function returns simulation time scaled and rounded to the calling module timescale, as a 32-bit integer data type. The system task returns only the lower 32 bits of the simulation time.
Accessing Simulation Time

Use \$\text{timeformat} to define how the \%t formatter displays time:

\$\text{timeformat}(unit, precision, suffix, min\_width)

`timescale 10ns / 100ps
module top;
reg in1;
not m1(o1, in1);
initial begin
 \$\text{timeformat}(-9, 2, " ns", 10);
in1 = 0;
#8 in1 = 1;
#10 $\text{display}("%t %b %b", $\text{realtime}, in1, o1);
#10 $\text{finish};
end
endmodule

180.00 ns 1 0
## Accessing Simulation Time (continued)

```bash
$timeformat(unit, precision, suffix, min_width)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>Integer between 0 (s) and -15 (fs), indicating the time scale</td>
</tr>
<tr>
<td>precision</td>
<td>Integer number of digits after decimal point</td>
</tr>
<tr>
<td>suffix</td>
<td>String to display after time value</td>
</tr>
<tr>
<td>min_width</td>
<td>Integer minimum field width</td>
</tr>
</tbody>
</table>
Accessing Simulation Time

This example shows $\verb|time|$ and $\verb|stime|$ rounding simulation time to integer values.

```verilog
`timescale 1 ns / 10 ps
module top;
    reg in1;
    not #9.53 n1 (o1,in1);
initial begin
    $display(" time stime realtime in1 o1");
    $display("------------------ ------- ---------- --- --");
    $timeformat(-9, 2, " ns", 10);
    $monitor("%d %d %t %b %b",$time,$stime,$realtime,in1,o1);
in1 = 0;
    #10 in1 = 1;
    #10 $finish;
end
endmodule
```

<table>
<thead>
<tr>
<th>time</th>
<th>stime</th>
<th>realtime</th>
<th>in1</th>
<th>o1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0.00 ns</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>9.53 ns</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10.00 ns</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>19.53 ns</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
The simulator rounds simple # delays to conform to the stated precision.

For example, with `timescale 1ns/100ps`, the simulator rounds 9.44 down to 9.4 and rounds 9.45 up to 9.5.
Displaying Signal Values

Use $display, $write, or $strobe to display signal values:

- **$display** immediately displays the values of its arguments
  
  ```
  $display([format_string,] argument_list);
  ```

- **$write** acts identically to $display, but does not output a newline character

- **$strobe** acts identically to $display, but waits until the end of the timeslice

  ```
  $display($time, " %b \t %o \t %d \t %h", sig1, sig2, sig3, sig4);
  $strobe ($time, " %b \t %o \t %d \t %h", sig1, sig2, sig3, sig4);
  $write ($time, " %b \t %o \t %d \t %h\n", sig1, sig2, sig3, sig4);
  ```
Displaying Signal Values

The simulator executes the `$display` and `$write` system tasks when it encounters them while processing block statements. These statements will display whatever value a signal has when the statement executes.

The simulator schedules the `$strobe` display to occur at the end of the current simulation time instant. This statement will display the final state of a signal for that simulation time instant.
# Displaying Signal Values

The `$display`, `$write`, or `$strobe` system tasks support multiple default radixes:

<table>
<thead>
<tr>
<th>Radix</th>
<th><code>$display</code></th>
<th><code>$write</code></th>
<th><code>$strobe</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>decimal</td>
<td><code>$display</code></td>
<td><code>$write</code></td>
<td><code>$strobe</code></td>
</tr>
<tr>
<td>binary</td>
<td><code>$displayb</code></td>
<td><code>$writeb</code></td>
<td><code>$strobeh</code></td>
</tr>
<tr>
<td>octal</td>
<td><code>$displayo</code></td>
<td><code>$writeo</code></td>
<td><code>$strobo</code></td>
</tr>
<tr>
<td>hexadecimal</td>
<td><code>$displayh</code></td>
<td><code>$writeh</code></td>
<td><code>$strobeh</code></td>
</tr>
</tbody>
</table>
Displaying Signal Values (continued)
Displaying Signal Values

You can use the following formatters and escaped literals in a format string. The format string overrides the default radix.

Format Specifiers:

<table>
<thead>
<tr>
<th>%b</th>
<th>%c</th>
<th>%d</th>
<th>%h</th>
<th>%m</th>
<th>%o</th>
<th>%s</th>
<th>%t</th>
<th>%v</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>ASCII</td>
<td>decimal</td>
<td>hex</td>
<td>module</td>
<td>octal</td>
<td>string</td>
<td>time</td>
<td>strength</td>
</tr>
</tbody>
</table>

Escaped Literals:

<table>
<thead>
<tr>
<th>&quot;</th>
<th>&lt;1-3 digit octal number&gt;</th>
<th>\ \ \</th>
<th>\n</th>
<th>\t</th>
</tr>
</thead>
<tbody>
<tr>
<td>double quote</td>
<td>ASCII representation of number</td>
<td>backslash</td>
<td>newline</td>
<td>tab</td>
</tr>
</tbody>
</table>

You can also use a format size specifier of 0 (the only value allowed) to indicate you want a value with no leading blanks (%0d, %0s) or leading zeros (%0h, %0b, %0o).
Displaying Signal Values (continued)

The simulator replaces the \%m formatter with the definition name of the current module. The simulator displays \%v formatted signal values with their Verilog signal strength.
Displaying Signal Values

Test your understanding of $display, $strobe, and $write:

module textio;
    reg flag;
    reg [31:0] data;
    initial begin
        $writeb("%d", $time,, "%h \t", data,, flag, "\n");
        #15 flag = 1; data = 16;
        $displayh($time,, data,, flag);
    end
    initial begin
        #10 data = 20;
        $strobe($time,, data);
        $display($time,, data);
        $display($time,, data);
        data = 30;
    end
endmodule

In what radix does each output statement display $time?

What values do $strobe and $display output?
Displaying Signal Values (continued)

This is the output of the example:

```vhdl
$writeb("%d", $time,"%h \t",data,, flag, "\n");
    0 xxxxxxxxx   x
```

The simulator requires 20 decimal digits to display 64-bit integer time. You can use the "%0d" formatter to suppress display of the leading blank characters.

The simulator displays an extra blank character for each null argument (double comma).

The simulator requires 8 hexadecimal digits to display the 32-bit `data` variable.

```vhdl
$display($time,, data);
    10   20
```

The simulator requires 10 decimal digits to display the 32-bit `data` variable.

```vhdl
$strobe($time,, data);
    10   30
```

The simulator displays the last value of the `data` variable for the simulation time instant.

```vhdl
$displayh($time,, data,, flag);
000000000000000f 00000010 1
```

The simulator requires 16 hexadecimal digits to display 64-bit integer time. You can use the "%0h" formatter to suppress display of the leading zeroes.
Monitoring Signal Values

Use `$monitor` to monitor changes of signal values:

```
$monitor([format_string,] argument_list)
```

- `$monitor` continuously monitors the variables of the argument list
- Whenever any of the variable arguments change within a timeslice, `$monitor` will display the values of the argument list just prior to the advance of simulation time (similar to the way `$strobe` works)
- A subsequent `$monitor` call overrides the previous call to `$monitor`
- `$monitoroff` temporarily suspends signal change monitoring
- `$monitoron` resumes signal change monitoring
- `$monitor` supports multiple bases. The default base is decimal
  ```
  $monitor ($time, "%b \t %h \t %d \t %o", sig1, sig2, sig3, sig4);
  ```
Monitoring Signal Values

The $monitor system task continually monitors its signal arguments, and displays the formatted arguments at the end of each simulation time instant in which any of the signal arguments changes value. The change of simulation time does not trigger a display.

You can have only one $monitor system task active. Any subsequent invocation of $monitor starts monitoring the new signal arguments and discontinues monitoring the previous signal arguments.

You can use the $monitoroff and $monitoron system tasks to restrict monitoring to intervals of time, rather than monitoring the entire remainder of the simulation session.

The $monitor system task accepts the same formatters and argument list as $display.

The $monitor system tasks support multiple default radixes. The default is decimal. Use $monitorb, $monitorh, and $monitoro for binary, hexadecimal, and octal (respectively).
File Output

You can open and write to a file.

Examine this code fragment:

```vhdl
integer MCD1;
initial begin
  MCD1 = $fopen("file.log");
  $fdisplay(MCD1, P1, P2, P3);
  $fmonitor(MCD1, P1, P2, P3);
  $fstrobe (MCD1, P1, P2, P3);
  $fwrite  (MCD1, P1, P2, P3, "\n");
  #10
  $fclose  (MCD1);
end
```

The multichannel descriptor (MCD) is a 32-bit register data type.

Each bit of the MCD represents one open channel.

Use `$fopen` to open a file and return an MCD — one bit set if successful.

Preface any display system task with `$f` to direct its output to the file channels set in the MCD argument (first argument).
File Output

A multichannel descriptor (MCD) is a 32-bit register data type, where each bit of the register represents a single output channel. The $fopen system task returns an MCD with one bit set if it successfully opened a channel, and none set if it was unsuccessful.

The simulator reserves channel 0 (MCD=1) for the standard output and log file.

The $fdisplay, $fmonitor, $fstrobe, and $fwrite system tasks write to the channels corresponding to the bits that are set in their MCD argument (the first argument). That means that you can write to more than one channel simultaneously, by providing an MCD that is the bit-wise OR of the MCDs returned by the $fopen system tasks.

The $fclose system task closes the channels specified in the MCD.
This example simultaneously writes to multiple files.

```verilog
integer messages, broadcast, cpu_chann, alu_chann;

initial begin
    cpu_chann = $fopen("cpu.log"); if(!cpu_chann) $finish;
    alu_chann = $fopen("alu.log"); if(!alu_chann) $finish;
    messages = cpu_chann | alu_chann; // cpu.log and alu.log
    broadcast = 1 | messages; // stdout, logfile, cpu.log, alu.log
end

always @(posedge clock) // print to alu.log
    $fdisplay(alu_chann, "acc= %h f=%h a=%h b=%h", acc, f, a, b);

always @(negedge reset) // stdout, logfile, cpu.log, alu.log
    $fdisplay(broadcast, "system reset at time %0d", $time);
```
File Output (continued)
File Input

Use `$readmemb` or `$readmemh` to load a `reg` array (memory) with file data.

- **$readmemb** — The file contains binary ASCII data
  
  ```
  $readmemb ("filename", memory_name, [start_addr, [finish_addr]]);
  ```

- **$readmemh** — The file contains hexadecimal ASCII data
  
  ```
  $readmemh ("filename", memory_name, [start_addr, [finish_addr]]);
  ```

You can default the addresses, or provide them in the system task or in the data file (or both). The system task by default writes from the left memory address (as declared) toward the right memory address. Addresses (if any) in the data file must reside with the range (if any) specified in the system task.

**Note:** The Cadence Verilog-XL and NC-Verilog simulators by default write from the lower memory address (as declared) toward the higher memory address. This is noncompliant with the IEEE Std. 1364-1995.
File Input
File Input

This example illustrates the loading of a Verilog memory.

/* Data File */

@0   // address always hex
0000_0000
0110_0001 0011_0010

// addresses 3-255 undefined

@100
1111_1100

// addresses 257-1022 undefined

@3FF
1110_0010
File Input (continued)

In the memory data file:

- You can include both single-line and multi-line comments
- You can assign a starting address for subsequent data:
  
  \texttt{@hex\_address}

  — The hexadecimal address must immediately follow the \texttt{@} character (no spaces)
  — The hexadecimal address is not case sensitive

- You can specify binary ($readmemb$) or hexadecimal ($readmemh$) ASCII data
- You can specify 0, 1, Z, and X data values
- You can use underscores (_ _) to increase readability
- You can separate memory words by spaces and lines
Summary

In this section you learned about:

- Accessing simulation time
- Displaying signal values
- Monitoring signal values
Summary
1. What system function returns 64-bit simulation time?

2. What system task displays the values of the argument list once at the end of the current timeslice?

3. True or false: Subsequent invocations of the `$monitor` system task add to the list of signals to be monitored.

4. What is the maximum number of channels you can simultaneously write to?

5. In what (IEEE standard) order does the `$readmemb` system task load memory if you do not specify addresses?
**Review**

1. What system function returns 64-bit simulation time?
   
   The `$time` system function returns simulation time scaled and rounded to the calling module timescale, as a 64-bit double-precision unsigned integer `time` data type.

2. What system task displays the values of the argument list once at the *end* of the current timeslice?

   The `$strobe` system task displays the values of the argument list once at the *end* of the current timeslice. This system task waits until just before the advance of simulation time to output signal values.

3. True or false: Subsequent invocations of the `$monitor` system task add to the list of signals to be monitored.

   A subsequent `$monitor` call overrides the previous call to `$monitor`.

4. What is the maximum number of channels you can simultaneously write to?

   The simulator automatically opens channel 0 to the standard output and log file. You can open up to 31 additional channels, and can simultaneously write to any combination of the 32 channels.

5. In what (IEEE standard) order does the `$readmemb` system task load memory if you do not specify addresses?

   The system task by default writes from the left memory address (as declared) toward the right memory address.
Chapter 15: Introduction to Verilog Testbenches

Objectives
In this section, you will learn about designing a testbench:

- Creating clocks
- Including files
- Strategic use of tasks and concurrent statements
- Controlling and observing the design
- Reporting warnings and errors
Introduction to Verilog Testbenches

This section offers a more detailed investigation of testbench design: how to create clocks, include other source files, and make effective use of Verilog tasks and concurrent statement blocks. The section then goes on to present several ways to apply stimulus, and ends with a discussion of warning and error reports.
The Simulation Environment

This is a simplified picture of the overall simulation environment.

This section concentrates on testbench development strategies.
The Simulation Environment
Creating Clocks

Example 1

You can define the clock in either the design or its testbench.

You can define the clock either behaviorally or structurally.

Here are examples of a symmetric clock:

![Clock Diagram]

### Behavioral

```verilog
reg clk;
always begin
  #10 clk = 1;
  #10 clk = 0;
end
```

### Structural

```verilog
reg start;
nand #10 (clk, clk, start);
initial begin
  start = 0;
  #10 start = 1;
end
```
Creating Clocks

Example 1

The behavioral version enters an **always** loop, where it initializes the clock to 1 at time 10 and thereafter toggles its value every 10 time units.

The structural version deasserts the *start* signal at time 0 and reasserts it at time 10. As the clock has a 10 time unit delay, this initializes the clock to 1 at time 10, and the clock toggles its value thereafter every 10 time units.

The Cadence Verilog-XL simulator accelerates primitive simulation with the XL engine and accelerates behavioral simulation with the Turbo engines. As there is significant overhead in propagating events between these two engines, you would want to use a behavioral clock with a primarily behavioral design and a structural clock with a primarily structural design. This is not an issue with the Cadence NC-Verilog simulator, which has no XL engine.
Creating Clocks

Example 2

Here are examples of a symmetric clock with delayed startup:

![Clock Waveform]

**Behavioral**

```verilog
reg clk;
initial begin
    #20 clk = 1;
    forever begin
        #10 clk = 0;
        #10 clk = 1;
    end
end
```

**Structural**

```verilog
reg start;
nand #10 (clk, clk, start);
initial begin
    #10 start = 0;
    #10 start = 1;
end
```
Example 2

The behavioral version initializes the clock to 1 at time 20 and thereafter toggles its value in a \texttt{forever} loop every 10 time units.

The structural version deasserts the \textit{start} signal at time 10 and reasserts it at time 20. As the clock has a 10 time unit delay, this initializes the clock to 1 at time 20, and the clock toggles its value thereafter every 10 time units.
Creating Clocks

Example 3

Here are examples of an asymmetric clock with delayed startup:

![Clock Waveform Diagram]

**Behavioral**

```verilog
reg clk;
initial begin
    #20 clk = 1;
    forever begin
        #5  clk = 0;
        #15 clk = 1;
    end
end
```

**Structural**

```verilog
reg start;
nand #(15,5) (clk, clk, start);
initial begin
    #5  start = 0;
    #15 start = 1;
end
```
Creating Clocks (continued)

Example 3

The behavioral version initializes the clock to 1 at time 20 and thereafter toggles its value in a `forever` loop with a 5/15 duty cycle.

The structural version deasserts the `start` signal at time 5 and reasserts it at time 20. As the clock has a 15 time unit rising delay and a 5 time unit falling delay, this initializes the clock to 1 at time 20, and the clock toggles its value thereafter with a 5/15 duty cycle.
Designing Your Testbench

You can make your testbench as simple or as complex as you want.

A complex testbench would perform response verification "on-the-fly".
Designing Your Testbench

A complex testbench might do some of the following:

- Make strategic use of included source files
- Encapsulate repetitive code in tasks
- Include self-checking mechanisms
- Incorporate configurable debugging tasks
- Perform progressively more detailed tests
Using Include Files

Use `include` files to ensure project-wide consistency of common source.

```
// defines.inc
`timescale 1 ns / 10 ps
`define PERIOD 20
`define TIMEOUT 10000000

`include "defines.inc"
module clkgen (clk);
output clk; reg clk;
    always begin
        #(`PERIOD/2) clk = 0;
        #(`PERIOD/2) clk = 1;
    end
    initial begin
        #(`TIMEOUT)
        $display("TIMEOUT ERROR");
        $finish;
    end
endmodule
```
Using Include Files

This example defines project-wide constants in a header file included at the top of every design file in the project. This enforces project-wide consistency. You could also potentially use include files to define such things as port maps (if several project teams are concurrently working on different flavors of the same ASIC).
Using Verilog Tasks

Use Verilog tasks in your testbench to encapsulate repeated operations.

```verilog
task cpu_read;
begin
    #30 data_valid = 1;
    wait (data_read == 1);
    #20 cpu_data = data_in;
    wait (data_read == 0);
    #20 cpu_data = 8'hzz;
    #30 data_valid = 0;
end
endtask
```
Using Verilog Tasks
Using Concurrent Statements

Use **fork-join** blocks in your testbench to concurrently activate parallel tasks.

```verilog
module inline_tb;
    // declare variables
    // instantiate designs
    // define tasks
    initial begin
        initialize_design;
        fork
            monitor_data;
            monitor_error;
            monitor_timeout;
            run_test;
        join
    end
endmodule
```
Using Concurrent Statements

The example concurrently starts four tasks. The monitor tasks presumably run throughout the simulation and never return, so you could not start them from within a sequential `begin-end` block. Another alternative would be to place the monitor activity in an `always` process. This is a less satisfactory solution, as you could then not disable and restart the monitor processes during the simulation session.

Here is another example. This example periodically starts the test task for two devices. It starts the first device at regular intervals, but upon each iteration starts the second device later and later within the interval. You can use this sort of loop to sweep the phases of the bus transactions of two devices across each other, to verify that no phase interferes with any other phase on a split-transaction bus.

```vhdl
task run_test;
    begin
        for (i=latency1; i<=latency2+2*latency1; i=i+1)
            fork
                #( `PERIOD*(latency1+latency2) ) test_device1;
                #( `PERIOD*(i) ) test_device2;
            join
        end
    endtask
```
Applying Stimulus

Some common stimulus application techniques include:

- In-line stimulus, applied from an **initial** block
- Stimulus applied from a loop or **always** block
- Stimulus applied from an array of vectors or integers
- Stimulus that is recorded during one simulation and played back in another simulation
Applying Stimulus
In-Line Stimulus

In-line stimulus has the following characteristics:

- You list the variables only when their values change
- You can easily define complex timing relationships between signals
- The testbench can become very long for tests of real designs

```verilog
module inline_tb;
    wire [7:0] results;
    reg [7:0] data_bus, addr;
    DUT u1 (results, data_bus, addr);
    initial fork
        #10 addr = 8'h01;
        #10 data_bus = 8'h23;
        #20 data_bus = 8'h45;
        #30 addr = 8'h67;
        #30 data_bus = 8'h89;
        #40 data_bus = 8'hAB;
        #45 $finish;
    join
    endmodule
```
Applying Stimulus (continued)

In-Line Stimulus
Stimulus From Loops

Stimulus applied from a loop has the following characteristics:

- For each iteration you assign a new stimulus vector
- The timing relationships between signals are regular in nature
- The testbench is compact

module loop_tb;
  wire [7:0] response;
  reg [7:0] stimulus;
  reg clk;
  integer i;
DUT u1 (response, stimulus);
initial clk = 0;
always begin
  #10 clk = 1;
  #10 clk = 0;
end
initial begin
  for (i = 0; i <= 255; i = i + 1)
    @(negedge clk) stimulus = i;
  #20 $finish;
end
endmodule
Applying Stimulus (continued)

Stimulus From Loops
Stimulus From Arrays

Stimulus applied from an array has the following characteristics:

- You can load the stimulus from a data file directly into the array
- For each iteration you read a new stimulus vector from the array

module array_tb;
    wire [7:0] response;
    reg [7:0] stimulus, stim_array[0:15];
    integer i;
    DUT u1 (response, stimulus);
    initial begin
        $readmemb("datafile",stim_array);
        for (i = 0; i <= 15; i = i + 1)
            #20 stimulus = stim_array[i];
            #20 $finish;
    end
endmodule
Applying Stimulus (continued)

Stimulus From Arrays
Vector Capture and Playback

You can capture manufacturing test vectors at the boundary of a device model.

```verilog
parameter period = 20
wire [7:0] response;
reg [7:0] stimulus;
DUT u1 (response, stimulus);

always apply (stimulus);
always verify (response);

task capture_tb;
integer MCDR, MCDS;
begin
    MCDR = $fopen("response.dat"); if (!MCDR) $finish;
    MCDS = $fopen("stimulus.dat"); if (!MCDS) $finish;
    forever @(posedge clk) #(period-1) begin
        $fdisplayb (MCDR, "%b", response);
        $fdisplayb (MCDS, "%b", stimulus);
    end
end
endtask
```
Applying Stimulus (continued)

Vector Capture and Playback

The vector capture task writes the stimulus and response vectors one time unit before each active edge of the clock. As the task misses the first clock edge, you should invoke the task one clock before you want to start capturing the vectors. This is a simplistic illustration that you will need to modify for your situation, for example if your device has negative setup or hold requirements that the testbench takes advantage of.
Vector Capture and Playback

You can play back the saved stimulus and response vectors.

```verilog
parameter MAX_VECTOR=255;
wire [7:0] response;
reg [7:0] stimulus, stim_array [0:255], resp_array [0:255];
DUT u1 (response, stimulus);

task playback_tb;
integer MCDR, MCDS, i;
begin
    $readmemb("response.dat", resp_array);
    $readmemb("stimulus.dat", stim_array);
    @(negedge clk) // synchronize to inactive clock
    stimulus = stim_array[0]; // apply 1st stimulus
    for (i=0; i&lt;=MAX_VECTOR; i=i+1) @(negedge clk) begin
        if (response != resp_array[i]) begin // check response
            $display("ERROR: response is %b, should be %b", response, resp_array[i], "\nTEST FAILED");
            $finish; end
        if (i==MAX_VECTOR) begin $display("TEST PASSED"); $finish; end
        stimulus = stim_array[i+1]; // apply next stimulus
    end
end
dontask
```
Vector Capture and Playback (continued)

The playback task loads the stimulus and response memory, synchronizes to the inactive edge of the clock, and enters a loop where at each inactive edge of the clock it verifies the response and provides new stimulus.
Forcing Stimulus

You can make two types of procedural continuous assignments:

- You can **assign** and **deassign** a register
  
  assign lvalue = expression

  This overrides any procedural assignment to the register

  initial begin
      #10 assign top.dut.fsml.state_reg = `init_state ;
      #20 deassign top.dut.fsml.state_reg ;
  end

- You can **force** and **release** a register or net

  This overrides all drivers of the signal

  initial begin
      #10 force top.dut.counter.scan_reg.q = 0 ;
      #20 release top.dut.counter.scan_reg.q ;
  end
Applying Stimulus (continued)

Forcing Stimulus

These examples assign a constant value to a register and net (respectively), overriding any procedural assignments or drivers between simulation time 10 and 20. You can also assign expressions involving variables, and the simulator updates the value of the register or net whenever a transition of the variable changes the expression value.

Use the **assign** keyword in a procedural block to make a continuous procedural assignment to a whole variable of a register data type. This continuous procedural assignment overrides any normal procedural assignments. A new **assign** also overrides a previous **assign**. Use the **deassign** keyword to remove the continuous procedural assignment. Use this construct to model asynchronous activity of a purely behavioral design. Synthesis tools do not accept continuous procedural assignments.

Use the **force** keyword in a procedural block to make a continuous procedural assignment to a whole variable of a register or net data type, or a bit or part select of an expandable vector net. This continuous procedural assignment overrides all drivers of a net. A new **force** also overrides a previous **force**. Use the **release** keyword to remove the continuous procedural assignment. Upon **release**, the variable returns to its **assign** value (if any). Use this construct in a testbench to temporarily patch a design. Synthesis tools do not accept continuous procedural assignments.
Reporting Warnings and Errors

Use file output system tasks to report errors and warnings.

A more sophisticated testbench would:

- Report an error to a centralized error handler
- Modify the test flow, depending upon the errors encountered
- Maintain error statistics, and report them at the end of the test

```verilog
task par_err_task;
    forever @(posedge par_err)
        err_handle_task (`NONFATAL, `PARITY);
endtask

task cor_err_task;
    forever @(posedge cor_err)
        err_handle_task (`NONFATAL, `CORRECTABLE);
endtask
```
Reporting Warnings and Errors
In this section, you learned about designing a testbench:

- Creating clocks
- Including files
- Strategic use of tasks and concurrent statements
- Controlling and observing the design
- Reporting warnings and errors
Summary
Review

1. What are some characteristics of a sophisticated testbench?
2. For what purpose might you define a Verilog task in your testbench?
3. What is the difference between a begin-end block and a fork-join block?
4. How can you efficiently generate regular stimulus?
5. What data type would you use to read stimulus from a file?
Review

1. What are some characteristics of a sophisticated testbench?
   A sophisticated testbench would make strategic use of included source files, encapsulate repetitive code in tasks, include self-checking mechanisms, incorporate configurable debugging tasks, and perform progressively more detailed tests.

2. For what purpose might you define a Verilog task in your testbench?
   You might define a Verilog task to encapsulate a process that you use repeatedly in several places in your testbench.

3. What is the difference between a begin-end block and a fork-join block?
   The simulator executes the statements in a begin-end block sequentially, finishing the previous one before starting the next one. The simulator executes the statements in a fork-join block concurrently, working on multiple statements at the same time.

4. How can you efficiently generate regular stimulus?
   You can efficiently generate regular stimulus in a forever loop or always block. For irregular stimulus you might use a large initial block or read the stimulus from a file.

5. What data type would you use to read stimulus from a file?
   Use arrays of the reg type (memories) with the $readmemb or $readmemh system tasks to read stimulus from a file.
About Lab 8

This lab is in two parts.

The objective of the first part is to use behavioral constructs to model a small combinational Arithmetic Logic Unit (ALU).

In this part, you will:
- Model an ALU at the behavioral level
- Test the ALU with the provided testbench

The objective of the second part is to create a model of a 5-bit counter using Verilog behavioral constructs.

In this part, you will:
- Model a 5-bit counter at the behavioral level
- Write a testbench for your counter design
About Lab 8
Objectives

In this section, you will learn about:

- Modeling a memory device
- Initializing a memory device
- Modeling bidirectional ports
Modeling Memories

This section describes the modeling and initialization of RAM and ROM. It contains a discussion of the correct use of bidirectional module ports.
Modeling a Memory Device

A memory model must do two things:

- Declare an appropriately-sized array of Verilog `reg`
  - You can parameterize the word and memory size
- Provide access to the memory contents
  - Read only
  - Read and write
  - Simultaneous read and write
Modeling a Memory Device
This simple ROM model initializes itself with data from a file.

```verilog
module rom16x4 (data, addr, read);
output [3:0] data;
input [3:0] addr;
input  read;
reg [3:0] data;
    reg [3:0] mem [0:15];
initial
    $readmemb ("rom_data", mem);
always @(addr or read)
    data = read ? mem[addr] : 4'bz;
endmodule
```
Modeling a Memory Device (continued)

Modeling a Simple ROM

The ROM model declares a two-dimensional array of Verilog `reg`. The model initializes the memory with data from a file.
This simple RAM model provides read and write access.

```verilog
module ram16x4 (data, addr, read, write);
    inout [3:0] data;
    input [3:0] addr;
    input read, write;
    reg [3:0] mem [0:15]; // 16x4
    //read
    assign data = read ? mem[addr] : 4'bz;
    //write
    always @(posedge write)
        mem[addr] = data;
endmodule
```
Modeling a Memory Device (continued)

Modeling a Simple RAM

RAM models are slightly more complicated than ROM models. RAM models must be capable of both read and write operations, usually using the same bidirectional data port. You will need to include circuitry to control the direction of the port.

This model drives a high-impedance (Z) value onto the data port when the read control is deasserted. This permits some external circuit to drive the data port during a write operation.

This model is synthesizable, but most tools will simply produce an array of flip-flop cells, which typically requires more silicon and routing than a true memory.
Parameterizing the Memory Size

You can parameterize the memory width and depth.

```verilog
module scale_RAM (data, addr, read, write);
parameter addrsize = 8; // size of address bus
parameter wordsize = 8; // width of a word
parameter words = (1 << addrsize); // size of mem
inout [wordsize:1] data; // word of memory
input [addrsize:1] addr; // address bus
input read, write;

reg [wordsize:1] mem [0 : words-1];
assign data = read ? mem[addr] : 'bz;
always @(posedge write)
    mem[addr] = data;
endmodule
```
Parameterizing the Memory Size

The example illustrates how you can easily define a memory device that is parameterizable by its word size and number of address bits. It uses the left shift operator to calculate the memory size from the number of address bits. As the address input directly addresses the memory, the memory range must start at 0. You could alternatively express the memory declaration and addressing as:

```
reg [wordsize:1] mem [1 : words]; // memory starts at word 1
wire [wordsize:1] mem_word = mem[address + 1]; // increment the address
```

The IEEE Std. 1364-1995 requires the simulator to expand unsized literals (such as 'bz) to 32 bits. If you intend to scale the word size beyond 32 bits, you could work around this by sizing the literal to the maximum word size you will ever use. The simulator will simply truncate the higher-order unneeded bits of the literal.
Initializing a Memory Device

You can use a loop or a system task to initialize memory:

- **Use a `for` loop**

  ```vHDL
  // initialize memory
  for (i=0; i < memsize; i = i+1)
    mema[i] = {wordsize{1'b1}};
  ```

- **Invoke the `$readmemb` or `$readmemh` system tasks**

  ```vHDL
  // load memory data from a file
  $readmemb("mem_file.txt", mema);
  ```
Initializing a Memory Device

You can use the `$readmemb` or `$readmemh` system tasks to load data from a file into a Verilog memory. For a ROM, this would be the actual memory content. For a RAM, this permits you to skip some of the write cycles you would otherwise need.

Note the parameterized word width and replication operator in the first example.
Modeling Bidirectional Ports

Use the **inout** keyword to declare a bidirectional port:

```vhdl
inout [7:0] databus;
```

- A bidirectional port is always a net, never a register
  - The simulator can resolve the value of multiple drivers of a net
- Your should drive a bidirectional port from only one direction at a time
  - You cannot place read and write data on the bus simultaneously
  - You must design logic around the port to ensure proper operation
Modeling Bidirectional Ports

You declare an **inout** port to enable driving data from either direction. The port data type defaults to the net data type. You cannot make a procedural assignment to a net, but you can continuously assign a register data type to it outside of a procedural block, or connect it to a primitive.

You must design logic around the **inout** port to ensure proper operation. When using the port as an input, you must disable the output logic.
Using Primitives

When $en_a_b = 1$, primitive $b1$ drives the $bus_a$ value onto $bus_b$

When $en_b_a = 1$, primitive $b2$ drives the $bus_b$ value onto $bus_a$

```verilog
module bus_xcvr(bus_a, bus_b, en_a_b, en_b_a);
inout bus_a, bus_b;
input en_a_b, en_b_a;
  bufif1 b1 (bus_b, bus_a, en_a_b);
  bufif1 b2 (bus_a, bus_b, en_b_a);
endmodule
```
Modeling Bidirectional Ports (continued)

Using Primitives

The \textit{en\_a\_b} and \textit{en\_b\_a} inputs control the enabling and disabling of the \texttt{bufif1} primitives.

The simulator resolves the result of multiple drivers driving the net.

If you assert both controls simultaneously, you will get unpredictable results.

An alternative design that avoids this problem is:

\begin{verbatim}
module bus_xcvr(bus_a, bus_b, direction);
inout bus_a, bus_b;
input direction;
    bufif1(bus_b, bus_a, direction); // bus_b gets bus_a
    bufif0(bus_a, bus_b, direction); // bus_a gets bus_b
endmodule
\end{verbatim}
Using Continuous Assignments

When $en_a_b = 1$, assignment $b1$ drives the $bus_a$ value onto $bus_b$

When $en_b_a = 1$, assignment $b2$ drives the $bus_b$ value onto $bus_a$

```
module bus_xcvr(bus_a, bus_b, en_a_b, en_b_a);
  inout bus_a, bus_b;
  input en_a_b, en_b_a;
  assign bus_b = en_a_b ? bus_a : 1'bz;
  assign bus_a = en_b_a ? bus_b : 1'bz;
endmodule
```
Modeling Bidirectional Ports (continued)

Using Continuous Assignments

The \texttt{en\_a\_b} and \texttt{en\_b\_a} inputs select the values driven by the continuous assignments.

The simulator resolves the result of multiple drivers driving the net.

If you assert both controls simultaneously, you will get unpredictable results.

An alternative design that avoids this problem is:

```verilog
module bus_xcvr(bus_a, bus_b, direction);
inout bus_a, bus_b;
input direction;
    assign bus_b = direction ? bus_a : 'bz;
    assign bus_a = direction ? 'bz : bus_b;
endmodule
```
When \textit{read} = 1 the assignment drives the \textit{datareg} value onto \textit{databus}.

The falling edge of \textit{write} stores the \textit{databus} value into \textit{datareg}.

\begin{verbatim}
module ram_cell(databus, read, write);
 inout databus;
 input read, write;
 reg datareg;
 assign databus = read ? datareg : 1'bz;
 always @(negedge write)
 datareg <= databus;
endmodule
\end{verbatim}
Example Bidirectional Port

This memory cell stores data on the falling edge of *write*. The module using this memory cell can drive write data onto the data bus while *write* is high. You must ensure that the write data persists long enough for the write to take place.

The memory cell drives read data onto the databus whenever *read* is high. As this is a single-port memory model, the results of asserting *write* and *read* at the same time are unpredictable.
Summary

In this section, you learned about:

- Modeling a memory device
- Initializing a memory device
- Modeling bidirectional ports
Summary
Review

1. With what Verilog data type do you declare a memory array?
2. With what syntax do you declare a memory array?
3. How can you initialize a memory with data?
4. How can you send values through a bidirectional (inout) port?
Review

1. With what Verilog data type do you declare a memory array?
   Declare a memory as a 2-dimensional array of Verilog `reg`.

2. With what syntax do you declare a memory array?
   Declare a memory as a 2-dimensional array of Verilog `reg`.

3. How can you initialize a memory with data?
   You can load a memory model with data using the `$readmemb` or the `$readmemh` system tasks, or by using procedural assignments (most likely in a `for` loop).

4. How can you send values through a bidirectional (`inout`) port?
   Because a bidirectional (`inout`) port must be a net data type, you can only drive values onto it with a primitive or continuous assignment. You must also ensure that the module and its environment do not both attempt to simultaneously drive the port.
About Lab 9

The objective of this lab is to create and test a memory model.

In this lab, you will:

- Create a simulation model for a memory device having asynchronous access and a bidirectional data bus
- Modify the provided testbench for the model, which does not correctly handle the data bus
About Lab 9
Chapter 17: High Level Verilog Constructs

Objectives

In this section, you will learn about:

- Verilog functions and tasks
- Named statement blocks
- Finite State Machines (FSMs)
High Level Verilog Constructs

This section contains a more detailed discussion of Verilog functions and tasks, named blocks, and Finite State Machines (FSMs).
Verilog Functions and Tasks

Verilog functions and tasks encapsulate repeated operations:

- **A function:**
  - Accepts `input` arguments, but not `inout` or `output` arguments
  - Returns a single value through the function name
  - Contains no timing controls, thus executes in zero simulation time
  - Can invoke other functions, but not tasks
  - Represents combinational logic

- **A task:**
  - Accepts `input`, `inout` and `output` arguments
  - Updates output arguments upon task return
  - Can contain timing controls (`#`, `@`, `wait`)
  - Can invoke other functions and tasks
  - Represents design behavior and testbench operation
Verilog Functions and Tasks

You define a Verilog function or task in a module. You can declare new temporary register data type local variables, such as integer, reg, or time, but you cannot declare a new design object, such as a net.

You can invoke a function from outside a procedural block, but can only invoke a task from inside a procedural block or another task.

Function and task inputs and outputs are variables local to the function or task definition. The simulator transfers values to and from these variables only when entering or returning from the function or task.
Verilog Functions

These are the key features of a Verilog function:

- You define a Verilog function within a `module` definition.

- Define a Verilog function with the `function` keyword, but without a port list. A function by default returns a scalar `reg`, but you can size or type the function definition.

- Declare the function inputs. A function must have at least one `input` and may not have an `output` or `inout`.

- Functions constitute a new level of scope. You can declare new local register data type variables. You cannot instantiate design components.

- A function can contain only one statement. Group multiple statements with the `begin-end` keywords.

- A function definition cannot contain any timing controls. Thus a function cannot invoke a task, which can contain timing controls.

- A function returns only one value, through its name. However, you can concatenate data and return it through an appropriately-sized function.

- Pass arguments to the function in the order in which you declare its inputs.
Verilog Functions and Tasks (continued)

Verilog Functions

When you invoke a Verilog function, pass the function arguments in the same order that you declare the function inputs. The simulator passes the input argument values to the function input ports upon function entry. The simulator passes the output value back upon function return. The function arguments and inputs are otherwise independent of each other, though you will probably want to name them differently to avoid confusion.

The function inputs and name are register variables local to the function definition. You can read and write them at will without affecting the function caller until the function returns. You can also declare other local register data type variables.

You can access any Verilog simulation object from anywhere in the design, thus functions can read and write signals outside the function definition. This type of access is called "side effects", and for functions, is considered poor programming practice, so is not discussed further here.

Only one instance of a function exists, thus all invocations of the function execute the same code and read and write the same input, name, and local variables. Verilog functions are thus nonrecursive. Although you can code a function to call itself, you will most likely not obtain the results you desire.
Functions cannot contain timing controls. Place the timing controls in the function invocation.

This example invokes the function from a procedural assignment.

```verilog
module aoi221_r (y,a,b,c,d,e,clk);
output y; reg y;
input a,b,c,d,e;
input clk;
    function aoi221_f;
    input a,b,c,d,e;
    if (e == 1)
        aoi221_f = 0;
    else
        aoi221_f = ~(a&b | c&d);
endfunction
always @(posedge clk)
    #3 y = aoi221_f (a,b,c,d,e);
endmodule
```
Verilog Functions (continued)

This function represents the familiar AND-OR-INVERT combinational logic.
Verilog Functions

A function can return a vector. Simply declare a range for the function.

This example invokes the function from a continuous assignment.

module count0 (count, ivector);
output [3:0] count;
input [7:0] ivector;

function [3:0] count0_f;
input [7:0] fvector;

integer i;
begin
  count0_f = 0;
  for (i=0; i<=7; i=i+1)
    if (fvector[i]==0)
      count0_f=count0_f+1;
end
endfunction
assign count=count0_f(ivector);
endmodule
Verilog Functions (continued)

This example function:

- Returns a vector. To return a vector, declare the function range when you define the function.

- Declares a local temporary variable. You can declare local function variables because a function definition is a new level of scope. You can declare only parameters and register data type variables in a function definition. You cannot declare nets.

- Groups multiple statements between **begin-end**. A function may contain only one statement. Group multiple statements to form a single compound statement.

- Makes multiple assignments to the function name. The function name represents a register data type inside the function. You can read and write the function name multiple times from within the function definition. The simulator passes the function value back to the invoking statement only when the function returns.
A function can also return **integer** or **real** data types. This function returns an integer.

This example invokes the function from a conditional expression.

```verilog
module checksub (neg, ina, inb);
output neg; reg neg;
input [7:0] ina, inb;

function integer subtract;
input [7:0] ina_f, inb_f;
subtract = ina_f - inb_f;
endfunction
always @ (ina or inb)
  if (subtract(ina, inb) < 0)
    neg = 1;
  else
    neg = 0;
endmodule
```
Verilog Functions (continued)

A function invoked from a procedural block can also return **integer** or **real** data types. This example function returns an integer.
Verilog Functions

You can assign to individual bits of the function name variable.

This example uses a module parameter to parameterize the function size and input. You cannot parameterize the function itself on a per-invocation basis.

```verilog
module reverse (ovector, ivector);
parameter WIDTH = 8;
output [WIDTH-1:0] ovector;
input [WIDTH-1:0] ivector;

function [WIDTH-1:0] reverse_f;
input [WIDTH-1:0] vector;
    integer i;
    for (i=0; i<WIDTH; i=i+1)
        reverse_f[i] = vector [WIDTH-i-1];
endfunction

always @ivector
    ovector = reverse_f (ivector);
endmodule
```
Verilog Functions  (continued)

This example makes assignments to individual bits of the function return variable.

You can access module parameters from inside a function, but cannot redefine the module parameters on a per-invocation basis.
Verilog Tasks

These are the key features of a Verilog task:

- You define a Verilog task within a module definition.
- Define a Verilog task with the task keyword, but without a port list.
- Declare the task ports. A task can have input, output, and inout ports.
- Tasks constitute a new level of scope. You can declare new local register data type variables. You cannot instantiate design components.
- A task can contain only one statement. Group multiple statements with the begin-end or fork-join keywords.
- A task definition can contain timing controls. A task can invoke a function.
- Pass arguments to the task in the order in which you declare its ports.
- You can disable a task. This cancels all scheduled task activity.
Verilog Functions and Tasks (continued)

Verilog Tasks

When you invoke a Verilog task, pass the task arguments in the same order that you declare the task ports. The simulator passes the input argument values to the task input ports upon task entry. The simulator passes the output port values to the task output arguments upon task return. The task arguments and ports are otherwise independent of each other, though you will probably want to name them differently to avoid confusion.

The task ports are register variables local to the task definition. You can read and write them at will without affecting the task arguments until the task returns. You can also declare other local register data type variables.

You can access any Verilog simulation object from anywhere in the design, thus tasks can read and write signals outside the task definition. This type of access is called "side effects". As the simulator passes task arguments by value, a task must directly read and write signals to observe and create transitions.

A task can directly reference signals declared in the module that defines the task, without using hierarchical signal names. Be aware that if you invoke a task from a module other than that which defines it, the task accesses signals in its own module, not the signals in the calling module, even if the signals have the same name.

Only one instance of a task exists, thus all invocations of the task execute the same code and read and write the same port and local variables. As tasks can have timing controls, you can easily invoke a task again before a previous invocation has returned. You normally would not want to do this.
Verilog Tasks

The simulator passes task argument values into and out of tasks only upon entry and return (respectively). Tasks cannot control and observe signal transitions through their ports.

This task contains a timing control. It observes signal transitions, and thus needs to access those signals directly.

```verilog
module aoi221_r (y,a,b,c,d,e,clk);
output y; reg y;
input a,b,c,d,e;
input clk;

task aoi221_t;
output y_t;
@(posedge clk)
#3 if (e == 1)
   y_t = 0;
else
   y_t = ~(a&b | c&d);
endtask
always
   aoi221_t (y);
endmodule
```
Verilog Tasks (continued)

This example continually invokes the task from an **always** block. The task waits for the positive edge of the clock before reading the module inputs, thus you cannot pass the clock and module input values to the task as input arguments.
You can assign to individual bits of a task output.

This example uses a module parameter to parameterize the task port. You cannot parameterize the task itself on a per-invocation basis.

```verilog
module reverse (ovector,ivector);
parameter WIDTH = 8;
output [WIDTH-1:0] ovector;
input [WIDTH-1:0] ivector;
reg [WIDTH-1:0] ovector;

task reverse_t;
inout [WIDTH-1:0] io_vector;
reg [WIDTH-1:0] temp_vect;
integer i;
begin
    temp_vect = io_vector;
    for (i=0; i<WIDTH; i=i+1)
        io_vector[i] = temp_vect [WIDTH-i-1];
end
endtask
reg [WIDTH-1:0] vector;
always @ivector begin
    vector = ivector;
    reverse_t (vector);
    ovector = vector; end
endmodule
```
Verilog Tasks (continued)

This example makes assignments to individual bits of the task `inout` port.

You can access module parameters from inside a task, but cannot redefine the module parameters on a per-invocation basis.

This example illustrates that you can access the task ports as register variables within the task. The simulator passes the input argument values to the task input ports upon task entry. You can modify the task port variables at will within the task. The simulator passes the output port values to the task output arguments upon task return.
Verilog Tasks

Supply task arguments in the order you declare the task ports.

The following task has two input ports and one output port.

```verilog
module mult (result, ina, inb, enb, clk);
output [7:0] result; reg [7:0] result;
input [3:0] ina, inb;
input enb, clk;
    task mult_t;
    input [3:0] a_t, b_t;
    output [7:0] result_t;
    result_t = a_t * b_t;
endtask
always @(posedge clk)
    if (enb)
        mult_t (ina, inb, result);
endmodule
```
Verilog Tasks (continued)
Verilog Tasks

You can **disable** a Verilog task.

This facilitates your control of design instrumentation.

```verilog
    task errmon;
        forever @(posedge datardy)
            if (parity !== ^databus) begin
                $display("ERR: data=%b, parity=%b",
                        databus, parity);
                $display("TEST FAILED");
                $finish;
            end
    endtask

    initial begin
        initialize;
        fork
            errmon;
            begin
                runtest;
                disable errmon;
            end
        join
    end
```
Verilog Tasks (continued)

You can place autonomous background activity, such as testbench instrumentation, in a Verilog task. This permits you to invoke and then disable the task multiple times throughout your test session. When you disable a task, the simulator cancels all scheduled task events.

You cannot control the testbench instrumentation in this manner if you place it in an initial or always block.
Named Blocks

You can name a **begin-end** or **fork-join** statement block:

- Append ": block_name" after the **begin** or **fork** keyword
  ```
  begin : seq_blk
  fork : par_blk
  ```
- Named blocks constitute a new level of scope
  — You can declare new local variables
- You can **disable** a named block
  — This cancels all scheduled block activity

```verilog
module namedblk;
  initial begin
    fork: fblk
      integer i;
      i=0;
      forever #1 i=i+1;
      #3 disable fblk;
    join
      $display("Disabled fblk at %0d",$stime);
  end
endmodule
```
Named Blocks

You can name a statement block by appending a colon (:) and the block name after the `begin` or `fork` keyword. A named block is a new level of scope, so a named block can declare new variables local to the block. A named block can still access all module-level variables.

You can also disable a named block with the `disable` keyword followed by the name of the block. Upon disabling a named block, the simulator cancels all scheduled block events.
Finite State Machines

Explicit FSMs:
- Make assignments to a state variable
- Debug more easily
- Handle complex state changes
- Are synthesizable

Implicit FSMs:
- Step through timing controls
- Simulate more efficiently
- Handle linear state changes well
- Are generally not synthesizable
Finite State Machines

For all synthesizable FSMs, you must provide a reset, and make state changes synchronous to one edge of one clock.

For the explicit FSM, you define the state transitions when you assign values to a state variable. You can easily code, debug, and synthesize a complex explicit FSM.

For the implicit FSM, you define the state transitions with the use of multiple timing controls and branching and looping statements. It is more difficult to code and debug an implicit FSM, and logic synthesis tools generally provide them little or no support. You would most likely use them only in a purely behavioral design.
An explicit FSM makes assignments to a state variable.

```verilog
module fsm (out, in, clk, rst);
output out;
input in, clk, rst;
    reg [1:0] state;
    assign out = (state==2);
    always @(posedge clk or posedge rst)
        if (rst)
            state <= 0;
        else
            case (state)
                0: state <= in ? 1 : 0;
                1,2: state <= in ? 2 : 0;
            endcase
endmodule
```
Finite State Machines (continued)

Explicit FSM

To describe an explicit FSM:

- Define a state variable
- Describe the FSM in a synchronous procedural block with a `case` statement
- Upon every active clock edge, update the state variable
  — Depending on the input values and current state
An implicit FSM steps through multiple timing controls.

module fsm(out,in,clk,rst);
output out; reg out;
input clk,in,rst;
always @(rst)
  if (rst)
    assign out = 0;
  else begin
    deassign out;
    disable machine;
  end
always @(posedge clk)
  if (in)
    @(posedge clk)
    begin
      while (in)
        begin
          out <= 1;
          @(posedge clk);
          end
      out <= 0;
      @(posedge clk);
    end
endmodule
Finite State Machines (continued)

Implicit FSM

To describe an implicit FSM:

- Describe the FSM in a synchronous procedural block with branching and looping statements
- Upon every active clock edge, step to the next timing control
  — Depending on the input values and current state

Note: Examine the unusual (and unsynthesizable) separate asynchronous reset block. While the reset signal is asserted, this block makes a procedural continuous assignment to the FSM outputs, overriding any assignment to the outputs that the FSM might make. The FSM can transition through several states during the reset, but the outputs remain initialized. When the reset deasserts, this block removes the assignment to the output and disables the FSM block, which cancels any scheduled FSM events, making the FSM block restart at the beginning.
Summary

In this section, you learned about:

- Verilog functions and tasks
- Named statement blocks
- Finite state machines (FSMs)
Summary
Review

1. What are the constructs that generate a new level of scope?
2. What constructs can you disable?
3. What are some characteristics of a named block?
4. What are some differences between explicit and implicit FSMs?
Review

1. What are the constructs that generate a new level of scope?
   Functions, modules, named blocks, and tasks are a new level of scope. The Verilog module is the main hierarchical divider. Functions and tasks provide additional partitioning and encapsulation of code.

2. What constructs can you disable?
   You can disable Verilog tasks and named blocks.

3. What are some characteristics of a named block?
   You can name a **begin-end** or **fork-join** block. A named block constitutes a new level of scope with new local variables. You can disable a named block.

4. What are some differences between explicit and implicit FSMs?
   Explicit FSMs make assignments to a state variable, debug more easily, handle complex state changes well, and are typically synthesizable. Implicit FSMs step through timing controls, simulate more efficiently, handle only linear state changes well, and are generally not synthesizable.
About Lab 10

The objective of this lab is to model a controller Finite State Machine (FSM), and test it using stimulus and response read from a file.

In this lab, you will:

- Model the sequence controller for the VeriRisc CPU at the RTL level
- Simulate the controller model with a provided testbench that reads stimulus and response vectors from files
About Lab 10
About Lab 11

The objective of this lab is to test the VeriRisc CPU design using diagnostic programs as stimulus.

In this lab, you will:

- Learn how to use microprograms as stimulus
- Learn how to use tasks to enhance interactive debugging capabilities
- Simulate a complete RISC CPU design
About Lab 11
Chapter 18: User-Defined Primitives

Objectives

In this section you will:

- Briefly review the built-in Verilog primitives
- Compare built-in and user-defined primitives
- Learn how to model logic with user-defined primitives
- Examine several user-defined primitive examples
User-Defined Primitives

This section contains a thorough explanation of user-defined primitives (UDPs). It compares UDPs with the built-in Verilog primitives and explains why you would want to define your own UDP and how to define it. The section ends with several complete examples of representative UDPs.
What is a Verilog Primitive?

The Verilog language offers 26 built-in primitives:

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>and, nand, or, nor, xor, xnor</td>
<td>(out, in &lt;, &lt;in&gt;&gt;*)</td>
</tr>
<tr>
<td>buf, not</td>
<td>(out &lt;, &lt;out&gt;&gt;*, in)</td>
</tr>
<tr>
<td>bufif0, notif0, bufif1, notif1</td>
<td>(out, in, ctrl)</td>
</tr>
<tr>
<td>pullup, pulldown</td>
<td>(out)</td>
</tr>
<tr>
<td>pmos, nmos, rpmos, rnmos</td>
<td>(out, in, ctrl)</td>
</tr>
<tr>
<td>cmos, rcmos</td>
<td>(out, in, n-ctrl, p-ctrl)</td>
</tr>
<tr>
<td>tran, rtran</td>
<td>(io1, io2)</td>
</tr>
<tr>
<td>tranif0, tranif1, rtranif0, rtranif1</td>
<td>(io1, io2, ctrl)</td>
</tr>
</tbody>
</table>


What is a Verilog Primitive?

The **and**, **nand**, **nor**, **or**, **xnor**, and **xor** primitives can have multiple inputs. The output is always the first port in the port list.

The **buf** and **not** primitives can have multiple outputs. The input is always the last port in the port list.

The **bufif0**, **bufif1**, **notif0**, and **notif1** primitives can drive a Z output value.

The **pullup** primitive applies a pull-strength logic 1 to its port. The **pulldown** primitive applies a pull-strength logic 0 its port.

The **nmos**, **pmos**, **rnmos**, and **rpmos** primitives can pass a Z value from input to output as well as drive a Z output.

The **cmos** and **rcmos** primitives can pass a Z value from input to output as well as drive a Z output. The **cmos** primitive is like a paired **nmos/cmos** device. The **rcmos** device is like a paired **rnmos/rcmos** device.

The **rtran**, **rtranif0**, **rtranif1**, **tran**, **tranif0**, and **tranif1** gates are bidirectional gates. They can pass Z values as well as drive Z values.

The "r" version of gates are "resistive" versions of the primitives. Resistive primitives drop one strength-value from input to output.
What is a User-Defined Primitive?

A user-defined primitive (UDP) acts only slightly differently than a built-in primitive:

- Verilog restricts UDPs to one output and up to 10 inputs
- Verilog restricts UDP ports types to input or output but not inout
- Verilog restricts UDPs to drive 0, 1, or X, but not Z
- You specify the output state for any given input state or transition
- You can specify an initial state for a UDP
- Synthesis tools do not generally accept UDPs

You can otherwise use a UDP just like a built-in Verilog primitive:

- You instantiate them like a built-in Verilog primitive
  - You can provide a delay specification
  - You can provide a drive strength specification
  - You can instantiate arrays of UDP instances
What is a User-Defined Primitive?

UDPs can have only one output. You need separate UDPs to model the functionality of additional outputs.

UDPs can have 1 to 10 inputs. However, the memory required for the look-up table is an exponentially increasing function of the number of inputs.
Why Define My Own Primitive?

Define your own primitive to take advantage of UDP features:

- You can enhance simulation performance by replacing several built-in primitives with a single UDP
- You can more accurately model the functionality of hardware because you have more control over the output response to input changes
Why Define My Own Primitive?

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.

Synthesis vendor simulation libraries typically utilize UDPs. This is OK 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.
How Do I Define My Own Primitive?

This is what a user-defined primitive looks like:

```
primitive udp_name (output_name, input_name ...);
output output_name;
reg output_name; // only for sequential UDP
input input_name;
initial output_name = binary_value;
  table
    ...
  endtable
endprimitive
```

- Define the UDP outside of any module description
- Place the output first in the port list
- Declare the output type `reg` for a sequential UDP
- You can initialize a sequential UDP to 0, 1, or 1'bX
How Do I Define My Own Primitive?
Defining the Truth Table

This is what truth table entries for a combination device may look like:

\[
\begin{align*}
0 & 0 & ? & : & 0 ; \\
0 & 1 & ? & : & 1 ; \\
1 & ? & 0 & : & 0 ; \\
1 & ? & 1 & : & 1 ; \\
\end{align*}
\]

- Enter input states in the same order as the input port list
- Then enter a colon (:), the output state, and a semicolon (;)

This is what truth table entries for a sequential device may look like:

\[
\begin{align*}
0 & r & : & ? & : & 0 ; \\
1 & r & : & ? & : & 1 ; \\
? & n & : & ? & : & - ; \\
* & ? & : & ? & : & - ; \\
\end{align*}
\]

- 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
How Do I Define My Own Primitive? (continued)

Defining the Truth Table

Table row entries for a combinational device consist of input state values of 0, 1, X, or ? and output state values of 0, 1, or X. The input symbol ‘?’ indicates any of the binary values (a don’t-care situation).

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 ‘-’. The output symbol ‘-’ indicates that the output does not change 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 affect 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

You can use symbols to improve the truth table readability.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Interpretation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>No change</td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>0 or 1 or x</td>
<td>Any value</td>
</tr>
<tr>
<td>b</td>
<td>0 or 1</td>
<td>Any known value</td>
</tr>
<tr>
<td>r</td>
<td>(01)</td>
<td>0-&gt;1 transition</td>
</tr>
<tr>
<td>f</td>
<td>(10)</td>
<td>1-&gt;0 transition</td>
</tr>
<tr>
<td>p</td>
<td>(01) or (0x) or (x1)</td>
<td>Any positive edge</td>
</tr>
<tr>
<td>n</td>
<td>(10) or (1x) or (x0)</td>
<td>Any negative edge</td>
</tr>
<tr>
<td>*</td>
<td>(??)</td>
<td>Any transition</td>
</tr>
</tbody>
</table>
How Do I Define My Own Primitive? (continued)

Truth Table Symbols

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’ state really means either the (10) or (X0) transition.
Examples of UDPs

This subsection presents the following UDP examples:

- Combinational example: 2-1 multiplexer
- Combinational example: full adder
- Level-sensitive sequential example: latch
- Latch with enable and reset
- Edge-sensitive sequential example: D flip-flop
- D flip-flop with synchronous reset
- D flip-flop using a notifier
Examples of UDPs
Combinational Example: 2-1 Multiplexerer

primitive UDP_MUX21 (o, s, a, b);
  output o;
  input s, a, b;
  table
    //s  a  b : o
    0  0  ? : 0; //select a
    0  1  ? : 1;
    1  ?  0 : 0; //select b
    1  ?  1 : 1;
    x  0  0 : 0; //reduce pessimism
    x  1  1 : 1;
  endtable
endprimitive

Define a UDP outside of a module.

Place the output first in the port list.

Make entries in input port list order.

Specify all combinations producing a known output.

The symbol ‘?’ indicates any of the values 0, 1, or X.
Examples of UDPs

Combinational Example: 2-1 Multiplexer

If the select signal $s$ is 0, the output follows input $a$ regardless of the value of input $b$.

If the select signal $s$ is 1, the output follows input $b$ regardless of the value of input $a$.

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 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.
Combinational Example: Full Adder

You can implement the 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
```
Examples of UDPs (continued)

Combinational Example: Full Adder

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

primitive UDP_LATCH (q, d, e);
output q; reg q;
input d, e;
    initial q = 1'b1;
endprimitive

Table

<table>
<thead>
<tr>
<th>d</th>
<th>e</th>
<th>q</th>
<th>qnext</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>?</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>?</td>
<td>1</td>
</tr>
<tr>
<td>?</td>
<td>0</td>
<td>?</td>
<td>-</td>
</tr>
</tbody>
</table>

Declare the output type **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.

The symbol ‘-’ indicates no output change.
Examples of UDPs (continued)

Level-Sensitive Sequential Example: Latch

If the enable signal $e$ is 1 the output $q$ follows the input $d$ regardless of the current state.

If the enable signal $e$ is 0 the output $q$ maintains its current state regardless of the input $d$.

You can use the **initial** statement to model power-up reset if the device you are modeling offers that feature.
Latch with Enable and Reset

For this latch the low-active reset \( r \) is enabled when the high-active enable \( en \) is 0.

```vhdl
primitive UDP_LATCHR (q, d, en, r);
output q; reg q;
input d, en, r;

  table
    // d e r : q : qn
    ? 0 1 : ? : -; // disabled
    ? 0 0 : ? : 0; // reset
    0 ? 0 : ? : 0; // reduce pessimism
    1 1 ? : ? : 1; // pass 1
    1 ? 1 : 1 : -; // reduce pessimism
    0 1 ? : ? : 0; // pass 0
    0 ? ? : 0 : -; // reduce pessimism

endtable
endprimitive
```
Examples of UDPs (continued)

Latch with Enable and Reset

If $d=0$ and $r=0$ then $q=0$ regardless of $e$.

If $d=q$ and $r=1$ then the output does not change regardless of $e$.

If $d=0$ and $q=0$ then the output does not change regardless of $e$ or $r$. 
Edge-Sensitive Sequential Example: D Flip-Flop

primitive UDP_DFF (q, d, ck);
output q; reg q;
input d, ck;

table
// d ck : q : qnext
  0 r : ? : 0 ; // clock 0
  0 p : 0 : - ; // reduce pessimism
  1 r : ? : 1 ; // clock 1
  1 p : 1 : - ; // reduce pessimism
  ? n : ? : - ; // falling clk
  * ? : ? : - ; // data edge
endtable
endprimitive

Put no more than one input transition symbol in each table row.

Specify all transitions producing a known output.

The symbol ’r’ means the rising transition (01).
The symbol ’p’ means any positive transition (0X), (01), (X1).
The symbol ’n’ means any negative transition (1X), (10), (X0).
Examples of UDPs (continued)

Edge-Sensitive Sequential Example: D Flip-Flop

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.

For a positive but not definitely rising clock edge, if the output value already matches the input value, then the output maintains its current state.

For negative clock transitions the output maintains its current state.

For any data transition the output maintains its current state.
D Flip-Flop With Synchronous Reset

primitive UDP_DFFR (q, d, ck, r);
output q; reg q;
input d, ck, r;
table
  //d ck r : q : qnext
  ? r 0 : ? : 0; // clk reset
  ? p 0 : 0 : -; // reduce pessimism
  0 r ? : ? : 0; // clock 0
  0 p ? : 0 : -; // reduce pessimism
  1 r 1 : ? : 1; // clock 1
  1 p 1 : 1 : -; // reduce pessimism
endtable
endprimitive
Examples of UDPs (continued)

Flip-Flop with Synchronous Reset

For a low-active synchronous reset, the clocked data is a AND function of the reset and data inputs. When either input is 0 the clocked data is 0 regardless of the other input.
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 register.

```
timescale 1 ns / 1 ns
module dff_nt (q, ck, d, rst);
input ck, d, rst;
output q;
  reg nt;
    U_FFD_RB  i1 (q, d, ck, rst, nt);

specify
  specparam tsu = 2;
  (ck => q) = (2:3:4);
$setup (d, posedge ck, tsu, nt);
endspecify
endmodule
```

```
primitive U_FFD_RB (Q,D,CP,RB,NT);
output Q; reg Q;
input D, CP, RB, NT;
table
 // D CP RB NT : Q : Qn
  ? ? 0 ? : ? : 0; // reset
  ? ? n ? : 0 : -; // pessimism
  0 r ? ? : ? : 0; // clock 0
  0 p ? ? : 0 : -; // pessimism
  1 r 1 ? : ? : 1; // clock 1
  1 p 1 ? : 1 : -; // pessimism
endtable
endprimitive
```
Examples of UDPs (continued)

D Flip-Flop Using a Notifier

Include the notifier register argument in the timing check system task.
The simulator toggles the notifier register value when the timing check fails.
You can use the notifier register transition to make the UDP output unknown.
Summary

In this section you:

- Briefly reviewed the built-in Verilog primitives
- Compared built-in and user-defined primitives
- Learned how to model logic with built-in primitives
- Examined several user-defined primitive examples
Summary
Review

1. How many inputs may you declare for a UDP?

2. True or false: You define a UDP inside a module definition as you do a Verilog function or task.

3. What output values can a UDP drive?

4. How many times does the simulator access the UDP truth table if three inputs to an instance of a UDP change state at the same simulation time?

5. For what logic would you most likely use a UDP?
Review

1. How many inputs may you declare for a UDP?
   You may declare from 1 to 10 inputs for a UDP. For a sequential UDP the current state counts as one of the 10 permitted inputs.

2. True or false: You define a UDP inside a module definition as you do a Verilog function or task.
   You define a UDP outside a module definition. Like a module, it is a basic building block of your design.

3. What output values can a UDP drive?
   A UDP can drive the output values 0, 1, X. It cannot drive the high-impedance value Z.

4. How many times does the simulator access the UDP truth table if three inputs to an instance of a UDP change state at the same simulation time?
   The simulator accesses the UDP truth table separately for each transition of each input. For this example the simulator accesses the table three times.

5. For what logic would you most likely use a UDP?
   You would use a UDP for small combinational and sequential (latch and flip-flop) circuits that you instantiate numerous times, that the testbench exercises heavily, and that you will not synthesize. UDPs are ideal for ASIC macro simulation libraries.
About Lab 12

The objective of this lab is to create and test a sequential UDP model of a D flip-flop.

In this lab you will:

- Create a UDP definition to model the functionality of a D flip-flop
- Modify a D flip-flop module to use the UDP
- Simulate the UDP version of the D flip-flop
About Lab 12
Chapter 19: Modeling for Synthesis

Objective

In this section you will learn how to model Verilog for synthesis.

This section discusses the following:

- Modeling combinational logic
  This includes a discussion of sensitivity lists, incomplete conditional expressions, inferring latches, and resource sharing.

- Modeling sequential logic
  This includes a discussion of blocking and non-blocking assignments and synchronous and asynchronous resets.

- Unsynthesizable Verilog constructs
Modeling for Synthesis

This section describes the Verilog constructs that the logic synthesis tools accept, and how to use those constructs to define combinational and sequential logic, and Finite State Machines (FSMs). The section ends with a quick reference list of Verilog constructs that you should not use in a design that represents synthesizable hardware.
Modeling Combinational Logic

Combinational logic: The current state of the output can be determined solely by the current state of the inputs.
Modeling Combinational Logic

The output of combinational logic can be determined solely by the current state of the inputs (ignoring propagation delay).

This implies no internal state storage.
Continuous Assignments

Continuous assignments, by their nature, imply combinational logic.

```
module or_and (y, a, b, c, d, e);
input a, b, c, d, e;
output y;
    assign y = (a | b) & c & (d | e);
endmodule
```
Continuous Assignments

A continuous assignment continuously drives a net. It responds immediately to any input transition, and if necessary drives a new value after any specified delay.
Asynchronous Procedural Blocks

Asynchronous procedural blocks imply combinational logic.

```
module or_and (y, a, b, c, d, e);
input a, b, c, d, e;
output y; reg y;
    always @(a or b or c or d or e)
        y = (a|b) & c & (d|e);
endmodule
```
Asynchronous Procedural Blocks

An asynchronous procedural block contains every edge of every input signal in its sensitivity list. The block responds immediately to any input transition, and if necessary assigns a new value after any specified delay.

The Verilog simulator executes the asynchronous block once for each input transition. You must declare Verilog register types (integer, real, reg, or time) to store the value of the left-hand side of procedural assignments. This use of the Verilog register types does not imply sequential logic.
**Sensitivity Lists**

Good coding practice means you complete the sensitivity list.

This is a simulation problem — not a synthesis problem!

This example has an incomplete sensitivity list.

This can cause simulation mismatches.

```verilog
module inc_sens (y, s, b, a);
input s, b, a;
output y; reg y;
always @(s)
  begin
    if (s)
      y = b;
    else
      y = a;
  end
endmodule
```
Modeling Combinational Logic (continued)

Sensitivity Lists

It is good modeling practice to include all inputs to the block in its sensitivity list. Different synthesis tools react in different ways to an incomplete sensitivity list. Some synthesis tools consider an incomplete sensitivity list illegal. Others produce a warning and proceed with the assumption that you meant to have a complete list. In this case, the synthesized output may simulate differently than the RTL description.

The complete sensitivity list for this example would be:

```verbatim
always @(s or a or b)
```
Verilog Registers

If the current state of the output can be determined by the current state of the inputs:

- Declare a Verilog reg to satisfy the Verilog semantics rules
- The Verilog reg keyword does not by itself imply sequential logic

```verilog
testbench:
    module no_reg (y, a, b, c);
    input a, b, c;
    output y; reg y;
    reg tmp;
    always @(a or b or c)
    begin
        if (a & b)
            tmp = c;
        else
            tmp = 0;
        y = tmp;
    end
endmodule
```
Modeling Combinational Logic (continued)

Verilog Registers

You declare Verilog register types to satisfy the Verilog semantics rules. These type declarations do not imply sequential logic.
Verilog Registers

If the current state of the output cannot be determined by the current state of the inputs:

- Declare a Verilog reg to satisfy the Verilog semantics rules
- The Verilog reg keyword does not by itself imply sequential logic
- The synthesis tool infers a latch to hold the output state

```verilog
c module latch (y, a, b, c);
  input a, b, c;
  output y; reg y;
  reg tmp;
  always @(a or b or c)
  begin
    if (a & b)
      tmp = c;
    y = tmp;
  end
endmodule
```
Verilog Registers (continued)

You declare Verilog register types to satisfy the Verilog semantics rules. These type declarations do not imply sequential logic. The synthesis tool infers a latch to hold the output state of the block during those input combinations for which you have not specified the output value.
Conditional Statements

Naturally complete conditional statements imply combinational logic.

```verilog
module comp_case (y,a,b,c,d);
    input a,b,c,d;
    output y; reg y;
    always @(a or b or c or d)
        case ({a,b})
            2'b00: y = 1'b0;
            2'b01: y = 1'b1;
            2'b10: y = c;
            2'b11: y = d;
        endcase
endmodule

module comp_if (y,a,b,c,d);
    input a,b,c,d;
    output y; reg y;
    always @(a or b or c or d)
        if (!a && !b)
            y = 1'b0;
        else if (!a &&  b)
            y = 1'b1;
        else if ( a && !b)
            y = c;
        else if ( a &&  b)
            y = d;
endmodule
```
Conditional Statements

This is purely combinational logic. The current state of the output can be determined solely by the current state of the inputs.
Conditional Statements

Incomplete conditional statements imply state storage.

module incp_case (y,a,b,c,d);
    input a,b,c,d;
    output y; reg y;
    always @(a or b or c or d)
        case ({a,b})
            2'b00: y = 1'b0;
            2'b01: y = 1'b1;
            2'b10: y = c;
        endcase
    endmodule

module incp_if (y,a,b,c,d);
    input a,b,c,d;
    output y; reg y;
    always @(a or b or c or d)
        if (!a && !b)
            y = 1'b0;
        else if (!a && b)
            y = 1'b1;
        else if ( a && !b)
            y = c;
        endmodule

What is the output when a=1 and b=1?
Conditional Statements (continued)

The current state of the output cannot be determined solely by the current state of the inputs. The synthesis tool infers sequential logic to hold the output state of the block during those input combinations for which you have not specified the output value.
Conditional Statements

Default complete conditional statements imply combinational logic.

module comp_case (y,a,b,c);
  input a,b,c;
  output y; reg y;
  always @(a or b or c)
    case ({a,b})
      2'b00: y = 1'b0;
      2'b01: y = 1'b1;
      2'b10: y = c;
    default: y = 1'bx;
  endcase
endmodule

module comp_if (y,a,b,c);
  input a,b,c;
  output y; reg y;
  always @(a or b or c)
    begin
      y = 1'bx;
      if (!a && !b) y = 1'b0;
      else if (!a && b) y = 1'b1;
      else if (a && !b) y = c;
    end
endmodule

Note: Synthesis tools treat 1'bx as a don't care.
Conditional Statements (continued)

This is purely combinational logic. You have specified an output state for all possible combinations of inputs. The synthesis tool will optimize the logic that produces the defined outputs. This optimization will determine what the state of the output is for the undefined input combinations.

The pre-synthesis and post-synthesis simulation results may differ. The simulator will produce an (unknown (X) output state for the undefined input combinations of the pre-synthesis design, and a binary (0 or 1) output state for the undefined input combinations of the post-synthesis design.
Conditional Statements

Directive complete case statements imply combinational logic.

```verilog
module dir_case (y,a,b,c);
input a,b,c;
output y; reg y;
always @(a or b or c)
    case ({a,b}) // ambit synthesis case = full
        2'b00: y = 1'b0;
        2'b01: y = 1'b1;
        2'b10: y = c;
    endcase
endmodule
```
Conditional Statements (continued)

Synthesis directives (sometimes called *pragmas*) direct the synthesis tool’s interpretation and transformation of the source code. The "case=full" directive instructs the synthesis tool to ignore any missing input combinations. You would use this directive if you, as the designer, know that certain input combinations cannot occur. For the synthesis tool, this directive is equivalent to a default statement that sets the output to the don’t-care state.

You embed synthesis directives in Verilog comments in the statement block they affect.

The pre-synthesis and post-synthesis simulation results may differ. The simulator will produce no output state change for the undefined input combinations of the pre-synthesis design, and a binary (0 or 1) output state for the undefined input combinations of the post-synthesis design.
Conditional Statements

The "case=full" synthesis directive applies only to the case match items.

The synthesis tool will still infer a latch for other missing input combinations!

```verilog
module select (y1, y2, in);
output y1, y2; reg y1, y2;
input [1:0] in;
always @(in)
    case (in) // ambit synthesis case = full
    2'b00: begin y1 = 1'b0; y2 = 1'b0; end
    2'b01: begin y1 = 1'b0; y2 = 1'b1; end
    2'b10: begin y1 = 1'b1; y2 = 1'b0; end
    2'b11: begin y1 = 1'b1; end
    default: begin y1 = 1'bx; y2 = 1'bx; end
endcase
endmodule
```
Conditional Statements (continued)

The synthesis tool infers a latch for this example because the 2'b11 case match item does not specify a value for the $y_2$ output.
**Synthesis case Directives**

The Verilog `case` statement implies priority.

Earlier `case` items have higher priority than later `case` items.

The synthesis tool by default builds a priority structure for a `case` statement.

You can direct the synthesis tool’s interpretation of a `case` statement:

- // ambit synthesis case = full
  Assume the `case` statement is fully defined, ignore missing case items
  This avoids inferring latches for the missing case items

- // ambit synthesis case = mux
  Infer a mux library component, if available

- // ambit synthesis case = parallel
  Assume case items are mutually exclusive, build parallel logic
Modeling Combinational Logic (continued)

Synthesis case Directives
Inferring Latches

An incomplete conditional statement in an asynchronous block infers a latch.

```
module latch (q, d, en);
output q; reg q;
input d, en;
  always @(d or en)
    if (en)
      q = d;
endmodule
```
Inferring Latches

The current state of the output cannot be determined solely by the current state of the inputs. The synthesis tool infers a latch to hold the output state of the block during those input combinations for which you have not specified the output value.
Latch Reset

A latch with reset is still an asynchronous device.

An asynchronous block has a complete sensitivity list.

It is the incomplete conditional statement that infers the latch.

module latch (q, d, en, set, rst);
output q; reg q;
input d, en, set, rst;
    always @(d or en or set or rst)
        if (rst)
            q = 0;
        else if (set)
            q = 1;
        else if (en)
            q = d;
endmodule
Inferring Latches (continued)
Functions

Functions cannot include timing controls.

Thus, functions always imply combinational logic.

You can call a function from a procedural block or a continuous assignment.

```
module or_and (y, a, b, c, d, e);
input a, b, c, d, e;
output y;
    function f_or_and;
        input a, b, c, d, e;
        forand = (a|b) & c & (d|e);
    endfunction
    assign y = f_or_and (a, b, c, d, e);
endmodule
```
Modeling Combinational Logic (continued)

Functions

Functions cannot contain timing controls, thus a function always implies combinational logic. You can call a function from a procedural block and assign its output value to a Verilog register. Depending upon how you do this, you can infer a sequential device in the procedural block, but the function itself represents combinational logic.

Synthesis tools do not infer a latch for missing function input combinations.
Tasks

Tasks can include timing controls — but if so, are not synthesizable.

Synthesizable tasks always imply combinational logic.

You can call a task only from a procedural block.

```vhdl
module or_and (y, a, b, c, d, e);
input a, b, c, d, e;
output y; reg y;
task t_or_and;
output y;
input a, b, c, d, e;
y = (a | b) & c & (d | e);
endtask
always @(a or b or c or d or e)
t_or_and (y, a, b, c, d, e);
endmodule
```
Tasks

As a synthesizable task cannot contain timing controls, there is little to differentiate it from a function. Perhaps a good rule-of-thumb might be to restrict tasks to your testbench and use functions to represent synthesizable combinational logic.
Complex Operators

Adders and multipliers are examples of high-level complex operators.

Synthesis tools map high-level operators to predefined library components:

- Possibly the silicon vendor’s technology library
- Could be a library of your own user-defined components
- More likely a separately-priced synthesis vendor’s library
  - These libraries can include various architectures for components such as adders, counters, decoders, FIFOs, and shifters
  - Select the architecture with the architecture selection directive
    
    // ambit synthesis architecture = architecture_name
Modeling Combinational Logic (continued)

Complex Operators

The architecture selection directive directs the synthesis tool to select a specific architecture, if it is available in the libraries. Carry Lookahead and Ripple Carry are examples of specific adder architectures.
Resource Sharing

More than one RTL statement can utilize an expensive computational resource.

```
always @ (a or b or c or d or s)
begin
    if (s)
        y = a + b;
    else
        y = c + d;
end
```

```
always @ (a or b or c or d or s)
begin: newscope
    reg tmp1, tmp2;
    tmp1 = s ? a : c;
    tmp2 = s ? b : d;
    y = tmp1 + tmp2;
end
```

Some synthesis tools automatically share resources.

What if your synthesis tool does not automatically share resources?

You can write your RTL code to do its own resource sharing!
A resource is a statement that implements an arithmetic or relational operator. Adders, comparators, and multipliers are examples of resources.

Resource sharing is dependent on the synthesis tool. Some do not share resources at all. Some may share the resources that you use multiple times in the same conditional statement.
Modeling Sequential Logic

Sequential logic: The current state of the output cannot be determined solely by the current state of the inputs. This implies internal state storage.

![Diagram of a D flip-flop](image-url)
Modeling Sequential Logic

The output of sequential logic cannot be determined solely by the current state of the inputs (ignoring propagation delay).

This implies internal state storage.
Synchronous Procedural Blocks

Synchronous procedural blocks imply sequential logic.

```verilog
always @(posedge clk or negedge rst)
  if (!rst)
    q <= 0;
  else
    q <= d;
```

![Synchronous Procedural Block Diagram](image-url)
Modeling Sequential Logic (continued)

Synchronous Procedural Blocks

A synchronous procedural block contains only one edge of input signals in its sensitivity list. The block responds only to the specified edges, and assigns a new value after any specified delay. The synthesis tool infers a register to hold the output value until the next triggering signal edge.
Verilog Registers

The synthesis tool can remove a Verilog reg you use in a synchronous block. It will remove a register you write and then read in the same clock cycle. The Verilog reg keyword does not imply sequential logic.

module one_reg (q, d, clk);
output q; reg q;
input d, clk;
    always @(posedge clk)
    begin: newscope
        reg tmp;
        tmp = 0;
        if (d) tmp = 1;
        q = tmp;
    end
endmodule
Verilog Registers

You can make procedural assignments only to Verilog register types. You declare the left-hand side of a procedural assignment to be a register type to satisfy Verilog semantics rules. This declaration does not imply sequential logic. You infer sequential logic by using only single edges of signals in the sensitivity list. The synthesis tool can remove variables not needed to hold values across clock boundaries.

This contrived example illustrates removal of temporary variables, which can only occur with blocking assignments. You would typically use nonblocking assignments in sequential procedural blocks.
Verilog Registers

The synthesis tool cannot remove a register you write and read in different clock cycles.

```verilog
module two_reg (q, d, clk);
output q; reg q;
input d, clk;
  reg tmp;
  always @(posedge clk)
    begin
      tmp = 0;
      if (d) tmp = 1;
    end
  always @(posedge clk)
    q = tmp;
endmodule
```
Verilog Registers

The simulator can process **always** blocks in any order. The simulator can read the value of $tmp$ that it wrote during the previous clock. As the $tmp$ register is not really temporary, the synthesis tool cannot remove it.

This design description is ambiguous. The synthesis tool may report an error, or it may report a warning and assume one of these two interpretations. If you do not correct this description, then your pre-synthesis simulation results may not match your post-synthesis simulation results.
Blocking and Non-Blocking Assignments

Use non-blocking assignments in synchronous blocks:

- Non-blocking assignments prevent simulation clock-data races
- Non-blocking and blocking assignments describe different functionality
- Non-blocking assignments describe unambiguous functionality

```verilog
module block (q, d, clk);
    input d, clk;
    output q;
    reg q;
    reg tmp;
    always @(posedge clk)
    begin
        tmp = d;
        q = tmp;
    end
endmodule

module noblock (q, d, clk);
    input d, clk;
    output q;
    reg q;
    reg tmp;
    always @(posedge clk)
    begin
        tmp <= d;
        q <= tmp;
    end
endmodule
```
Modeling Sequential Logic (continued)

Blocking and Non-Blocking Assignments

The synthesis tool can remove the temporary variable $tmp$ in the example with blocking assignments, as the example writes, and then reads, the variable in the same clock cycle.

The synthesis tool cannot remove the variable $tmp$ in the example with nonblocking assignments, as the example reads its value from the previous clock cycle while writing a new value in the current clock cycle.
Modeling Resets

You can use synchronous or asynchronous resets in a synchronous block.

module sync(q, ck, d, r);
output q;
reg q;
input ck, d, r;
always @(posedge ck)
if (r)
    q <= 0;
else
    q <= d;
endmodule

module async(q, ck, d, r);
output q;
reg q;
input ck, d, r;
always @(posedge ck or posedge r)
if (r)
    q <= 0;
else
    q <= d;
endmodule
Modeling Resets

To model a synchronous reset, treat the reset as an overriding data input.

Place the reset in the first conditional branch or first `case` match item.

To model an asynchronous reset, include the active edge of the reset in the sensitivity list and model the reset exactly as you would for a synchronous reset.
Finite State Machines

You can encode FSM states in two ways:

- Explicit encoding
  You declare a state `reg` and code a `case` statement to transition states
  All synthesis tools accept explicit state encoding

- Implicit encoding
  You use multiple `@(posedge clk)` statements to indicate state transitions
  Few synthesis tools accept implicit state encoding
Finite State Machines
Example Explicit FSM

\`timescale 1ns/100ps
module state4 (out, clk, rst);
output [1:0] out;
reg [1:0] out;
input clk, rst;

// Enumerate state information
parameter stateA = 2'b00,
    stateB = 2'b01,
    stateC = 2'b10,
    stateD = 2'b11;

reg [1:0] state, nextstate;

// Synchronous block for state
always @(posedge clk)
    if (rst)
        state <= stateA;
    else
        state <= nextstate;

// Combination block for next state
always @(state)
    case (state)
        stateA: nextstate = stateB;
        stateB: nextstate = stateC;
        stateC: nextstate = stateD;
        stateD: nextstate = stateA;
    endcase

// Output is function of state
always @(state)
    case (state)
        stateA: out = 2'b00;
        stateB: out = 2'b11;
        stateC: out = 2'b10;
        stateD: out = 2'b00;
    endcase
endmodule
Finite State Machines (continued)

Example Explicit FSM
Synthesis FSM Directives

You can direct the synthesis tool’s optimization of the FSM:

- // ambit synthesis enum tag_name
  — Use this directive to enumerate the FSM states
  — Use this directive to bind the state vector to the state enumeration

- // ambit synthesis state_vector reg_name [optimizations]
  — Use this directive to identify the state vector and specify optimizations
Finite State Machines (continued)

Synthesis FSM Directives
Synthesis FSM Directives

`timescale 1ns/100ps
module state4 (clock, reset, out);
input reset, clock;
output [1:0] out;
reg [1:0] out;

// Define state enumeration information
parameter /*ambit synthesis enum state_info */
    stateA = 2'b00,
    stateB = 2'b01,
    stateC = 2'b10,
    stateD = 2'b11;

// Declare state variables and bind to enumeration
reg [1:0] /* ambit synthesis enum state_info */ state;
reg [1:0] /* ambit synthesis enum state_info */ nextstate;

// Declare state machine and specify FSM optimizations
always @(posedge clock)
/*ambit synthesis state_vector state -encoding one_hot */
    if (reset)
        state <= stateA;
    else
        state <= nextstate;
Synthesis FSM Directives (continued)
Additional Synthesis Directives

This is a partial list of Ambit BuildGates synthesis directives not already discussed:

- Code selection directives
  
  // ambit synthesis {off, on}

- Module template directive

  // ambit synthesis template

- Function and task mapping directives

  // ambit synthesis map_to_module module_name
  
  // ambit synthesis return_port_name port_name

- Set and reset directives

  // ambit synthesis set_reset [a]synchronous blocks = {block_list}
  
  // ambit synthesis set_reset [a]synchronous signals = {signal_list}
  
  // ambit synthesis set_reset [a]synchronous block(block_name) = {signal_list}
Additional Synthesis Directives

Use the code selection directives to hide portions of the source code from the parser.

Use the module template directive to accommodate Verilog module parameterization. When you write a module definition utilizing at least one module parameter to parameterize the module definition, you do not synthesize this module "template", but only synthesize its individually configured instances. This directive prevents the inadvertent synthesis of the template definition.

Use the function and task mapping directives to map a Verilog function or task to a module. The `map_to_module` directive directs the synthesis tool to map any call to the given task or function to an instance of the specified module that you have defined elsewhere (the synthesis tool ignores statements in the task or function body). The tool positionally maps function or task arguments onto ports of the module.

When mapping a function, you should also specify what output port provides the function return value. The `return_port_name` directive specifies the output port that provides the function return value.

Use the set and reset directives to direct how the synthesis tool handles set and reset signals. When the `do_build_generic` command infers a register from a HDL description, it also infers set and reset control of the register and defines whether these controls are synchronous or asynchronous. The synthesis tool infers set and reset from the HDL description, even if you do not use these directives. The synthesis directives direct the tool to attempt to implement the set and reset operation by using the set and reset pins on the storage element components.
Unsynthesizable Verilog Constructs

Synthesis tools generally do not support these Verilog constructs:

<table>
<thead>
<tr>
<th>deassign</th>
<th>fork</th>
<th>real</th>
<th>wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>event</td>
<td>initial</td>
<td>release</td>
<td>while</td>
</tr>
<tr>
<td>force</td>
<td>join</td>
<td>repeat</td>
<td>=== !==</td>
</tr>
<tr>
<td>forever</td>
<td>primitive</td>
<td>time</td>
<td></td>
</tr>
</tbody>
</table>

Synthesis tools accept the assign keyword as a continuous assignment outside a procedural block, but not as a procedural continuous assignment.

Synthesis tools require the indexes of a for loop to be statically computable.
Unsynthesizable Verilog Constructs
Summary

In this section you learned how to model Verilog for synthesis.

This section discussed the following:

- **Modeling combinational logic**
  This included a discussion of sensitivity lists, incomplete conditional expressions, inferring latches, and resource sharing

- **Modeling sequential logic**
  This included a discussion of blocking and non-blocking assignments and synchronous and asynchronous resets

- **Unsynthesizable Verilog constructs**
Summary
Review

1. What does the sensitivity list look like for a block that infers a latch with an asynchronous reset?

2. True or false: Blocking and non-blocking assignments always produce the same logic.

3. What are some obvious characteristics of an explicit FSM?

4. What Ambit BuildGates synthesis directive would you use to hide a testbench construct from the tool’s parser?

5. In what context does a synthesis tool accept the Verilog assign keyword?
Review

1. What does the sensitivity list look like for a block that infers a latch with an asynchronous reset?

A latch is an asynchronous construct. The block inferring the latch is sensitive to all changes on any input signals. The synthesis tool infers a latch when the output is not defined for some combination of inputs.

2. True or False: Blocking and non-blocking assignments always produce the same logic.

Blocking and non-blocking assignments can produce different logic. The synthesis tool can remove a temporary reg written with a blocking assignment and subsequently read in the same block.

3. What are some obvious characteristics of an explicit FSM?

An explicit FSM declares a state reg and in a case statement transitions between states by assigning values to the state reg.

4. What Ambit BuildGates synthesis directive would you use to hide a testbench construct from the tool’s parser?

You would use the ambit synthesis {off, on} directive to hide a testbench construct from the tool’s parser.

5. In what context does a synthesis tool accept the Verilog assign keyword?

Synthesis tools accept the Verilog assign keyword as a continuous assignment outside a procedural block. They do not accept the assign keyword inside a procedural block, function, or task.
About Lab 13

The objective of this lab is to practice modeling a design for synthesis.

In this lab, you will:

- Model an ALU in synthesizable Verilog
- Synthesize the design
- Test the design
About Lab 13
About Lab 14

The objective of this lab is to perform FSM synthesis optimizations.

In this lab, you will:
- Model an FSM in synthesizable Verilog
- Synthesize the FSM design with directives
- Test the FSM design

 FSM State Diagram

Legend: (in,out)
About Lab 14
Chapter 20: Modeling for Performance

Objectives

In this section you will learn about:

- The Verilog-XL Turbo and XL algorithms
- Verilog-XL and NC-Verilog performance
- Optimizing behavioral performance
- Command-line simulation control
- Code coverage
Modeling for Performance

This Cadence Verilog-XL and NC-Verilog simulator-specific section tells you how to obtain the best performance from the simulator. It covers design rules and simulator operational modes. The section ends with a discussion of code coverage, a technique for determining a figure of merit for your testbench, and code profiling.
Verilog-XL Simulation Algorithms

Verilog-XL utilizes the following simulation algorithms by default:

- The XL algorithm
  Accelerates simulation of primitives and unidirectional switches
- The Turbo algorithm
  Accelerates simulation of procedural blocks

You can select additional optimizations:

- Higher levels of Turbo optimizations
- Twin Turbo optimizations
  Precompile behavioral code blocks
- Switch-XL acceleration
  Acceleration of bidirectional switch simulation
Verilog-XL Simulation Algorithms

The Verilog-XL XL algorithm accelerates simulation of primitives and unidirectional switches. It accelerates nets and built-in and user-defined primitives.

The Turbo algorithm accelerates procedural blocks. It does not affect primitives and nets.
## Selecting Verilog-XL Optimizations

Use command-line options to enable or disable acceleration algorithms:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Simulate with default XL and Turbo algorithms</td>
</tr>
<tr>
<td>-a</td>
<td>Enable the XL algorithm (default since version 1.7)</td>
</tr>
<tr>
<td>+caxl</td>
<td>Accelerate continuous assignments</td>
</tr>
<tr>
<td>+noxl</td>
<td>Disable the XL algorithm</td>
</tr>
<tr>
<td>+no_speedup</td>
<td>Disable all behavioral performance improvements since 1.6c</td>
</tr>
<tr>
<td>+no_turbo</td>
<td>Disable all Turbo optimizations</td>
</tr>
<tr>
<td>+speed_up</td>
<td>Enable behavioral performance improvements (default since 1.7)</td>
</tr>
<tr>
<td>+switchxl</td>
<td>Accelerate bidirectional switches</td>
</tr>
<tr>
<td>+turbo</td>
<td>Enable additional Turbo optimizations (default since 2.0)</td>
</tr>
<tr>
<td>+turbo+2</td>
<td>Enable Turbo 2 optimizations</td>
</tr>
<tr>
<td>+turbo+3</td>
<td>Enable Turbo 3 optimizations</td>
</tr>
<tr>
<td>+twin_turbo</td>
<td>Enable Twin Turbo optimizations</td>
</tr>
</tbody>
</table>
Verilog-XL Simulation Algorithms

Selecting Verilog-XL Optimizations

Disable the XL algorithm if your design contains almost no primitives.

Disable the Turbo algorithm if your design contains almost no procedural code.

You can use the +noxl option to disable the XL algorithm. You would do this if your design contains almost no primitives. Switching simulation between the XL and non-XL algorithms entails some overhead that negatively impacts simulation performance for a design with very few acceleratable components.

You can use the +caxl option to enable XL acceleration of continuous assignments, and primitives with an operator on an input. The Turbo algorithms also accelerate continuous assignments, so use the +caxl option only for primarily structural designs.

You can use the +switchxl option to enable XL acceleration of bidirectional switches. This option also provides finer resolution of charge strengths and drive strengths.
## Selecting Turbo Optimizations

You can select one of three levels of Turbo optimizations:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+no_turbo</td>
<td>Disable all Turbo optimizations</td>
</tr>
<tr>
<td>+turbo (default)</td>
<td>Simulate procedural code with the default Turbo algorithm Disable the event counter and behavior profiler</td>
</tr>
<tr>
<td>+turbo+2</td>
<td>Convert scalar nets to compact nets Perform assignment optimizations</td>
</tr>
<tr>
<td></td>
<td>[Prevents retrieval of result values during event tracing]</td>
</tr>
<tr>
<td>+turbo+3</td>
<td>Delay evaluation of assignments until they occur</td>
</tr>
<tr>
<td></td>
<td>[May change zero-delay event ordering, thus expose pre-existing race conditions]</td>
</tr>
</tbody>
</table>
Selecting Turbo Optimizations

There is no additional licensing required for Turbo optimizations.

The Turbo option adds function calls to the data structure, optimizes some memory operations, and rewrites behavioral code. As of version 1.7, Verilog-XL performs these optimizations by default. Use the +no_turbo option to disable these optimizations, or use the +no_speedup option to disable all behavioral optimizations since version 1.6c.

The +turbo option improves performance by disabling the event counter and behavior profiler. As of version 2.0, Verilog-XL also performs these optimizations by default.

The +turbo+2 option improves performance by converting some scalar nets to compact nets and performing some assignment optimizations. When you trace assignments you may no longer see the actual value assigned.

The +turbo+3 option improves performance by delaying evaluation of the right-hand side of an assignment until the assignment actually occurs. This option reorders execution of procedural code. If for example, you have inadvertently coded a zero-delay race between writing and reading a variable, the simulation may produce different results with this option.
Selecting Twin Turbo Optimizations

You can use the `+twin_turbo` option with any level of Turbo:

- Generates compiled code for procedural blocks
- May change zero-delay event-ordering (races)
  - Use `+compat_twin_turbo` to restore event ordering
  - Restoring event ordering negatively impacts Twin Turbo performance
- Restricts your use of Verilog-XL features
  - Use `+trace_twin_turbo` to restore optimized trace results
  - Use `+save_twin_turbo` to restore save/restart capability
  - Incremental save not possible; `$incsave` works like `$save`
  - Restoring these features negatively impacts Twin Turbo performance

Note: The `+twin_turbo` option is not available on the NT platform.
Selecting Twin Turbo Optimizations

There is no additional licensing required for Twin Turbo optimizations.

The terms compiled code simulation and interpretive simulation refer to the way in which a simulator operates on a design’s source description.

An interpretive simulator reads the source code and builds an internal representation of all primitive operations. During simulation, it interprets and processes the internal representation.

A compiled code simulator converts the source code into machine code, then executes the program to run the simulation. Many simulators convert the HDL description into a standard programming language, such as C, and then use a compiler to convert the program into an executable.

In the Twin Turbo mode, Verilog-XL converts all behavioral Verilog code directly into machine code, thus bypassing the inefficient step of translating to standard C. The simulator executes the machine code at the system speed of the host computer. This mode increases simulation performance with only slight loss of flexibility and interactive debugging capability.
Combining Performance Options

You can combine multiple Verilog-XL performance options for best results:

+twin_turbo
+turbo+3
+caxl
+switchxl
Combining Performance Options

As a general rule, Turbo increases simulation performance, regardless of the use of the other command-line options.

Use the `+caxl` option to accelerate continuous assignments in a primarily structural design.

Use the `+switchxl` option to accelerate bidirectional switches.

There is some overhead in switching between the XL and non-XL components of your design.

- Use the `+noxl` option if you have very few primitives in your design
- Use the `+no_turbo` option if you have very little procedural code in your design
NC-Verilog Simulation Performance

NC-Verilog by default performs the following optimizations:

- Fully precompiles all source code to native instructions
- Takes advantage of incremental compilation
- Utilizes a single simulation kernel for all source code
- Collapses unnecessary data structure
- Collapses unnecessary events

NC-Verilog runs in performance mode "out of the box".

The following `ncverilog` command line options degrade performance:

- The `+noupdate` option forces recompilation of all design data units
- The `+nclinedebug` option restores line debug capability
- The `+ncaccess` option restores user debug access to simulation objects
NC-Verilog Simulation Performance
Optimizing XL Algorithm Performance

Do the following where practical to increase XL performance:

- Use the largest timescale precision possible
- Use the `+caxl` option to accelerate continuous assignments
- Avoid algorithm switching — minimize non-accelerated constructs
  - Model the clock with primitives if your design is mostly structural
  - Model storage devices with sequential UDPs in a gate-level design
- Use compiler directives to remove unneeded identifiers
  - `remove_gatenames`
  - `remove_netnames`
Optimizing XL Algorithm Performance

Try to use the largest timescale precision possible. Very precise timing requires more memory for delay tables and more simulation time to process the extra timeslices.

Structure the Verilog models to minimize the switching between XL and non-XL algorithms. For example, model flip-flops with sequential UDPs in a primarily gate-level design.

The IEEE Std. 1364-1995 does not describe the `\[no\]remove_gatenames` and `\[no\]remove_netnames` compiler directives. These are features specific to the Cadence Verilog-XL and NC-Verilog simulators.
Items Accelerated by XL

The XL algorithm accelerates primitives and nets.

Supported primitive types:

buf  and  or  xor  bufif0
not  nand  nor  xnor  notif0
bufif1  pmos  rpmos  cmos  pulldown
notif1  nmos  rnmos  rcmos  pullup
UDP

Supported net types:

wire  wand  wor  supply0  supply1
tri  triand  trior  tri0  tri1  trireg
Optimizing XL Algorithm Performance (continued)

Items Accelerated by XL

The table shows the primitives and net types that the XL algorithm accelerates:

- The XL algorithm also accelerates arrays of instances of those primitives
- The non-XL algorithm continues to simulate the non-acceleratable items
Items not Accelerated by XL

Unaccelerated Nets

The XL algorithm does not accelerate the following nets:

- A net that is or has been forced
  ```
  force RST=1; #50 release RST;
  ```

- A net declared with the `vectored` keyword
  ```
  wire vectored [63:0] busa;
  ```

- A net with a non-zero delay
  ```
  wire #3 node157;
  ```

**Note:** Use the `$shownonxl()` system task to show non-accelerated structures. Give it an instance name argument, or default to the entire design.
Items not Accelerated by XL

The **force** command forces a net or register to a value. Once you have forced a net, Verilog-XL does not accelerate that net, even if you release it, during the remainder of the simulation session.

The **vectored** keyword specifies that the simulator may not access bit or part selects of the vector net. This similarly restricts most PLI access to the vector net.

The **scalared** keyword specifies that the simulator may access bit or part selects of the vector net.

The simulator will expand a net declared without either keyword only when necessary.
Items not Accelerated by XL

Unaccelerated Primitives

The XL algorithm does not accelerate the following primitives:

- A bidirectional primitive
  \[
  \text{tran tranif}0 \ \text{tranif}1 \ \text{rtran rtranif}1 \ \text{rtranif}0
  \]

- A \textbf{buf} or \textbf{not} gate with more than one output
  \[
  \text{buf u1 (out1, out2, out3, in)};
  \]

- A primitive with a non-constant delay expression
  \[
  \text{and} \ (a + 1) \ (\text{out, ina, inb});
  \]

- A primitive with an input connected to a bit-select of a vector register
  \[
  \text{and} \ (\text{out, ina, vreg}[5]);
  \]

- A primitive with an expression on an input, or a net with a continuous assignment made to it (unless +caxl is used)
  \[
  \text{or} \ (\text{out, a1 & a2, b1 ^ b2})
  \]

- A primitive with a delay in excess of 65535 distinct delays
Items not Accelerated by XL (continued)

The XL algorithm does not by default accelerate bidirectional primitives. Use the +switchxl option to accelerate bidirectional primitives.

The XL algorithm can support no more than 65,535 distinct gate delays. Once you reach this limit, Verilog-XL does not accelerate subsequent gates having new distinct delays. For example, the delays #(3, 4) and #(3, 2) are distinct, but #3 and #(3,3) are not.
Optimizing Behavioral Performance

Do the following to increase simulator performance:

- Minimize your use of system tasks, especially file output tasks
- Minimize simulator work, such as expression evaluations
- Use compiler directives to reduce timing accuracy
  - `delay_mode_unit`
  - `delay_mode_zero`

Note: The IEEE Std. 1364-1995 describes, but does not require, the `delay_mode_unit` and `delay_mode_zero` compiler directives. Not all simulators support these features.
Optimizing Behavioral Performance

Try to reduce unnecessary scheduling activity and scopes. For example, naming code blocks can impact the performance of some simulators.

For Verilog-XL, structure the Verilog models to minimize the switching between the XL and Turbo algorithms. For example, model flip-flops with RTL in a primarily behavioral design.

For Verilog-XL, group statements having the same delay. This allows Verilog-XL to optimize the scheduling associated with these statements.
Minimize File Output System Tasks

Minimize your use of system tasks, especially for output:

// Less efficient                      // More efficient
always @(posedge clk)                always @(posedge error)
if (error)                            $display("ERROR");
   $display("ERROR");
else                                    else
   $display("OK");
Minimize File Output System Tasks

Minimize the use of statements that write to the disk or display to the terminal. Use them for interactive debugging, and shut them off during normal simulation.

You should also minimize the recording of waveform data. Do not get into the habit of arbitrarily capturing "everything" in the design, as the more data you capture, the more time your simulation requires. Not only do you significantly degrade simulation performance by making the simulator stop to write out data, that data is written to one of the slowest devices on a computer (the hard disk, or, even worse, a network connection to a remote hard disk).

Of course, in the debug phase of a design, you will need to capture waveform data. But even here you should only capture the data you need to debug a specific block of a design, not the "entire" design.

For regression testing, you should use self checking testbenches to monitor simulations rather than capturing simulation data for a manual, or even automated, review.
Keep Operations Vectored

// Less efficient

wire [7:0] lfsr;
reg [7:0] nlfsr;
always @(lfsr)
    if (lfsr & 8’h80)
        nlfsr = lfsr << 1 ^ 8’h63;
    else
        nlfsr = lfsr << 1;
DFF_RTL d[7:0] (lfsr,nlfsr,clk);

// More efficient

reg [7:0] lfsr;
reg [7:0] nlfsr;
always @(lfsr)
    if (lfsr & 8’h80)
        nlfsr = lfsr << 1 ^ 8’h63;
    else
        nlfsr = lfsr << 1;
always @(posedge clk) lfsr <= nlfsr;
Optimizing Behavioral Performance (continued)

Keep Operations Vectored

Event simulation CPU process time is approximately proportional to the total number of simulation events. Every simulation object and connection potentially creates additional events. Where practical, describe your design’s behavior as operations on vectors rather than operations on scalars.

The Cadence Verilog-XL simulator executes the vector version of the LFSR approximately 5 times faster than the scalar version.
Command-Line Simulation Control

You can use command-line options to customize the simulation:

- **Use the `+define+** option to define or redefine text macros
  ```
  ifdef MY_MACRO
  // compile if MY_MACRO is defined
  ...
  `else
  // compile if MY_MACRO not defined
  ...
  `endif
  ```

- **Use the `$test$plusargs` system function to detect user-defined plus options**
  - Make its argument the name of the plus option in question
  - It returns 1 (true) if the plus options exists, 0 otherwise
  - Cadence Verilog simulators assume misspelled plus options are your own plus options!
Command-Line Simulation Control

The Verilog-XL simulator sets text macros and performs any subsequent conditional compilation before it begins simulation. You can use conditional compilation to optionally reduce the size of your data structure and to select a mode of execution for a particular simulation run. This type of customization does not always efficiently port to compiled-code simulators such as the NC-Verilog simulator, that use precompiled libraries.

You can configure the simulation mode with user-defined command-line plus (+) arguments. You can detect the option with the $test$plusargs system function in your Verilog source code, or the mc_scan_plusargs library routine in your PLI source code.

Redefining text macros can defeat the incremental compilation capability. For NC-Verilog use $test$plusargs rather than `ifdef where practical.

The IEEE Std. 1364-1995 does not describe the $test$plusargs system task. This feature is specific to the Cadence Verilog-XL and NC-Verilog simulators.
Command-Line Simulation Control

```
verilog top.v +wav +define+accel+verbose=2 +define+stats=2

`timescale 1 ns / 1 ns
module top;
  reg a, b, c;
`ifdef accel
  and (tmp, b, c);
`else
  wire o = a | b & c;
`endif
initial
begin
  if ($test$plusargs("wav"))
  begin
    begin
      $shm_open ("waves.shm");
      $shm_probe ("AS");
      {a,b,c} = 0;
      repeat(7) #10
      {a,b,c} = {a,b,c} + 1;
    `ifdef verbose
      `$finish(`stats + 0);
    `else
      `$finish;
    `endif
  end
endmodule
```
Command-Line Simulation Control

Add 0 to text macros in expressions to ensure that a null macro value does not cause an error.

An engineer running the above example can choose:

- Whether to store value changes to an SHM database
- Whether to be verbose (display value changes to the screen), and if so, how verbose?
  - Verbosity level 0 (the default) monitors only the results (the output).
  - Verbosity level 1 monitors only the stimulus (the inputs).
  - Verbosity level 2 monitors all outputs and inputs in formatted columns.
- How much detail about the simulation to display
  - `$finish(0)$ displays information as follows:
    0 simulation events (use +profile or +listcounts option to count)
    CPU time: 0.6 secs to compile + 0.1 secs to link + 0.1 secs in simulation
  - `$finish(1)$ or `$finish$ also displays:
    L31 "top.v": $finish at simulation time 70
  - `$finish(2)$ also displays:
    Data structure takes 83788 bytes of memory

Note: Resource statistics reported by Verilog-XL are for the simulation only, not the system (file or terminal I/O, page faults). NC-Verilog reports actual process size and CPU time.
Code Coverage

Use code coverage tools to measure code activity at run time:

- **Statements** (execution, branches)
- **Variables** (specific changes, toggle)
- **Conditions** (evaluation, branches, execution paths)

Use code coverage to:

- Improve your design
  - Identify redundant or unexercisable paths
- Improve your testbench
  - Identify untested areas of the design
  - Identify inefficient testbench code
  - Identify unnecessary testbench code
Code Coverage

Code coverage tools run in conjunction with the simulator, and generate a report at the end of simulation. Simulators can embed code coverage recording capability, or the code coverage tools can either insert additional Verilog source code in a copy of your design, or monitor your design with the PLI.
Verilog-XL Statement Coverage

The Verilog-XL simulator offers integrated statement-coverage capability.

The event counter counts the number of times each line of code executes.

- Use the `+listcounts` option to enable this feature
- Use the `$listcounts` system task to report the execution count
Verilog-XL Statement Coverage

The Turbo option disables the event counter. Use the +listcounts option to re-enable it.

The $listcounts system task produces a line-numbered decompilation that includes an execution count — the number of times Verilog-XL executed the statements in each line.
Verilog-XL Statement Coverage

Example Source Code

module top;
   sub1 m1();
   sub2 m2();
   initial
      begin
         #1000
         $listcounts;
         $listcounts(m1);
         $listcounts(m2);
         $finish;
      end
endmodule

module sub1;
   reg r;
   initial begin
      r=0;
      forever
         r = ~r;
      end
endmodule

module sub2;
   reg r;
   initial begin
      r=1;
      forever
         r = ~r;
      end
endmodule

verilog +listcounts top.v
Verilog-XL Statement Coverage (continued)
Verilog-XL Statement Coverage

$\text{listcounts}

// top.v
  1 module top;
  2 sub1
  2 m1();
  3 sub2
  3 m2();
  4 initial
  5 begin
  6 #1000
  7 $\text{listcounts};
  8* $\text{listcounts}(m1);
  9 $\text{listcounts}(m2);
 10 $\text{finish};
11 end
12 endmodule

// top.v
 14 module sub1;
15 reg
15  r; // = 1'h1, 1
16 initial
17 begin
17  r = 0;
18 forever
19* #1
20 r = ~r;
21 end
22 endmodule
Verilog-XL Statement Coverage (continued)

The simulator writes the $listcounts output to the terminal and the log file. The output is a statement-by-statement record of the number of times the simulator executed each statement. The simulator marks with an asterisk (*) the line numbers of currently scheduled statements.
NC-Sim Code Coverage

The NC-Sim simulator offers integrated code-coverage capability:

- Use the `+nccoverage` option to enable this feature
- Use the `coverage` interactive command to start code coverage
- Use the `nccov` utility to report the code coverage
  - Statement coverage
  - State machine coverage
  - Expression coverage
Code Coverage

NC-Sim Code Coverage

Statement coverage reports whether a statement has executed. You can analyze statement coverage on either a design unit or an instance basis. You can also analyze the coverage of VHDL `generate` statements. The code coverage tool allows you to identify statements as *Required*, *Illegal*, or *Exclude*.

State machine coverage reports state visitation, state transitions, and paired state visitations. The code coverage tool allows you to identify each transition or paired state as *Required*, *Illegal*, or *Exclude*.

Expression coverage reports term contributions to true and false expression values. You can evaluate any included expression in either the *Standard* or the *Exclusive* (default) mode. Standard expression evaluation assigns Boolean values to each of the terms of the expression, from left to right, where needed, until it can determine the expression value. Exclusive expression evaluation establishes whether each input can independently propagate true and false values to the result.

For more details please refer to the Cadence online document *Code Coverage User Guide*. 
NC-Sim Code Coverage

Example Source Code

```verilog
module top;   module sub1;   module sub2;
sub1 m1();   reg r;         reg r;
sub2 m2();   initial begin   initial begin
    initial r=0;         r=1;
    begin              forever
        #1000 #1 r = ~r;
        $finish;          #2 r = ~r;
        end              end
    end              endmodule
endmodule
```

ncverilog +nccoverage +nclinedebug +ncinput+@"coverage;run;exit" top.v
NC-Sim Code Coverage (continued)

Use the `+nccoverage` option to enable the code coverage feature.

Use the `+nclinedebug` option to preserve debug access to source lines for expression coverage. Expression coverage requires that you preserve this form of access. NC-Verilog by default provides no interactive access to simulation components.

Use the `+ncaccess+r` option to preserve read access to Verilog signals for state machine coverage. State machine coverage requires that you preserve this form of access. NC-Verilog by default provides no interactive access to simulation components.

Use the `+ncinput` option to provide a script of interactive simulation commands. You can provide either a script filename, or a script string prefaced with the '@' character.

Use the `coverage` interactive simulation command to start code coverage. The `coverage` command by default starts recording statement and expression coverage for the entire design. You can optionally provide a subscope name, and can use the `coverage` command `-statement` or `-expression` options to restrict coverage to only statements, or only expressions. For state machine coverage you use the `-fsm` option to provide the state variable name, and the `-global_event` option to specify the signal edge where the simulator samples the variable value.
NC-Sim Code Coverage

nccov -batch -stmt cover.cov

Generated by nccov 3.40.(p1) on 12-17-2001 at 12:16

Design Name: cover.cov/top-1
Run/RunSet: ALLRUNS is 1
Report Type: Instance Summary Failing
Sort Criteria: SortByName

Selections used in this analysis:
defselect -stmt . -depth all -required

<table>
<thead>
<tr>
<th>Instance Name</th>
<th>Category</th>
<th>Total Lines</th>
<th>Problem Lines</th>
<th>%Cover</th>
</tr>
</thead>
<tbody>
<tr>
<td>top(@worklib.top)</td>
<td>Required</td>
<td>7</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>top.m1(@worklib.sub1)</td>
<td>Required</td>
<td>3</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>top.m2(@worklib.sub2)</td>
<td>Required</td>
<td>3</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>
NC-Sim Code Coverage (continued)

Use the `nccov` code coverage analysis tool to analyze information the simulator writes to a code coverage database. You can use the tool in either the textual or graphical mode, to report summary or detailed information on statement, expression, and FSM coverage. The tool offers wide flexibility with respect to the coverage elements to report and the format of the report.

The tool by default analyzes all simulation runs in the database, and produces a summary report of all failing statements, sorted by instance name. The tool by default assumes that the simulator should execute all instrumented statements at least once.

This example restricts the summary report to statements, and the report indicates that all statements have executed.
Summary

In this presentation you learned about:

- The XL and Turbo Algorithms
- Verilog-XL and NC-Verilog performance
- Optimizing behavioral performance
- Command-line simulation control
- Code coverage
Summary
Review

1. What options would you probably use for highest simulation performance?

2. Name an item normally accelerated by the Verilog-XL XL algorithm that can become unaccelerated during simulation.

3. True or false: Probing waveform data can significantly negatively impact simulation performance.

4. What do code coverage tools do?
Review

1. What options would you probably use for highest simulation performance?

   Use the Verilog-XL +twin_turbo and +turbo+3 options to obtain highest simulation performance for most designs. Use the +caxl option if you have more than a few continuous assignments in a primarily gate-level design. The NC-Verilog simulator executes in the highest performance mode by default.

2. Name an item normally accelerated by the Verilog-XL XL algorithm that can become unaccelerated during simulation.

   An expandable zero-delay net is normally accelerated by the XL algorithm, but becomes unaccelerated if forced during simulation.

3. True or false: Probing waveform data can significantly negatively impact simulation performance.

   Unnecessary probing of waveform data may be the single testbench feature most negatively impacting simulation performance.

4. What do code coverage tools do?

   Code coverage is a measurement of how completely the testbench exercises the device under test.
About Lab 15

The objective of this lab is to simulate several versions of a shift register, noting the performance differences among them.

In this lab you will run a series of simulations that demonstrate the effects of coding for performance. You will:

- Simulate several versions of a shift register design with different simulator options
- Note and compare the performance differences among the various shift register designs and simulator options

### Shift Register

![Shift Register Diagram]
About Lab 15
Chapter 21: Annotating SDF Timing

Objectives

In this section you will learn about:

- Delay calculators
- The Standard Delay Format (SDF)
- Annotating SDF data
Annotating SDF Timing

This section introduces delay calculation and annotation, and the standard delay format (SDF). It presents small examples of correct SDF files, and explains how to annotate this information with the Cadence Verilog-XL and NC-Verilog simulators.
Timing Annotation

A silicon vendor simulation library typically contains estimated intrinsic timing. For accurate timing simulation you need additional data:

- Drive strength
- Interconnect parasitics
- Total load
- Environmental factors
  - process
  - temperature
  - voltage

You also need to simulate fast clock with slow data and slow clock with fast data. Most event simulators cannot directly do this.
Timing Annotation

The Verilog language permits you to specify minimum, typical, and maximum timing information, but you can select only one of these sets of timing information for a given simulation run. 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

- Design Capture
- Functional Simulation
- Logic Synthesis
- Gate-level Simulation
- Timing Analysis
- Floorplanning Place & Route
- SPF Data
- Delay Calculator
- $sdf_annotate
- SDF Data

SDF Data

Annotating SDF Timing
Timing Data Flow

The delay calculator requires:

- A structural representation of the design
- Reduced parasitics from layout tools

The delay calculator can generate:

- Estimated delays, based only upon design connectivity and hierarchy
- Detailed delays, using parasitic information extracted by layout tools

Several synthesis tools now include some level of built-in layout tools
Many EDA tools accept SDF.
Delay Calculators

Two major categories of delay calculators exist:

- Delay calculators embedded in the tools
  - The Ambit BuildGates synthesis tool
  - The Pearl timing analyzer

- Custom delay calculators
  - User-defined
  - Vendor-supplied

Delay calculators can generate SDF data, or directly annotate timing data with the PLI.
Delay Calculators

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. These are often written in C and use the PLI to directly annotate the design during simulation, although they can also be standalone programs that generate SDF for the built-in timing data annotator.
**Standard Delay Format**

SDF provides a tool-independent, uniform way to specify timing information.

It can specify absolute or incremental delays (which can be conditional), timing checks (which can also be conditional), and timing constraints:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NETDELAY</td>
<td>Wire delay for all paths on the net</td>
</tr>
<tr>
<td>PORT</td>
<td>Wire delay for all paths to an input port</td>
</tr>
<tr>
<td>INTERCONNECT</td>
<td>Wire delay for a specific path</td>
</tr>
<tr>
<td>DEVICE</td>
<td>Module delay for all paths to an output</td>
</tr>
<tr>
<td>IOPATH</td>
<td>Module delay for a specific path</td>
</tr>
<tr>
<td>TIMINGCHECK</td>
<td>Setup, Hold, ..., Skew, Width</td>
</tr>
<tr>
<td>TIMINGENV</td>
<td>Path constraints, ... skew constraints</td>
</tr>
</tbody>
</table>
Standard Delay Format

You cannot specify interconnect delay in a **specify** block. To simulate with interconnect delay, you must annotate timing. Most of the information in a typical SDF file is interconnect delay.

Be aware that annotating a large number of interconnect delays can significantly increase simulation memory requirements. This is because SDF annotated interconnect delays actually add new elements to the simulation data structure.

Verilog-XL by default simulates with inertial delays and treats INTERCONNECT delays as PORT delays.

Use the Verilog-XL `+transport_int_delays` option to simulate with transport interconnect delays.

Use the Verilog-XL `+multisource_int_delays` option to accommodate multiple sources with different delays.
SDF Example

(DELAYFILE
  (DESIGN "system")
  (DATE "Mon Dec 17 11:21:44 EST 2001")
  (VENDOR "Cadence")
  (PROGRAM "delay_calc")
  (VERSION "v03.40.(p002)")
  (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))))
)
Standard Delay Format (continued)

SDF Example

You place SDF file configuration information in the header.

You can specify delays for all instances of a cell type or for specific instances.

You can specify absolute or incremental delays.

You can specify typical delays (a single value) or \textit{min:typ:max} delay triplets. If you omit a value from a delay triplet, that delay will default to the delay specified in the original Verilog HDL source. The example illustrates this omission.
The SDF Annotator

Use the `$sdf_annotate` system task to annotate SDF timing information.

Verilog-XL also permits you to invoke this system task interactively:

```bash
$sdf_annotate (sdf_file, [module_instance, config_file, log_file, mtm_spec, scale_factors, scale_type ]);
```

**Note**: The only required argument is `sdf_file`. 
The SDF Annotator

sdf_file — The full or relative path of the SDF file

module_instance — The scope at which to perform annotation. The default is the scope at which you invoke `$sdf_annotate`

config_file — The full or relative path of the configuration file. The default is to use preset settings

log_file — The log file name. The default is `sdf.log`. You can use the `+sdf_verbose` option to generate a log file.

mtm_spec — Select the timing value set for annotation; one of `{ MINIMUM, TYPICAL, MAXIMUM, TOOL_CONTROL }`. The default is TOOL_CONTROL (command-line option). This argument overrides the configuration file MTM keyword.

scale_factors — Scale factors in min:typ:max format; default is 1.0:1.0:1.0. This argument overrides the configuration file SCALE_FACTORS keyword.

scale_type — Select the timing value set for scaling; one of `{ FROM_MINIMUM, FROM_TYPICAL, FROM_MAXIMUM, FROM_MTM }`. The default is FROM_MTM. This argument overrides the configuration file SCALE_TYPE keyword.

Note: The annotator applies the `scale_factors` as directed by the `scale_type` to generate a new MTM timing triplet, then selects a number from that triplet based upon the `mtm_spec`. 
Running the SDF Annotator

The following example annotates scaled SDF data to the top-level design:

module top;
    . . .
    initial
        $sdf_annotate ("my.sdf",,,,1.6:1.4:1.2);
    . . .
endmodule

The following example annotates separate SDF data to different instances:

module top;
    . . .
    cpu u1 ( . . .
    fpu u2 ( . . .
    dma u3 ( . . .
    . . .
    initial
        begin
            $sdf_annotate ("sdffiles/cpu.sdf",u1,,,"logfiles/cpu_sdf.log");
            $sdf_annotate ("sdffiles/fpu.sdf",u2,,,"logfiles/fpu_sdf.log");
            $sdf_annotate ("sdffiles/dma.sdf",u3,,,"logfiles/dma_sdf.log");
        end
    . . .
endmodule
The SDF Annotator (continued)

Running the SDF Annotator

These are some of the Verilog-XL command-line options related to the SDF annotator:

<table>
<thead>
<tr>
<th>Option</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+annotate_any_time</td>
<td>Allows SDF annotation to occur after time 0</td>
</tr>
<tr>
<td>+sdf_cputime</td>
<td>Log the number of CPU seconds for annotation</td>
</tr>
<tr>
<td>+sdf_error_info</td>
<td>Display PLI annotator error messages</td>
</tr>
<tr>
<td>+sdf_file filename</td>
<td>Override the file name in the $sdf_annotate system task</td>
</tr>
<tr>
<td>+sdf_no_errors</td>
<td>Disable SDF annotator error messages</td>
</tr>
<tr>
<td>+sdf_no_warnings</td>
<td>Disable SDF annotator warning messages</td>
</tr>
<tr>
<td>+sdf_nocheck_celltype</td>
<td>Disable instance-by-instance cell type validation</td>
</tr>
<tr>
<td>+sdf_nomsrc_int</td>
<td>Inform annotator you have no MITDs; increases performance</td>
</tr>
<tr>
<td>+sdf_VERBOSE</td>
<td>Log detailed annotation process information</td>
</tr>
</tbody>
</table>

Note: The NC-Verilog simulator does not accept the +annotate_any_time, +sdf_error_info, and +sdf_file options.
Summary

In this section you learned about:

- Delay calculators
- The Standard Delay Format (SDF)
- Annotating SDF data
Summary
Review

1. What factors affect device timing?
2. True or false: Some delay calculators can directly annotate the calculated delay into the simulation.
3. Which SDF keyword denotes source-to-sink interconnect delay?
4. What are the required $sdf_annotate system task arguments?
Review

1. What factors affect device timing?
   
   Factors that affect device timing include drive strength, slew rate, interconnect parasitics, fanout load, and the environment (process, temperature, and voltage).

2. True or false: Some delay calculators can directly annotate the calculated delay into the simulation.

   Some delay calculators produce an SDF file for the simulator to annotate, and some delay calculators directly annotate the calculated timing data into the simulation.

3. Which SDF keyword denotes source-to-sink interconnect delay?

   The SDF INTERCONNECT keyword denotes source-to-sink interconnect delay.

4. What are the required $sdf_annotate system task arguments?

   The only required $sdf_annotate system task argument is the SDF file name. The Verilog-XL simulator permits you to override this file name with a command-line option.
About Lab 16

The objective of this lab is annotate timing data.

In this lab, you will:

- Simulate using annotated timing data and maximum delays
- Simulate using annotated timing data and minimum delays
- Examine and explain any reported timing violations
About Lab 16
Chapter 22: Overview of the NC-Verilog Simulator

Objectives

In this section, you will learn about:

- Interleaved Native Compiled-Code Architecture (INCA)
- Support of the Verilog Hardware Description Language (HDL)
- Setting up your design environment
- Running the simulator
Overview of the NC-Verilog Simulator

This section presents a brief overview of the Cadence NC-Verilog simulator. It presents the Interleaved Native-compiled Code Architecture (INCA), defines support of the Verilog language standard, and describes setting up and running the NC-Verilog simulator.
Interleaved Native Compiled Code

Extensions to Native Compiled Code address the performance challenges of a single-simulation strategy:

- Multiple representations — behavioral / RTL / gate / UDP
- Multiple languages — Verilog / VHDL
- Multiple paradigms — Event / Cycle
- Mixed signal — Analog / Digital
Interleaved Native Compiled Code

These are some of the performance challenges of a single-simulation strategy:

- Design size and complexity demand multiple parallel approaches to design verification simulation. For example, the designer of a particular ASIC might configure an overall system-level model that uses behavioral representations of other ASICs with an RTL representation of the ASIC undergoing design.

- Some portion of the design might instantiate pre-existing third-party intellectual property, such as an embedded CPU or bus interface core, that is only available in a second simulation language.

- Verification of design functionality is a problem of 4th-order complexity. As design geometries halve, both the number of cells and the required simulation test vectors quadruple. Design teams need to develop new hardware descriptions that accommodate high-performance cycle-based simulation techniques while maintaining compatibility with large inherited testbench suites that do not accommodate cycle-based simulation.

- It is preferable to maintain a single strategy for capture, control, and verification of design intent. Many design teams capture the entire system-wide architecture, including both digital and analog components, in an HDL. The ability to use a single simulator to simultaneously verify any combination of design elements enhances the productivity of the team.
How Native Compiled Code is Interleaved

- Verilog
  - Compile
  - VST

- VHDL
  - Compile
  - AST

- Simulator-X
  - Compile

Elaborate

Update

- Time update code

Verilog
- VHDL
- Simulator-X

Event scheduling kernel (distributed)
Interleaved Native Compiled Code (continued)

With INCA, all supported simulation styles leverage a single high-performance engine. Compilers for each input language or format create an optimized sequence of instructions that are interleaved to create a single contiguous code stream. This code stream is, in effect, a simulation engine customized for the specifically configured blend of simulation languages and techniques.

This approach lays the foundation for mixed-language, mixed-paradigm, and mixed-signal simulations. Event scheduling tasks are removed from the simulation kernel and distributed into the individual code streams. A special time management code stream, based on the chosen languages, models, and scheduling methods, is created and interleaved with the behavior and event scheduling streams.
Library-Based Design Data Management

Module Hierarchy

mychip

mod1

mod2

File | Cell | View | Reference
--- | --- | --- | ---
mychip.v | mychip | structural | worklib.mychip:module
mod1.vb | mod1 | behavioral RTL | worklib.mod1:beh
mod1.vr | mod1 | RTL | worklib.mod1:rtl
mod2.vg | mod2 | gate RTL | worklib.mod2:gate
mod2.vr | mod2 | RTL | worklib.mod2:rtl

Directory Structure

sources/

mychip.v

mod1.vb

mod1.vr

mod2.vg

mod2.vr

Library

Cells

mychip

mod1

mod2

module

beh

rtl

rtl

gate

Views

Cell References

File | Cell | View | Reference
--- | --- | --- | ---
mychip.v | mychip | structural | worklib.mychip:module
mod1.vb | mod1 | behavioral RTL | worklib.mod1:beh
mod1.vr | mod1 | RTL | worklib.mod1:rtl
mod2.vg | mod2 | gate RTL | worklib.mod2:gate
mod2.vr | mod2 | RTL | worklib.mod2:rtl
Library-Based Design Data Management

The NC-Verilog simulator stores intermediate design data in libraries. It organizes library structures using the *Library.Cell:View* approach, where:

- A library is a cohesive collection of associated simulation modules
- A cell is a specific simulation module
- A view is a specific representation of a module

The library used for the current design work is called the *work* library. The NC-Verilog simulator, by default, stores derived data (such as compiled design units) in the *work* library.

Each library has a unique logical name and is represented by a unique UNIX directory. Inside that directory is a compressed binary `.pak` file, whose contents you can list with the `ncls` utility.

You can create each library directory and enter, in the `cds.lib` file, the mapping between the logical library name and the physical directory path for each library.

You can define the `work` library either on the command line as you use the tools or in the `hdl.var` file as the WORK variable.

You can create a `setup.loc` file to change the search path order for the `cds.lib` and `hdl.var` files.

You don’t need to do any of these things. If you don’t, the simulator simply uses default values.
Running the NC-Verilog Simulator

Running the NC-Verilog simulator is separated into three major steps:

1. **ncvlog** (compilation)
   - Performs syntax and static semantics checks
   - Creates Verilog Syntax Tree (VST)

2. **ncelab** (elaboration: expansion and linking)
   - Constructs design hierarchy as configured
   - Establishes signal connectivity
   - Creates Signature (SIG)
   - Invokes code generator
   - Creates executable code (COD)
   - Creates time-zero Simulation SnapShot (SSS)

3. **ncsim** (simulation)
   - Executes simulation code

**Note:** Separation of steps allows iteration of only the required steps!
Running the NC-Verilog Simulator

A VST is an intermediate data structure containing linked tokens (such as identifiers and operators) that represents one original Verilog source module in an efficiently accessible and interpretable format. There is one VST for each Verilog module definition.

A SIG is a unique signature of a Verilog module. There is one SIG for each unique Verilog module instantiation. A SIG contains resolutions of:

- Instantiation parameters (such as port widths)
- References to external (out of module) identifiers (such as task calls)

A COD is machine-specific executable code. There is one COD for each VST-SIG pair that contains behavioral Verilog source code.

An SSS is a snapshot of the state of the entire elaborated design hierarchy. An SSS contains:

- Current values (reg/wire)
- Current process state
  - Sensitized structures
  - Execution point
- Current simulation state
  - Simulation time
  - Scheduled events and methods
  - File status
The NC-Verilog Tool Flow

Path Legend:
- primary
- transparent
- optional
The NC-Verilog Tool Flow

You can invoke the individual tools on the primary path, but not on the transparent paths. For example, if you invoke the elaborator, it automatically spawns the code generator. You do not need to invoke any of these tools individually. If you run `ncverilog`, all of these tools are automatically run for you in the proper sequence.

Transparent paths indicate that:

- The code generator uses the VST and SIG
- The simulator’s user interface uses the VST

**Note:** A *method* is a procedure associated with an object. For example, a routine that is executed upon each change of signal state is considered a *method* of the signal.
Summary

In this section, you learned about:

- Interleaved Native Compiled-Code Architecture (INCA)
- Support of the Verilog Hardware Description Language (HDL)
- Setting up your design environment
- Running the simulator
Summary
Review

1. True or False: With the Interleaved Native Compiled Code Architecture, a single executable simulation process handles all the different kinds of simulation.

2. Where do your original Verilog source files reside in relationship to the Library.Cell:View library of derived simulation objects?

3. Which of the NC-Verilog software tools spawns the code generator?

4. How many VST (Verilog Syntax Tree) objects will a library contain?
Review

1. True or False: With the Interleaved Native Compiled Code Architecture, a single executable simulation process handles all the different kinds of simulation.
   Within INCA all contemplated simulation disciplines execute within a single process.

2. Where do your original Verilog source files reside in relationship to the Library.Cell:View library of derived simulation objects?
   You can store the Verilog source description anywhere that is accessible to ncvlog. To avoid any possible future conflicts, you may want to store your sources outside of the NC-Verilog library directories.

3. Which of the NC-Verilog software tools spawns the code generator?
   The code generator is invoked by the ncelab elaborator as needed. This operation is transparent to the user.

4. How many VST (Verilog Syntax Tree) objects will a library contain?
   A VST is an intermediate data structure containing linked tokens (such as identifiers and operators) that represents one original Verilog source module in an efficiently accessible and interpretable format. There is one VST for each Verilog module definition.
Objectives

In this section, you will learn about:

- Setting up your local design environment to run the NC-Verilog simulator.
Getting Started with the NC-Verilog Simulator

This section introduces and explains the *cds.lib*, *hdl.var*, and *setup.loc* files. It describes the *hdl.var* variables you set to direct the simulation. It also introduces the *nchelp*, *ncls*, and *ncrm* utilities, which you will use in the lab associated with this section.
Creating the cds.lib File

The *cds.lib* file:

- Is an ASCII file (easily edited)
- Tells the NC-Verilog simulator which libraries may be used
- Tells the NC-Verilog simulator where the libraries are located
- Maps logical library names to physical directory locations
- Is used by all NC-Verilog tools

This is an example of a *cds.lib* file:

```
INCLUDE /users/${USER}/cds.lib
DEFINE aludesign ./design
DEFINE ic_lib /usr1/libraries/lsi_library
ASSIGN ic_lib TMP ./tmp_lib
```

This example *cds.lib* file includes a *cds.lib* file from the home directory and defines two additional libraries. It temporarily attaches a local directory to the *ic_lib* library, into which NC-Verilog places derived *ic_lib* data.

Please refer to the Cadence online document *NC-Verilog Simulator Help* for additional information concerning *cds.lib* syntax rules.
Creating the cds.lib File

Keywords in the cds.lib File

You can use the following keywords in a cds.lib file:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFINE library path</td>
<td>Defines library as the directory specified in path</td>
</tr>
<tr>
<td>UNDEFINE library</td>
<td>Undefines library (useful if defined in another file)</td>
</tr>
<tr>
<td>INCLUDE file</td>
<td>Includes the contents of file in the cds.lib file</td>
</tr>
<tr>
<td>SOFTINCLUDE file</td>
<td>Includes the contents of file in the cds.lib file, but does not report a missing file</td>
</tr>
<tr>
<td>ASSIGN library attribute value</td>
<td>Assigns attribute and its value to library</td>
</tr>
<tr>
<td>UNASSIGN library attribute</td>
<td>Removes attribute from library</td>
</tr>
</tbody>
</table>

Note: You may specify a directory in no more than one DEFINE statement. You may provide either absolute or relative paths. Relative paths are relative to the location of the cds.lib file.

The NC-Verilog simulator currently specifies only the TMP attribute. Use the TMP attribute to bind a second temporary directory to a previously defined library. The temporary directory will contain only new derived data, not source files. This allows multiple designers to reference a common library, but store compiled objects in their own design directories.
Creating the hdl.var File

The *hdl.var* file:

- Is an ASCII file (easily edited)
- Contains variables
  - Default configuration variables, such as WORK
  - File name to library name mapping, such as LIB_MAP
  - Default command line options, such as NCVLOGOPTS
  - File name variables, such as NCSIMRC

This is an example of an *hdl.var* file:

```bash
INCLUDE /users/${USER}/hdl.var
DEFINE WORK my_lib
DEFINE VERILOG_SUFFIX ( .v, .vr, .vb, .vg)
```

This example *hdl.var* file includes the *hdl.var* file in the home directory, defines the *work* library to be *my_lib*, and defines several valid Verilog source file suffixes.

Please refer to the Cadence online document *NC-Verilog Simulator Help* for additional information concerning *hdl.var* syntax rules.
Creating the hdl.var File

Keywords in the hdl.var File

You can use the following keywords in a hdl.var file:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFINE variable value</td>
<td>Defines a variable and assigns a value to it</td>
</tr>
<tr>
<td>UNDEFINE variable</td>
<td>Undefines a variable (use if defined in an included file)</td>
</tr>
<tr>
<td>INCLUDE file</td>
<td>Includes the contents of file in the hdl.var file</td>
</tr>
<tr>
<td>SOFTINCLUDE file</td>
<td>Includes the contents of file in the hdl.var file, but does not report a missing file</td>
</tr>
</tbody>
</table>

The NC-Verilog simulator, by default, stores derived data, such as compiled versions of design units, in the work library. You define the work library by assigning a value to the WORK variable. You may alternatively specify the work library on the command line using the -work option, which overrides the setting of the WORK variable.
Variables in the hdl.var File

Here is an example of a larger hdl.var file:

DEFINE NCELABOPTS -messages -errormax 10
DEFINE NCSIMOPTS -messages -errormax 10
DEFINE NCSIMRC ./my_ncsimrc
DEFINE NCVLOGOPTS -messages -errormax 10 -file ./proj_file
#DEFINE VIEW behav
#DEFINE WORK worklib

#specify to ncvlog the list of valid Verilog source file extensions
DEFINE VERILOG_SUFFIX (.v, .vr, .vb, .vg)

#specify to ncvlog the libraries into which to compile sources
#specify to ncelab the library search order for instance resolution
DEFINE LIB_MAP (myfile.v => mylib, ./cell_lib => techlib, + => worklib)

#specify to ncvlog the views into which to compile sources
#specify to ncelab the view search order for instance resolution
DEFINE VIEW_MAP (.vb => behav, .vr => rtl, .vg => gate, + => module)

Please refer to the Cadence online document NC-Verilog Simulator Help for additional information concerning hdl.var variables.
Creating the hdl.var File (continued)

Variables in the hdl.var File

This is a partial list of variables you can set in a hdl.var file:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB_MAP</td>
<td>Maps files and directories to library names</td>
</tr>
<tr>
<td>NCELABOPTS</td>
<td>Sets elaborator command line options</td>
</tr>
<tr>
<td>NCSIMOPTS</td>
<td>Sets simulator command line options</td>
</tr>
<tr>
<td>NCSIMRC</td>
<td>Executes a file of run-time commands upon invocation</td>
</tr>
<tr>
<td>NCVLOGOPTS</td>
<td>Sets compiler command line options</td>
</tr>
<tr>
<td>SRC_ROOT</td>
<td>Defines the root directory of Verilog source files</td>
</tr>
<tr>
<td>VERILOG_SUFX</td>
<td>Defines valid source file extensions</td>
</tr>
<tr>
<td>VIEW</td>
<td>Sets the view name</td>
</tr>
<tr>
<td>VIEW_MAP</td>
<td>Maps file extensions to view names</td>
</tr>
<tr>
<td>WORK</td>
<td>Defines the current work library</td>
</tr>
</tbody>
</table>

**Note:** You cannot include -append_log, -cdslib, -hdlvar, or -logfile option in a tool option variable.
Creating the Optional setup.loc File

The setup.loc file:

- Is an ASCII file (easily edited)
- Specifies an alternative search order for the cds.lib and hdl.var files

The default search order is:

1. Current directory
2. $CDS_WORKAREA
3. $CDS_SEARCHDIR
4. $HOME
5. $CDS_PROJECT
6. $CDS_SITE
7. your_install_dir/share

Note: The NC-Verilog simulator uses only the first cds.lib and hdl.var files found during the search, but you can use the INCLUDE keyword to explicitly include additional files.
Creating the Optional setup.loc File

This feature is provided to facilitate administration of the design environment. You can customize the order in which the NC-Verilog simulator searches for *cds.lib* and *hdl.var* files. You can also explicitly specify alternative *cds.lib* and *hdl.var* files as NC-Verilog command-line options.

Although you can change the directory search order for *cds.lib* and *hdl.var* by editing the *setup.loc* file, you cannot change the directory search order for the *setup.loc* file itself:

1. `$CDS_SITE/cdssetup` — your site’s version
2. `your_install_dir/share/cdssetup` — original version

Thus, the *setup.loc* file must reside in one of the locations listed above.

**Note:** Unlike the *cds.lib* file, relative paths in *setup.loc* are relative to the current working directory, not to the location of the *setup.loc* file. If the simulator cannot access a directory specified in a *setup.loc* file, it advances to succeeding locations without issuing a warning or error.
Using the nchelp Utility

Use **nchelp** in two different ways:

- To explain tool error and warning messages
  
  Syntax: `nchelp [ -options ] tool_name message_code`
  
  ```
  nchelp ncvlog BADCLP
  ```

- To print the contents of *cds.lib* and *hdl.var* files
  
  Syntax: `nchelp [ -options ] { -hdlvar | -cdslib } [ filename ]`
  
  ```
  nchelp -hdlvar ~/proj/cpu/hdl.var
  ```
Using the nchelp Utility

This example explains the `ncvlog` BADCLP message

```bash
% nchelp ncvlog BADCLP
ncvlog/BADCLP =
  The path specified for the -CDSLIB argument is invalid.
  Check that the specified path exists and is readable.
```

This example prints the variables defined in the local `hdl.var` file

```bash
% nchelp -hdlvar
Parsing -HDLVAR file ./hdl.var.
hdl.var files:
  1: ./hdl.var
Variables defined:
Defined in ./hdl.var:
Line #    Name                  Value
-------  --------              -----
  4  LIB_MAP ( . => projectlib , ./cell_lib => techlib )
  1  WORK                   lab2reglib
  2  NCVLOGOPTS              -messages
  3  VIEW                   behav
  5  VIEW_MAP ( + => module , .vg => gate , .vr => rtl )
```
Using the ncls Utility

Use **ncls** to list the derived objects stored in the library system

Syntax: ncls [ -options ] [lib.][cell][:view]

Use the `cell` argument to list all objects of a specific cell

   ncls top

Use the `library` argument or option to list all objects in a specific library

   ncls projlib.
   ncls -library projlib

Use the `view` argument or option to list all objects with a specific view

   ncls :rtl
   ncls -view rtl

Use the `-all` option to list all objects in all libraries

   ncls -all
Using the ncls Utility

This example lists the contents of the INCA_libs libraries

```
% ncls -cdslib INCA_libs/cds.lib -hdlvar INCA_libs/hdl.var
    module worklib.drv:v (VST)
    module worklib.drv:v (SIG) <0x6e18694e>
    module worklib.dut:v (VST)
    module worklib.dut:v (SIG) <0x05b84916>
    module worklib.top:v (VST)
    module worklib.top:v (SIG) <0x3e8e45bc>
    module worklib.tst:v (VST)
    module worklib.tst:v (SIG) <0x79eac46c>
    module worklib.tst:v (COD) <0x79eac46c>
    snapshot worklib.top:v (SSS)
```
Using the ncrm Utility

Use **ncrm** to delete the derived objects stored in the library system

Syntax: `ncrm [-options] [lib.]cell[:view]`

Just enter the arguments to remove one or all views of a specified cell

```
ncrm project_lib.top
```

Use the **-snapshot** option to remove only the snapshot from one or all views

```
ncrm -snapshot project_lib.top:tmp_snap
```

Use the **-library** option to completely remove a specified library

```
ncrm -library project_lib
```

**Note:** You can omit the library specification if the cell exists in only one library.
Using the ncrm Utility
Summary

In this section, you learned about setting up your local design environment to run the NC-Verilog simulator.
Summary
Review

1. What is the purpose of the `cds.lib` DEFINE keyword?
2. What does the `hdl.var` WORK variable define?
3. In what location does the simulator search first for the `hdl.var` file?
4. How would you use `nchelp` to list all defined libraries?
Review

1. What is the purpose of the *cds.lib* DEFINE keyword?
   Use the DEFINE keyword in the *cds.lib* file to define a library and map it to a filesystem directory.

2. What does the *hdl.var* WORK variable define?
   Use the WORK variable to define your working library. Define the WORK variable in the *hdl.var* file and give it a value that is the logical name of the work library.

3. In what location does the simulator search first for the *hdl.var* file?
   Unless otherwise reordered in the *setup.loc* file, the NC-Verilog simulator will first look for the *cds.lib* and *hdl.var* files in the current directory. It will stop looking when a version of the file is found. To use more than one *cds.lib* file or *hdl.var* file, you must specify the additional files with the INCLUDE or SOFTINCLUDE keywords.

4. How would you use *nchelp* to list all defined libraries?
   Use the *nchelp –cdslib* option to list the contents of the *cds.lib* file(s). With no option argument, *nchelp* lists the contents of all applicable *cds.lib* files.
About Lab 17

The objective of this lab is to become familiar with the NC-Verilog library structure and tool invocation.

In this lab, you will simulate a small sample design with the NC-Verilog simulator.
About Lab 17
Chapter 24: Transitioning to the NC-Verilog Simulator

Objectives

In this section, you will learn about:

- Transitioning an existing Verilog-XL simulator design environment to the NC-Verilog simulator
- The trade-offs among the transition methods
- Incremental compilation
Transitioning to the NC-Verilog Simulator

This section describes the three methods with which to transition a design from Verilog-XL to NC-Verilog: with `ncverilog`, with `ncprep`, or manually. It also discusses the advantages and disadvantages of each method, and the situations in which one or another method is most appropriate. This section takes a "cookbook" orientation, covering step-by-step what you need to do.
Transitioning a Design to the NC-Verilog Simulator

There are three ways to transition your Verilog-XL simulator environment to the NC-Verilog simulator:

- Run `ncverilog`
- Transition with the help of `ncprep`
  - First you run `ncprep`, then you customize the new environment
- Transition manually
  - You restructure the design environment yourself
Transitioning a Design to the NC-Verilog Simulator
Using the ncverilog Command

With ncverilog, you maintain only your existing source control methodology when you transition to the NC-Verilog simulator.

In this section, we:

■ Introduce ncverilog
■ Show you how ncverilog works
■ Show you how to run ncverilog
Using the `ncverilog` Command

Prior to NC-Verilog version 1.2, you transitioned your existing Verilog-XL simulator design environment to an NC-Verilog environment manually, with or without using `ncprep`. Users of NC-Verilog reported that:

- The manual transition can be time-consuming to users of the Verilog-XL simulator
- The use of `ncprep` can require significant manual intervention for highly interdependent design data files
- The library-based use model is not appropriate for Verilog-XL users who want to maintain only their existing source control methodology

These persons can use `ncverilog`. 
Introducing ncverilog

Let us address the initial questions first:

- What is ncverilog?
- What does ncverilog do?
- Should you use ncverilog?
Using the ncverilog Command (continued)

Introducing ncverilog
Introducing ncverilog

What is ncverilog?

ncverilog automatically and almost invisibly transitions your Verilog-XL simulator design simulation environment into an NC-Verilog simulator design environment:

- Requires only a single command (ncverilog)
- Accepts Verilog-XL simulator command line options and arguments
- Uses the same library search order as the Verilog-XL simulator
- Duplicates the binding rules of the Verilog-XL simulator
- Propagates compiler directives and text macros as the Verilog-XL simulator
Introducing nCVerilog (continued)
Introducing ncverilog

What Does ncverilog Do?

ncverilog does the following:

- Creates the *cds.lib* and *hdl.var* files
- Creates the NC-Verilog library structure for compiled elements
- Translates Verilog-XL options into tool-specific NC-Verilog options
- Sequentially invokes the NC-Verilog simulation tools as needed
  - The *ncvlog* compiler
  - The *ncelab* elaborator
  - The *ncsim* simulator
Introducing ncverilog (continued)
Introducing ncverilog

Should You Use ncverilog?

Use ncverilog if you use the Verilog-XL simulator and you:

- Want the incremental compilation capability of the NC-Verilog simulator
- Want the improved simulation performance of the NC-Verilog simulator
- Want to maintain only your existing source control methodology
- Expect to use the Verilog-XL simulator again to simulate your design project
Introducing ncverilog (continued)
Understanding ncverilog

ncverilog responds to these situations differently:

- The first time you invoke it to simulate a design
- All subsequent invocations to simulate the same design hierarchy
Using the ncverilog Command (continued)

Understanding ncverilog
Understanding ncverilog

First-Time Behavior

The first time you run ncverilog, it:

- Creates the INCA_libs directory
- Creates the INCA_libs/cds.lib and INCA_libs/hdl.var files
- Creates the INCA_libs/worklib work library directory
- Creates the INCA_libs/snap.nc scratch directory
- Creates the INCA_libs/snap.nc/cds.lib and INCA_libs/snap.nc/hdl.var files
- Writes the SNAPSHOT variable to INCA_libs/snap.nc/hdl.var — It stores the snapshot name for future reference.

more on the next page...

Note: This behavior assumes you have simply invoked ncverilog instead of verilog, but that all the other options remain identical. You can also include options specific to ncverilog, but this is not shown here.
Understanding ncverilog (continued)

The ncverilog utility uses the Verilog-XL simulator command line parser to determine the order in which to search the source library files and directories.

The ncverilog utility uses the INCA_libs/snap.nc directory to pass data between tools. This directory contains invocation-specific information related to the most recently built snapshot. You can use the +name+<snapshot_name> option to specify the snapshot and invocation directory name. You can alternatively use the +ncuid+<unique_id> option to concurrently run multiple simulations sharing a common directory structure.

The cds.lib and hdl.var files in the invocation directory contain INCLUDE statements for the cds.lib and hdl.var files in the INCA_libs directory. This mechanism supports multiple snapshots of the same design.
Understanding ncverilog

First-Time Behavior (continued)

The first time you run ncverilog, it also:

■ Parses the command line to determine the library search order
■ Creates library directories for any libraries specified with the \(-y\) or \(-v\) options
■ Translates the Verilog-XL simulator command line options into tool-specific arguments for the NC-Verilog ncvlog, ncelab, and ncsim tools
■ Invokes ncvlog, ncelab, and ncsim sequentially
  — All tools share a common log file (ncverilog.log)
  — If any tool fails, it issues an error message and terminates the process
Understanding ncverilog (continued)
Understanding ncverilog

Subsequent Behavior

If the options are different:

For subsequent invocations, ncverilog compares the new command line options to those stored in the library. If options that affect compilation are not the same, it reverts to first-time behavior.

Note: ncverilog does not factor into the comparison some command line options, such as +gui, −q, and −s, which only affect run-time behavior. To take advantage of incremental compilation, you must enter all other options identically to the previous session and enter any −v and −y in the same order as before.
Understanding ncverilog (continued)
Understanding ncverilog

Subsequent Behavior (continued)

If the options are identical, ncverilog:

- Reads the *cds.lib* and *hdl.var* files in the *snap.nc* directory
- Examines the SNAPSHOT variable to determine which snapshot to use
- Examines the date/time stamp of all source and derived files
  - If the snapshot is up to date:
    - Invokes only *ncsim*
  - If the snapshot is up to date but an SDF file has been modified:
    - Invokes both *ncelab* and *ncsim*
  - If the snapshot is not up to date:
    - Invokes the NC-Verilog *ncvlog*, *ncelab*, and *ncsim* tools. Invokes *ncvlog* with the *-update* option by default to recompile only modified source files and their other interdependent files
Understanding ncverilog (continued)
Simulating with ncverilog

Now let us address some technical details:

- How to invoke **ncverilog**
- The **ncverilog** library structure
- Some useful **ncverilog** options
Using the ncverilog Command (continued)

Simulating with ncverilog
Simulating with ncverilog

Invoking ncverilog

You can invoke ncverilog exactly as you would verilog

Syntax:

    ncverilog [ncverilog_options] verilog-xl_arguments

Example:

If you currently invoke the Verilog-XL simulator with command-line arguments:

    verilog $DESIGN/top.v -v $DESIGN/libs/comb_logic.v -y
    $DESIGN/models +libext+.v

You run ncverilog with the same command-line arguments:

    ncverilog $DESIGN/top.v -v $DESIGN/libs/comb_logic.v -y
    $DESIGN/models +libext+.v
Simulating with ncverilog (continued)
Simulating with ncverilog

The NC-Verilog Library Structure

Given this existing Verilog-XL simulator directory structure:

To run ncverilog, enter:

```
ncverilog -f run.f
```
Simulating with ncverilog (continued)

This is the contents of the four design files for this example:

---

**top.v**

```verilog
module top;
    reg in1, in2;
    glue u1 (in1, in2, out1, out2);
    initial
        begin
            #1000 in1 = 1'b1;
            #1000 in2 = 1'b1;
            #1000 in2 = 1'b0;
            #1000 in1 = 1'b0;
            #1000 $stop;
        end
endmodule
```

**models/and2.v**

```verilog
module and2 (A, B, X);
    input A, B;
    output X;
    assign X = A && B;
endmodule
```

---

**libs/comb_logic.v**

```verilog
module glue (in1, in2, out1, out2);
    input in1, in2;
    output out1, out2;
    and2 u1 (in1, in2, out1);
    or2  u2 (in1, in2, out2);
endmodule
```

**models/or2.v**

```verilog
module or2 (A, B, X);
    input A, B;
    output X;
    assign X = A || B;
endmodule
```
Simulating with ncverilog

The NC-Verilog Library Structure (continued)

ncverilog adds the INCA_libs library structure:

top.v
-s
-v libs/comb_logic.v
-y ./models
+libext+.v
+libverbose
+maxdelays
+max_err_count+25
+notimingchecks
+nowarnTFMPS
+nowarnTFNPC
+turbo+3
+twin_turbo
Simulating with ncverilog (continued)

% ncverilog -f run.f
ncverilog: v03.40.(p002): Started on Dec 17, 2001 at 12:43:19
ncverilog
-f
run.f
+ncsimargs"-exit"

file: top.v
  module worklib.top:v
    errors: 0, warnings: 0

file: libs/comb_logic.v
  module comb_logic.glue:v
    errors: 0, warnings: 0

file: libs/comb_logic.v
file: models/and2.v
  module models.and2:v
    errors: 0, warnings: 0

file: models/or2.v
  module models.or2:v
    errors: 0, warnings: 0
    Caching library 'models' ------- Done
    Caching library 'comb_logic' ------- Done
    Caching library 'worklib' ------- Done

Elaborating the design hierarchy:
Resolved module/udp 'glue' at 'top.u1' to 'comb_logic.glue:v'.
Resolved module/udp 'and2' at 'top.u1@glue<module>..u1' to 'models.and2:v'.
Resolved module/udp 'or2' at 'top.u1@glue<module>..u2' to 'models.or2:v'.
Building instance overlay tables: .................... Done
Generating native compiled code:
  models.and2:v <0x38872c8b>
    streams: 0, words: 0
  models.or2:v <0x38872c8b>
    streams: 0, words: 0
  worklib.top:v <0x3c2ffe4f>
    streams: 1, words: 247
Loading native compiled code: ....................... Done
Building instance specific data structures.
Design hierarchy summary:

  Instances    Unique
  Modules:     4       4
  Registers:   2       2
  Scalar wires: 4       -
  Initial blocks: 1      1
  Cont. assignments: 0   2

Writing initial simulation snapshot: worklib.top:v
Loading snapshot worklib.top:v .................... Done
ncsim>
Simulating with ncverilog

Some ncverilog Options

You can pass tool-specific options directly to the NC-Verilog simulation tools:

- `+ncvlogargs"quoted_argument_string"
  ncverilog -f run.f +ncvlogargs+-use5x
  Generate a full 5.x-compliant library

- `+ncelabargs"quoted_argument_string"
  ncverilog -f run.f +ncelabargs+-binding sub:gate"
  Bind the gate view of module sub

- `+ncsimargs"quoted_argument_string"
  ncverilog -f run.f +ncsimargs+-exit
  Force the simulation to exit upon executing $stop or upon completion`
Simulating with ncverilog (continued)

The NC-Verilog compiler, elaborator, and simulator first determine that the options you pass to them are compatible with the translated Verilog-XL simulator and ncverilog options, and issue an error message if they are not.

Use the -use5x option to generate a fully 5.x-compliant library. It includes the master.tag, pc.db, and verilog.v files which, although not required for logic verification, may be required for subsequent design tools and frameworks, for example the library browser.

Use the -binding option to bind a specific design subunit that might otherwise not be selected by the library and view search rules. This option forces the specified binding on all instances of the subunit, except those otherwise directed by the `uselib compiler directive.

Use the -exit option to cause the simulator to exit upon executing $stop or upon running the Tcl script or the simulation to completion. The -batch option only overrides the -tcl option, and is not sufficient to prevent entry into interactive mode. The -exit option is the equivalent of entering exit as a run-time Tcl command or selecting File — Exit from the SimControl menu.

Enclose the argument string in double quotes if it contains more than one word!
Simulating with ncverilog

Some ncverilog Options (continued)

Some NC-Verilog simulator tool options have identical ncverilog counterparts:

- **+nclinedebug**
  
  ncverilog -f run.f +nclinedebug

  Set source line access for debug purposes

- **+ncaccess+access_spec**
  
  ncverilog -f run.f +ncaccess+rwc

  Set full read, write, and connectivity access for debug purposes

- **+ncredmem**
  
  ncverilog -f verilog.run +ncredmem

  Used reduced memory image size
Simulating with ncverilog (continued)

Use the `+nclinedebug` option to re-enable line debug capabilities. Some of the default performance optimizations disable the ability to perform line debug operations, such as single stepping through behavioral code and setting line breakpoints. This option also provides read, write, and connectivity access to all simulation objects.

Use the `+ncaccess` option to set default access for all objects. The default access is no read, write, or connectivity access. Use `+ncaccess+<args>` to turn on access and `+ncaccess-<args>` to turn off access. The arguments can be any combination of the letters R, W, and C in upper or lower case.

**Important**

You must include the `+ncaccess+r` option if you intend to record waveform data. If you do not do this, you will get a warning message and the simulation will continue, but not record waveform data.

Use the `+ncredmem` option to enhance memory performance by deferring loading of VST files until the information in them is needed. This design information is not needed for simulation, but may be needed for debugging. Do not recompile design data while a simulation started with this option is still running as it could subsequently load design data inconsistent with the currently running simulation.
Simulating with ncverilog

The NC-Verilog Command-Line Interface

The command-line interface is somewhat different, but equally capable:

- You control the Verilog-XL simulator with Verilog-XL run-time commands
  — The Verilog-XL run-time commands are action-oriented
- You control NC-Verilog simulation with Tcl simulator commands
  — The Tcl command extensions are primarily object-oriented
- Some Tcl commands correspond closely with Verilog and some do not

<table>
<thead>
<tr>
<th>Correspondence</th>
<th>Verilog-XL Command</th>
<th>NC-Verilog Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>acceptable</td>
<td>$showvars</td>
<td>describe, drivers, scope</td>
</tr>
<tr>
<td>more closely</td>
<td>$input</td>
<td>input</td>
</tr>
<tr>
<td>even closer</td>
<td>$deposit</td>
<td>deposit</td>
</tr>
<tr>
<td>closest</td>
<td>force</td>
<td>force</td>
</tr>
</tbody>
</table>
Simulating with ncverilog (continued)

You cannot use the Verilog-XL simulator –i option to input a file of Verilog-XL commands. ncverilog does not translate the Verilog-XL commands in the file and does not translate the –i option. After you have manually translated the Verilog-XL commands into Tcl commands, you can use the +ncinput option to execute the command file.

You cannot expect NC-Verilog simulator commands to exactly replicate the operation of Verilog-XL simulator commands. For example, the force command, though named similarly, produces slightly different results. You can force a Verilog-XL object to an expression, and the object’s value will continue to respond to new inputs to the expression. When you force an NC-Verilog object to an expression value, it does not respond to new input values to the expression.
Manually Transitioning a Design Using ncprep

This subsection discusses:

- What does ncprep do?
- Should you use ncprep?
- How does ncprep work?
- Customizing the ncprep environment
Manually Transitioning a Design Using ncprep
What Does ncprep Do?

Use **ncprep** to convert an existing Verilog-XL simulator environment to an NC-Verilog simulator environment. The **ncprep** utility:

- Requires only a single command (**ncprep**)
- Accepts Verilog-XL simulator command line options and arguments
- Creates the *cds.lib* and *hdl.var* files
- Creates the required NC-Verilog directories and libraries
- Translates all applicable Verilog-XL simulator options into tool-specific NC-Verilog options
- Creates (but does not run) a shell script called **RUN_NC**

Syntax:

```
ncprep [ncprep_options] verilog-xl_arguments
```
Manually Transitioning a Design Using ncprep (continued)

What Does ncprep Do?

Use the ncprep utility to create initial libraries and cds.lib and hdl.var files and to convert Verilog-XL simulator command line options into NC-Verilog command line options for the compiler, elaborator, and simulator tools.

Invoke ncprep with Verilog-XL command line options and arguments.

Note: Please refer to the Cadence online document NC-Verilog Simulator Help for the complete list of ncprep command line options and translated Verilog-XL simulator options.
Should You Use ncprep?

Use **ncprep** if you use the Verilog-XL simulator and you:

- Want to completely transition your existing Verilog-XL simulator design simulation environment to an NC-Verilog design simulation environment
- Do not expect to return to the Verilog-XL simulator environment
- Are completely committed to using the library-based approach to future NC-Verilog simulation

The primary function of the **ncprep** utility is to create the required libraries, directories, and set-up files necessary for using NC-Verilog simulation on a design currently working with the Verilog-XL simulator.

**Important**

The **ncprep** utility is a one-time transition tool, meant for users who want to completely adopt the library-based NC-Verilog simulation methodology.
Manually Transitioning a Design Using ncprep (continued)

Should You Use ncprep?
How Does ncprep Work?

The ncprep utility creates:

- `.INCA_libs` library directories for each library (including `work`)
- `.INCA_libs/worklib` work directory
- `cds.lib` file containing one entry for each library (including `work`)
- `hdl.var` file containing definitions of LIB_MAP and VIEW_MAP
- `ncvlog.args` file containing options and source file names for use by the NC-Verilog `ncvlog` compiler
- `ncelab.args` file containing options and top-level module names for use by the NC-Verilog `ncelab` elaborator
- `ncsim.args` file containing options and the snapshot name for use by the NC-Verilog `ncsim` simulator
- `ncprep.log` file containing a log of all actions taken
- `RUN_NC` shell script invoking `ncvlog`, `ncelab`, and `ncsim`, which use command line options in the `.args` files
Manually Transitioning a Design Using ncprep (continued)

How Does ncprep Work?

The *ncvlog.args* file lists all of the HDL files required for a particular design. The *ncprep* utility searches library directories and only includes the files that are needed from the directories. However, *ncprep* can’t determine which modules are required in the library files, so it includes the entire file in the *ncvlog.args* file. Thus *ncvlog* compiles the complete library file, even though only a few modules are needed. This behavior can result in a longer compile time.
Customizing the Environment

After \texttt{ncprep} has finished executing:

- Review the generated files and conversion script
  Consider whether your coding style, library management, or design environment contain features that mandate manual intervention

- If necessary:
  - Edit the initial \texttt{cds.lib} and \texttt{hdl.var} files as desired to customize tool operation
  - Edit the RUN NC script if your \texttt{ncelab} or \texttt{ncsim} executables are named something else, and to customize as desired

- Execute the RUN NC script to compile, elaborate, and simulate your design

\textbf{Important}

Please have your Cadence online documentation available while transitioning with \texttt{ncprep}. It contains a substantial amount of information on how to transition a design to the NC-Verilog simulator using \texttt{ncprep}, as well as many helpful hints on trouble-shooting the transition process.
Customizing the Environment

If the files to be created by `ncprep` already exist, `ncprep` reports an error message and stops. Use the `+overwrite` option to overwrite existing files. Do not abbreviate any `ncprep` plus options.

**Note:** The `ncprep` utility cannot convert every Verilog-XL simulator option to an equivalent NC-Verilog option. It is not designed to create an optimized simulation flow. You may want to customize the `RUN_NC` script and argument files, and possibly rewrite some of the source.
Manually Transitioning a Design

This subsection discusses:

- Modifying the source for best results
- Defining the libraries and variables
- Invoking the NC-Verilog tools
Manually Transitioning a Design
Modifying the Source for Best Results

To simplify manual transition of your design, and best take advantage of incremental compilation, Cadence recommends that you review your source:

- **Merge multi-file modules** (those using `include`) wherever practical
  - The use of *include* files is best reserved for sharing control parameters and directives common to all or a subset of design units

- **Split multi-module files** wherever practical
  - Rule of thumb: one file <=> one module
  - Exception: libraries of numerous small, similar cells that are seldom modified, such as vendor technology libraries

- **Remove cross-file inheritance** (use `undefineall`, `resetall`, and `include`)
  - Reduces unnecessary compilation
  - Prevents incorrect inheritance due to out-of-order compilation

**Note:** Cadence recommends that you reserve the sharing of common control information to a single or small set of files included immediately after the `resetall` directive at the beginning of a module definition file. Follow that with any control information that is needed only by the individual module in the file.
Modifying the Source for Best Results

Carefully review your use of the `include compiler directive to incorporate a separate design source file into a design unit, especially if that separate source file contains additional module definitions. The included source is compiled whenever the file that includes it is compiled. This might not be desired if it contains additional module definitions.

The NC-Verilog simulator by default compiles all modules in a file submitted for compilation, even if only one module has been modified and needs to be compiled. This is not efficient. The space saved by keeping multiple modules in a single source file is insignificant.

Compiler directives are by default active between source files of a compilation session. The assumption of compiler directives requires that the interrelated modules be compiled together and in the correct order. This reduces efficiency and increases risk that the directives may be incorrectly inherited.
Defining the Libraries and Variables

The next step to manually transition your design is to define libraries and variables:

- **Define libraries in the *cds.lib* file**
  
  ```
  DEFINE projectlib ./proj.lib
  DEFINE worklib ./work.lib
  ```

- **Define variables in the *hdl.var* file**
  
  - Map source directories to specific libraries
    
    ```
    DEFINE LIB_MAP (.src => projectlib, + => worklib)
    ```
  
  - Map source extensions to specific views
    
    ```
    DEFINE VIEW_MAP (.vg => gate, + => module)
    ```
Manually Transitioning a Design (continued)

Defining the Libraries and Variables

The NC-Verilog library of compiled design data elements is not obvious to you if you run `ncverilog`. However, it is very evident to you if you choose to run the individual tools.

You precompile libraries prior to using them. You set the LIB_MAP and VIEW_MAP variables to map source files and extensions to libraries and views. You can override these variables on the `ncvlog` compiler command line, for example:

```bash
ncvlog -work cmos /usr/me/lib/cmos.v
```

You can alternatively override LIB_MAP and VIEW_MAP with WORK and VIEW.

You use the `hdl.var` LIB_MAP variable to specify the libraries and the order for which the `ncelab` elaborator to search them. The elaborator by default first tries to find subunits in the library in which their parent is found. Then it searches the following LIB_MAP libraries, wrapping around to the beginning of the list if necessary.

You use the `hdl.var` VIEW_MAP variable to specify the views and the order for which the `ncelab` elaborator to search them. The elaborator by default first tries to find subunits in the view in which their parent is found. Then it searches the following VIEW_MAP views, wrapping around to the beginning of the list if necessary.

The `ncelab` elaborator does not accept the `-v` library file option, the `-y` library directory option, or the `+libext+` file extension option. The `ncprep` utility accepts these options. You can use `ncprep` to initially prepare your Verilog-XL design environment if you choose to run the individual NC-Verilog tools. It translates Verilog-XL command-line options and generates the `cds.lib` and `hdl.var` files and a script to run the individual NC-Verilog tools.
Invoking the NC-Verilog Tools

Lastly, to manually transition a design:

- Convert Verilog-XL simulator options to NC-Verilog options
- Invoke the `ncvlog`, `ncelab`, and `ncsim` tools sequentially
Manually Transitioning a Design (continued)

Invoking the NC-Verilog Tools

Many Verilog-XL simulator options have identical or approximate NC-Verilog counterparts. Some of these are intuitively obvious, yet others produce results that are not exactly identical. There are also some Verilog-XL simulator command line options that are not supported by the NC-Verilog simulator when you run it in the library-based mode, as they are not applicable or necessary. To use the NC-Verilog simulator with the Verilog-XL options, run `ncverilog`. Please refer to the Cadence online document *NC-Verilog Simulator Help* for additional information concerning the exact syntax and option arguments.

You can place the command line options in a file. Invoke the file on the command line using the `-file` option.
Making the Most of Incremental Compilation

This subsection discusses:

- Incremental compilation with `ncverilog`
- Incremental compilation in the library-based mode
- Some things to be aware of
Making the Most of Incremental Compilation
Incremental Compilation with ncverilog

ncverilog performs incremental compilation by default:

- It examines all the source code and intermediate files to determine if they have been modified
- It invokes the appropriate tools
  - The ncvlog compiler
    To compile only the modified design units
  - The ncelab elaborator
    To write new code for only the modified design units
    To annotate timing data
    To generate a new snapshot
  - The ncsim simulator
Incremental Compilation with ncverilog

The library-based user-directed mode of operation is incompatible with ncverilog. The NC-Verilog simulator tags the snapshot with the mode in which it is generated. This tag prohibits you from directly updating an ncverilog snapshot with the ncsim -update option or the ncuupdate utility. The NC-Verilog simulator issues an error message if you attempt this.

The performance of an incremental recompilation when running ncverilog will vary widely, depending on the interdependency of the design files. It can range from recompiling the entire design to recompiling only one or a very small number of units. In general, the recompilation time with ncverilog is not different from that of the library-based mode.
Automatic Incremental Compilation

You can perform automatic incremental compilation in the library-based mode:

- **ncsim -update**
  - Invokes *ncupdate*
  - Invokes *ncvlog* and *ncelab* (if needed)
  - Invokes *ncsim*

- **ncelab -update**
  - Invokes *ncupdate*
  - Invokes *ncvlog* (if needed)
  - Invokes *ncelab*

- **ncvlog -update**
  - Recompiles design units
  - Maintains file interdependencies
Making the Most of Incremental Compilation (continued)

Incremental Compilation in the Library-Based Mode

You can use the `ncsim -update` option to update and resimulate a snapshot. You can use this option only if you first compiled and elaborated the design in the library-based mode. You can use it with the `-nosource` option to bypass recompilation, if you have made multiple changes and have already compiled the ones you want to use. The NC-Verilog simulator invokes the `ncupdate` utility to recompile any changed design units and re-elaborate the design, then starts the simulator. It retrieves compiler and elaborator options from the VST in the library for each cell.

You can use the `ncelab -update` option similarly. In this situation the NC-Verilog simulator does not start the simulation.

You can use the `ncvlog -update` option to recompile modified design units. You can then re-elaborate and resimulate your design. You must take this approach if you have added or deleted files or modules, changed compiler directives, or want to compile with different options. The `ncupdate` utility invoked by `ncsim -update` or `ncelab -update` does not maintain compiler directive inheritance. Use the `ncvlog -update` option to automatically read any other source files needed to maintain compiler directive inheritance. This is the default `ncverilog` mode.

After the initial elaboration, subsequent elaboration generally takes much less time, as `ncelab` regenerates the SIG and COD library objects only if a cell instantiation changes. Although elaboration during update must process the entire design hierarchy, it only creates new SIG and COD objects where necessary.
Incremental Compilation in the Library-Based Mode

Manual Incremental Compilation

You can perform manual incremental compilation in the library-based mode with:

■ **ncupdate**
  — Wide range of options available
  — Requires one previous full elaboration of the entire design
  Cannot be used if a module or file has been added to the design
  — Recompiles and elaborates all out-of-date design units
    Uses Date/Time stamp
    Uses links saved from previous full elaboration
  — **ncvlog** and **ncelab** option arguments default to those previously used
    Saved in VST for each design unit
Incremental Compilation in the Library-Based Mode (continued)

You can use `ncupdate` to automatically recompile only modified design units, re-elaborate the design, and re-invoke the simulator, but only if you previously compiled and elaborated the design in the library-based mode. The NC-Verilog simulator retrieves compiler and elaborator options from the VST in the library for each cell.
Things to Be Aware Of

The following are beneficial in both ncverilog and the library-based modes:

- Avoid cross-file inheritance
  - Compiler directives embedded in source file text (such as `define)
  - Included source files containing new module declarations

- Use one file for each module definition
  - Except vendor technology libraries
Making the Most of Incremental Compilation (continued)

Things to Be Aware Of

Cross-file inheritance creates dependencies that reduce the efficiency of incremental compilation. You must simultaneously recompile every (possibly unmodified) source file that the modified source file is dependent upon, and also all files that depend upon the modified file. You must recompile these files in the same order as they were previously compiled.

Multimodule source files force incremental compilation to perform unnecessary work. Recompilation of even one module contained in the file may cause all modules contained in the file to recompile.
Comparing ncverilog and the Library-Based Mode

The ncverilog utility is right for users who want high performance, but:

- Want to maintain only their existing source control methodology
- Want to intermingle Verilog-XL and NC-Verilog simulator sessions

Manually transitioning a design (perhaps with ncrep) is right for those that:

- Want to use mixed language simulation
- Want the highest performance incremental compilation

⚠️ Important
Manually transitioning a design requires a commitment to the library-based methodology!
Comparing ncverilog and the Library-Based Mode

ncverilog scans the libraries each time you invoke it. Its recompilation performance may be slightly less than that of the library-based approach, where you can explicitly invoke ncsim without scanning libraries that you know are already up to date.

The library-based use model, where you run each NC-Verilog tool individually, provides you with the most control over the compilation, elaboration, and simulation process.

The ncprep utility can quickly transition a running Verilog-XL simulator design to the NC-Verilog simulator. It will create the necessary directory structure, but it cannot create an optimized simulation flow. Your coding style, library management process, and design environment define the ultimate performance of the simulator, especially in the area of incremental compilation. You may want to make modifications to the script that ncprep generates.
Summary

In this section, you learned about:

- Transitioning an existing Verilog-XL simulator design environment to the NC-Verilog simulator
- Incremental compilation
- The trade-offs among the transition methods
Summary
Review

1. What options and arguments would you normally provide to the `ncverilog` command?

2. True or false: `ncprep` optimizes your simulation flow for maximum performance.

3. What utilities does `ncsim -update` invoke to update design timing?

4. For what reason might you want to separately invoke the NC-Verilog compiler, elaborator, and simulator?
Review

1. What options and arguments would you normally provide to the `ncverilog` command?
   
   Provide to the `ncverilog` command the options and arguments you provide to the `verilog` command and any additional options and arguments applicable only to the `ncverilog` command.

2. True or false: `ncprep` optimizes your simulation flow for maximum performance.
   
   The `ncprep` utility prepares your existing Verilog-XL design environment for simulation with the NC-Verilog simulator. It creates the necessary directory structure and files. You must manually modify your design environment for optimal NC-Verilog performance.

3. What utilities does `ncsim -update` invoke to update design timing?
   
   The `ncupdate` utility will determine that only the SDF timing data is out of date, so will invoke the `ncelab` elaborator to annotate the new timing data and write a new snapshot for the `ncsim` simulator to simulate.

4. For what reason might you want to separately invoke the NC-Verilog compiler, elaborator, and simulator?

   The `ncverilog` utility automatically performs an incremental update, for which it needs to compare the stored date and time of the most recent update with the date and time stamp of the source files. You can manually invoke the tools to avoid this slight overhead.
About Lab 18

The objective of this lab is to transition a design from the Verilog-XL simulator to the NC-Verilog simulator.

In this lab, you will:

- Run `ncverilog` to transition a Verilog-XL simulator design environment to the NC-Verilog simulator
  - Correct `ncverilog` compile-time errors
  - Correct mismatches between the Verilog-XL and NC-Verilog simulators
- Compare your waveform databases with Comparescan
About Lab 18
Chapter 25: Introduction to Cycle Simulation

Objectives

In this section you will learn about cycle simulation:

- How it works
- Its advantages
- Adoption issues
- Integrating it into your design flow
Introduction to Cycle Simulation

This section briefly describes cycle simulation technology, and discusses the advantages and disadvantages, handling of adoption issues, and incorporating the technology into your design flow.
Introducing Cycle Simulation

Cycle simulators:

- Preprocess the design into a symbolic representation
  - Levelized Compiled Code (LCC)
  - Binary Decision Diagrams (BDD)
  - Boolean and arithmetic operations

- Update all circuit states simultaneously when triggered
  - Using lookup tables and computational methods
  - Typically on an externally defined clock edge
  - The software counterpart of a hardware model

- Are old technology recently gaining wider use
  - Previously was usually proprietary software
Introducing Cycle Simulation

Cycle simulators precompile the design into a representation that they can more efficiently execute during runtime. This technology is also more or less applicable to event simulation, depending upon how much access to internal objects and events the user wishes to retain.

The essential difference between event and cycle simulation is that event simulation "pushes" events through the design, iteratively calculating node states, until the design state settles, while cycle simulation calculates the design state "as a whole" only once for each external trigger. Cycle simulation in this way acts much as a hardware modeler would.

Cycle simulation is not new technology. Prior to the widespread adoption of event simulation, proprietary simulation was primarily cycle-based. Cycle simulation has become more mainstream recently, partly due to its performance advantage over event simulation.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Event-based</th>
<th>Cycle-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circuit state evaluation</td>
<td>Signal edges immediately propagate through the design</td>
<td>Signal states update once for each evaluation</td>
</tr>
<tr>
<td>Number of states</td>
<td>4 - 48</td>
<td>2 - 4</td>
</tr>
<tr>
<td>Timing analysis</td>
<td>As accurate as you want it</td>
<td>No timing analysis</td>
</tr>
<tr>
<td>Simulatable representations</td>
<td>All abstractions from behavioral to transistor</td>
<td>A &quot;synthesizable&quot; subset</td>
</tr>
</tbody>
</table>
How Does it Work?

Event-based simulation evaluates circuit components in response to external or internal triggers.

Cycle-based simulation evaluates circuit as a whole in response to external triggers.

Verilog-XL Event Trace

L25 "ADD.v": (a, b, ci) = 6;
L4 "ADD.v": wire ci >>> NET = St0
L4 "ADD.v": wire b >>> NET = St1
L8 "ADD.v": xor x4 >>> XL GATE = St1
L7 "ADD.v": and a3 >>> XL GATE = St0
L6 "ADD.v": xor x2 >>> XL GATE = St0
L5 "ADD.v": and a1 >>> XL GATE = St1
L8 "ADD.v": xor x4 >>> XL GATE = St0
L9 "ADD.v": or o5 >>> XL GATE = St1
L14 "ADD.v": wire s >>> FROMXL NET = St0
L14 "ADD.v": wire co >>> FROMXL NET = St1

Levelized Compiled-code

LDA 1, a
LDA 2, b
XOR 1, 2
LDA 1, c
XOR 1, 2
STA 2, s

Table Lookup

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>ci</th>
<th>co</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>
Introducing Cycle Simulation (continued)

How Does it Work?

This is an illustration of the difference between event and cycle simulation:

- Event simulation typically traces every signal transition as it propagates through the circuit, and continues to do so until the circuit reaches a stable state. The illustration displays part of an event trace produced by the Verilog-XL simulator. Note that the simulator may need to calculate the state of a component several times during the same simulation time instant.

- Cycle simulation typically re-evaluates the state of the circuit as a whole, once upon each external trigger, usually without evaluating any intermediate node states. Cycle simulation can implement combinational control logic as an optimized lookup table. Cycle simulation can map vector functions, data flow, and arithmetic calculations into native machine instructions. For example, it may implement a multiply operation in machine instructions, rather than as a lookup table.
Taking Advantage of Cycle Simulation

Some of the advantages of cycle simulation are:

- It typically executes significantly faster than event simulation
- It more closely represents the operation of real hardware
  - The modeling style closely resembles that used for synthesis
  - Without events, cycle simulation has no run-time event ordering issues

Some advantages of adopting the cycle simulation design style are that:

- Synchronous designs are easier to debug than asynchronous designs
- The synchronous design style also enhances event simulation performance!
Taking Advantage of Cycle Simulation

The primary advantage of cycle simulation is that of performance. Depending on many factors, including platform, your coding style, the degree to which you monitor the simulation, and the vendor’s algorithms, the cycle simulation process executes at many times the speed of event simulation.

Simulation vendors have incorporated cycle simulation technologies into event simulation. If you are willing to give up timing analysis, signal strength resolution, and access to individual design objects and simulation events, your event simulator may approach the performance of pure cycle simulation, especially for highly synchronous RTL simulations.

This level of performance allows the verification of large designs, that was previously done with the use of servers, to be farmed out to significantly less expensive desktop machines for distributed test regression.
Adopting Cycle Simulation

The cycle-simulatable model:

- Requires synchronous code
  - Few circuits are 100% synchronous
  - Expect code and methodology changes

- Cannot handle behavioral constructs typically found in testbenches
  - Expect to rewrite a significant portion of your testbench code

- Maintains many fewer states
  - Cannot prove initialization
  - Cannot resolve signal strength

- Has no concept of simulation time
  - Does not handle propagation delay or timing checks
  - Cannot handle delay-dependent components

- Offers less observability
  - Does not by default provide debug access to intermediate signal states
Adopting Cycle Simulation

The cycle model is somewhat like a hardware model, and thus also somewhat like an emulation. As with a hardware emulator, you will need to precompile the cycle model. As with the hardware model, when triggered, the cycle model generates new outputs based upon the current state of the model and the current inputs.

The designs that benefit the most from cycle simulation are those that most closely reflect the hardware they describe, especially those with a high degree of parallelism or multiply-instantiated components. Although many vendors provide means to incorporate asynchronous and behavioral components into the simulation, the event-based engine executes these components, thus significantly degrading overall simulation performance.

You can regain some of this performance by implementing your testbench in C and using the PLI or VPI to interface your testbench with your design. Several vendors offer testbench generation tools that hide the details of the application programming interface (API).
Handling Methodology Issues

Consider the following issues when adopting cycle simulation:

- Immature modeling standards
- Third-party intellectual property (IP) integration
- Legacy event-based testbench code
- Special considerations
  - Initialization, bus contention, pullup/pulldown, debug capability
- Tool suite integration
  - Use event simulation for initialization and sign-off
  - Use static timing analysis
Handling Methodology Issues

There is no cycle simulation modeling standard. Each vendor publishes the subset of the HDL that their tool accepts. Several vendors automatically partition the simulation between components they accept for cycle simulation and components that require event simulation, and this partitioning is often sufficiently coarse to exclude much of the design and most or all of the testbench from cycle simulation.

Unless specifically advertised as such, you are unlikely to be able to cycle-simulate any third-party models. You may also have much testbench code developed for event simulation that you will find difficult to rewrite for cycle simulation.

While developing a model for cycle simulation you should carefully consider how you will accommodate initialization of the model, signal strength resolution, and access into the model for debugging purposes.
Cycle Simulation in Your Design Flow

Integrate cycle simulation into your design flow in the following manner:

■ Keep behavioral testbench constructs out of the model
  — Stimulus generation, response verification, file I/O

■ First use event simulation to verify asynchronous behaviors
  — initialization, arbitration, bus contention

■ Then use cycle simulation to verify system functionality
  — Use with pseudo-random stimulus requiring many simulation cycles
  — Use for resource-intensive simulations
    microprocessors, video processors, digital signal processors, data communications

■ Use a static timing analysis tool to verify system timing
Cycle Simulation in Your Design Flow

Cycle simulation does not replace event simulation. Event simulation handles asynchronous behaviors and signal strength resolution. Cycle simulation is best focused on the functional verification of large, synchronous designs of the 100K to 1M equivalent-gate size, that require tens of millions, or even billions, of clock cycles to adequately verify. Such systems as microprocessors, multimedia, and digital signal processing (DSP) fit this criteria.

As designs become larger and more complex, designers spend proportionately more time ensuring the correctness of their design before releasing it to the factory.

Cycle simulation enables designers to do more testing in less time, by restricting access to internal objects and events, evaluating logic states only once for each clock trigger, and ignoring all time-related information.

Timing verification is also a very important part of the design process, thus static timing analysis complements cycle simulation. The cycle simulator verifies the logic, while the static timing analyzers verify the timing.
Summary

In this section you learned about cycle simulation:

- How it works
- Its advantages
- Adoption issues
- Integrating it into your design flow
Summary
Review

1. Name a benefit for the use of cycle simulation.

2. How does cycle simulation obtain its performance advantage over event simulation?

3. What factors make a design an ideal candidate for cycle simulation?

4. True or false: Cycle simulation replaces event simulation in a typical verification flow.
Review

1. Name a benefit for the use of cycle simulation.
   Among the advantages that cycle simulation offers, are: It typically executes significantly faster than event simulation; It more closely represents the operation of real hardware, thus forcing a synthesis-friendly coding style.

2. How does cycle simulation obtain its performance advantage over event simulation?
   Cycle simulation evaluates the state of the design once for each stimulus cycle, does not simulate design timing, and does not resolve the conflict of multiple drivers of a net.

3. What factors make a design an ideal candidate for cycle simulation?
   The ideal candidate for cycle simulation is a synchronous design that contains complex combinational circuitry, and for which no signal strength resolution or timing data is needed during simulation.

4. True or false: Cycle simulation replaces event simulation in a typical verification flow.
   Cycle simulation does not replace event simulation in a typical verification flow. You still need event simulation to verify asynchronous behaviors, such as initialization, bus arbitration, and value conflicts.
Chapter 26: Switch-Level Modeling

Objectives

In this section, you will learn about:

- Switch-level modeling
- The Switch-XL algorithm
- Choose an acceleration algorithm
Switch-Level Modeling

This section describes the Verilog switch primitives: how to instantiate them and assign delays and drive strength to them. It explains how the simulator uses the drive strengths to resolve the value of a net driven by multiple devices. The section then goes on to describe the Cadence Verilog-XL simulator Switch-XL engine, that permits much more detailed relative strength specifications, and accelerates the simulation of networks of switch primitives.
Verilog Switch Primitives

Modeling transistor networks at the switch-level more accurately represents their operation.

Verilog provides unidirectional and bidirectional primitives that you can use to model switch networks:

- The following are unidirectional primitives:
  
<table>
<thead>
<tr>
<th>cmos</th>
<th>nmos</th>
<th>pmos</th>
<th>pullup</th>
</tr>
</thead>
<tbody>
<tr>
<td>rcmos</td>
<td>rnmos</td>
<td>rpmos</td>
<td>pulldown</td>
</tr>
</tbody>
</table>

- The following are bidirectional primitives:

<table>
<thead>
<tr>
<th>tran</th>
<th>tranif0</th>
<th>tranif1</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtran</td>
<td>rtranif0</td>
<td>rtranif1</td>
</tr>
</tbody>
</table>
Verilog Switch Primitives
Switch Instantiation

This chart illustrates the order of switch connections.

nmos (drain, source, gate);

pmos (drain, source, gate);

cmos (drain, source, ngate, pgate);

tranif0 (data1, data2, control);
Verilog Switch Primitives (continued)

Switch Instantiation
Switch Delay

You can assign delays to some switch types:

- The unidirectional \texttt{cmos}, \texttt{nmos}, and \texttt{pmos} switches can have rise, fall, and turn-off delays
  
  cmos #(<delay>) (d, s, ng, pg);
  
  nmos #(<rise\_delay>,<fall\_delay>) (d, s, g);
  
  pmos #(<rise\_delay>,<fall\_delay>,<turnoff\_delay>) (d, s, g);

- The bidirectional switches \texttt{tranif0} and \texttt{tranif1} can have turn-on and turn-off delays, but no source-drain channel delays
  
  tranif0 #(<delay>) (d, s, g);
  
  tranif1 #(<turnon\_delay>,<turnoff\_delay>) (d, s, g);

- The \texttt{pulldown}, \texttt{pullup}, and \texttt{tran} gates cannot have delays

\textbf{Note:} You can specify delays in min:typ:max format.
Verilog Switch Primitives (continued)

Switch Delay

For the `cmos`, `nmos`, and `pmos` unidirectional switches:

- 1 delay — The delay for all transitions
- 2 delays — First delay for transitions to 1; Second delay for transitions to 0; Smaller delay for transitions to Z and X
- 3 delays — First delay for transitions to 1; Second delay for transitions to 0; Third delay for transitions to Z; Smaller delay for transitions to X

For the `tranif0` and `tranif1` bidirectional switches:

- 1 delay — The delay to turn on and to turn off
- 2 delays — First delay to turn on; Second delay to turn off
Drive Strength

You can assign strengths to some primitive types:

- The **pulldown** and **pullup** primitives can have one or two drive strengths
  - The simulator ignores the unneeded strength specification
    pullup (weak1, weak0) (net1);

- The boolean primitives can have two drive strengths
  - You must specify both drive strengths, or none
    nand (highz1, strong0) (net1, net2, net3);

- The **trireg** net type can have charge strengths
  trireg (small) net1;

- The switch primitives CANNOT have drive strengths!

<table>
<thead>
<tr>
<th>Level</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drive</td>
<td>supply</td>
<td>strong</td>
<td>pull</td>
<td>weak</td>
<td></td>
<td></td>
<td>highz</td>
<td></td>
</tr>
<tr>
<td>Charge</td>
<td></td>
<td>large</td>
<td>medium</td>
<td>small</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Verilog Switch Primitives (continued)

Drive Strength

The simulator uses drive strengths to resolve the value of a net driven by multiple drivers.

You cannot assign drive strengths to the switch primitives. That is because the switch primitives are the only primitives that pass (or reduce) an already existing strength on the net.

The boolean primitives by default drive strong0 and strong1 strengths.

The pulldown and pullup primitives by default drive pull0 and pull1 strengths.

The default trireg charge strength is medium. This is the drive strength that remains on the net during the charge decay time after all drivers stop driving the net.
Strength Reduction

The switches can reduce the strength of signals passing through them:

- The `cmos`, `nmos`, `pmos`, `tran`, `tranif0`, and `tranif1` primitives reduce a supply strength signal to a strong signal.

- The `rcmos`, `rnmos`, `rpmos`, `rtran`, `rtranif0`, and `rtranif1` primitives reduce signal strength according to the following table:

<table>
<thead>
<tr>
<th>Input strength</th>
<th>Reduced strength</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 - supply</td>
<td>5 - pull</td>
</tr>
<tr>
<td>6 - strong</td>
<td>5 - pull</td>
</tr>
<tr>
<td>5 - pull</td>
<td>3 - weak</td>
</tr>
<tr>
<td>4 - large</td>
<td>2 - medium</td>
</tr>
<tr>
<td>3 - weak</td>
<td>2 - medium</td>
</tr>
<tr>
<td>2 - medium</td>
<td>1 - small</td>
</tr>
<tr>
<td>1 - small</td>
<td>1 - small</td>
</tr>
<tr>
<td>0 - highz</td>
<td>0 - highz</td>
</tr>
</tbody>
</table>
**Verilog Switch Primitives** (continued)

**Strength Reduction**

The resistive switches reduce the strength of a signal passing through them. These primitives otherwise act identically to their non-resistive counterparts.

The simulator uses drive strengths to resolve the value of a net driven by multiple drivers:

- Stronger strengths override weaker strengths
- If the signals have unambiguous strengths, the result is the signal value with the highest strength
- If the signals have ambiguous strengths, the result is a signal with ambiguous strength
- The conflict of ambiguous strengths can result in a shorter or wider strength range

The values of unknown signals with ambiguous strengths are denoted with this nomenclature: The first digit is the strongest 0 strength; and the second digit is the strongest 1 strength. For example, if a signal driving a \( \text{WeL} \) (a strength between weak0 and highz0) conflicts with a signal driving a \( \text{PuH} \) (a strength between highz1 and pull1), the result is a signal driving a \( 35X \) (an unknown value between weak0 and pull1).

The values of known signals with ambiguous strengths are denoted with this nomenclature: The first digit is the strongest strength; the second digit is the weakest strength; and the third digit is the value. Thus \( \text{WeL} \) is also known as \( 300 \) and \( \text{PuH} \) is also known as \( 501 \). The conflict of \( 530 \) and \( 651 \) results in \( 56X \). However, the conflict of \( 430 \) and \( 651 \) results in \( 651 \), as the range of 1 strengths does not overlap with the range of 0 strengths.
Switch-Level Networks

Switch networks may contain unidirectional and bidirectional switches. Verilog-XL partitions switch-level networks into channel-connected regions.
Switch-Level Networks

Switch-level networks contain unidirectional switches, bidirectional switches, or both.

Verilog-XL partitions switch-level networks into channel-connected regions:

- A single switch and the nets that connect to its source and drain terminals
- A group of switches connected together through their source and drain terminals, and nets connecting them

The non-XL and XL algorithms simulate the unidirectional switches.

The default and Switch-XL algorithms simulate the bidirectional switches. These algorithms do not provide scheduled values. For the default algorithm, the $showvars system task will show that a value is scheduled, but will not show the value. The Switch-XL algorithm will not even show that a value is scheduled.
The Switch-XL Algorithm

Use the Switch-XL algorithm to:

- Accelerate simulation of bidirectional switches
  — The XL algorithm does not accelerate bidirectional switches
- Simulate up to 250 relative drive strengths on switches
- Simulate up to 250 relative charge strengths on trireg nets
The Switch-XL Algorithm

The XL algorithm does not accelerate the bidirectional switch primitives:

\texttt{tran, tranif0, tranif1, rtran, rtranif0, rtranif1}

Without the Switch-XL algorithm, you specify one of two switch drive strengths (conductances) by instantiating either the non-resistive or the resistive version of the switch. With the Switch-XL algorithm, you can specify relative switch drive strengths in integer increments from 1 to 250.

Without the Switch-XL algorithm, you specify one of three \texttt{trireg} net charge strengths by optionally using the \texttt{small}, \texttt{medium}, and \texttt{large} keywords when you declare the net. With the Switch-XL algorithm, you can alternatively specify relative net charge strengths in integer increments from 0 (no charge) to 250.
Selecting the Switch-XL Algorithm

Use the `+switchxl` option to globally enable the Switch-XL algorithm.

```verilog
verilog source.v +switchxl
```

Use the `switch` compiler directive to selectively enable the Switch-XL algorithm.

```verilog
`switch XL
  // control networks here

`switch default
  // datapath networks here
```

- Use the Switch-XL algorithm:
  - For a significantly-sized network of bidirectional switches
  - For a network of switches you cannot otherwise correctly functionally model with only 2 switch drive strengths and 4 (including none) net charge strengths

- Use the default algorithm:
  - For a densely-packed significantly-sized regularly-structured network of pass transistors
Selecting the Switch-XL Algorithm

Use the Switch-XL algorithm for "sea of gates" networks of bidirectional transistors.

Use the default algorithm for regular structures of pass transistors, such as memories.

If you specify different algorithms for portions of the same channel-connected network, the simulator will first attempt to use the Switch-XL algorithm, then revert to the default algorithm if unsuccessful.

The Switch-XL algorithm requires significant overhead. You would not want to use the Switch-XL algorithm to simulate a design which the XL algorithm can simulate, and for which you can use the standard strength model. For these circuits, the XL algorithm may outperform Switch-XL by a factor of up to 5.

The XL algorithm does not accelerate bidirectional switches. For a circuit consisting mainly of bidirectional switches, the Switch-XL algorithm may outperform the default algorithm by a factor of up to 15.
The Switch-XL Strength Model

Drive Strengths

- These statements declare `tran` switches and assign relative drive strengths.
- The drive strength expression must evaluate to a number from 1 to 250.
- Switch `t1` has the largest conductance relative to `t0` and `rt`.

```
tranif1 strength(3) t1 (s0,d0,g0);
tranif0 strength(2) t0 (s1,d1,g1);
rtran   strength(1) rt (s2,d2);
```

Charge Strengths

- These statements declare `trireg` nets and assign relative charge strengths.
- The charge strength expression must evaluate to a number from 0 to 250.
- Net `a` has the largest capacitance relative to `b`, `c` and `d`.

```
trireg strength(25) a;
trireg strength(10) b;
trireg strength(5) c;
trireg strength(1) d;
```
The Switch-XL Strength Model

The Switch-XL strength model allows you to specify a wider range of relative charge strengths (capacitance) for triregs and drive strengths (conductance) for switches. You specify charge strengths and drive strengths with the `strength` keyword followed by an expression in parentheses. The expression specifies the relative capacitance of a trireg net, or the relative conductance of a switch, compared to other triregs or switches in a channel-connected network. The Switch-XL algorithm must be able to evaluate the expression to a constant value. If the expression evaluates to a real number, the Switch-XL algorithm truncates it to the nearest integer. Keep the relative charge and drive strengths small. The Switch-XL strength model limits the sum of all strengths in any one channel-connected region to no more than 254.

<table>
<thead>
<tr>
<th>Primitives</th>
<th>Range</th>
<th>Default</th>
<th>To Change Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>trireg</td>
<td>0-250</td>
<td>2</td>
<td><code>default_trireg_strength</code></td>
</tr>
<tr>
<td>cmos nmos pmos</td>
<td>1-250</td>
<td>3</td>
<td><code>default_switch_strength</code></td>
</tr>
<tr>
<td>tran tranif0 tranif1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rcmos rnmos rpmos</td>
<td>1-250</td>
<td>2</td>
<td><code>default_rswitch_strength</code></td>
</tr>
<tr>
<td>rtran rtranif0 rtranif1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pulldown pullup tri0 tri1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Switch-XL Strength Reduction

The Switch-XL and default algorithms reduce signal strength differently:

- The default algorithm:
  - Reduces signal strength by 0, 1, or 2 levels for each switch instance
  - An rtran reduces a weak drive to a medium charge

- The Switch-XL algorithm:
  - Treats all drive strengths as higher than charge strengths
  - Maps standard strengths into the range of network strengths
  - Reduces strength once (by the highest resistance) in the channel

![Diagram showing signal strength reduction for default and Switch-XL algorithms.]
Switch-XL Strength Reduction

This is the algorithm by which Switch-XL maps standard strengths into the strengths you specify for a channel-connected region. Switch-XL does this on a region-by-region basis for a design that uses both standard and Switch-XL strength models.

<table>
<thead>
<tr>
<th>Strength</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>supply</td>
<td>( \max \left( \left( \maxd + \maxc + 1 \right), \maxc + 4 \right) )</td>
</tr>
<tr>
<td>strong</td>
<td>( \max \left( \left( \maxd + \maxc \right), \maxc + 3 \right) )</td>
</tr>
<tr>
<td>pull</td>
<td>( \max \left( \left( \left( \maxd + 1 \right) / 2 + \maxc \right), \maxc + 2 \right) )</td>
</tr>
<tr>
<td>weak</td>
<td>( \maxc + 1 )</td>
</tr>
<tr>
<td>large</td>
<td>( \max (\maxc, 3) )</td>
</tr>
<tr>
<td>medium</td>
<td>( (\maxc + 1) / 2 )</td>
</tr>
<tr>
<td>small</td>
<td>1</td>
</tr>
<tr>
<td>highz</td>
<td>0</td>
</tr>
</tbody>
</table>

**Note:** \( \maxc \) is the strongest defined charge strength, \( \maxd \) is the strongest defined drive strength.
How Switch-XL Works

The Switch-XL algorithm performs the following steps:

1. Detects channel-connected switch networks containing at least one bidirectional switch

2. Converts the timing model of unidirectional switches in these networks from the rise/fall/turn-off model to the turn-on/turn-off model of bidirectional switches

3. Optimizes these networks, including removing nets

4. Compiles these networks into accelerative expressions for the XL engine

Switch-XL can remove net n1
The Switch-XL Algorithm (continued)

How Switch-XL Works

Switch-XL accelerates simulation of channel-connected switch networks that contain at least one bidirectional switch. To do this it needs to:

- Convert the timing model of any unidirectional switches in that network to the bidirectional switch timing model
- Remove nets from that network which:
  - Have no out-of-channel connection
  - Are not referenced by the source code
- Convert the network to acceleratable expressions

You cannot interactively obtain value information for removed nets. The simulator will always report their value as a strong X.

You can use the +sxl_keepDeclared option to preserve explicitly declared nets.

You can use the +sxl_keepAll option to preserve all nets.

You can use the +sxl_unidirect option to force all unidirectional switches to use the bidirectional switch timing model.
Timing Model Conversion

In channel-connected regions containing at least one bidirectional switch, Switch-XL converts the unidirectional switch timing model to the bidirectional switch timing model, so that it can accelerate the region:

- It converts rise/fall delays into turn-on/turn-off delays
  — The rise delay becomes the turn-on delay, fall becomes turn-off
- It converts rise/fall/turn-off delays into turn-on/turn-off delays
  — The smaller of the rise and fall delays becomes the turn-on delay

![Diagram showing the conversion of timing models](image-url)
The Switch-XL Algorithm (continued)

Timing Model Conversion

If the Switch-XL algorithm finds a bidirectional switch in a channel-connected switch network, it converts the delay models on unidirectional switches in that network to a turn-on/turn-off delay model, so that all switches in the network share the same delay model.

If you specify only a rise and fall delay for a MOS switch, then the rise and fall delays become the turn-on and turn-off delays, respectively.

If you specify a rise, fall, and turn-off delay for a MOS switch, the Switch-XL algorithm converts the rise and fall delays to a turn-on delay. It uses the smaller of the rise and fall delays as the turn-on delay. The turn-off delay is unchanged.
Summary

In this section, you learned about:

- Switch-level modeling
- The Switch-XL algorithm
- Choose an acceleration algorithm
Summary
Review

1. Name the Verilog bidirectional primitives.
2. How do the delay models for the unidirectional and bidirectional switches differ?
3. A Verilog-XL switch-level network contains which types of switches?
4. What is the range of charge strengths you can assign to a Switch-XL `trireg` net?
5. How does Switch-XL reduce signal strength in a channel?
Review

1. Name the Verilog bidirectional primitives.

The Verilog bidirectional primitives are `tran`, `tranif0`, and `tranif1`, and their resistive versions `rtran`, `rtranif0`, and `rtranif1`.

2. How do the delay models for the unidirectional and bidirectional switches differ?

You can specify rise, fall, and turn-off delay for the unidirectional switches. You can specify only turn-on and turn-off delays for the `tranif0` and `tranif1` bidirectional switches. The `tran` switch cannot have delays.

3. A Verilog-XL switch-level network contains which types of switches?

Switch-level networks contain unidirectional switches, bidirectional switches, or both. Verilog-XL partitions switch-level networks into channel-connected regions.

4. What is the range of charge strengths you can assign to a Switch-XL `trireg` net?

You can assign relative integer charge strengths from 0 to 250 to a `trireg` net accelerated by the Switch-XL algorithm. The zero strength indicates no charge storage.

5. How does Switch-XL reduce signal strength in a channel?

The Switch-XL algorithm treats all drive strengths as higher than charge strengths, maps standard strengths into the range of network strengths, and reduces strength once (by the highest resistance) in the channel.
Appendix A

Glossary
**ALU**

Arithmetic and Logic Unit. The part of the CPU that performs logical and integer arithmetic operations.

**API**

Application Programming Interface. The published conventions by which an application invokes the services provided by some host software.

**ASCII**

American Standard Code for Information Interchange. ASCII uses the lower seven bits to represent some control codes, and space, number, punctuation, and letter characters.

**ASIC**

Application-Specific Integrated Circuit. An integrated circuit defined by the interconnection of vendor-supplied library components to support a specific application.

**AST**

Abstract Syntax Tree. The Cadence 5.X library object which represents the VHDL language in an intermediate form for the Cadence NC-Sim simulator.

**BDD**

Binary Decision Diagram. An acyclic binary graph. All Boolean logic can be represented as a BDD and thus mathematically minimized.

**Blocking Assignment**

In the Verilog HDL, a procedural assignment using the "=" operator, that until complete, blocks execution of the subsequent sequential statement (See Nonblocking Assignment).
**Bourne Shell**

A CLI shell and script language for Unix written by S.R. Bourne of Bell Laboratories in 1978.

**C**

A terse, low-level, popular programming language based on B and developed by Dennis Ritchie at AT&T Bell Labs in 1972.

**CLI**

Command Line Interpreter. A program which reads and executes textual commands from the user or from a file.

**COD**

Code. The Cadence 5.X library object that stores platform-specific machine code for the Cadence NC-Sim simulator.

**Combinational Logic**

Logic for which the state of the inputs solely determines the state of the outputs (See Sequential Logic).

**Compiler Directive**

In the Verilog HDL, a directive to the compiler, that until overridden or reset, affects its subsequent interpretation of Verilog source. All Verilog compiler directives start with the `''` (accent grave) character.

**CPU**

Central Processing Unit. That part of a computer that controls all the other parts. A CPU typically consists of a control unit, an arithmetic/logic unit, and memory.

**CRC**

Customer Response Center. The Cadence hotline for customers with a current support agreement. 1-877-237-4911
CTLF
Compiled Timing Library Format. Standard format for technology-specific component timing data.

DSP
Digital Signal Processing. Processing of the digital form of an analog signal (typically audio or video).

DUT
Device Under Test. That part of the manufacturing test environment which represents the product. The remainder of the test environment is considered the testbench.

ECL
Emitter Coupled Logic. A logic design technology coupled through the emitter (as apposed to the collector).

EDA
Electronic Design Automation. Software tools for development of microelectronic circuits and systems.

EVCD
Extended Value Change Dump. The file format used to store port value, direction, and strength data for postprocessing tools.

FIFO
First-In-First-Out. A data structure from which items are taken out in the same order they are put in.

FMI
Foreign Model Interface. A component of the Cadence C interface to its VHDL simulators. It consists of a library of routines by which you import models written in C.

FPGA
Field-Programmable Gate Array. A gate array where the logic function can be programmed in the field.
FSM

Finite State Machine. A function that maps a sequence of input events into a sequence of output events.

GCF

General Constraint Format. A Cadence standard format for constraint data. Physically knowledgeable synthesis does not use GCF.

GUI

Graphical User Interface. Interface with the user by way of pictures (instead of words).

HAL

HDL Analysis and Lint. A component of the Cadence Verification Cockpit that verifies semantic consistency of Verilog or VHDL source code.

HDL

Hardware Description Language. A language (such as Verilog or VHDL) that describes the behavior of electronic circuitry.

Instance

An individual object of a given defined type. All instances of a given defined type share the features of that type, but each instance has its own internal state and its own environment.

I/O

Input/Output. Communication between a computer and its environment, i.e. other computers, peripheral equipment, and users.

IC

Integrated Circuit. A microelectronic device consisting of many interconnected semiconductor components fabricated on a die typically cut from silicon.
**Identifier**

A user-provided object name. Verilog functions, modules, primitives, nets, statement blocks, tasks, and variables are examples of such objects.

**IEEE**

Institute of Electrical and Electronics Engineers. The predominant technical professional society supporting electrical and electronic issues.

**INCA**


**Integer Number**

Any rational number containing no fractional part (See Real Number).

**IP**

Intellectual Property. Commonly refers to material which qualifies for copyright status, but not ideas or physical objects.

**LCC**

Levelized Compiled Code. A technique for simulation of digital logic that implements the evaluation of levelized boolean logic as a sequence of machine instructions.

**Lexeme**

The minimal meaningful unit of a language (see Lexical, Token). Verilog lexemes include identifiers, keywords, literals, and punctuation.

**Lexical**

Of or relating to the vocabulary of a language (see Syntax).
LFSR

Linear Feedback Shift Register. A shift register whose feedback function is the modulo 2 addition of a subset of its current terms. The subset can be chosen to make the LFSR a maximal-length low-autocorrelation counter in the Galois domain, i.e. a pseudo-random bit stream generator.

LHS

Left Hand Side. Often used to refer to the destination of an assignment operation.

LIFO

Last-In-First-Out. A data structure from which items are taken out in the reverse of the order they are put in.

Literal

An unidentified constant directly included in the executable text.

LMC


LMG

Logic Modeling Group (of Synopsys, Inc.). Purveyor of hardware and software simulation models. Acquired as Logic Modeling Corporation (LMC) in 1994. LMC formed in 1992 as the merger of Logic Automation and LMSI.

LMSFI

LMC Simulator Function Interface. A Cadence VHDL simulator interface to the LMC hardware modeler.

LRM

LSB

Least Significant Bit. The least significant bit of a binary number. Usually the rightmost.

MCD

MultiChannel Descriptor. In the Verilog language, a 32-bit quantity whose bit positions that are 1 represent the channels for a write operation.

MIPD

Module Input Port Delay. A Cadence Verilog-XL simulator term describing delay to a module input port. Delays are inertial and use the three-transition model. MIPDs distribute the same delay to each accelerated primitive in the fanout of the module input port, regardless of hierarchy. The Cadence NC-Verilog simulator does not use MIPDs for interconnect delay.

MITD

Multi-Source Interconnect Transport Delay. A Cadence Verilog-XL simulator term. A MITD is similar to a SITD, but permits you to specify unique source-to-load paths. The NC-Verilog simulator does not use MITDs for interconnect delay.

Module

A packaged assembly of electronic subcomponents. A Verilog module is the basic building block of the HDL. A module encapsulates and defines a piece of the design, as an object that you can instantiate one or more times.

Module Path

In the Verilog HDL, a connection defined inside a specify block between a source input or inout signal and a destination inout or output signal, for the purpose of assigning a delay.

Module Path Delay

In the Verilog HDL, the propagation delay assigned to a module path.
Module Parameter

In the Verilog HDL, a flexible data type constant declared with the parameter keyword, that you can use to parameterize the module, and can redefine on a per-instance basis to reconfigure the individual instance.

MOS

Metal Oxide Semiconductor. Named for the three materials in the gate of a MOSFET.

MSB

Most Significant Bit. The most significant bit of a binary number. Usually the leftmost.

Nonblocking Assignment

In the Verilog HDL a procedural assignment using the "<=" operator, that does not block execution of the subsequent sequential statement (See Blocking Assignment).

NT


OMI

Open Model Interface. IEEE Std. 1499. A simulation interface to protected design or testbench components written in C or an HDL.

OOMR

Out-Of-Module Reference. Reference to a Verilog design object declared in a module definition other than the module which references the object.

OVI

Open Verilog International. A special interest group promoting the specification and use of the Verilog language.
PDF
Portable Document Format. Adobe file format for delivery of documents independent of the editing tool that created them.

PLI
Programming Language Interface. An applications programming interface for Verilog tools.

Primitive
An essential function built into a programming language. A Verilog primitive is a built-in transistor or gate-level model.

Procedural Assignment
In the Verilog HDL, any assignment made within a procedural code block (See Procedural Block).

Procedural Block
In the Verilog HDL, a code block following the always or initial keywords.

Procedural Delay
In the Verilog HDL, a delay made with the "#" token that prefaces a procedural statement.

Procedural Statement
In the Verilog HDL, any statement made within a procedural code block (See Procedural Block).

RAM
Random-Access Memory. Nominally, a data storage device for which access is equally efficient in any order. This term has come to commonly mean read/write memory, as apposed to ROM.

Real Number
Any rational or irrational number containing no imaginary part (See Integer Number).
RHS
Right Hand Side. Often used to refer to the source expression of an assignment operation.

RISC
Reduced Instruction Set Computer. A processor that executes a small set of simple instructions very rapidly.

ROM
Read-Only Memory. A data storage device, which contents are determined by the manufacturing process, and not alterable in the field.

RTL
Register Transfer Level. The subset of an HDL that describes hardware in terms of its registers and the transfer of data between them.

SDF
Standard Delay Format. The file format used to exchange timing data between EDA tools.

SDPD
State-Dependent Path Delay. Module path delay specified to apply only when module input ports are in the specified state.

Semantics
The meaning of a constituent of a language (see Syntax).

Sequential Logic
Logic for which internal state storage at least partially determines the state of the outputs (See Combinational Logic).
**SFB**


**SHM**

Simulation History Manager. A Cadence tool that manages simulation object value transition data for display and comparison.

**SIG**

Signature. The Cadence 5.X library object which represents a unique instantiation of a Verilog module for the Cadence NC-Sim simulator.

**SITD**

Single-Source Interconnect Transport Delay. A Cadence Verilog-XL simulator term. A SITD is similar to a MIPD, but delays are transport, under pulse control, and use the six-transition model. The NC-Verilog simulator does not use SITDs for interconnect delay.

**Specify Block**

In the Verilog HDL, a statement block between the `specify` and `endspecify` keywords, defined within a `module`, that contains module path timing information (See Module Path).

**SPF**


**SPICE**

Simulated Program (with) Integrated Circuit Emphasis. A general-purpose circuit simulation program for the analog domain.

**SSS**

Simulation Snapshot. The Cadence 5.X library object that stores the run-time state of an elaborated design for the Cadence NC-Sim simulator.
**SWIFT™**


**Switch-XL**

The Cadence Verilog-XL simulator engine that accelerates the simulation of bidirectional Verilog primitives.

**Syntax**

The rules governing the assembly of lexemes into constituents of a language (see Semantics).

**System Task**

In the Verilog HDL, a procedure that is either built-in or defined by the user via the API (SEE API). All Verilog system tasks start with the ‘$’ (dollar sign) character.

**Tcl**

Tool Command Language. A Lisp-like textual interpreted string processing language for interacting with software tools, developed by John Ousterhout at UCB in 1998.

**Timing Check**

In the Verilog HDL, a system task that compares the time between two events and logs an error report if the time violates the specified limits.

**Token**

An essential unit of a language (see Lexeme). Verilog tokens include identifiers, keywords, and operators.

**Turbo Mode**

Cadence Verilog-XL simulator acceleration modes. You can select increasing levels of Turbo acceleration, each at the expense of some design visibility.
UDP
User-Defined Primitive. Gate-level combinational or sequential Verilog primitive defined by the user.

UNIX®
An interactive time-sharing operating system invented in 1969 by Ken Thompson after Bell Labs left the Multics project. A registered trademark of The Open Group.

VCC
The power supply rail most closely associated with the bipolar npn transistor collector.

VCD
Value Change Dump. The file format used to store object value transition data for postprocessing tools.

VCL
Value Change Link. A PLI mechanism for dynamically monitoring signal value changes during simulation.

VDD
The power supply rail most closely associated with the n-channel MOS transistor drain.

Verilog
A C-like HDL. IEEE std. 1364.

VHDL
VHSIC Hardware Description Language. An Ada-like HDL. IEEE std. 1076.

VLSI
Very Large Scale Integration. ICs containing several hundred thousand gates.
VPI
Verilog Procedural Interface. An applications programming interface for Verilog tools. Originally the OVI 2.0 PLI specification. Still sometimes referred to as "PLI 2.0".

VSS
The power supply rail most closely associated with the $n$-channel MOS transistor source.

VST
Verilog Syntax Tree. The Cadence 5.X library object which represents the Verilog language in an intermediate form for the Cadence NC-Sim simulator.

White Space
The tab, newline, formfeed, return, and space characters. Verilog parsers ignore these characters except where needed to separate tokens, or in string literals.

XL
The Cadence Verilog-XL simulator engine that accelerates the simulation of most Verilog primitives.
Index

Command Line Plus Options
+access 3-22
+autoprotect 9-39, 9-41
+define+ 9-23, 20-29
  and library search paths 9-16
+delay_mode_distributed 9-31
+delay_mode_path 9-31
+delay_mode_unit 9-31
+delay_mode_zero 9-31
+gui 13-3
+incdir+ 9-21
+nclinedebug 3-22
+protect 9-43
+save_twin_turbo 20-9
+trace_twin_turbo 20-9
+transport_path_delays 9-33
+twin_turbo 20-9

Command Line Options
-c (compile) 3-19
-f (file) 3-19
-i 12-63
-q (quiet) 3-19
-r 12-67
-s (stop) 12-5
-s (suspend) 3-19
-t (trace) 12-49
-v (library file scan) 9-11
-y (library directory scan) 9-13

Compiler Directives
`celldefine 9-7
`define 5-19, 5-21
  and library search paths 9-16
`delay_mode_distributed 9-31
`delay_mode_path 9-31
`delay_mode_unit 9-31, 20-23
`delay_mode_zero 9-31, 20-23
`endcelldefine 9-7
`endprotect 9-43
`endprotected 9-40
`include 5-19, 5-23, 9-21
`protect 9-43
`protected 9-40
`resetall 5-19, 9-21
`timescale 5-19, 5-25
`undef 5-19, 9-21
`undefineall 9-21
`uselib
  defining macros for use with 9-16
  example of 9-17

Symbols
# (simple delay) 11-19

System Tasks and Functions
$cleartrace 12-49
$countdrivers 12-33
$cupetime 12-75
$display 5-15, 14-9, 14-11
$dumpall 4-27
$dumpfile 4-27
$dumpflush 4-27
$dumplimit 4-27
$dumpoff 4-27
$dumpon 4-27
$dumpvars 4-27
  syntax and use 4-29
$fclose 14-19
$fdisplay 14-19
$finish 4-22, 5-15, 12-71, 20-32
Index

$monitor 14-19
$open 14-19
$fprobe 14-19
fwrite 14-19
hold 8-27
incsave 12-67
input 12-63
list 12-21
list_forces 12-37
monitor 4-23, 5-15, 14-17
monitoroff 14-17
monitoron 14-17
nochange 8-28
period 8-28
readmemb 14-23, 16-11
readmemh 14-23, 16-11
realtime 14-3
recrem 8-28
removal 8-28
reset 12-67
restart 12-67
save 12-67
scope 12-17
sdf_annotate 21-13
settrace 12-49
setup 8-27
setuphold 8-27
shm_close 3-25
shm_open 3-25
shm Probe 3-25
example of node specifiers 3-27
shm_save 3-25
showscapes 12-17
showvariables 12-33
showvars 12-33
skew 8-28
stime 14-3
stop 5-15, 12-5
strobe 14-9, 14-11
strobe 4-23, 5-15, 14-3
timeformat 14-5
width 8-28
write 14-9, 14-11

A

abstraction
behavioral level 2-17
gate level 2-17
RTL (register transfer logic) level 2-17
acceleration algorithms
Turbo 20-3
XL 20-3, 20-17
always 11-5
procedural blocks 11-5
arithmetic operators 10-15
ASIC library modeling 18-1
assignments
continuous 11-53

B

begin and end 11-7
behavioral modeling 11-3
behavioral-level abstraction
definition 2-17
example 2-19
bidirectional port 16-13
bit-wise operators 10-27
blocking procedural assignment 11-13
breakpoints
setting 12-41, 13-15

C

case 19-25, 19-51
case equality operator 10-25
case sensitivity 9-19
case statement 11-41
CLI (Command Line Interface) 12-3
comments 5-3
compiled code simulation 20-10
compiler directives (see Symbols) 5-19, 9-21
concatenation operator 10-7
conditional operator 10-31
conditional statements
default 19-19
directive complete case 19-21
incomplete 19-17
naturally complete 19-15
conditional timing checks 8-33
conflict resolution 6-13
continuous assignments 11-53
cosimulation 9-46

D

data types
  choosing the correct type 6-27
  definition 6-5
  nets 6-7
  parameters 6-21
  registers 6-15
defparam 6-23

delay
  inertial 9-33
  inertial versus transport 8-21, 9-33
  min, typ, max 8-11
  rise, fall, turn-off 8-11
  simple delay specification 5-17
  transport 9-33
distributed delays 8-7, 8-11

E

distributed delays 8-7, 8-11

e downloaded library
edge-sensitive sequential UDP 18-25, 18-26
edge-sensitive timing control 11-21
equality operator 10-23
equality operators 10-21
escaped identifiers 5-11
escaped literals 14-13
examples
  backannotation 21-15
  SDF 21-11
  UDP D flip flop 18-25, 18-26
  UDP latch 18-21

F

file output 14-19 to 14-21
for loop 11-45
force 12-37
forever loop 11-47
fork/join statements 11-7
format specifiers 5-8, 14-13
FSM directives
  enum 19-55
  state_vector 19-55
functions 19-31, 19-33
  system, see System Tasks and System Functions

G

gate-level abstraction
  definition 2-17
  example 2-23

graphical user interface (GUI) 13-5

H

HDL (Hardware Description Language)
  benefits 2-5
  definition 2-3
  features 2-4
  hold check 8-25

I

identifiers 5-9
identity operator 10-25
if and if-else statements 11-39
importing VHDL models 9-45
inequality operator 10-23
inertial delay 9-33
initial 11-5
initial procedural blocks 11-5
inout ports 4-8, 16-13
input ports 4-8
instantiation
  array of instances 7-17
  modules 7-15
  primitives 7-13
  integers 5-5
interpretive simulation 20-10
invoking the Verilog environment 13-3

K

keywords
  endmodule 4-6, 4-14
  module 4-6, 4-14

L

level-sensitive sequential UDP 18-21
levels of abstraction
  behavioral 2-17
  gate 2-17
  RTL (register transfer logic) 2-17
lexical conventions 5-1 to 5-29
### L (continued)

library
- ASIC 18-1

library search
- `uselib` 9-15

 logical equality operator 10-23
 logical operators 10-29
 looping statements 11-43 to 11-51
 lumped delays 8-5, 8-11

### M

MCD (see multichannel descriptor)

memories
- addressing 6-35
- declaration 6-33
- loading 14-23
- modeling 16-3

Modeling
- ASIC Libraries 9-5
- modeling memories 16-3
- module
  - definition 4-6
  - endmodule 4-6, 4-14
  - instances 4-9
  - instantiation 7-15
  - ports 4-7
- module parameters 6-21
- module path
  - transport delays 9-33
  - module path delays 8-9
  - module paths
    - full connection 8-17
    - parallel connection 8-17
- monitoring signal values 14-17
- multichannel descriptor 14-19

### N

named blocks 17-25

Navigator 13-17

NC-Verilog
- invoking 3-19
- ncelab 3-15
- ncsim 3-15
- ncvg 3-15
- negation operators 10-11
- net types 6-9 to 6-13
- nonblocking procedural assignment 11-15

### O

operators
- arithmetic 10-15
- bit-wise 10-27
- case equality 10-25
- concatenation 10-7
- conditional 10-31
- equality 10-21, 10-23
- inequality 10-23
- logical 10-29
- logical equality 10-23
- negation 10-11
- relational 10-19
- replication 10-9
- shift 10-17
- unary reduction 10-13

output ports 4-8

### P

parallel block 11-7

parameter 6-21
- defparam 6-23
- overriding parameters 6-23

parameters
- specify block parameters 8-15
- path delays 8-7, 8-9
- PATHPULSES 8-23
- period check 8-25
- ports
  - inout (bidirectional) 4-8, 16-13
  - input 4-8
  - module 4-7
  - output 4-8

primitives
- conditional type 7-9
- instantiation 7-13
- table of Verilog primitives 7-5
- user defined 7-14

procedural assignments 11-11

procedural blocks 11-5
- always 4-20, 11-5
- initial 4-20, 11-5
R
real numbers  5-5
recovery check  8-25
reg  19-43
register types
  integer  6-17
  real  6-17
  realtime  6-18
  reg  6-17
  time  6-17
relational operators  10-19
release  12-37
repeat loop  11-49
replication operator  10-9
RTL-level abstraction
  definition  2-17
  example  2-21

S
SDF (Standard Delay Format)  21-9 to 21-16
SDF (Standard Delay Format) file
  definition  8-37
  interconnect delays  8-39
  IOPATH Delays  8-41
selecting objects  13-13
sensitivity lists  19-9
sequential block  11-7
setting breakpoints  12-41, 13-15
setup check  8-25
shift operators  10-17
SHM (simulation history manager) database  3-25
SimControl  13-5
simple timing control  11-19
simulation
  algorithms  3-4
  compiled code  20-10
  interpretive  20-10
simulation algorithms  3-4
  cycle-based  3-4
  event-based  3-4
  time-based  3-4
SimVision
  selecting objects  13-13
  setting breakpoints  13-15
skew check  8-25
specify block
  features of  8-13
  parameters  8-15
  syntax  8-17
specparam  8-15
strength modeling  7-21
string constants  5-7
structural modeling
  basics  7-3
  Verilog primitives  7-5
structural-level abstraction
  see gate-level abstraction
system tasks and system functions  5-15
T
tasks
  system, see system tasks and system functions
Tcl (Tool Command Language)  12-3
test fixture
  template  4-15
testbench  4-15 to 4-25
  applying stimulus  4-21
  describing stimulus  4-19
  instantiating a submodule  4-17
text inclusion  5-23
text macro substitutions
  define+  9-23
  'define  5-21
text macros  20-29
time wheel  3-5
timescale  5-25
timing checks  8-25 to 8-35
  conditional  8-33
  notifiers  8-29
timing control  11-17
  edge-sensitive  11-21
  simple delay  11-19
trace mode  12-49
tracing simulation activity  12-49
transport delay  9-33
Twin Turbo  20-9
U
UDP
  combinational  18-17
  edge-sensitive sequential  18-25, 18-26
  level-sensitive sequential  18-21
UDP (user defined primitives)  7-14
Index

unary reduction operators  10-13
unknown logic values  7-12

V
Verilog
   applications  2-11
   definition  2-7
   history  2-9
Verilog-XL  3-13, 3-15
   invoking  3-19
VHDL model import  9-45

W
Watch Objects windows  13-19
waveform display tool
   group  3-28
   menu bar  3-28
   time-display region  3-28
   title bar  3-28
   tool bar  3-28
   waveforms region  3-28
while loop  11-51
white space  5-3
width check  8-25

X
XL algorithm  20-17 to 20-21

Z
zero-delay loop  11-27