

## Contents

- [Main Page](#)
- [Table of content](#)
- [Copyright](#)
- [About the Author](#)
- [List of Figures](#)
- [List of Tables](#)
- [List of Examples](#)
- [Foreword](#)
- [Preface](#)
  - [Who Should Use This Book](#)
  - [How This Book Is Organized](#)
  - [Conventions Used in This Book](#)
- [Acknowledgments](#)
- [Part 1: Basic Verilog Topics](#)
  - [Chapter 1. Overview of Digital Design with Verilog HDL](#)
    - [1.1 Evolution of Computer-Aided Digital Design](#)
    - [1.2 Emergence of HDLs](#)
    - [1.3 Typical Design Flow](#)
    - [1.4 Importance of HDLs](#)
    - [1.5 Popularity of Verilog HDL](#)
    - [1.6 Trends in HDLs](#)
  - [Chapter 2. Hierarchical Modeling Concepts](#)
    - [2.1 Design Methodologies](#)
    - [2.2 4-bit Ripple Carry Counter](#)
    - [2.3 Modules](#)
    - [2.4 Instances](#)
    - [2.5 Components of a Simulation](#)
    - [2.6 Example](#)
    - [2.7 Summary](#)
    - [2.8 Exercises](#)
  - [Chapter 3. Basic Concepts](#)
    - [3.1 Lexical Conventions](#)
    - [3.2 Data Types](#)
    - [3.3 System Tasks and Compiler Directives](#)
    - [3.4 Summary](#)
    - [3.5 Exercises](#)
  - [Chapter 4. Modules and Ports](#)
    - [4.1 Modules](#)

 [4.2 Ports](#)

 [4.3 Hierarchical Names](#)

 [4.4 Summary](#)

 [4.5 Exercises](#)

 [Chapter 5. Gate-Level Modeling](#)

 [5.1 Gate Types](#)

 [5.2 Gate Delays](#)

 [5.3 Summary](#)

 [5.4 Exercises](#)

 [Chapter 6. Dataflow Modeling](#)

 [6.1 Continuous Assignments](#)

 [6.2 Delays](#)

 [6.3 Expressions, Operators, and Operands](#)

 [6.4 Operator Types](#)

 [6.5 Examples](#)

 [6.6 Summary](#)

 [6.7 Exercises](#)

 [Chapter 7. Behavioral Modeling](#)

 [7.1 Structured Procedures](#)

 [7.2 Procedural Assignments](#)

 [7.3 Timing Controls](#)

 [7.4 Conditional Statements](#)

 [7.5 Multiway Branching](#)

 [7.6 Loops](#)

 [7.7 Sequential and Parallel Blocks](#)

 [7.8 Generate Blocks](#)

 [7.9 Examples](#)

 [7.10 Summary](#)

 [7.11 Exercises](#)

 [Chapter 8. Tasks and Functions](#)

 [8.1 Differences between Tasks and Functions](#)

 [8.2 Tasks](#)

 [8.3 Functions](#)

 [8.4 Summary](#)

 [8.5 Exercises](#)

 [Chapter 9. Useful Modeling Techniques](#)

 [9.1 Procedural Continuous Assignments](#)

 [9.2 Overriding Parameters](#)

 [9.3 Conditional Compilation and Execution](#)

 [9.4 Time Scales](#)

- [?](#) [9.5 Useful System Tasks](#)
    - [?](#) [9.6 Summary](#)
    - [?](#) [9.7 Exercises](#)
  -  [Part 2: Advanced Verilog Topics](#)
    -  [Chapter 10. Timing and Delays](#)
      - [?](#) [10.1 Types of Delay Models](#)
      - [?](#) [10.2 Path Delay Modeling](#)
      - [?](#) [10.3 Timing Checks](#)
      - [?](#) [10.4 Delay Back-Annotation](#)
      - [?](#) [10.5 Summary](#)
      - [?](#) [10.6 Exercises](#)
    -  [Chapter 11. Switch-Level Modeling](#)
      - [?](#) [11.1 Switch-Modeling Elements](#)
      - [?](#) [11.2 Examples](#)
      - [?](#) [11.3 Summary](#)
      - [?](#) [11.4 Exercises](#)
    -  [Chapter 12. User-Defined Primitives](#)
      - [?](#) [12.1 UDP basics](#)
      - [?](#) [12.2 Combinational UDPs](#)
      - [?](#) [12.3 Sequential UDPs](#)
      - [?](#) [12.4 UDP Table Shorthand Symbols](#)
      - [?](#) [12.5 Guidelines for UDP Design](#)
      - [?](#) [12.6 Summary](#)
      - [?](#) [12.7 Exercises](#)
    -  [Chapter 13. Programming Language Interface](#)
      - [?](#) [13.1 Uses of PLI](#)
      - [?](#) [13.2 Linking and Invocation of PLI Tasks](#)
      - [?](#) [13.3 Internal Data Representation](#)
      - [?](#) [13.4 PLI Library Routines](#)
      - [?](#) [13.5 Summary](#)
      - [?](#) [13.6 Exercises](#)
    -  [Chapter 14. Logic Synthesis with Verilog HDL](#)
      - [?](#) [14.1 What Is Logic Synthesis?](#)
      - [?](#) [14.2 Impact of Logic Synthesis](#)
      - [?](#) [14.3 Verilog HDL Synthesis](#)
      - [?](#) [14.4 Synthesis Design Flow](#)
      - [?](#) [14.5 Verification of Gate-Level Netlist](#)
      - [?](#) [14.6 Modeling Tips for Logic Synthesis](#)
      - [?](#) [14.7 Example of Sequential Circuit Synthesis](#)
      - [?](#) [14.9 Exercises](#)

 [Chapter 15. Advanced Verification Techniques](#)

-  [15.1 Traditional Verification Flow](#)
-  [15.2 Assertion Checking](#)
-  [15.3 Formal Verification](#)
-  [15.4 Summary](#)

 [Part 3: Appendices](#)

 [Appendix A. Strength Modeling and Advanced Net Definitions](#)

-  [A.1 Strength Levels](#)
-  [A.2 Signal Contention](#)
-  [A.3 Advanced Net Types](#)

 [Appendix B. List of PLI Routines](#)

-  [B.1 Conventions](#)
-  [B.2 Access Routines](#)
-  [B.3 Utility \(tf\\_\) Routines](#)

 [Appendix C. List of Keywords, System Tasks, and Compiler Directives](#)

-  [C.1 Keywords](#)
-  [C.2 System Tasks and Functions](#)
-  [C.3 Compiler Directives](#)

 [Appendix D. Formal Syntax Definition](#)

-  [D.1 Source Text](#)
-  [D.2 Declarations](#)
-  [D.3 Primitive Instances](#)
-  [D.4 Module and Generated Instantiation](#)
-  [D.5 UDP Declaration and Instantiation](#)
-  [D.6 Behavioral Statements](#)
-  [D.7 Specify Section](#)
-  [D.8 Expressions](#)
-  [D.9 General](#)
-  [Endnotes](#)

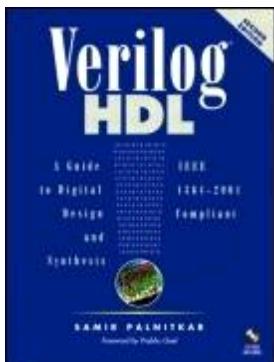
 [Appendix E. Verilog Tidbits](#)

-  [Origins of Verilog HDL](#)
-  [Interpreted, Compiled, Native Compiled Simulators](#)
-  [Event-Driven Simulation, Oblivious Simulation](#)
-  [Cycle-Based Simulation](#)
-  [Fault Simulation](#)
-  [General Verilog Web sites](#)
-  [Architectural Modeling Tools](#)
-  [High-Level Verification Languages](#)
-  [Simulation Tools](#)
-  [Hardware Acceleration Tools](#)

- [?](#) [In-Circuit Emulation Tools](#)
- [?](#) [Coverage Tools](#)
- [?](#) [Assertion Checking Tools](#)
- [?](#) [Equivalence Checking Tools](#)
- [?](#) [Formal Verification Tools](#)
- [?](#) [Appendix F. Verilog Examples](#)
  - [?](#) [F.1 Synthesizable FIFO Model](#)
  - [?](#) [F.2 Behavioral DRAM Model](#)
- [?](#) [Bibliography](#)
  - [?](#) [Manuals](#)
  - [?](#) [Books](#)
  - [?](#) [Quick Reference Guides](#)
- [?](#) [About the CD-ROM](#)
  - [?](#) [Using the CD-ROM](#)
  - [?](#) [Technical Support](#)

[\[ Team LiB \]](#)

[NEXT ▶](#)



[Table of Contents](#)

[Examples](#)

**Verilog HDL: A Guide to Digital Design and Synthesis, Second Edition**

By [Samir Palnitkar](#)

[START READING](#)

Publisher: Prentice Hall PTR

Pub Date: February 21, 2003

ISBN: 0-13-044911-3

Pages: 496

Written for both experienced and new users, this book gives you broad coverage of Verilog HDL. The book stresses the practical design and verification perspective of Verilog rather than emphasizing only the

language aspects. The information presented is fully compliant with the IEEE 1364-2001 Verilog HDL standard.

- 
- Describes state-of-the-art verification methodologies
- 
- Provides full coverage of gate, dataflow (RTL), behavioral and switch modeling
- 
- Introduces you to the Programming Language Interface (PLI)
- 
- Describes logic synthesis methodologies
- 
- Explains timing and delay simulation
- 
- Discusses user-defined primitives
- 
- Offers many practical modeling tips

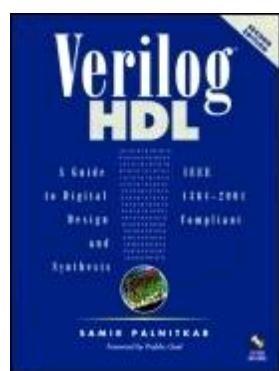
Includes over 300 illustrations, examples, and exercises, and a Verilog resource list. Learning objectives and summaries are provided for each chapter.

[\[ Team LiB \]](#)

[NEXT ▶](#)

[\[ Team LiB \]](#)

[◀ PREVIOUS](#) [NEXT ▶](#)



[Table of Contents](#)

[Examples](#)

**Verilog HDL: A Guide to Digital Design and Synthesis, Second Edition**

By [Samir Palnitkar](#)

[START READING](#)

Publisher: Prentice Hall PTR

Pub Date: February 21, 2003

ISBN: 0-13-044911-3

Pages: 496

[Copyright](#)

[About the Author](#)

[List of Figures](#)

[List of Tables](#)

[List of Examples](#)

[Foreword](#)

[Preface](#)

[Who Should Use This Book](#)

[How This Book Is Organized](#)

[Conventions Used in This Book](#)

[Acknowledgments](#)

[Part 1. Basic Verilog Topics](#)

[Chapter 1. Overview of Digital Design with Verilog HDL](#)

[Section 1.1. Evolution of Computer-Aided Digital Design](#)

[Section 1.2. Emergence of HDLs](#)

[Section 1.3. Typical Design Flow](#)

[Section 1.4. Importance of HDLs](#)

[Section 1.5. Popularity of Verilog HDL](#)

[Section 1.6. Trends in HDLs](#)

[Chapter 2. Hierarchical Modeling Concepts](#)

[Section 2.1. Design Methodologies](#)

[Section 2.2. 4-bit Ripple Carry Counter](#)

[Section 2.3. Modules](#)

[Section 2.4. Instances](#)

[Section 2.5. Components of a Simulation](#)

[Section 2.6. Example](#)

[Section 2.7. Summary](#)

[Section 2.8. Exercises](#)

[Chapter 3. Basic Concepts](#)

[Section 3.1. Lexical Conventions](#)

[Section 3.2. Data Types](#)

[Section 3.3. System Tasks and Compiler Directives](#)

[Section 3.4. Summary](#)

[Section 3.5. Exercises](#)

[Chapter 4. Modules and Ports](#)

[Section 4.1. Modules](#)

[Section 4.2. Ports](#)

[Section 4.3. Hierarchical Names](#)

[Section 4.4. Summary](#)

[Section 4.5. Exercises](#)

[Chapter 5. Gate-Level Modeling](#)

[Section 5.1. Gate Types](#)

[Section 5.2. Gate Delays](#)

[Section 5.3. Summary](#)

[Section 5.4. Exercises](#)

[Chapter 6. Dataflow Modeling](#)

[Section 6.1. Continuous Assignments](#)

[Section 6.2. Delays](#)

[Section 6.3. Expressions, Operators, and Operands](#)

[Section 6.4. Operator Types](#)

[Section 6.5. Examples](#)

[Section 6.6. Summary](#)

[Section 6.7. Exercises](#)

[Chapter 7. Behavioral Modeling](#)

[Section 7.1. Structured Procedures](#)

[Section 7.2. Procedural Assignments](#)

[Section 7.3. Timing Controls](#)

[Section 7.4. Conditional Statements](#)

[Section 7.5. Multiway Branching](#)

[Section 7.6. Loops](#)

[Section 7.7. Sequential and Parallel Blocks](#)

[Section 7.8. Generate Blocks](#)

[Section 7.9. Examples](#)

[Section 7.10. Summary](#)

[Section 7.11. Exercises](#)

[Chapter 8. Tasks and Functions](#)

[Section 8.1. Differences between Tasks and Functions](#)

[Section 8.2. Tasks](#)

[Section 8.3. Functions](#)

[Section 8.4. Summary](#)

[Section 8.5. Exercises](#)

[Chapter 9. Useful Modeling Techniques](#)

[Section 9.1. Procedural Continuous Assignments](#)

[Section 9.2. Overriding Parameters](#)

[Section 9.3. Conditional Compilation and Execution](#)

[Section 9.4. Time Scales](#)

[Section 9.5. Useful System Tasks](#)

[Section 9.6. Summary](#)

[Section 9.7. Exercises](#)

[Part 2. Advanced Verilog Topics](#)

[Chapter 10. Timing and Delays](#)

[Section 10.1. Types of Delay Models](#)

[Section 10.2. Path Delay Modeling](#)

[Section 10.3. Timing Checks](#)

[Section 10.4. Delay Back-Annotation](#)

[Section 10.5. Summary](#)

[Section 10.6. Exercises](#)

[Chapter 11. Switch-Level Modeling](#)

[Section 11.1. Switch-Modeling Elements](#)

[Section 11.2. Examples](#)

[Section 11.3. Summary](#)

[Section 11.4. Exercises](#)

[Chapter 12. User-Defined Primitives](#)

[Section 12.1. UDP basics](#)

[Section 12.2. Combinational UDPs](#)

[Section 12.3. Sequential UDPs](#)

[Section 12.4. UDP Table Shorthand Symbols](#)

[Section 12.5. Guidelines for UDP Design](#)

[Section 12.6. Summary](#)

[Section 12.7. Exercises](#)

[Chapter 13. Programming Language Interface](#)

[Section 13.1. Uses of PLI](#)

[Section 13.2. Linking and Invocation of PLI Tasks](#)

[Section 13.3. Internal Data Representation](#)

[Section 13.4. PLI Library Routines](#)

[Section 13.5. Summary](#)

[Section 13.6. Exercises](#)

[Chapter 14. Logic Synthesis with Verilog HDL](#)

[Section 14.1. What Is Logic Synthesis?](#)

[Section 14.2. Impact of Logic Synthesis](#)

[Section 14.3. Verilog HDL Synthesis](#)

[Section 14.4. Synthesis Design Flow](#)

[Section 14.5. Verification of Gate-Level Netlist](#)

[Section 14.6. Modeling Tips for Logic Synthesis](#)

[Section 14.7. Example of Sequential Circuit Synthesis](#)

[Section 14.9. Exercises](#)

[Chapter 15. Advanced Verification Techniques](#)

[Section 15.1. Traditional Verification Flow](#)

[Section 15.2. Assertion Checking](#)

[Section 15.3. Formal Verification](#)

[Section 15.4. Summary](#)

[Part 3. Appendices](#)

[Appendix A. Strength Modeling and Advanced Net Definitions](#)

[Section A.1. Strength Levels](#)

[Section A.2. Signal Contention](#)

[Section A.3. Advanced Net Types](#)

[Appendix B. List of PLI Routines](#)

[Section B.1. Conventions](#)

[Section B.2. Access Routines](#)

[Section B.3. Utility \(tf\\_\) Routines](#)

[Appendix C. List of Keywords, System Tasks, and Compiler Directives](#)

[Section C.1. Keywords](#)

[Section C.2. System Tasks and Functions](#)

[Section C.3. Compiler Directives](#)

[Appendix D. Formal Syntax Definition](#)

[Section D.1. Source Text](#)

[Section D.2. Declarations](#)

[Section D.3. Primitive Instances](#)

[Section D.4. Module and Generated Instantiation](#)

[Section D.5. UDP Declaration and Instantiation](#)

[Section D.6. Behavioral Statements](#)

[Section D.7. Specify Section](#)

[Section D.8. Expressions](#)

[Section D.9. General](#)

[Endnotes](#)

[Appendix E. Verilog Tidbits](#)

[Origins of Verilog HDL](#)

[Interpreted, Compiled, Native Compiled Simulators](#)

[Event-Driven Simulation, Oblivious Simulation](#)

[Cycle-Based Simulation](#)

[Fault Simulation](#)

[General Verilog Web sites](#)

[Architectural Modeling Tools](#)

[High-Level Verification Languages](#)

[Simulation Tools](#)

[Hardware Acceleration Tools](#)

[In-Circuit Emulation Tools](#)

[Coverage Tools](#)

[Assertion Checking Tools](#)

[Equivalence Checking Tools](#)

[Formal Verification Tools](#)

[Appendix F. Verilog Examples](#)

[Section F.1. Synthesizable FIFO Model](#)

[Section F.2. Behavioral DRAM Model](#)

[Bibliography](#)

[Manuals](#)

[Books](#)

[Quick Reference Guides](#)

[About the CD-ROM](#)

[Using the CD-ROM](#)

[Technical Support](#)

[\[ Team LiB \]](#)



[\[ Team LiB \]](#)



# Copyright

2003 Sun Microsystems, Inc. 2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

All rights reserved. This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX system and from the Berkeley 4.3 BSD system, licensed from the University of California. Third-party software, including font technology in this product, is protected by copyright and licensed from Sun's Suppliers.

**RESTRICTED RIGHTS LEGEND:** Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

## TRADEMARKS

Sun, Sun Microsystems, the Sun logo, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and may be protected as trademarks in other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. OPEN LOOK is a registered trademark of Novell, Inc. PostScript and Display PostScript are trademarks of Adobe Systems, Inc. Verilog is a registered trademark of Cadence Design Systems, Inc. Verilog-XL is a trademark of Cadence Design Systems, Inc. VCS is a trademark of Viewlogic Systems, Inc. Magellan is a registered trademark of Systems Science, Inc. VirSim is a trademark of Simulation Technologies, Inc. Signalscan is a trademark of Design Acceleration, Inc. All other product, service, or company names mentioned herein are claimed as trademarks and trade names by their respective companies.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. in the United States and may be protected as trademarks in other countries. SPARCcenter, SPARCcluster, SPARCompiler, SPARCdesign, SPARC811, SPARCengine, SPARCprinter, SPARCserver, SPARCstation, SPARCstorage, SPARCworks, microSPARC, microSPARC-II, and UltraSPARC are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who develop OPEN LOOK GUIs under license to Sun. Sun's licensees include Apple Computer, Inc.,

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## About the Author

Samir Palnitkar is currently the President of Jambo Systems, Inc., a leading ASIC design and verification services company which specializes in high-end designs for microprocessor, networking, and communications applications. Mr. Palnitkar is a serial entrepreneur. He was the founder of Integrated Intellectual Property, Inc., an ASIC company that was acquired by Lattice Semiconductor, Inc. Later he founded Obongo, Inc., an e-commerce software firm that was acquired by AOL Time Warner, Inc.

Mr. Palnitkar holds a Bachelor of Technology in Electrical Engineering from Indian Institute of Technology, Kanpur, a Master's in Electrical Engineering from University of Washington, Seattle, and an MBA degree from San Jose State University, San Jose, CA.

Mr. Palnitkar is a recognized authority on Verilog HDL, modeling, verification, logic synthesis, and EDA-based methodologies in digital design. He has worked extensively with design and verification on various successful microprocessor, ASIC, and system projects. He was the lead developer of the Verilog framework for the shared memory, cache coherent, multiprocessor architecture, popularly known as the UltraSPARCTM Port Architecture, defined for Sun's next generation UltraSPARC-based desktop systems. Besides the UltraSPARC CPU, he has worked on a number of diverse design and verification projects at leading companies including Cisco, Philips, Mitsubishi, Motorola, National, Advanced Micro Devices, and Standard Microsystems.

Mr. Palnitkar was also a leading member of the group that first experimented with cycle-based simulation technology on joint projects with simulator companies. He has extensive experience with a variety of EDA tools such as Verilog-NC, Synopsys VCS, Specman, Vera, System Verilog, Synopsys, SystemC, Verplex, and Design Data Management Systems.

Mr. Palnitkar is the author of three US patents, one for a novel method to analyze finite state machines, a second for work on cycle-based simulation technology and a third(pending approval) for a unique e-commerce tool. He has also published several technical papers. In his spare time, Mr. Palnitkar likes to play cricket, read books, and travel the world.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

# List of Figures

[Figure 1-1](#) Typical Design Flow

[Figure 2-1](#) Top-down Design Methodology

[Figure 2-2](#) Bottom-up Design Methodology

[Figure 2-3](#) Ripple Carry Counter

[Figure 2-4](#) T-flipflop

[Figure 2-5](#) Design Hierarchy

[Figure 2-6](#) Stimulus Block Instantiates Design Block

[Figure 2-7](#) Stimulus and Design Blocks Instantiated in a Dummy Top-Level Module

[Figure 2-8](#) Stimulus and Output Waveforms

[Figure 3-1](#) Example of Nets

[Figure 4-1](#) Components of a Verilog Module

[Figure 4-2](#) SR Latch

[Figure 4-3](#) I/O Ports for Top and Full Adder

[Figure 4-4](#) Port Connection Rules

[Figure 4-5](#) Design Hierarchy for SR Latch Simulation

[Figure 5-1](#) Basic Gates

[Figure 5-2](#) Pass-1 Net Goto

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

# List of Tables

[Table 3-1](#) Value Levels

[Table 3-2](#) Strength Levels

[Table 3-3](#) Special Characters

[Table 3-4](#) String Format Specifications

[Table 5-1](#) Truth Tables for And/Or Gates

[Table 5-2](#) Truth Tables for Buf/Not Gates

[Table 5-3](#) Truth Tables for Bufif/Notif Gates

[Table 6-1](#) Operator Types and Symbols

[Table 6-2](#) Equality Operators

[Table 6-3](#) Truth Tables for Bitwise Operators

[Table 6-4](#) Operator Precedence

[Table 8-1](#) Tasks and Functions

[Table 11-1](#) Logic Tables for NMOS and PMOS

[Table 11-2](#) Strength Reduction by Resistive Switches

[Table 11-3](#) Delay Specification on MOS and CMOS Switches

[Table 11-4](#) Delay Specification for Bidirectional Switches

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

# List of Examples

[Example 2-1](#) Module Instantiation

[Example 2-2](#) Illegal Module Nesting

[Example 2-3](#) Ripple Carry Counter Top Block

[Example 2-4](#) Flipflop T\_FF

[Example 2-5](#) Flipflop D\_F

[Example 2-6](#) Stimulus Block

[Example 2-7](#) Output of the Simulation

[Example 3-1](#) Example of Register

[Example 3-2](#) Signed Register Declaration

[Example 3-3](#) \$display Task

[Example 3-4](#) Special Characters

[Example 3-5](#) Monitor Statement

[Example 3-6](#) Stop and Finish Tasks

[Example 3-7](#) `define Directive

[Example 3-8](#) `include Directive

[Example 4-1](#) Components of SR Latch

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

# Foreword

From a modest beginning in early 1984 at Gateway Design Automation, the Verilog hardware description language has become an industry standard as a result of extensive use in the design of integrated circuit chips and digital systems. Verilog came into being as a proprietary language supported by a simulation environment that was the first to support mixed-level design representations comprising switches, gates, RTL, and higher levels of abstractions of digital circuits. The simulation environment provided a powerful and uniform method to express digital designs as well as tests that were meant to verify such designs.

There were three key factors that drove the acceptance and dominance of Verilog in the marketplace. First, the introduction of the Programming Language Interface (PLI) permitted users of Verilog to literally extend and customize the simulation environment. Since then, users have exploited the PLI and their success at adapting Verilog to their environment has been a real winner for Verilog. The second key factor which drove Verilog's dominance came from Gateways paying close attention to the needs of the ASIC foundries and enhancing Verilog in close partnership with Motorola, National, and UTMC in the 1987-1989 time-frame. The realization that the vast majority of logic simulation was being done by designers of ASIC chips drove this effort. With ASIC foundries blessing the use of Verilog and even adopting it as their internal sign-off simulator, the industry acceptance of Verilog was driven even further. The third and final key factor behind the success of Verilog was the introduction of Verilog-based synthesis technology by Synopsys in 1987. Gateway licensed its proprietary Verilog language to Synopsys for this purpose. The combination of the simulation and synthesis technologies served to make Verilog the language of choice for the hardware designers.

The arrival of the VHDL (VHSIC Hardware Description Language), along with the powerful alignment of the remaining EDA vendors driving VHDL as an IEEE standard, led to the placement of Verilog in the public domain. Verilog was inducted as the IEEE 1364 standard in 1995. Since 1995, many enhancements were made to Verilog HDL based on requests from Verilog users. These changes were incorporated into the latest IEEE 1364-2001 Verilog standard. Today, Verilog has become the language of choice for digital design and is the basis for synthesis, verification, and place and route technologies.

Samir's book is an excellent guide to the user of the Verilog language. Not only does it explain the language constructs with a rich variety of examples, it also goes into details of the usage of the PLI and the application of synthesis technology. The topics in the book are arranged logically and flow very smoothly. This book is written from a very practical design perspective rather than with a focus simply on the syntax aspects of the language.

This second edition of Samir's book is unique in two ways. Firstly, it incorporates all enhancements described in IEEE 1364-2001 standard. This ensures that the readers of the book are working with the latest information on Verilog. Secondly, a new chapter has been added on advanced verification techniques that are now an integral part of Verilog-based methodologies. Knowledge of these techniques is critical to Verilog users who design and verify multi-million gate systems.

I can still remember the challenges of teaching Verilog and its associated design and verification methodologies to users. By using Samir's book, beginning users of Verilog will become productive sooner, and experienced Verilog users will get the latest in a convenient reference book that can refresh their understanding of Verilog. This book is a must for any Verilog user.

Prabhu Goel

Former President of Gateway Design Automation

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## Preface

During my earliest experience with Verilog HDL, I was looking for a book that could give me a "jump start" on using Verilog HDL. I wanted to learn basic digital design paradigms and the necessary Verilog HDL constructs that would help me build small digital circuits, using Verilog and run simulations. After I had gained some experience with building basic Verilog models, I wanted to learn to use Verilog HDL to build larger designs. At that time, I was searching for a book that broadly discussed advanced Verilog-based digital design concepts and real digital design methodologies. Finally, when I had gained enough experience with digital design and verification of real IC chips, though manuals of Verilog-based products were available, from time to time, I felt the need for a Verilog HDL book that would act as a handy reference. A desire to fill this need led to the publication of the first edition of this book.

It has been more than six years since the publication of the first edition. Many changes have occurred during these years. These years have added to the depth and richness of my design and verification experience through the diverse variety of ASIC and microprocessor projects that I have successfully completed in this duration. I have also seen state-of-the-art verification methodologies and tools evolve to a high level of maturity. The IEEE 1364-2001 standard for Verilog HDL has been approved. The purpose of this second edition is to incorporate the IEEE 1364-2001 additions and introduce to Verilog users the latest advances in verification. I hope to make this edition a richer learning experience for the reader.

This book emphasizes breadth rather than depth. The book imparts to the reader a working knowledge of a broad variety of Verilog-based topics, thus giving the reader a global understanding of Verilog HDL-based design. The book leaves the in-depth coverage of each topic to the Verilog HDL language reference manual and the reference manuals of the individual Verilog-based products.

This book should be classified not only as a Verilog HDL book but, more generally, as a digital design book. It is important to realize that Verilog HDL is only a tool used in digital design. It is the means to an end—the digital IC chip. Therefore, this book stresses the practical design perspective more than the mere language aspects of Verilog HDL. With HDL-based digital design having become a necessity, no digital designer can afford to ignore HDLs.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## Who Should Use This Book

The book is intended primarily for beginners and intermediate-level Verilog users. However, for advanced Verilog users, the broad coverage of topics makes it an excellent reference book to be used in conjunction with the manuals and training materials of Verilog-based products.

The book presents a logical progression of Verilog HDL-based topics. It starts with the basics, such as HDL-based design methodologies, and then gradually builds on the basics to eventually reach advanced topics, such as PLI or logic synthesis. Thus, the book is useful to Verilog users with varying levels of expertise as explained below.

- 
- Students in logic design courses at universities
- [Part 1](#) of this book is ideal for a foundation semester course in Verilog HDL-based logic design. Students are exposed to hierarchical modeling concepts, basic Verilog constructs and modeling techniques, and the necessary knowledge to write small models and run simulations.
- 
- New Verilog users in the industry
- Companies are moving to Verilog HDL- based design. [Part 1](#) of this book is a perfect jump start for designers who want to orient their skills toward HDL-based design.
- 
- Users with basic Verilog knowledge who need to understand advanced concepts
- [Part 2](#) of this book discusses advanced concepts, such as UDPs, timing simulation, PLI, and logic synthesis, which are necessary for graduation from small Verilog models to larger designs.
- 
- Verilog experts
- All Verilog topics are covered, from the basics modeling constructs to advanced topics like PLIs, logic synthesis, and advanced verification techniques. For Verilog experts, this book is a handy reference to be used along with the IEEE Standard Verilog Hardware Description Language reference manual.

The material in the book sometimes leans toward an Application Specific Integrated Circuit (ASIC) design methodology. However, the concepts explained in the book are general enough to be applicable to the design of FPGAs, PALs, buses, boards, and systems. The book uses Medium Scale Integration (MSI) logic examples to simplify discussion. The same concepts apply to VLSI designs.



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## How This Book Is Organized

This book is organized into three parts.

[Part 1](#), Basic Verilog Topics, covers all information that a new user needs to build small Verilog models and run simulations. Note that in [Part 1](#), gate-level modeling is addressed before behavioral modeling. I have chosen to do so because I think that it is easier for a new user to see a 1-1 correspondence between gate-level circuits and equivalent Verilog descriptions. Once gate-level modeling is understood, a new user can move to higher levels of abstraction, such as data flow modeling and behavioral modeling, without losing sight of the fact that Verilog HDL is a language for digital design and is not a programming language. Thus, a new user starts off with the idea that Verilog is a language for digital design. New users who start with behavioral modeling often tend to write Verilog the way they write their C programs. They sometimes lose sight of the fact that they are trying to represent hardware circuits by using Verilog. [Part 1](#) contains nine chapters.

[Part 2](#), Advanced Verilog Topics, contains the advanced concepts a Verilog user needs to know to graduate from small Verilog models to larger designs. Advanced topics such as timing simulation, switch-level modeling, UDPs, PLI, logic synthesis, and advanced verification techniques are covered. [Part 2](#) contains six chapters.

[Part 3](#), Appendices, contains information useful as a reference. Useful information, such as strength-level modeling, list of PLI routines, formal syntax definition, Verilog tidbits, and large Verilog examples is included. [Part 3](#) contains six appendices.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## Conventions Used in This Book

Table PR-1 describes the type changes and symbols used in this book.

Table PR-1. Typographic Conventions

Typeface or Symbol	Description	Examples
AaBbCc123	Keywords, system tasks and compiler directives that are a part of Verilog HDL	and, nand, \$display, `define
AaBbCc123	Emphasis	cell characterization, instantiation
AaBbCc123	Names of signals, modules, ports, etc.	fulladd4, D_FF, out

A few other conventions need to be clarified.

- 
- In the book, use of Verilog and Verilog HDL refers to the "Verilog Hardware Description Language." Any reference to a Verilog-based simulator is specifically mentioned, using words such as Verilog simulator or trademarks such as Verilog-XL or VCS.
- 
- The word designer is used frequently in the book to emphasize the digital design perspective. However, it is a general term used to refer to a Verilog HDL user or a verification engineer.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

# Acknowledgments

The first edition of this book was written with the help of a great many people who contributed their energies to this project. Following were the primary contributors to my creation: John Sanguinetti, Stuart Sutherland, Clifford Cummings, Robert Emberley, Ashutosh Mauskar, Jack McKeown, Dr. Arun Soman, Dr. Michael Ciletti, Larry Ke, Sunil Sabat, Cheng-I Huang, Maqsoodul Mannan, Ashok Mehta, Dick Herlein, Rita Glover, Ming-Hwa Wang, Subramanian Ganesan, Sandeep Aggarwal, Albert Lau, Samir Sanghani, Kiran Buch, Anshuman Saha, Bill Fuchs, Babu Chilukuri, Ramana Kalapatapu, Karin Ellison and Rachel Borden. I would like to start by thanking all those people once again.

For this second edition, I give special thanks to the following people who helped me with the review process and provided valuable feedback:

Anders Nordstrom

Stefen Boyd

Clifford Cummings

Harry Foster

Yatin Trivedi

Rajeev Madhavan

John Sanguinetti

Dr. Arun Soman

Michael McNamara

Berend Ozceri

Shrenik Mehta

Mike Meredith

ASIC Consultant

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

# Part 1: Basic Verilog Topics

## 1 Overview of Digital Design with Verilog HDL

Evolution of CAD, emergence of HDLs, typical HDL-based design flow, why Verilog HDL?, trends in HDLs.

## 2 Hierarchical Modeling Concepts

Top-down and bottom-up design methodology, differences between modules and module instances, parts of a simulation, design block, stimulus block.

## 3 Basic Concepts

Lexical conventions, data types, system tasks, compiler directives.

## 4 Modules and Ports

Module definition, port declaration, connecting ports, hierarchical name referencing.

## 5 Gate-Level Modeling

Modeling using basic Verilog gate primitives, description of and/or and buf/not type gates, rise, fall and turn-off delays, min, max, and typical delays.

## 6 Dataflow Modeling

Continuous assignments, delay specification, expressions, operators, operands, operator types.

## 7 Behavioral Modeling

Structured procedures, initial and always, blocking and nonblocking statements, delay control, generate statement, event control, conditional statements, multiway branching, loops, sequential and parallel blocks.

## 8 Tasks and Functions

Differences between tasks and functions, declaration, invocation, automatic tasks and functions.

## 9 Useful Modeling Techniques

Procedural continuous assignments, overriding parameters, conditional compilation and execution, useful system tasks.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

# Chapter 1. Overview of Digital Design with Verilog HDL

- [Section 1.1. Evolution of Computer-Aided Digital Design](#)
- [Section 1.2. Emergence of HDLs](#)
- [Section 1.3. Typical Design Flow](#)
- [Section 1.4. Importance of HDLs](#)
- [Section 1.5. Popularity of Verilog HDL](#)
- [Section 1.6. Trends in HDLs](#)

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 1.1 Evolution of Computer-Aided Digital Design

Digital circuit design has evolved rapidly over the last 25 years. The earliest digital circuits were designed with vacuum tubes and transistors. Integrated circuits were then invented where logic gates were placed on a single chip. The first integrated circuit (IC) chips were SSI (Small Scale Integration) chips where the gate count was very small. As technologies became sophisticated, designers were able to place circuits with hundreds of gates on a chip. These chips were called MSI (Medium Scale Integration) chips. With the advent of LSI (Large Scale Integration), designers could put thousands of gates on a single chip. At this point, design processes started getting very complicated, and designers felt the need to automate these processes. Electronic Design Automation (EDA)<sup>[1]</sup> techniques began to evolve. Chip designers began to use circuit and logic simulation techniques to verify the functionality of building blocks of the order of about 100 transistors. The circuits were still tested on the breadboard, and the layout was done on paper or by hand on a graphic computer terminal.

[1] The earlier edition of the book used the term CAD tools. Technically, the term Computer-Aided Design (CAD) tools refers to back-end tools that perform functions related to place and route, and layout of the chip . The term Computer-Aided Engineering (CAE) tools refers to tools that are used for front-end processes such HDL simulation, logic synthesis, and timing analysis. Designers used the terms CAD and CAE interchangeably. Today, the term Electronic Design Automation is used for both CAD and CAE. For the sake of simplicity, in this book, we will refer to all design tools as EDA tools.

With the advent of VLSI (Very Large Scale Integration) technology, designers could design single chips with more than 100,000 transistors. Because of the complexity of these circuits, it was not possible to verify these circuits on a breadboard. Computer-aided techniques became critical for verification and design of VLSI digital circuits. Computer programs to do automatic placement and routing of circuit layouts also became popular. The designers were now building gate-level digital circuits manually on graphic terminals. They would build small building blocks and then derive higher-level blocks from them. This process would continue until they had built the top-level block. Logic simulators came into existence to verify the functionality of these circuits before they were fabricated on chip.

As designs got larger and more complex, logic simulation assumed an important role in the design process. Designers could iron out functional bugs in the architecture before the chip was designed further.

[\[ Team LiB \]](#)

◀ PREVIOUS | NEXT ▶

[\[ Team LiB \]](#)

◀ PREVIOUS | NEXT ▶

## 1.2 Emergence of HDLs

For a long time, programming languages such as FORTRAN, Pascal, and C were being used to describe computer programs that were sequential in nature. Similarly, in the digital design field, designers felt the need for a standard language to describe digital circuits. Thus, Hardware Description Languages (HDLs) came into existence. HDLs allowed the designers to model the concurrency of processes found in hardware elements. Hardware description languages such as Verilog HDL and VHDL became popular. Verilog HDL originated in 1983 at Gateway Design Automation. Later, VHDL was developed under contract from DARPA. Both Verilog and VHDL simulators to simulate large digital circuits quickly gained acceptance from designers.

Even though HDLs were popular for logic verification, designers had to manually translate the HDL-based design into a schematic circuit with interconnections between gates. The advent of logic synthesis in the late 1980s changed the design methodology radically. Digital circuits could be described at a register transfer level (RTL) by use of an HDL. Thus, the designer had to specify how the data flows between registers and how the design processes the data. The details of gates and their interconnections to implement the circuit were automatically extracted by logic synthesis tools from the RTL description.

Thus, logic synthesis pushed the HDLs into the forefront of digital design. Designers no longer had to manually place gates to build digital circuits. They could describe complex circuits at an abstract level in terms of functionality and data flow by designing those circuits in HDLs. Logic synthesis tools would implement the specified functionality in terms of gates and gate interconnections.

HDLs also began to be used for system-level design. HDLs were used for simulation of system boards, interconnect buses, FPGAs (Field Programmable Gate Arrays), and PALs (Programmable Array Logic). A common approach is to design each IC chip, using an HDL, and then verify system functionality via simulation.

Today, Verilog HDL is an accepted IEEE standard. In 1995, the original standard IEEE 1364-1995 was approved. IEEE 1364-2001 is the latest Verilog HDL standard that made significant improvements to the original standard.

[ Team LiB ]

[◀ PREVIOUS](#) [NEXT ▶](#)

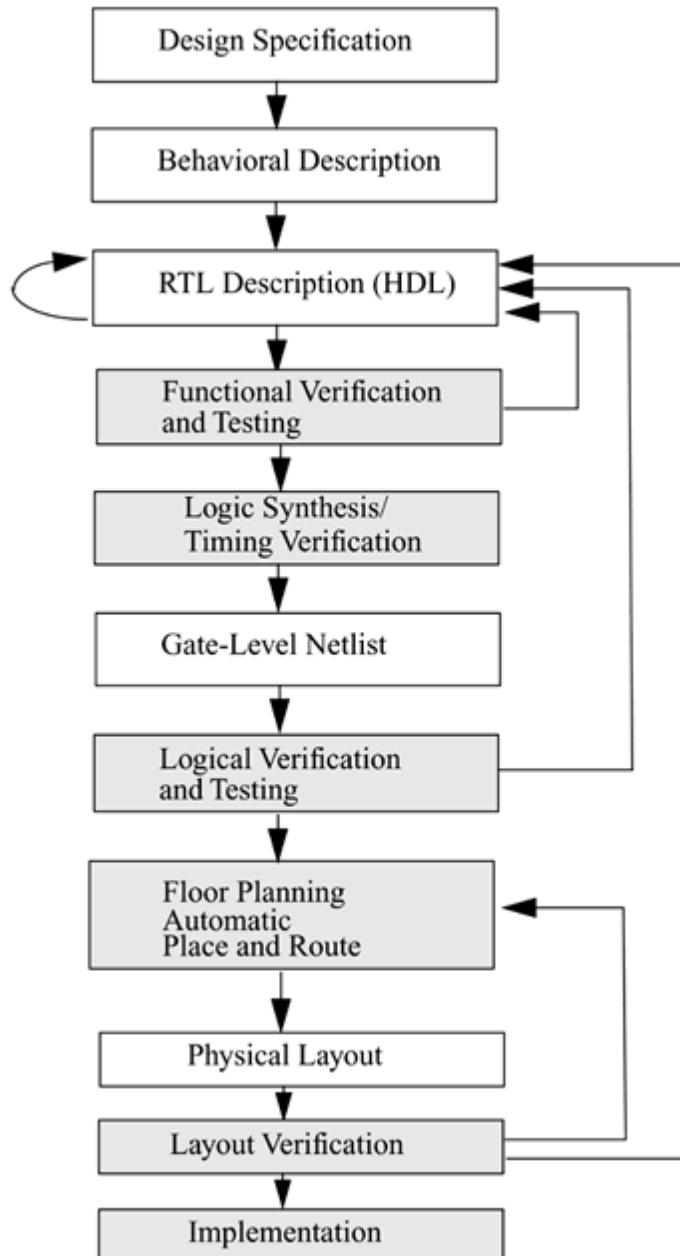
[ Team LiB ]

[◀ PREVIOUS](#) [NEXT ▶](#)

## 1.3 Typical Design Flow

A typical design flow for designing VLSI IC circuits is shown in [Figure 1-1](#). Unshaded blocks show the level of design representation; shaded blocks show processes in the design flow.

**Figure 1-1. Typical Design Flow**



The design flow shown in [Figure 1-1](#) is typically used by designers who use HDLs. In any design, specifications are written first. Specifications describe abstractly the functionality, interface, and overall architecture of the digital circuit to be designed. At this point, the architects do not need to think about how they will implement this circuit. A behavioral description is then created to analyze the design in terms of functionality, performance, compliance to standards, and other high-level issues. Behavioral descriptions are often written with HDLs.[\[2\]](#)

[2] New EDA tools have emerged to simulate behavioral descriptions of circuits. These tools combine the powerful concepts from HDLs and object oriented languages such as C++. These tools can be used instead of writing behavioral descriptions in Verilog HDL.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 1.4 Importance of HDLs

HDLs have many advantages compared to traditional schematic-based design.

- 
- Designs can be described at a very abstract level by use of HDLs. Designers can write their RTL description without choosing a specific fabrication technology. Logic synthesis tools can automatically convert the design to any fabrication technology. If a new technology emerges, designers do not need to redesign their circuit. They simply input the RTL description to the logic synthesis tool and create a new gate-level netlist, using the new fabrication technology. The logic synthesis tool will optimize the circuit in area and timing for the new technology.
- 
- By describing designs in HDLs, functional verification of the design can be done early in the design cycle. Since designers work at the RTL level, they can optimize and modify the RTL description until it meets the desired functionality. Most design bugs are eliminated at this point. This cuts down design cycle time significantly because the probability of hitting a functional bug at a later time in the gate-level netlist or physical layout is minimized.
- 
- Designing with HDLs is analogous to computer programming. A textual description with comments is an easier way to develop and debug circuits. This also provides a concise representation of the design, compared to gate-level schematics. Gate-level schematics are almost incomprehensible for very complex designs.

HDL-based design is here to stay.[\[3\]](#) With rapidly increasing complexities of digital circuits and increasingly sophisticated EDA tools, HDLs are now the dominant method for large digital designs. No digital circuit designer can afford to ignore HDL-based design.

[3] New tools and languages focused on verification have emerged in the past few years. These languages are better suited for functional verification. However, for logic design, HDLs continue as the preferred choice.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 1.5 Popularity of Verilog HDL

Verilog HDL has evolved as a standard hardware description language. Verilog HDL offers many useful features

- 
- Verilog HDL is a general-purpose hardware description language that is easy to learn and easy to use. It is similar in syntax to the C programming language. Designers with C programming experience will find it easy to learn Verilog HDL.
- 
- Verilog HDL allows different levels of abstraction to be mixed in the same model. Thus, a designer can define a hardware model in terms of switches, gates, RTL, or behavioral code. Also, a designer needs to learn only one language for stimulus and hierarchical design.
- 
- Most popular logic synthesis tools support Verilog HDL. This makes it the language of choice for designers.
- 
- All fabrication vendors provide Verilog HDL libraries for postlogic synthesis simulation. Thus, designing a chip in Verilog HDL allows the widest choice of vendors.
- 
- The Programming Language Interface (PLI) is a powerful feature that allows the user to write custom C code to interact with the internal data structures of Verilog. Designers can customize a Verilog HDL simulator to their needs with the PLI.

[ Team LiB ]

[◀ PREVIOUS](#) [NEXT ▶](#)



[ Team LiB ]

[◀ PREVIOUS](#) [NEXT ▶](#)

## 1.6 Trends in HDLs

The speed and complexity of digital circuits have increased rapidly. Designers have responded by designing at higher levels of abstraction. Designers have to think only in terms of functionality. EDA tools take care of the implementation details. With designer assistance, EDA tools have become sophisticated enough to achieve a close-to-optimum implementation.

The most popular trend currently is to design in HDL at an RTL level, because logic synthesis tools can create gate-level netlists from RTL level design. Behavioral synthesis allowed engineers to design directly in terms of algorithms and the behavior of the circuit, and then use EDA tools to do the translation and optimization in each phase of the design. However, behavioral synthesis did not gain widespread acceptance. Today, RTL design continues to be very popular. Verilog HDL is also being constantly enhanced to meet the needs of new verification methodologies.

Formal verification and assertion checking techniques have emerged. Formal verification applies formal mathematical techniques to verify the correctness of Verilog HDL descriptions and to establish equivalency between RTL and gate-level netlists. However, the need to describe a design in Verilog HDL will not go away. Assertion checkers allow checking to be embedded in the RTL code. This is a convenient way to do checking in the most important parts of a design.

New verification languages have also gained rapid acceptance. These languages combine the parallelism and hardware constructs from HDLs with the object oriented nature of C++. These languages also provide support for automatic stimulus creation, checking, and coverage. However, these languages do not replace Verilog HDL. They simply boost the productivity of the verification process. Verilog HDL is still needed to describe the design.

For very high-speed and timing-critical circuits like microprocessors, the gate-level netlist provided by logic synthesis tools is not optimal. In such cases, designers often mix gate-level description directly into the RTL description to achieve optimum results. This practice is opposite to the high-level design paradigm, yet it is frequently used for high-speed designs because designers need to squeeze the last bit of timing out of circuits, and EDA tools sometimes prove to be insufficient to achieve the desired results.

Another technique that is used for system-level design is a mixed bottom-up methodology where the designers use either existing Verilog HDL modules, basic building blocks, or vendor-supplied core blocks to quickly bring up their system simulation. This is done to reduce development costs and compress design schedules. For example, consider a system that has a CPU, graphics chip, I/O chip, and a system bus. The CPU designers would build the next-generation CPU themselves at an RTL level, but they would use behavioral models for the graphics chip and the I/O chip and would buy a vendor-supplied model for the system bus. Thus, the system-level simulation for the CPU could be up and running very quickly and long before the RTL descriptions for the graphics chip and the I/O chip are completed.

[ Team LiB ]



## Chapter 2. Hierarchical Modeling Concepts

Before we discuss the details of the Verilog language, we must first understand basic hierarchical modeling concepts in digital design. The designer must use a "good" design methodology to do efficient Verilog HDL-based design. In this chapter, we discuss typical design methodologies and illustrate how these concepts are translated to Verilog. A digital simulation is made up of various components. We talk about the components and their interconnections.

### Learning Objectives

- 
- Understand top-down and bottom-up design methodologies for digital design.
- 
- Explain differences between modules and module instances in Verilog.
- 
- Describe four levels of abstraction?ehavioral, data flow, gate level, and switch level?o represent the same module.
- 
- Describe components required for the simulation of a digital design. Define a stimulus block and a design block. Explain two methods of applying stimulus.

[ Team LiB ]



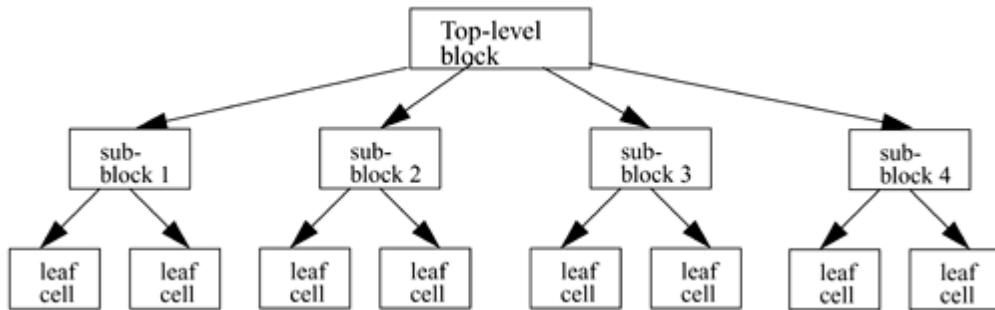
[ Team LiB ]



## 2.1 Design Methodologies

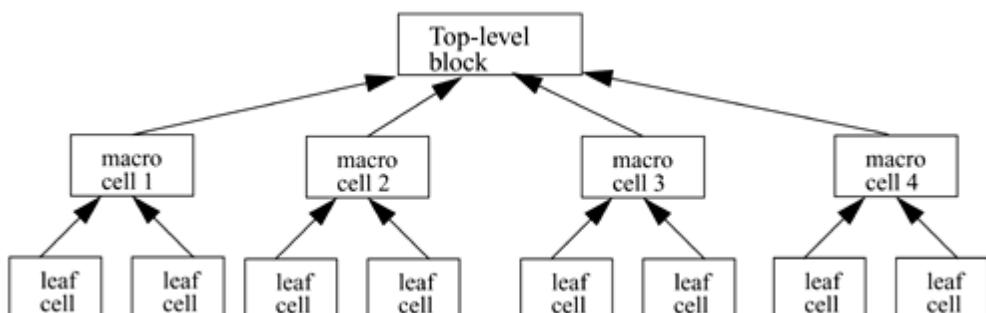
There are two basic types of digital design methodologies: a top-down design methodology and a bottom-up design methodology. In a top-down design methodology, we define the top-level block and identify the sub-blocks necessary to build the top-level block. We further subdivide the sub-blocks until we come to leaf cells, which are the cells that cannot further be divided. [Figure 2-1](#) shows the top-down design process.

**Figure 2-1. Top-down Design Methodology**



In a bottom-up design methodology, we first identify the building blocks that are available to us. We build bigger cells, using these building blocks. These cells are then used for higher-level blocks until we build the top-level block in the design. [Figure 2-2](#) shows the bottom-up design process.

**Figure 2-2. Bottom-up Design Methodology**



Typically, a combination of top-down and bottom-up flows is used. Design architects define the specifications of the top-level block. Logic designers decide how the design should be structured by breaking up the functionality into blocks and sub-blocks. At the same time, circuit designers are designing optimized circuits for leaf-level cells. They build higher-level cells by using these leaf cells. The flow meets at an intermediate point where the switch-level circuit designers have created a library of leaf cells by using switches, and the logic level designers have designed from top-down until all modules are defined in terms of leaf cells.

To illustrate these hierarchical modeling concepts, let us consider the design of a negative edge-triggered 4-bit ripple carry counter described in [Section 2.2](#), 4-bit Ripple Carry Counter.

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

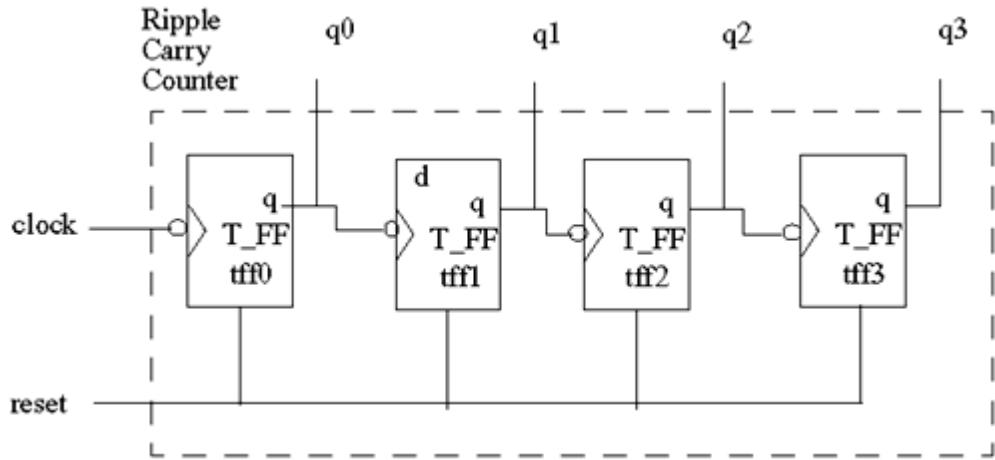
[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## 2.2 4-bit Ripple Carry Counter

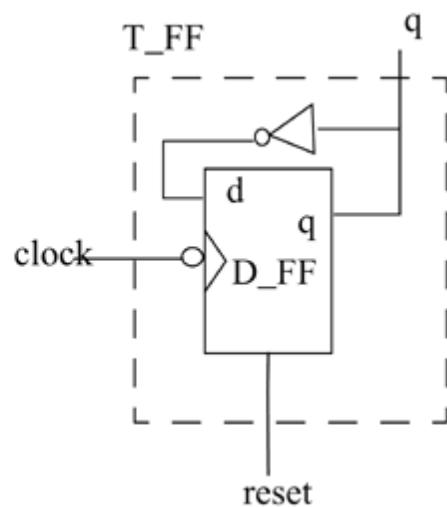
The ripple carry counter shown in [Figure 2-3](#) is made up of negative edge-triggered toggle flipflops (T\_FF). Each of the T\_FFs can be made up from negative edge-triggered D-flipflops (D\_FF) and inverters (assuming q\_bar output is not available on the D\_FF), as shown in [Figure 2-4](#).

**Figure 2-3. Ripple Carry Counter**



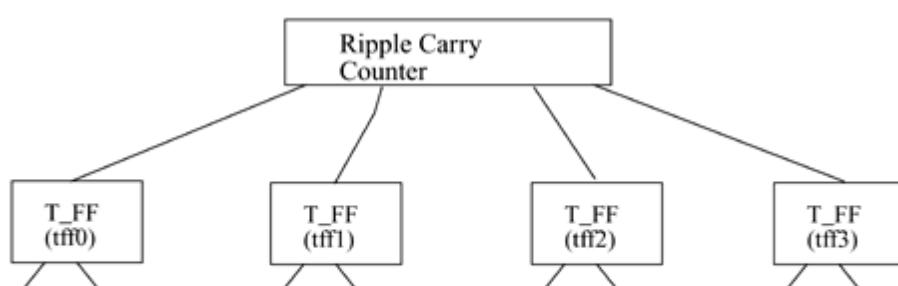
**Figure 2-4. T-flipflop**

reset	$q_n$	$q_{n+1}$
1	1	0
1	0	0
0	0	1
0	1	0
0	0	0



Thus, the ripple carry counter is built in a hierarchical fashion by using building blocks. The diagram for the design hierarchy is shown in [Figure 2-5](#).

**Figure 2-5. Design Hierarchy**



[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## 2.3 Modules

We now relate these hierarchical modeling concepts to Verilog. Verilog provides the concept of a module. A module is the basic building block in Verilog. A module can be an element or a collection of lower-level design blocks. Typically, elements are grouped into modules to provide common functionality that is used at many places in the design. A module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs), but hides the internal implementation. This allows the designer to modify module internals without affecting the rest of the design.

In [Figure 2-5](#), ripple carry counter, T\_FF, D\_FF are examples of modules. In Verilog, a module is declared by the keyword module. A corresponding keyword endmodule must appear at the end of the module definition. Each module must have a module\_name, which is the identifier for the module, and a module\_terminal\_list, which describes the input and output terminals of the module.

```
module <module_name> (<module_terminal_list>);  
...  
<module internals>  
...  
...  
endmodule
```

Specifically, the T-flipflop could be defined as a module as follows:

```
module T_FF (q, clock, reset);  
.  
.  
<functionality of T-flipflop>  
.  
.  
endmodule
```

Verilog is both a behavioral and a structural language. Internals of each module can be defined at four levels of abstraction, depending on the needs of the design. The module behaves identically with the external environment irrespective of the level of abstraction at which the module is described. The internals of the module are hidden from the environment. Thus, the level of abstraction to describe a module can be changed without any change in the environment. These levels will be studied in detail in separate chapters later in the book. The levels are defined below.

- 
- Behavioral or algorithmic level
- This is the highest level of abstraction provided by Verilog HDL. A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. Designing at this level is very similar to C programming.
- 
- Dataflow level
- At this level, the module is designed by specifying the data flow. The designer is aware of how

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 2.4 Instances

A module provides a template from which you can create actual objects. When a module is invoked, Verilog creates a unique object from the template. Each object has its own name, variables, parameters, and I/O interface. The process of creating objects from a module template is called instantiation, and the objects are called instances. In [Example 2-1](#), the top-level block creates four instances from the T-flipflop (T\_FF) template. Each T\_FF instantiates a D\_FF and an inverter gate. Each instance must be given a unique name. Note that // is used to denote single-line comments.

### Example 2-1 Module Instantiation

```
// Define the top-level module called ripple carry
// counter. It instantiates 4 T-flipflops. Interconnections are
// shown in Section 2.2, 4-bit Ripple Carry Counter.
module ripple_carry_counter(q, clk, reset);

output [3:0] q; //I/O signals and vector declarations
               //will be explained later.
input clk, reset; //I/O signals will be explained later.

//Four instances of the module T_FF are created. Each has a unique
//name.Each instance is passed a set of signals. Notice, that
//each instance is a copy of the module T_FF.
T_FF tff0(q[0],clk, reset);
T_FF tff1(q[1],q[0], reset);
T_FF tff2(q[2],q[1], reset);
T_FF tff3(q[3],q[2], reset);

endmodule

// Define the module T_FF. It instantiates a D-flipflop. We assumed
// that module D-flipflop is defined elsewhere in the design. Refer
// to Figure 2-4 for interconnections.
module T_FF(q, clk, reset);

//Declarations to be explained later
output q;
input clk, reset;
wire d;

D_FF dff0(q, d, clk, reset); // Instantiate D_FF. Call it dff0.
not n1(d, q); // not gate is a Verilog primitive. Explained later.

endmodule
```

In Verilog, it is illegal to nest modules. One module definition cannot contain another module definition within the module and endmodule statements. Instead, a module definition can incorporate copies of other modules by instantiating them. It is important not to confuse module definitions and instances of a module. Module definitions simply specify how the module will work, its internals, and its interface. Modules must be instantiated for use in the design.

[Example 2-2](#) shows an illegal module nesting where the module T\_FF is defined inside the module definition of the ripple carry counter.

### Example 2-2 Illegal Module Nesting

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

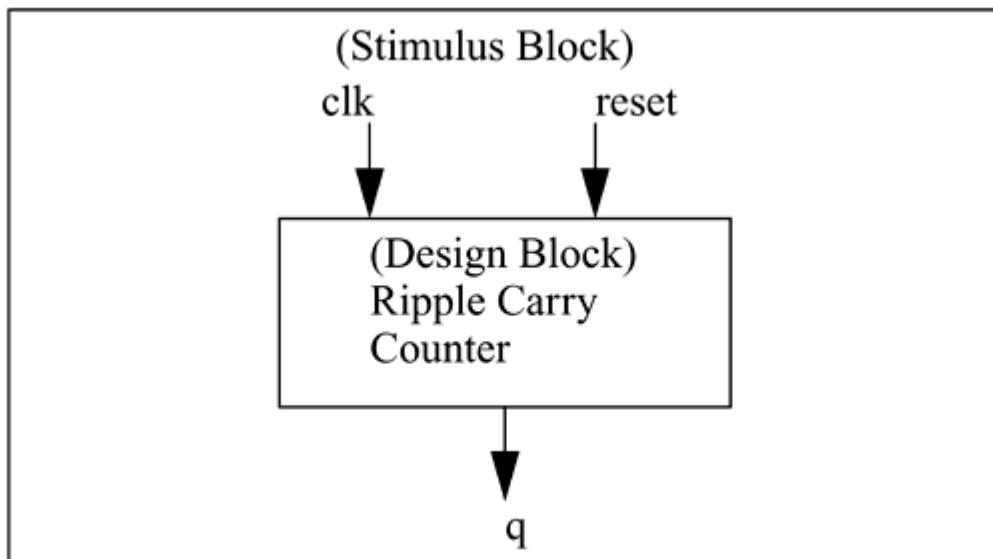
◀ PREVIOUS | NEXT ▶

## 2.5 Components of a Simulation

Once a design block is completed, it must be tested. The functionality of the design block can be tested by applying stimulus and checking results. We call such a block the stimulus block. It is good practice to keep the stimulus and design blocks separate. The stimulus block can be written in Verilog. A separate language is not required to describe stimulus. The stimulus block is also commonly called a test bench. Different test benches can be used to thoroughly test the design block.

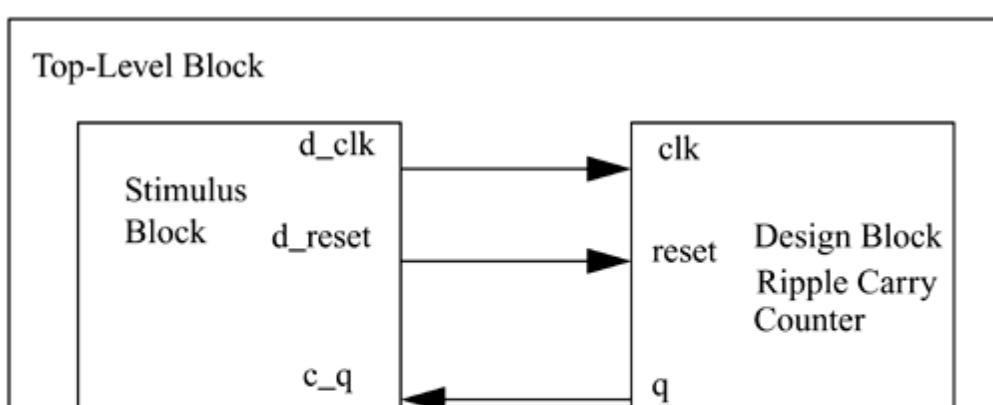
Two styles of stimulus application are possible. In the first style, the stimulus block instantiates the design block and directly drives the signals in the design block. In [Figure 2-6](#), the stimulus block becomes the top-level block. It manipulates signals clk and reset, and it checks and displays output signal q.

**Figure 2-6. Stimulus Block Instantiates Design Block**



The second style of applying stimulus is to instantiate both the stimulus and design blocks in a top-level dummy module. The stimulus block interacts with the design block only through the interface. This style of applying stimulus is shown in [Figure 2-7](#). The stimulus module drives the signals d\_clk and d\_reset, which are connected to the signals clk and reset in the design block. It also checks and displays signal c\_q, which is connected to the signal q in the design block. The function of top-level block is simply to instantiate the design and stimulus blocks.

**Figure 2-7. Stimulus and Design Blocks Instantiated in a Dummy Top-Level Module**



[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## 2.6 Example

To illustrate the concepts discussed in the previous sections, let us build the complete simulation of a ripple carry counter. We will define the design block and the stimulus block. We will apply stimulus to the design block and monitor the outputs. As we develop the Verilog models, you do not need to understand the exact syntax of each construct at this stage. At this point, you should simply try to understand the design process. We discuss the syntax in much greater detail in the later chapters.

### 2.6.1 Design Block

We use a top-down design methodology. First, we write the Verilog description of the top-level design block ([Example 2-3](#)), which is the ripple carry counter (see [Section 2.2](#), 4-bit Ripple Carry Counter).

#### Example 2-3 Ripple Carry Counter Top Block

```
module ripple_carry_counter(q, clk, reset);
    output [3:0] q;
    input clk, reset;

    //4 instances of the module T_FF are created.
    T_FF tff0(q[0],clk, reset);
    T_FF tff1(q[1],q[0], reset);
    T_FF tff2(q[2],q[1], reset);
    T_FF tff3(q[3],q[2], reset);

endmodule
```

In the above module, four instances of the module T\_FF (T-flipflop) are used. Therefore, we must now define ([Example 2-4](#)) the internals of the module T\_FF, which was shown in [Figure 2-4](#).

#### Example 2-4 Flipflop T\_FF

```
module T_FF(q, clk, reset);
    output q;
    input clk, reset;
    wire d;
    D_FF dff0(q, d, clk, reset);
    not nl(d, q); // not is a Verilog-provided primitive. case sensitive
endmodule
```

Since T\_FF instantiates D\_FF, we must now define ([Example 2-5](#)) the internals of module D\_FF. We assume asynchronous reset for the D\_FFF.

#### Example 2-5 Flipflop D\_F

```
// module D_FF with synchronous reset
module D_FF(q, d, clk, reset);

    output q;
    input d, clk, reset;
    reg q;
```

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 2.7 Summary

In this chapter we discussed the following concepts.

- 
- Two kinds of design methodologies are used for digital design: top-down and bottom-up. A combination of these two methodologies is used in today's digital designs. As designs become very complex, it is important to follow these structured approaches to manage the design process.
- 
- Modules are the basic building blocks in Verilog. Modules are used in a design by instantiation. An instance of a module has a unique identity and is different from other instances of the same module. Each instance has an independent copy of the internals of the module. It is important to understand the difference between modules and instances.
- 
- There are two distinct components in a simulation: a design block and a stimulus block. A stimulus block is used to test the design block. The stimulus block is usually the top-level block. There are two different styles of applying stimulus to a design block.
- 
- The example of the ripple carry counter explains the step-by-step process of building all the blocks required in a simulation.

This chapter is intended to give an understanding of the design process and how Verilog fits into the design process. The details of Verilog syntax are not important at this stage and will be dealt with in later chapters.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 2.8 Exercises

1:

An interconnect switch (IS) contains the following components, a shared memory (MEM), a system controller (SC) and a data crossbar (Xbar).

a.

- a. Define the modules MEM, SC, and Xbar, using the module/endmodule keywords. You do not need to define the internals. Assume that the modules have no terminal lists.

b.

- b. Define the module IS, using the module/endmodule keywords. Instantiate the modules MEM, SC, Xbar and call the instances mem1, sc1, and xbar1, respectively. You do not need to define the internals. Assume that the module IS has no terminals.

c.

- c. Define a stimulus block (Top), using the module/endmodule keywords. Instantiate the design block IS and call the instance is1. This is the final step in building the simulation environment.

2:

A 4-bit ripple carry adder (Ripple\_Add) contains four 1-bit full adders (FA).

a.

- a. Define the module FA. Do not define the internals or the terminal list.

b.

- b. Define the module Ripple\_Add. Do not define the internals or the terminal list. Instantiate four full adders of the type FA in the module Ripple\_Add and call them fa0, fa1, fa2, and fa3.



[ Team LiB ]

[PREVIOUS](#)  [NEXT](#)

## Chapter 3. Basic Concepts

In this chapter, we discuss the basic constructs and conventions in Verilog. These conventions and constructs are used throughout the later chapters. These conventions provide the necessary framework for Verilog HDL. Data types in Verilog model actual data storage and switch elements in hardware very closely. This chapter may seem dry, but understanding these concepts is a necessary foundation for the successive chapters.

### Learning Objectives

- 
- Understand lexical conventions for operators, comments, whitespace, numbers, strings, and identifiers.
- 
- Define the logic value set and data types such as nets, registers, vectors, numbers, simulation time, arrays, parameters, memories, and strings.
- 
- Identify useful system tasks for displaying and monitoring information, and for stopping and finishing the simulation.
- 
- Learn basic compiler directives to define macros and include files.

[ Team LiB ]

[PREVIOUS](#)  [NEXT](#)

[ Team LiB ]

[PREVIOUS](#)  [NEXT](#)



## 3.1 Lexical Conventions

The basic lexical conventions used by Verilog HDL are similar to those in the C programming language. Verilog contains a stream of tokens. Tokens can be comments, delimiters, numbers, strings, identifiers, and keywords. Verilog HDL is a case-sensitive language. All keywords are in lowercase.

### 3.1.1 Whitespace

Blank spaces (\b), tabs (\t) and newlines (\n) comprise the whitespace. Whitespace is ignored by Verilog except when it separates tokens. Whitespace is not ignored in strings.

### 3.1.2 Comments

Comments can be inserted in the code for readability and documentation. There are two ways to write comments. A one-line comment starts with "//". Verilog skips from that point to the end of line. A multiple-line comment starts with "/\*" and ends with "\*/". Multiple-line comments cannot be nested. However, one-line comments can be embedded in multiple-line comments.

```
a = b && c; // This is a one-line comment  
/* This is a multiple line  
comment */  
/* This is /* an illegal */ comment */  
/* This is //a legal comment */
```

### 3.1.3 Operators

Operators are of three types: unary, binary, and ternary. Unary operators precede the operand. Binary operators appear between two operands. Ternary operators have two separate operators that separate three operands.

```
a = ~ b; // ~ is a unary operator. b is the operand  
a = b && c; // && is a binary operator. b and c are operands  
a = b ? c : d; // ?: is a ternary operator. b, c and d are operands
```

### 3.1.4 Number Specification

There are two types of number specification in Verilog: sized and unsized.

#### Sized numbers

Sized numbers are represented as <size>'<base format><number>.

<size> is written only in decimal and specifies the number of bits in the number. Legal base formats are decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and octal ('o or 'O). The number is specified

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## 3.2 Data Types

This section discusses the data types used in Verilog.

### 3.2.1 Value Set

Verilog supports four values and eight strengths to model the functionality of real hardware. The four value levels are listed in [Table 3-1](#).

Table 3-1. Value Levels

Value Level	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown logic value
z	High impedance, floating state

In addition to logic values, strength levels are often used to resolve conflicts between drivers of different strengths in digital circuits. Value levels 0 and 1 can have the strength levels listed in [Table 3-2](#).

Table 3-2. Strength Levels

Strength Level	Type	Degree
supply	Driving	strongest
strong	Driving	
pull	Driving	
large	Storage	
weak	Driving	
medium	Storage	
small	Storage	

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## 3.3 System Tasks and Compiler Directives

In this section, we introduce two special concepts used in Verilog: system tasks and compiler directives.

### 3.3.1 System Tasks

Verilog provides standard system tasks for certain routine operations. All system tasks appear in the form \$<keyword>. Operations such as displaying on the screen, monitoring values of nets, stopping, and finishing are done by system tasks. We will discuss only the most useful system tasks. Other tasks are listed in Verilog manuals provided by your simulator vendor or in the IEEE Standard Verilog Hardware Description Language specification.

#### Displaying information

\$display is the main system task for displaying values of variables or strings or expressions. This is one of the most useful tasks in Verilog.

Usage: \$display(p1, p2, p3,....., pn);

p1, p2, p3,..., pn can be quoted strings or variables or expressions. The format of \$display is very similar to printf in C. A \$display inserts a newline at the end of the string by default. A \$display without any arguments produces a newline.

Strings can be formatted using the specifications listed in [Table 3-4](#). For more detailed specifications, see IEEE Standard Verilog Hardware Description Language specification.

Table 3-4. String Format Specifications

Format	Display
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex
%c or %C	Display ASCII character
%m or %M	Display hierarchical name (no argument required)

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 3.4 Summary

We discussed the basic concepts of Verilog in this chapter. These concepts lay the foundation for the material discussed in the further chapters.

- 
- Verilog is similar in syntax to the C programming language . Hardware designers with previous C programming experience will find Verilog easy to learn.
- 
- Lexical conventions for operators, comments, whitespace, numbers, strings, and identifiers were discussed.
- 
- Various data types are available in Verilog. There are four logic values, each with different strength levels. Available data types include nets, registers, vectors, numbers, simulation time, arrays, memories, parameters, and strings. Data types represent actual hardware elements very closely.
- 
- Verilog provides useful system tasks to do functions like displaying, monitoring, suspending, and finishing a simulation.
- 
- Compiler directive `define is used to define text macros, and `include is used to include other Verilog files.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

 PREVIOUS | NEXT |

## 3.5 Exercises

1:

Practice writing the following numbers:

a.

a. Decimal number 123 as a sized 8-bit number  
in binary. Use \_ for readability.

b.

b. A 16-bit hexadecimal unknown number with  
all x's.

c.

c. A 4-bit negative 2 in decimal . Write the 2's  
complement form for this number.

d.

d. An unsized hex number 1234.

2:

Are the following legal strings? If not, write the  
correct strings.

a.

a. "This is a string displaying the % sign"  
b.

b. "out = in1 + in2"

c.

c. "Please ring a bell \007"  
d.

d. "This is a backslash \ character\n"

3:

Are these legal identifiers?

a.

a. system1  
b.

b. 1reg

c.

c. \$latch  
d.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## Chapter 4. Modules and Ports

In the previous chapters, we acquired an understanding of the fundamental hierarchical modeling concepts, basic conventions, and Verilog constructs. In this chapter, we take a closer look at modules and ports from the Verilog language point of view.

### Learning Objectives

- 
- Identify the components of a Verilog module definition, such as module names, port lists, parameters, variable declarations, dataflow statements, behavioral statements, instantiation of other modules, and tasks or functions.
- 
- Understand how to define the port list for a module and declare it in Verilog.
- 
- Describe the port connection rules in a module instantiation.
- 
- Understand how to connect ports to external signals, by ordered list, and by name.
- 
- Explain hierarchical name referencing of Verilog identifiers.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

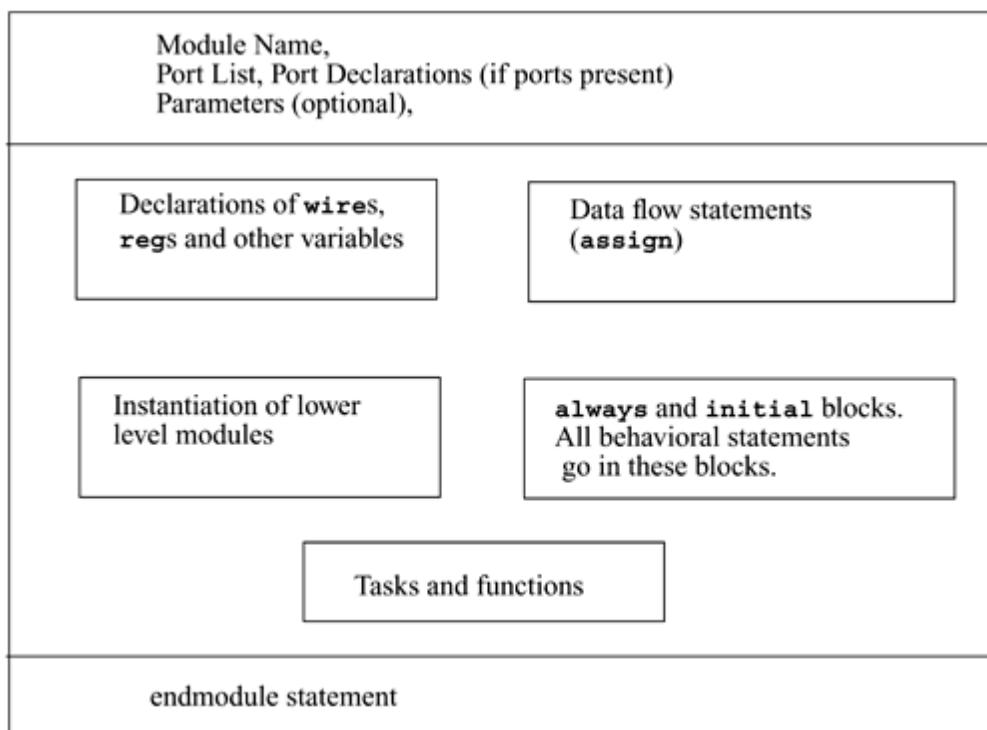
[ PREVIOUS ] [ NEXT ]

## 4.1 Modules

We discussed how a module is a basic building block in [Chapter 2](#), Hierarchical Modeling Concepts. We ignored the internals of modules and concentrated on how modules are defined and instantiated. In this section, we analyze the internals of the module in greater detail.

A module in Verilog consists of distinct parts, as shown in [Figure 4-1](#).

**Figure 4-1. Components of a Verilog Module**



A module definition always begins with the keyword `module`. The module name, port list, port declarations, and optional parameters must come first in a module definition. Port list and port declarations are present only if the module has any ports to interact with the external environment. The five components within a module are: variable declarations, dataflow statements, instantiation of lower modules, behavioral blocks, and tasks or functions. These components can be in any order and at any place in the module definition. The `endmodule` statement must always come last in a module definition. All components except `module`, module name, and `endmodule` are optional and can be mixed and matched as per design needs. Verilog allows multiple modules to be defined in a single file. The modules can be defined in any order in the file.

To understand the components of a module shown above, let us consider a simple example of an SR latch, as shown in [Figure 4-2](#).

**Figure 4-2. SR Latch**



[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

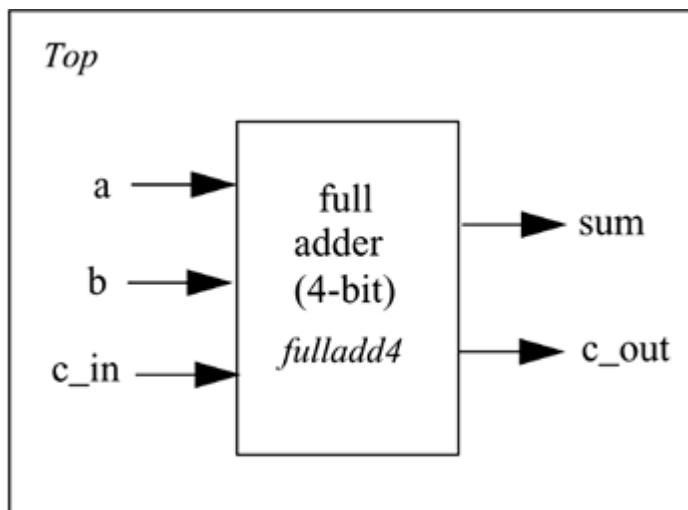
## 4.2 Ports

Ports provide the interface by which a module can communicate with its environment. For example, the input/output pins of an IC chip are its ports. The environment can interact with the module only through its ports. The internals of the module are not visible to the environment. This provides a very powerful flexibility to the designer. The internals of the module can be changed without affecting the environment as long as the interface is not modified. Ports are also referred to as terminals.

### 4.2.1 List of Ports

A module definition contains an optional list of ports. If the module does not exchange any signals with the environment, there are no ports in the list. Consider a 4-bit full adder that is instantiated inside a top-level module Top. The diagram for the input/output ports is shown in [Figure 4-3](#).

**Figure 4-3. I/O Ports for Top and Full Adder**



Notice that in the above figure, the module Top is a top-level module. The module fulladd4 is instantiated below Top. The module fulladd4 takes input on ports a, b, and c\_in and produces an output on ports sum and c\_out. Thus, module fulladd4 performs an addition for its environment. The module Top is a top-level module in the simulation and does not need to pass signals to or receive signals from the environment. Thus, it does not have a list of ports. The module names and port lists for both module declarations in Verilog are as shown in [Example 4-2](#).

### Example 4-2 List of Ports

```

module fulladd4(sum, c_out, a, b, c_in); //Module with a list of ports
module Top; // No list of ports, top-level module in simulation
  
```

### 4.2.2 Port Declaration

All ports in the list of ports must be declared in the module. Ports can be declared as follows:

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

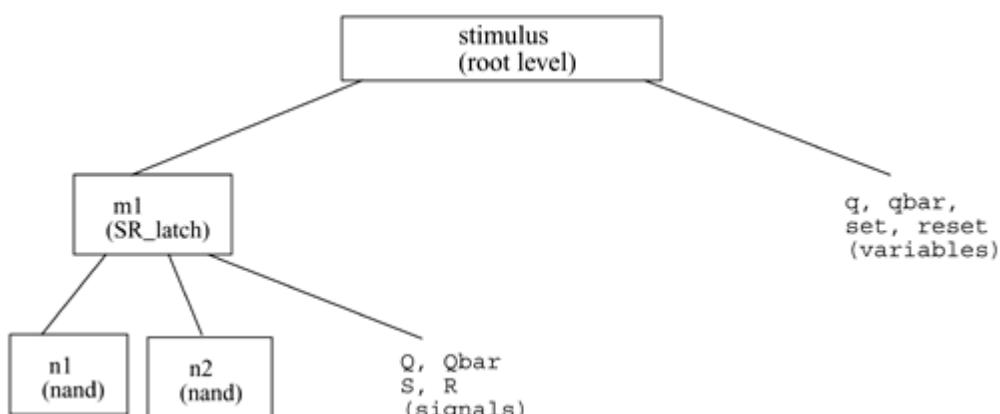
◀ PREVIOUS | NEXT ▶

## 4.3 Hierarchical Names

We described earlier how Verilog supports a hierarchical design methodology. Every module instance, signal, or variable is defined with an identifier. A particular identifier has a unique place in the design hierarchy. Hierarchical name referencing allows us to denote every identifier in the design hierarchy with a unique name. A hierarchical name is a list of identifiers separated by dots (".") for each level of hierarchy. Thus, any identifier can be addressed from any place in the design by simply specifying the complete hierarchical name of that identifier.

The top-level module is called the root module because it is not instantiated anywhere. It is the starting point. To assign a unique name to an identifier, start from the top-level module and trace the path along the design hierarchy to the desired identifier. To clarify this process, let us consider the simulation of SR latch in [Example 4-1](#). The design hierarchy is shown in [Figure 4-5](#).

**Figure 4-5. Design Hierarchy for SR Latch Simulation**



For this simulation, stimulus is the top-level module. Since the top-level module is not instantiated anywhere, it is called the root module. The identifiers defined in this module are q, qbar, set, and reset. The root module instantiates m1, which is a module of type SR\_latch. The module m1 instantiates nand gates n1 and n2. Q, Qbar, S, and R are port signals in instance m1. Hierarchical name referencing assigns a unique name to each identifier. To assign hierarchical names, use the module name for root module and instance names for all module instances below the root module. [Example 4-8](#) shows hierarchical names for all identifiers in the above simulation. Notice that there is a dot (.) for each level of hierarchy from the root module to the desired identifier.

### Example 4-8 Hierarchical Names

stimulus	stimulus.q
stimulus.qbar	stimulus.qbar
stimulus.reset	stimulus.reset
stimulus.m1.Q	stimulus.m1.Q
stimulus.m1.S	stimulus.m1.S
stimulus.n1	stimulus.n1

Each identifier in the design is uniquely specified by its hierarchical path name. To display the level of hierarchy, use the special character %m in the \$display task. See [Table 3-4](#), String Format Specifications, for details.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 4.4 Summary

In this chapter, we discussed the following aspects of Verilog:

- 
- Module definitions contain various components. Keywords module and endmodule are mandatory. Other components?ort list, port declarations, variable and signal declarations, dataflow statements, behavioral blocks, lower-level module instantiations, and tasks or functions?re optional and can be added as needed.
- 
- Ports provide the module with a means to communicate with other modules or its environment. A module can have a port list. Ports in the port list must be declared as input, output, or inout. When instantiating a module, port connection rules are enforced by the Verilog simulator. An ANSI C style embeds the port declarations in the module definition statement.
- 
- Ports can be connected by name or by ordered list.
- 
- Each identifier in the design has a unique hierarchical name. Hierarchical names allow us to address any identifier in the design from any other level of hierarchy in the design.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 4.5 Exercises

1:

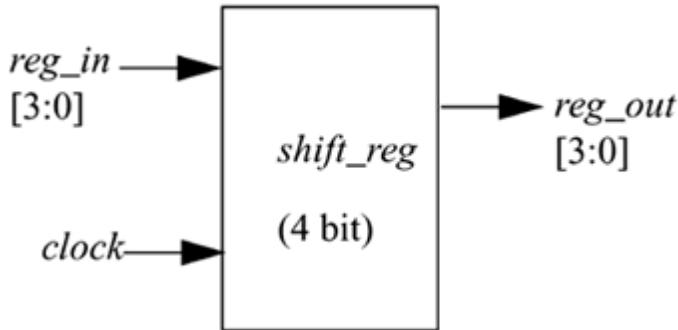
What are the basic components of a module? Which components are mandatory?

2:

Does a module that does not interact with its environment have any I/O ports? Does it have a port list in the module definition?

3:

A 4-bit parallel shift register has I/O pins as shown in the figure below. Write the module definition for this module *shift\_reg*. Include the list of ports and port declarations. You do not need to show the internals.



4:

Declare a top-level module stimulus. Define REG\_IN (4 bit) and CLK (1 bit) as reg register variables and REG\_OUT (4 bit) as wire. Instantiate the module *shift\_reg* and call it sr1. Connect the ports by ordered list.

5:

Connect the ports in Step 4 by name.

6:

Write the hierarchical names for variables REG\_IN, CLK, and REG\_OUT.

7:

Write the hierarchical name for the instance sr1. Write the hierarchical names for its ports clock and reg\_in.

[ Team LiB ]

◀ PREVIOUS    NEXT ▶

[ Team LiB ]

◀ PREVIOUS    NEXT ▶

# Chapter 5. Gate-Level Modeling

In the earlier chapters, we laid the foundations of Verilog design by discussing design methodologies, basic conventions and constructs, modules and port interfaces. In this chapter, we get into modeling actual hardware circuits in Verilog.

We discussed the four levels of abstraction used to describe hardware. In this chapter, we discuss a design at a low level of abstraction—gate level. Most digital design is now done at gate level or higher levels of abstraction. At gate level, the circuit is described in terms of gates (e.g., and, nand). Hardware design at this level is intuitive for a user with a basic knowledge of digital logic design because it is possible to see a one-to-one correspondence between the logic circuit diagram and the Verilog description. Hence, in this book, we chose to start with gate-level modeling and move to higher levels of abstraction in the succeeding chapters.

Actually, the lowest level of abstraction is switch- (transistor-) level modeling. However, with designs getting very complex, very few hardware designers work at switch level. Therefore, we will defer switch-level modeling to [Chapter 11](#), Switch-Level Modeling, in [Part 2](#) of this book.

## Learning Objectives

- 
- Identify logic gate primitives provided in Verilog.
- 
- Understand instantiation of gates, gate symbols, and truth tables for and/or and buf/not type gates.
- 
- Understand how to construct a Verilog description from the logic diagram of the circuit.
- 
- Describe rise, fall, and turn-off delays in the gate-level design.
- 
- Explain min, max, and typ delays in the gate-level design.

[ [Team LiB](#) ]



[ [Team LiB](#) ]



## 5.1 Gate Types

A logic circuit can be designed by use of logic gates. Verilog supports basic logic gates as predefined primitives. These primitives are instantiated like modules except that they are predefined in Verilog and do not need a module definition. All logic circuits can be designed by using basic gates. There are two classes of basic gates: and/or gates and buf/not gates.

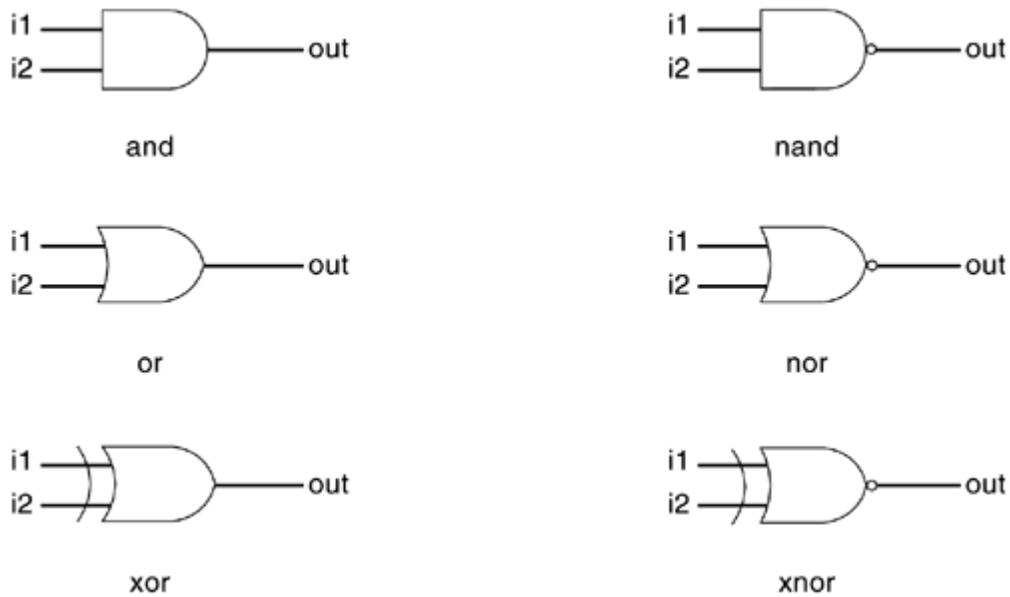
### 5.1.1 And/Or Gates

And/or gates have one scalar output and multiple scalar inputs. The first terminal in the list of gate terminals is an output and the other terminals are inputs. The output of a gate is evaluated as soon as one of the inputs changes. The and/or gates available in Verilog are shown below.

and	or	xor
nand	nor	xnor

The corresponding logic symbols for these gates are shown in [Figure 5-1](#). We consider gates with two inputs. The output terminal is denoted by out. Input terminals are denoted by i1 and i2.

**Figure 5-1. Basic Gates**



These gates are instantiated to build logic circuits in Verilog. Examples of gate instantiations are shown below. In [Example 5-1](#), for all instances, OUT is connected to the output out, and IN1 and IN2 are connected to the two inputs i1 and i2 of the gate primitives. Note that the instance name does not need to be specified for primitives. This lets the designer instantiate hundreds of gates without giving them a name.

More than two inputs can be specified in a gate instantiation. Gates with more than two inputs are instantiated by simply adding more input ports in the gate instantiation (see [Example 5-1](#)). Verilog automatically instantiates the appropriate gate.

#### [Example 5-1](#) Gate Instantiation of And/Or Gates

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## 5.2 Gate Delays

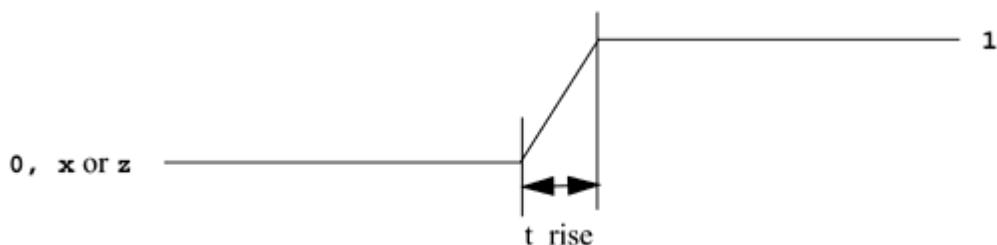
Until now, we described circuits without any delays (i.e., zero delay). In real circuits, logic gates have delays associated with them. Gate delays allow the Verilog user to specify delays through the logic circuits. Pin-to-pin delays can also be specified in Verilog. They are discussed in [Chapter 10](#), Timing and Delays.

### 5.2.1 Rise, Fall, and Turn-off Delays

There are three types of delays from the inputs to the output of a primitive gate.

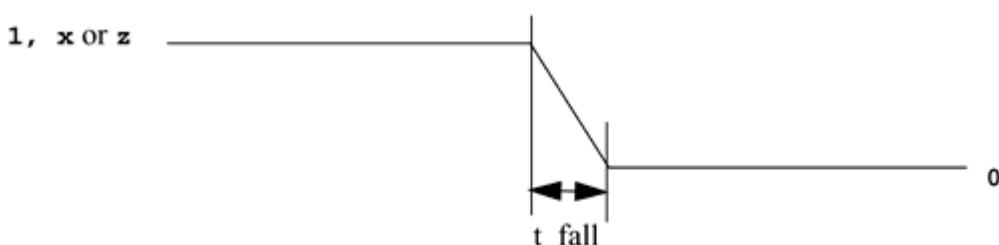
#### Rise delay

The rise delay is associated with a gate output transition to a 1 from another value.



#### Fall delay

The fall delay is associated with a gate output transition to a 0 from another value.



#### Turn-off delay

The turn-off delay is associated with a gate output transition to the high impedance value (z) from another value.

If the value changes to x, the minimum of the three delays is considered.

Three types of delay specifications are allowed. If only one delay is specified, this value is used for all transitions. If two delays are specified, they refer to the rise and fall delay values. The turn-off delay is the minimum of the two delays. If all three delays are specified, they refer to rise, fall, and turn-off delay values. If no delays are specified, the default value is zero. Examples of delay specification are shown in [Example 5-10](#).

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 5.3 Summary

In this chapter, we discussed how to model gate-level logic in Verilog. We also discussed different aspects of gate-level design.

- 
- The basic types of gates are and, or, xor, buf, and not. Each gate has a logic symbol, truth table, and a corresponding Verilog primitive. Primitives are instantiated like modules except that they are predefined in Verilog. The output of a gate is evaluated as soon as one of its inputs changes.
- 
- Arrays of built-in primitive instances and user-defined modules can be defined in Verilog.
- 
- For gate-level design, start with the logic diagram, write the Verilog description for the logic by using gate primitives, provide stimulus, and look at the output. Two design examples, a 4-to-1 multiplexer and a 4-bit full adder, were discussed. Each step of the design process was explained.
- 
- Three types of delays are associated with gates: rise, fall, and turn-off. Verilog allows specification of one, two, or three delays for each gate. Values of rise, fall, and turn-off delays are computed by Verilog, based on the one, two, or three delays specified.
- 
- For each type of delay, a minimum, typical, and maximum value can be specified. The user can choose which value to apply at simulation time. This provides the flexibility to experiment with three delay values without changing the Verilog code.
- 
- The effect of propagation delay on waveforms was explained by the simple, two-gate logic example. For each gate with a delay of  $t$ , the output changes  $t$  time units after any of the inputs change.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 5.4 Exercises

1:

Create your own 2-input Verilog gates called my-or, my-and and my-not from 2-input nand gates. Check the functionality of these gates with a stimulus module.

2:

A 2-input xor gate can be built from my\_and, my\_or and my\_not gates. Construct an xor module in Verilog that realizes the logic function,  $z = xy' + x'y$ . Inputs are x and y, and z is the output. Write a stimulus module that exercises all four combinations of x and y inputs.

3:

The 1-bit full adder described in the chapter can be expressed in a sum of products form.

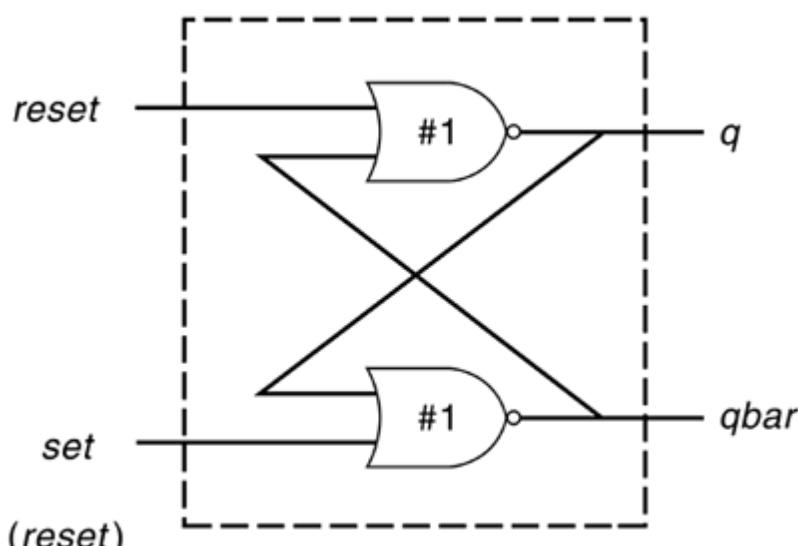
$$\text{sum} = a.b.c_{\text{in}} + a'.b.c_{\text{in}}' + a'.b'.c_{\text{in}} + a.b'.c_{\text{in}}'$$

$$c_{\text{out}} = a.b + b.c_{\text{in}} + a.c_{\text{in}}$$

Assuming a, b, c\_in are the inputs and sum and c\_out are the outputs, design a logic circuit to implement the 1-bit full adder, using only and, not, and or gates. Write the Verilog description for the circuit. You may use up to 4-input Verilog primitive and and or gates. Write the stimulus for the full adder and check the functionality for all input combinations.

4:

The logic diagram for an RS latch with delay is shown below.



Write the Verilog description for the RS latch. Include delays of 1 unit when instantiating the nor gates. Write the stimulus module for the RS latch, using the

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

# Chapter 6. Dataflow Modeling

For small circuits, the gate-level modeling approach works very well because the number of gates is limited and the designer can instantiate and connect every gate individually. Also, gate-level modeling is very intuitive to a designer with a basic knowledge of digital logic design. However, in complex designs the number of gates is very large. Thus, designers can design more effectively if they concentrate on implementing the function at a level of abstraction higher than gate level. Dataflow modeling provides a powerful way to implement a design. Verilog allows a circuit to be designed in terms of the data flow between registers and how a design processes data rather than instantiation of individual gates. Later in this chapter, the benefits of dataflow modeling will become more apparent.

With gate densities on chips increasing rapidly, dataflow modeling has assumed great importance. No longer can companies devote engineering resources to handcrafting entire designs with gates. Currently, automated tools are used to create a gate-level circuit from a dataflow design description. This process is called logic synthesis. Dataflow modeling has become a popular design approach as logic synthesis tools have become sophisticated. This approach allows the designer to concentrate on optimizing the circuit in terms of data flow. For maximum flexibility in the design process, designers typically use a Verilog description style that combines the concepts of gate-level, data flow, and behavioral design. In the digital design community, the term RTL (Register Transfer Level) design is commonly used for a combination of dataflow modeling and behavioral modeling.

## Learning Objectives

- 
- Describe the continuous assignment (assign) statement, restrictions on the assign statement, and the implicit continuous assignment statement.
- 
- Explain assignment delay, implicit assignment delay, and net declaration delay for continuous assignment statements.
- 
- Define expressions, operators, and operands.
- 
- List operator types for all possible operations?arithmetric, logical, relational, equality, bitwise, reduction, shift, concatenation, and conditional.
- 
- Use dataflow constructs to model practical digital circuits in Verilog.

[ Team LiB ]

 PREVIOUS | NEXT |

## 6.1 Continuous Assignments

A continuous assignment is the most basic statement in dataflow modeling, used to drive a value onto a net. This assignment replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction. The assignment statement starts with the keyword assign. The syntax of an assign statement is as follows.

```
continuous_assign ::= assign [ drive_strength ] [ delay3 ]
                     list_of_net_assignments ;
list_of_net_assignments ::= net_assignment { , net_assignment }
net_assignment ::= net_lvalue = expression
```

Notice that drive strength is optional and can be specified in terms of strength levels discussed in [Section 3.2.1](#), Value Set. We will not discuss drive strength specification in this chapter. The default value for drive strength is strong1 and strong0. The delay value is also optional and can be used to specify delay on the assign statement. This is like specifying delays for gates. Delay specification is discussed in this chapter. Continuous assignments have the following characteristics:

- 1.
2. The left hand side of an assignment must always be a scalar or vector net or a concatenation of scalar and vector nets. It cannot be a scalar or vector register. Concatenations are discussed in [Section 6.4.8](#), Concatenation Operator.
- 3.
4. Continuous assignments are always active. The assignment expression is evaluated as soon as one of the right-hand-side operands changes and the value is assigned to the left-hand-side net.
- 5.
6. The operands on the right-hand side can be registers or nets or function calls. Registers or nets can be scalars or vectors.
- 7.
8. Delay values can be specified for assignments in terms of time units. Delay values are used to control the time when a net is assigned the evaluated value. This feature is similar to specifying delays for gates. It is very useful in modeling timing behavior in real circuits.

Examples of continuous assignments are shown below. Operators such as &, ^, |, {, } and + used in the examples are explained in [Section 6.4](#), Operator Types. At this point, concentrate on how the assign statements are specified.

### Example 6-1 Examples of Continuous Assignment

```
// Continuous assign. out is a net. i1 and i2 are nets.
assign out = i1 & i2;

// Continuous assign for vector nets. addr is a 16-bit vector net
// addr1 and addr2 are 16-bit vector registers.
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];

// Concatenation. Left-hand side is a concatenation of a scalar
// net and a vector net.
assign [a_out : sum[3:0]] = a[3:0] + b[3:0] + c_in;
```

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## 6.2 Delays

Delay values control the time between the change in a right-hand-side operand and when the new value is assigned to the left-hand side. Three ways of specifying delays in continuous assignment statements are regular assignment delay, implicit continuous assignment delay, and net declaration delay.

### 6.2.1 Regular Assignment Delay

The first method is to assign a delay value in a continuous assignment statement. The delay value is specified after the keyword assign. Any change in values of in1 or in2 will result in a delay of 10 time units before recomputation of the expression in1 & in2, and the result will be assigned to out. If in1 or in2 changes value again before 10 time units when the result propagates to out, the values of in1 and in2 at the time of recomputation are considered. This property is called inertial delay. An input pulse that is shorter than the delay of the assignment statement does not propagate to the output.

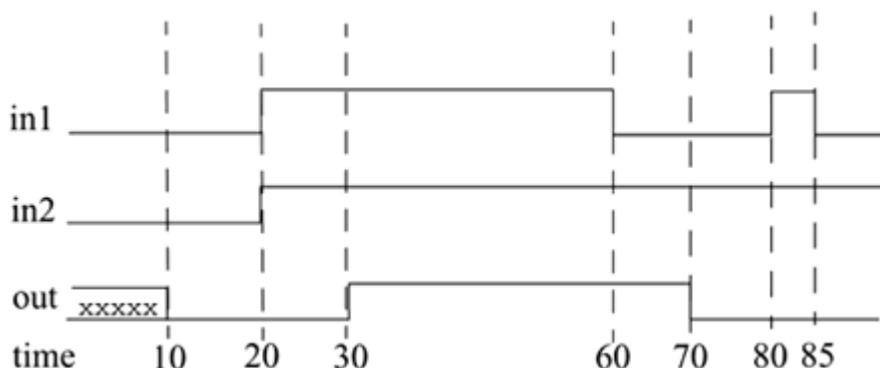
```
assign #10 out = in1 & in2; // Delay in a continuous assign
```

The waveform in [Figure 6-1](#) is generated by simulating the above assign statement. It shows the delay on signal out. Note the following change:

1.

1. When signals in1 and in2 go high at time 20, out goes to a high 10 time units later (time = 30).
- 2.
2. When in1 goes low at 60, out changes to low at 70.
- 3.
3. However, in1 changes to high at 80, but it goes down to low before 10 time units have elapsed.
- 4.
4. Hence, at the time of recomputation, 10 units after time 80, in1 is 0. Thus, out gets the value 0. A pulse of width less than the specified assignment delay is not propagated to the output.

**Figure 6-1. Delays**



Inertial delays also apply to gate delays, discussed in [Chapter 5](#), Gate-Level Modeling.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 6.3 Expressions, Operators, and Operands

Dataflow modeling describes the design in terms of expressions instead of primitive gates. Expressions, operators, and operands form the basis of dataflow modeling.

### 6.3.1 Expressions

Expressions are constructs that combine operators and operands to produce a result.

```
// Examples of expressions. Combines operands and operators
a ^ b
addr1[20:17] + addr2[20:17]
in1 | in2
```

### 6.3.2 Operands

Operands can be any one of the data types defined in [Section 3.2](#), Data Types. Some constructs will take only certain types of operands. Operands can be constants, integers, real numbers, nets, registers, times, bit-select (one bit of vector net or a vector register), part-select (selected bits of the vector net or register vector), and memories or function calls (functions are discussed later).

```
integer count, final_count;
final_count = count + 1; //count is an integer operand

real a, b, c;
c = a - b; //a and b are real operands

reg [15:0] reg1, reg2;
reg [3:0] reg_out;
reg_out = reg1[3:0] ^ reg2[3:0]; //reg1[3:0] and reg2[3:0] are
                                //part-select register operands

reg ret_value;
ret_value = calculate_parity(A, B); //calculate_parity is a
                                    //function type operand
```

### 6.3.3 Operators

Operators act on the operands to produce desired results. Verilog provides various types of operators. Operator types are discussed in detail in [Section 6.4](#), Operator Types.

```
d1 && d2 // && is an operator on operands d1 and d2
!a[0] // ! is an operator on operand a[0]
B >> 1 // >> is an operator on operands B and 1
```

[ Team LiB ]

[◀ PREVIOUS](#) [NEXT ▶](#)

## 6.4 Operator Types

Verilog provides many different operator types. Operators can be arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, or conditional. Some of these operators are similar to the operators used in the C programming language. Each operator type is denoted by a symbol. [Table 6-1](#) shows the complete listing of operator symbols classified by category.

Table 6-1. Operator Types and Symbols

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power (exponent)	two
Logical	!	logical negation	one
	&&	logical and	two
		logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## 6.5 Examples

A design can be represented in terms of gates, data flow, or a behavioral description. In this section, we consider the 4-to-1 multiplexer and 4-bit full adder described in [Section 5.1.4](#), Examples. Previously, these designs were directly translated from the logic diagram into a gate-level Verilog description. Here, we describe the same designs in terms of data flow. We also discuss two additional examples: a 4-bit full adder using carry lookahead and a 4-bit counter using negative edge-triggered D-flipflops.

### 6.5.1 4-to-1 Multiplexer

Gate-level modeling of a 4-to-1 multiplexer is discussed in [Section 5.1.4](#), Examples. The logic diagram for the multiplexer is given in [Figure 5-5](#) and the gate-level Verilog description is shown in [Example 5-5](#). We describe the multiplexer, using dataflow statements. Compare it with the gate-level description. We show two methods to model the multiplexer by using dataflow statements.

#### Method 1: logic equation

We can use assignment statements instead of gates to model the logic equations of the multiplexer (see [Example 6-2](#)). Notice that everything is same as the gate-level Verilog description except that computation of out is done by specifying one logic equation by using operators instead of individual gate instantiations. I/O ports remain the same. This is important so that the interface with the environment does not change. Only the internals of the module change. Notice how concise the description is compared to the gate-level description.

#### Example 6-2 4-to-1 Multiplexer, Using Logic Equations

```
// Module 4-to-1 multiplexer using data flow. logic equation
// Compare to gate-level model
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;

//Logic equation for out
assign out = (~s1 & ~s0 & i0) |
            (~s1 & s0 & i1) |
            (s1 & ~s0 & i2) |
            (s1 & s0 & i3) ;

endmodule
```

#### Method 2: conditional operator

There is a more concise way to specify the 4-to-1 multiplexers. In [Section 6.4.10](#), Conditional Operator, we described how a conditional statement corresponds to a multiplexer operation. We will use this operator to write a 4-to-1 multiplexer. Convince yourself that this description ([Example 6-3](#)) correctly models a multiplexer.

#### Example 6-3 4-to-1 Multiplexer, Using Conditional Operators

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 6.6 Summary

- 
- Continuous assignment is one of the main constructs used in dataflow modeling. A continuous assignment is always active and the assignment expression is evaluated as soon as one of the right-hand-side variables changes. The left-hand side of a continuous assignment must be a net. Any logic function can be realized with continuous assignments.
- 
- Delay values control the time between the change in a right-hand-side variable and when the new value is assigned to the left-hand side. Delays on a net can be defined in the assign statement, implicit continuous assignment, or net declaration.
- 
- Assignment statements contain expressions, operators, and operands.
- 
- The operator types are arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, replication, and conditional. Unary operators require one operand, binary operators require two operands, and ternary require three operands. The concatenation operator can take any number of operands.
- 
- The conditional operator behaves like a multiplexer in hardware or like the if-then-else statement in programming languages.
- 
- Dataflow description of a circuit is more concise than a gate-level description. The 4-to-1 multiplexer and the 4-bit full adder discussed in the gate-level modeling chapter can also be designed by use of dataflow statements. Two dataflow implementations for both circuits were discussed. A 4-bit ripple counter using negative edge-triggered D-flipflops was designed.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 6.7 Exercises

**1:**

A full subtractor has three 1-bit inputs x, y, and z (previous borrow) and two 1-bit outputs D (difference) and B (borrow). The logic equations for D and B are as follows:

$$D = x'.y'.z + x'.y.z' + x.y'.z' + x.y.z$$

$$B = x'.y + x'.z + y.z$$

Write the full Verilog description for the full subtractor module, including I/O ports (Remember that + in logic equations corresponds to a logical or operator (`|`) in dataflow). Instantiate the subtractor inside a stimulus block and test all eight possible combinations of x, y, and z given in the following truth table.

x	y	z	B	D
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

**2:**

A magnitude comparator checks if one number is greater than or equal to or less than another number. A 4-bit magnitude comparator takes two 4-bit numbers, A and B, as input. We write the bits in A and B as follows. The leftmost bit is the most significant bit.

$$A = A(3) A(2) A(1) A(0)$$

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

# Chapter 7. Behavioral Modeling

With the increasing complexity of digital design, it has become vitally important to make wise design decisions early in a project. Designers need to be able to evaluate the trade-offs of various architectures and algorithms before they decide on the optimum architecture and algorithm to implement in hardware. Thus, architectural evaluation takes place at an algorithmic level where the designers do not necessarily think in terms of logic gates or data flow but in terms of the algorithm they wish to implement in hardware. They are more concerned about the behavior of the algorithm and its performance. Only after the high-level architecture and algorithm are finalized, do designers start focusing on building the digital circuit to implement the algorithm.

Verilog provides designers the ability to describe design functionality in an algorithmic manner. In other words, the designer describes the behavior of the circuit. Thus, behavioral modeling represents the circuit at a very high level of abstraction. Design at this level resembles C programming more than it resembles digital circuit design. Behavioral Verilog constructs are similar to C language constructs in many ways. Verilog is rich in behavioral constructs that provide the designer with a great amount of flexibility.

## Learning Objectives

- 
- Explain the significance of structured procedures always and initial in behavioral modeling.
- 
- Define blocking and nonblocking procedural assignments.
- 
- Understand delay-based timing control mechanism in behavioral modeling. Use regular delays, intra-assignment delays, and zero delays.
- 
- Describe event-based timing control mechanism in behavioral modeling. Use regular event control, named event control, and event OR control.
- 
- Use level-sensitive timing control mechanism in behavioral modeling.
- 
- Explain conditional statements using if and else.
- 
- Describe multiway branching, using case, casex, and casez statements.
- 
- Understand looping statements such as while, for, repeat, and forever.
- 
- Define sequential and parallel blocks.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## 7.1 Structured Procedures

There are two structured procedure statements in Verilog: always and initial. These statements are the two most basic statements in behavioral modeling. All other behavioral statements can appear only inside these structured procedure statements.

Verilog is a concurrent programming language unlike the C programming language, which is sequential in nature. Activity flows in Verilog run in parallel rather than in sequence. Each always and initial statement represents a separate activity flow in Verilog. Each activity flow starts at simulation time 0. The statements always and initial cannot be nested. The fundamental difference between the two statements is explained in the following sections.

### 7.1.1 initial Statement

All statements inside an initial statement constitute an initial block. An initial block starts at time 0, executes exactly once during a simulation, and then does not execute again. If there are multiple initial blocks, each block starts to execute concurrently at time 0. Each block finishes execution independently of other blocks. Multiple behavioral statements must be grouped, typically using the keywords begin and end. If there is only one behavioral statement, grouping is not necessary. This is similar to the begin-end blocks in Pascal programming language or the { } grouping in the C programming language. [Example 7-1](#) illustrates the use of the initial statement.

#### Example 7-1 initial Statement

```
module stimulus;

reg x,y, a,b, m;

initial
    m = 1'b0; //single statement; does not need to be grouped

initial
begin
    #5 a = 1'b1; //multiple statements; need to be grouped
    #25 b = 1'b0;
end

initial
begin
    #10 x = 1'b0;
    #25 y = 1'b1;
end

initial
#50 $finish;

endmodule
```

In the above example, the three initial statements start to execute in parallel at time 0. If a delay #<delay> is seen before a statement, the statement is executed <delay> time units after the current simulation time. Thus, the execution sequence of the statements inside the initial blocks will be as follows.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## 7.2 Procedural Assignments

Procedural assignments update values of reg, integer, real, or time variables. The value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value. These are unlike continuous assignments discussed in [Chapter 6](#), Dataflow Modeling, where one assignment statement can cause the value of the right-hand-side expression to be continuously placed onto the left-hand-side net. The syntax for the simplest form of procedural assignment is shown below.

```
assignment ::= variable_lvalue = [ delay_or_event_control ]
               expression
```

The left-hand side of a procedural assignment <lvalue> can be one of the following:

- 
- A reg, integer, real, or time register variable or a memory element
- 
- A bit select of these variables (e.g., addr[0])
- 
- A part select of these variables (e.g., addr[31:16])
- 
- A concatenation of any of the above

The right-hand side can be any expression that evaluates to a value. In behavioral modeling, all operators listed in [Table 6-1](#) on page 96 can be used in behavioral expressions.

There are two types of procedural assignment statements: blocking and nonblocking.

### 7.2.1 Blocking Assignments

Blocking assignment statements are executed in the order they are specified in a sequential block. A blocking assignment will not block execution of statements that follow in a parallel block. Both parallel and sequential blocks are discussed in [Section 7.7](#), Sequential and Parallel Blocks. The = operator is used to specify blocking assignments.

#### Example 7-6 Blocking Statements

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;

//All behavioral statements must be inside an initial or always block
initial
begin
    x = 0; y = 1; z = 1; //Scalar assignments
    count = 0; //Assignment to integer variables
```

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## 7.3 Timing Controls

Various behavioral timing control constructs are available in Verilog. In Verilog, if there are no timing control statements, the simulation time does not advance. Timing controls provide a way to specify the simulation time at which procedural statements will execute. There are three methods of timing control: delay-based timing control, event-based timing control, and level-sensitive timing control.

### 7.3.1 Delay-Based Timing Control

Delay-based timing control in an expression specifies the time duration between when the statement is encountered and when it is executed. We used delay-based timing control statements when writing few modules in the preceding chapters but did not explain them in detail. In this section, we will discuss delay-based timing control statements. Delays are specified by the symbol #. Syntax for the delay-based timing control statement is shown below.

```

delay3 ::= # delay_value | # ( delay_value [ , delay_value [ ,
                                             delay_value ] ] )
delay2 ::= # delay_value | # ( delay_value [ , delay_value ] )
delay_value ::=
    unsigned_number
  | parameter_identifier
  | specparam_identifier
  | mintypmax_expression

```

Delay-based timing control can be specified by a number, identifier, or a mintypmax\_expression. There are three types of delay control for procedural assignments: regular delay control, intra-assignment delay control, and zero delay control.

#### Regular delay control

Regular delay control is used when a non-zero delay is specified to the left of a procedural assignment. Usage of regular delay control is shown in [Example 7-10](#).

#### Example 7-10 Regular Delay Control

```

//define parameters
parameter latency = 20;
parameter delta = 2;
//define register variables
reg x, y, z, p, q;

initial
begin
    x = 0; // no delay control
    #10 y = 1; // delay control with a number. Delay execution of
                // y = 1 by 10 units

    #latency z = 0; // Delay control with identifier. Delay of 20 units
    #(latency + delta) p = 1; // Delay control with expression

    #y x = x + 1; // Delay control with identifier. Take value of y.

    #(4:5:6) q = 0; // Minimum, typical and maximum delay values.

```

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## 7.4 Conditional Statements

Conditional statements are used for making decisions based upon certain conditions. These conditions are used to decide whether or not a statement should be executed. Keywords if and else are used for conditional statements. There are three types of conditional statements. Usage of conditional statements is shown below. For formal syntax, see [Appendix D](#), Formal Syntax Definition.

```
//Type 1 conditional statement. No else statement.
//Statement executes or does not execute.
if (<expression>) true_statement ;

//Type 2 conditional statement. One else statement
//Either true_statement or false_statement is evaluated
if (<expression>) true_statement ; else false_statement ;

//Type 3 conditional statement. Nested if-else-if.
//Choice of multiple statements. Only one is executed.
if (<expression1>) true_statement1 ;
else if (<expression2>) true_statement2 ;
else if (<expression3>) true_statement3 ;
else default_statement ;
```

The <expression> is evaluated. If it is true (1 or a non-zero value), the true\_statement is executed. However, if it is false (zero) or ambiguous (x), the false\_statement is executed. The <expression> can contain any operators mentioned in [Table 6-1](#) on page 96. Each true\_statement or false\_statement can be a single statement or a block of multiple statements. A block must be grouped, typically by using keywords begin and end. A single statement need not be grouped.

### Example 7-18 Conditional Statement Examples

```
//Type 1 statements
if(!lock) buffer = data;
if(enable) out = in;

//Type 2 statements
if (number_queued < MAX_Q_DEPTH)
begin
    data_queue = data;
    number_queued = number_queued + 1;
end
else
    $display("Queue Full. Try again");

//Type 3 statements
//Execute statements based on ALU control signal.
if (alu_control == 0)
    y = x + z;
else if(alu_control == 1)
    y = x - z;
else if(alu_control == 2)
    y = x * z;
else
    $display("Invalid ALU control signal");
```

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 7.5 Multiway Branching

In type 3 conditional statement in [Section 7.4](#), Conditional Statements, there were many alternatives, from which one was chosen. The nested if-else-if can become unwieldy if there are too many alternatives. A shortcut to achieve the same result is to use the case statement.

### 7.5.1 case Statement

The keywords case, endcase, and default are used in the case statement..

```
case (expression)
    alternative1: statement1;
    alternative2: statement2;
    alternative3: statement3;
    ...
    ...
    default: default_statement;
endcase
```

Each of statement1, statement2 , default\_statement can be a single statement or a block of multiple statements. A block of multiple statements must be grouped by keywords begin and end. The expression is compared to the alternatives in the order they are written. For the first alternative that matches, the corresponding statement or block is executed. If none of the alternatives matches, the default\_statement is executed. The default\_statement is optional. Placing of multiple default statements in one case statement is not allowed. The case statements can be nested. The following Verilog code implements the type 3 conditional statement in [Example 7-18](#).

```
//Execute statements based on the ALU control signal
reg [1:0] alu_control;
...
...
case (alu_control)
    2'd0 : y = x + z;
    2'd1 : y = x - z;
    2'd2 : y = x * z;
    default : $display("Invalid ALU control signal");
endcase
```

The case statement can also act like a many-to-one multiplexer. To understand this, let us model the 4-to-1 multiplexer in [Section 6.5](#), Examples, on page 106, using case statements. The I/O ports are unchanged. Notice that an 8-to-1 or 16-to-1 multiplexer can also be easily implemented by case statements.

#### Example 7-19 4-to-1 Multiplexer with Case Statement

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;
```

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## 7.6 Loops

There are four types of looping statements in Verilog: while, for, repeat, and forever. The syntax of these loops is very similar to the syntax of loops in the C programming language. All looping statements can appear only inside an initial or always block. Loops may contain delay expressions.

### 7.6.1 While Loop

The keyword while is used to specify this loop. The while loop executes until the while-expression is not true. If the loop is entered when the while-expression is not true, the loop is not executed at all. Each expression can contain the operators in [Table 6-1](#) on page 96. Any logical expression can be specified with these operators. If multiple statements are to be executed in the loop, they must be grouped typically using keywords begin and end. [Example 7-22](#) illustrates the use of the while loop.

#### Example 7-22 While Loop

```
//Illustration 1: Increment count from 0 to 127. Exit at count 128.
//Display the count variable.
integer count;

initial
begin
    count = 0;

    while (count < 128) //Execute loop till count is 127.
        //exit at count 128
    begin
        $display("Count = %d", count);
        count = count + 1;
    end
end

//Illustration 2: Find the first bit with a value 1 in flag (vector variable)
#define TRUE 1'b1';
#define FALSE 1'b0;
reg [15:0] flag;
integer i; //integer to keep count
reg continue;

initial
begin
    flag = 16'b 0010_0000_0000_0000;
    i = 0;
    continue = 'TRUE;

    while((i < 16) && continue) //Multiple conditions using operators.
    begin
        if (flag[i])
            begin
                $display("Encountered a TRUE bit at element number %d", i);
                continue = 'FALSE;
            end
        i = i + 1;
    end
end
```

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## 7.7 Sequential and Parallel Blocks

Block statements are used to group multiple statements to act together as one. In previous examples, we used keywords begin and end to group multiple statements. Thus, we used sequential blocks where the statements in the block execute one after another. In this section we discuss the block types: sequential blocks and parallel blocks. We also discuss three special features of blocks: named blocks, disabling named blocks, and nested blocks.

### 7.7.1 Block Types

There are two types of blocks: sequential blocks and parallel blocks.

#### Sequential blocks

The keywords begin and end are used to group statements into sequential blocks. Sequential blocks have the following characteristics:

- 
- The statements in a sequential block are processed in the order they are specified. A statement is executed only after its preceding statement completes execution (except for nonblocking assignments with intra-assignment timing control).
- 
- If delay or event control is specified, it is relative to the simulation time when the previous statement in the block completed execution.

We have used numerous examples of sequential blocks in this book. Two more examples of sequential blocks are given in [Example 7-26](#). Statements in the sequential block execute in order. In Illustration 1, the final values are x = 0, y = 1, z = 1, w = 2 at simulation time 0. In Illustration 2, the final values are the same except that the simulation time is 35 at the end of the block.

#### Example 7-26 Sequential Blocks

```
//Illustration 1: Sequential block without delay
reg x, y;
reg [1:0] z, w;

initial
begin
    x = 1'b0;
    y = 1'b1;
    z = {x, y};
    w = {y, x};
end

//Illustration 2: Sequential blocks with delay.
reg x, y;
reg [1:0] z, w;

initial
begin
```

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## 7.8 Generate Blocks

Generate statements allow Verilog code to be generated dynamically at elaboration time before the simulation begins. This facilitates the creation of parametrized models. Generate statements are particularly convenient when the same operation or module instance is repeated for multiple bits of a vector, or when certain Verilog code is conditionally included based on parameter definitions.

Generate statements allow control over the declaration of variables, functions, and tasks, as well as control over instantiations. All generate instantiations are coded with a module scope and require the keywords generate - endgenerate.

Generated instantiations can be one or more of the following types:

- 
- Modules
- 
- User defined primitives
- 
- Verilog gate primitives
- 
- Continuous assignments
- 
- initial and always blocks

Generated declarations and instantiations can be conditionally instantiated into a design. Generated variable declarations and instantiations can be multiply instantiated into a design. Generated instances have unique identifier names and can be referenced hierarchically. To support interconnection between structural elements and/or procedural blocks, generate statements permit the following Verilog data types to be declared within the generate scope:

- 
- net, reg
- 
- integer, real, time, realtime
- 
- event

Generated data types have unique identifier names and can be referenced hierarchically.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## 7.9 Examples

In order to illustrate the use of behavioral constructs discussed earlier in this chapter, we consider three examples in this section. The first two, 4-to-1 multiplexer and 4-bit counter, are taken from [Section 6.5](#), Examples. Earlier, these circuits were designed by using dataflow statements. We will model these circuits with behavioral statements. The third example is a new example. We will design a traffic signal controller, using behavioral constructs, and simulate it.

### 7.9.1 4-to-1 Multiplexer

We can define a 4-to-1 multiplexer with the behavioral case statement. This multiplexer was defined, in [Section 6.5.1](#), 4-to-1 Multiplexer, by dataflow statements. It is described in [Example 7-35](#) by behavioral constructs. The behavioral multiplexer can be substituted for the dataflow multiplexer; the simulation results will be identical.

#### Example 7-35 Behavioral 4-to-1 Multiplexer

```
// 4-to-1 multiplexer. Port list is taken exactly from
// the I/O diagram.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
//output declared as register
reg out;

//recompute the signal out if any input signal changes.
//All input signals that cause a recomputation of out to
//occur must go into the always @(...) sensitivity list.
always @(s1 or s0 or i0 or i1 or i2 or i3)
begin
    case ({s1, s0})
        2'b00: out = i0;
        2'b01: out = i1;
        2'b10: out = i2;
        2'b11: out = i3;
        default: out = 1'bx;
    endcase
end

endmodule
```

### 7.9.2 4-bit Counter

In [Section 6.5.3](#), Ripple Counter, we designed a 4-bit ripple carry counter. We will now design the 4-bit counter by using behavioral statements. At dataflow or gate level, the counter might be designed in hardware as ripple carry, synchronous counter, etc. But, at a behavioral level, we work at a very high level of abstraction and do not care about the underlying hardware implementation. We will design only functionality. The counter can be designed by using behavioral constructs, as shown in [Example 7-36](#). Notice how concise the behavioral counter description is compared to its dataflow counterpart. If we substitute the counter in place of the dataflow counter, the simulation results will be exactly the same, assuming that there are no x and z values on the inputs.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 7.10 Summary

We discussed digital circuit design with behavioral Verilog constructs.

- 
- A behavioral description expresses a digital circuit in terms of the algorithms it implements. A behavioral description does not necessarily include the hardware implementation details. Behavioral modeling is used in the initial stages of a design process to evaluate various design-related trade-offs. Behavioral modeling is similar to C programming in many ways.
- 
- Structured procedures initial and always form the basis of behavioral modeling. All other behavioral statements can appear only inside initial or always blocks. An initial block executes once; an always block executes continuously until simulation ends.
- 
- Procedural assignments are used in behavioral modeling to assign values to register variables. Blocking assignments must complete before the succeeding statement can execute. Nonblocking assignments schedule assignments to be executed and continue processing to the succeeding statement.
- 
- Delay-based timing control, event-based timing control, and level-sensitive timing control are three ways to control timing and execution order of statements in Verilog. Regular delay, zero delay, and intra-assignment delay are three types of delay-based timing control. Regular event, named event, and event OR are three types of event-based timing control. The wait statement is used to model level-sensitive timing control.
- 
- Conditional statements are modeled in behavioral Verilog with if and else statements. If there are multiple branches, use of case statements is recommended. casex and casez are special cases of the case statement.
- 
- Keywords while, for, repeat, and forever are used for four types of looping statements in Verilog.
- 
- Sequential and parallel are two types of blocks. Sequential blocks are specified by keywords begin and end . Parallel blocks are expressed by keywords fork and join. Blocks can be nested and named. If a block is named, the execution of the block can be disabled from anywhere in the design. Named blocks can be referenced by hierarchical names.
- 
- Generate statements allow Verilog code to be generated dynamically at elaboration time before the simulation begins. This facilitates the creation of parametrized models. Generate statements are particularly convenient when the same operation or module instance is repeated for multiple bits of a vector, or when certain Verilog code is conditionally included based on parameter definitions. Generate loop, generate conditional, and generate case are the three types of generate statements.

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## 7.11 Exercises

1:

Declare a register called oscillate. Initialize it to 0 and make it toggle every 30 time units. Do not use always statement (Hint: Use the forever loop).

2:

Design a clock with time period = 40 and a duty cycle of 25% by using the always and initial statements. The value of clock at time = 0 should be initialized to 0.

3:

Given below is an initial block with blocking procedural assignments. At what simulation time is each statement executed? What are the intermediate and final values of a, b, c, d?

```
initial
begin
    a = 1'b0;
    b = #10 1'b1;
    c = #5 1'b0;
    d = #20 {a, b, c};
end
```

4:

Repeat exercise 3 if nonblocking procedural assignments were used.

5:

What is the order of execution of statements in the following Verilog code? Is there any ambiguity in the order of execution? What are the final values of a, b, c, d?

```
initial
begin
    a = 1'b0;
    #0 c = b;
end
initial
begin
    b = 1'b1;
    #0 d = a;
end
```

6:

What is the final value of d in the following example? (Hint: See intra-assignment delays.)

```
initial
begin
```

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## Chapter 8. Tasks and Functions

A designer is frequently required to implement the same functionality at many places in a behavioral design. This means that the commonly used parts should be abstracted into routines and the routines must be invoked instead of repeating the code. Most programming languages provide procedures or subroutines to accomplish this. Verilog provides tasks and functions to break up large behavioral designs into smaller pieces. Tasks and functions allow the designer to abstract Verilog code that is used at many places in the design.

Tasks have input, output, and inout arguments; functions have input arguments. Thus, values can be passed into and out from tasks and functions. Considering the analogy of FORTRAN, tasks are similar to SUBROUTINE and functions are similar to FUNCTION.

Tasks and functions are included in the design hierarchy. Like named blocks, tasks or functions can be addressed by means of hierarchical names.

### Learning Objectives

- 
- Describe the differences between tasks and functions.
- 
- Identify the conditions required for tasks to be defined. Understand task declaration and invocation.
- 
- Explain the conditions necessary for functions to be defined. Understand function declaration and invocation.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 8.1 Differences between Tasks and Functions

Tasks and functions serve different purposes in Verilog. We discuss tasks and functions in greater detail in the following sections. However, first it is important to understand differences between tasks and functions, as outlined in [Table 8-1](#).

Table 8-1. Tasks and Functions

Functions	Tasks
A function can enable another function but not another task.	A task can enable other tasks and functions.
Functions always execute in 0 simulation time.	Tasks may execute in non-zero simulation time.
Functions must not contain any delay, event, or timing control statements.	Tasks may contain delay, event, or timing control statements.
Functions must have at least one input argument. They can have more than one input.	Tasks may have zero or more arguments of type input, output, or inout.
Functions always return a single value. They cannot have output or inout arguments.	Tasks do not return with a value, but can pass multiple values through output and inout arguments.

Both tasks and functions must be defined in a module and are local to the module. Tasks are used for common Verilog code that contains delays, timing, event constructs, or multiple output arguments. Functions are used when common Verilog code is purely combinational, executes in zero simulation time, and provides exactly one output. Functions are typically used for conversions and commonly used calculations.

Tasks can have input, output, and inout arguments; functions can have input arguments. In addition, they can have local variables, registers, time variables, integers, real, or events. Tasks or functions cannot have wires. Tasks and functions contain behavioral statements only. Tasks and functions do not contain always or initial statements but are called from always blocks, initial blocks, or other tasks and functions.

[ [Team LiB](#) ]

[ PREVIOUS ] [ NEXT ]

[ [Team LiB](#) ]

[ PREVIOUS ] [ NEXT ]

## 8.2 Tasks

Tasks are declared with the keywords task and endtask. Tasks must be used if any one of the following conditions is true for the procedure:

- 
- There are delay, timing, or event control constructs in the procedure.
- 
- The procedure has zero or more than one output arguments.
- 
- The procedure has no input arguments.

### 8.2.1 Task Declaration and Invocation

Task declaration and task invocation syntax are as follows.

#### Example 8-1 Syntax for Tasks

```

task_declaration ::= 
    task [ automatic ] task_identifier ;
    { task_item_declarator }
    statement
    endtask
  | task [ automatic ] task_identifier ( task_port_list ) ;
    { block_item_declarator }
    statement
    endtask

task_item_declarator ::= 
    block_item_declarator
  | { attribute_instance } tf_input_declarator ;
  | { attribute_instance } tf_output_declarator ;
  | { attribute_instance } tf inout_declarator ;
task_port_list ::= task_port_item { , task_port_item }
task_port_item ::= 
    { attribute_instance } tf_input_declarator
  | { attribute_instance } tf_output_declarator
  | { attribute_instance } tf inout_declarator
tf_input_declarator ::= 
    input [ reg ] [ signed ] [ range ] list_of_port_identifiers
  | input [ task_port_type ] list_of_port_identifiers
tf_output_declarator ::= 
    output [ reg ] [ signed ] [ range ] list_of_port_identifiers
  | output [ task_port_type ] list_of_port_identifiers
tf inout_declarator ::= 
    inout [ reg ] [ signed ] [ range ] list_of_port_identifiers
  | inout [ task_port_type ] list_of_port_identifiers
task_port_type ::= 
    time | real | realtime | integer

```

I/O declarations use keywords input, output, or inout, based on the type of argument declared. Input and inout arguments are passed into the task. Input arguments are processed in the task statements.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 8.3 Functions

Functions are declared with the keywords function and endfunction. Functions are used if all of the following conditions are true for the procedure:

- 
- There are no delay, timing, or event control constructs in the procedure.
- 
- The procedure returns a single value.
- 
- There is at least one input argument.
- 
- There are no output or inout arguments.
- 
- There are no nonblocking assignments.

### 8.3.1 Function Declaration and Invocation

The syntax for functions is follows:

#### Example 8-6 Syntax for Functions

```
function_declaration ::=  
    function [ automatic ] [ signed ] [ range_or_type ]  
    function_identifier ;  
    function_item_declaraction { function_item_declaraction }  
    function_statement  
    endfunction  
| function [ automatic ] [ signed ] [ range_or_type ]  
    function_identifier (function_port_list ) ;  
    block_item_declaraction { block_item_declaraction }  
    function_statement  
    endfunction  
function_item_declaraction ::=  
    block_item_declaraction  
    | tf_input_declaraction ;  
function_port_list ::= { attribute_instance } tf_input_declaraction { ,  
                      { attribute_instance } tf_input_declaraction }  
range_or_type ::= range | integer | real | realtime | time
```

There are some peculiarities of functions. When a function is declared, a register with name function\_identifier is declared implicitly inside Verilog. The output of a function is passed back by setting the value of the register function\_identifier appropriately. The function is invoked by specifying function name and input arguments. At the end of function execution, the return value is placed where the function was invoked. The optional range\_or\_type specifies the width of the internal register. If no range or type is specified, the default bit width is 1. Functions are very similar to FUNCTION in FORTRAN.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 8.4 Summary

In this chapter, we discussed tasks and functions used in behavior Verilog modeling.

- 
- Tasks and functions are used to define common Verilog functionality that is used at many places in the design. Tasks and functions help to make a module definition more readable by breaking it up into manageable subunits. Tasks and functions serve the same purpose in Verilog as subroutines do in C.
- 
- Tasks can take any number of input, inout, or output arguments. Delay, event, or timing control constructs are permitted in tasks. Tasks can enable other tasks or functions.
- 
- Re-entrant tasks defined with the keyword automatic allow each task call to operate in an independent space. Therefore, re-entrant tasks work correctly even with concurrent tasks calls.
- 
- Functions are used when exactly one return value is required and at least one input argument is specified. Delay, event, or timing control constructs are not permitted in functions. Functions can invoke other functions but cannot invoke other tasks.
- 
- A register with name as the function name is declared implicitly when a function is declared. The return value of the function is passed back in this register.
- 
- Recursive functions defined with the keyword automatic allow each function call to operate in an independent space. Therefore, recursive or concurrent calls to such functions will work correctly.
- 
- Tasks and functions are included in a design hierarchy and can be addressed by hierarchical name referencing.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 8.5 Exercises

1:

Define a function to calculate the factorial of a 4-bit number. The output is a 32-bit value. Invoke the function by using stimulus and check results.

2:

Define a function to multiply two 4-bit numbers a and b. The output is an 8-bit value. Invoke the function by using stimulus and check results.

3:

Define a function to design an 8-function ALU that takes two 4-bit numbers a and b and computes a 5-bit result out based on a 3-bit select signal. Ignore overflow or underflow bits.

Select Signal	Function Output
3'b000	a
3'b001	a + b
3'b010	a - b
3'b011	a / b
3'b100	a % 1 (remainder)
3'b101	a << 1
3'b110	a >> 1
3'b111	(a > b) (magnitude compare)

4:

Define a task to compute the factorial of a 4-bit number. The output is a 32-bit value. The result is assigned to the output after a delay of 10 time units.

5:

Define a task to compute even parity of a 16-bit number. The result is a 1-bit value that is assigned to the output after three positive edges of clock.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## Chapter 9. Useful Modeling Techniques

We learned the basic features of Verilog in the preceding chapters. In this chapter, we will discuss additional features that enhance the Verilog language, making it powerful and flexible for modeling and analyzing a design.

### Learning Objectives

- 
- Describe procedural continuous assignment statements assign, deassign, force, and release. Explain their significance in modeling and debugging.
- 
- Understand how to override parameters by using the defparam statement at the time of module instantiation.
- 
- Explain conditional compilation and execution of parts of the Verilog description.
- 
- Identify system tasks for file output, displaying hierarchy, strobing, random number generation, memory initialization, and value change dump.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## 9.1 Procedural Continuous Assignments

We studied procedural assignments in [Section 7.2](#), Procedural Assignments. Procedural assignments assign a value to a register. The value stays in the register until another procedural assignment puts another value in that register. Procedural continuous assignments behave differently. They are procedural statements which allow values of expressions to be driven continuously onto registers or nets for limited periods of time. Procedural continuous assignments override existing assignments to a register or net. They provide an useful extension to the regular procedural assignment statement.

### 9.1.1 assign and deassign

The keywords assign and deassign are used to express the first type of procedural continuous assignment. The left-hand side of procedural continuous assignments can be only be a register or a concatenation of registers. It cannot be a part or bit select of a net or an array of registers. Procedural continuous assignments override the effect of regular procedural assignments. Procedural continuous assignments are normally used for controlled periods of time.

A simple example is the negative edge-triggered D-flipflop with asynchronous reset that we modeled in [Example 6-8](#). In [Example 9-1](#), we now model the same D\_FF, using assign and deassign statements.

#### Example 9-1 D-Flipflop with Procedural Continuous Assignments

```
// Negative edge-triggered D-flipflop with asynchronous reset
module edge_dff(q, qbar, d, clk, reset);

// Inputs and outputs
output q,qbar;
input d, clk, reset;
reg q, qbar; //declare q and qbar are registers

always @(negedge clk) //assign value of q & qbar at active edge of clock.
begin
    q = d;
    qbar = ~d;
end

always @(reset) //Override the regular assignments to q and qbar
                //whenever reset goes high. Use of procedural continuous
                //assignments.
if(reset)
begin //if reset is high, override regular assignments to q with
      //the new values, using procedural continuous assignment.
      assign q = 1'b0;
      assign qbar = 1'b1;
end
else
begin //If reset goes low, remove the overriding values by
      //deassigning the registers. After this the regular
      //assignments q = d and qbar = ~d will be able to change
      //the registers on the next negative edge of clock.
      deassign q;
      deassign qbar;
end

endmodule
```

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 9.2 Overriding Parameters

Parameters can be defined in a module definition, as was discussed earlier in [Section 3.2.8](#), Parameters. However, during compilation of Verilog modules, parameter values can be altered separately for each module instance. This allows us to pass a distinct set of parameter values to each module during compilation regardless of predefined parameter values.

There are two ways to override parameter values: through the defparam statement or through module instance parameter value assignment.

### 9.2.1 defparam Statement

Parameter values can be changed in any module instance in the design with the keyword defparam. The hierarchical name of the module instance can be used to override parameter values. Consider [Example 9-2](#), which uses defparam to override the parameter values in module instances.

#### Example 9-2 Defparam Statement

```
//Define a module hello_world
module hello_world;
parameter id_num = 0; //define a module identification number = 0

initial //display the module identification number
    $display("Displaying hello_world id number = %d", id_num);
endmodule

//define top-level module
module top;
//change parameter values in the instantiated modules
//Use defparam statement
defparam w1.id_num = 1, w2.id_num = 2;

//instantiate two hello_world modules
hello_world w1();
hello_world w2();

endmodule
```

In [Example 9-2](#), the module hello\_world was defined with a default id\_num = 0. However, when the module instances w1 and w2 of the type hello\_world are created, their id\_num values are modified with the defparam statement. If we simulate the above design, we would get the following output:

```
Displaying hello_world id number = 1
Displaying hello_world id number = 2
```

Multiple defparam statements can appear in a module. Any parameter can be overridden with the defparam statement. The defparam construct is now considered to be a bad coding style and it is recommended that alternative styles be used in Verilog HDL code.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## 9.3 Conditional Compilation and Execution

A portion of Verilog might be suitable for one environment but not for another. The designer does not wish to create two versions of Verilog design for the two environments. Instead, the designer can specify that the particular portion of the code be compiled only if a certain flag is set. This is called conditional compilation.

A designer might also want to execute certain parts of the Verilog design only when a flag is set at run time. This is called conditional execution.

### 9.3.1 Conditional Compilation

Conditional compilation can be accomplished by using compiler directives `ifdef, `ifndef, `else, `elsif, and `endif. [Example 9-5](#) contains Verilog source code to be compiled conditionally.

#### Example 9-5 Conditional Compilation

```
//Conditional Compilation
//Example 1
'ifdef TEST //compile module test only if text macro TEST is defined
module test;
...
...
endmodule
'else //compile the module stimulus as default
module stimulus;
...
...
endmodule
'endif //completion of 'ifdef directive

//Example 2
module top;

bus_master b1(); //instantiate module unconditionally
'ifdef ADD_B2
    bus_master b2(); //b2 is instantiated conditionally if text macro
                     //ADD_B2 is defined
'elsif ADD_B3
    bus_master b3(); //b3 is instantiated conditionally if text macro
                     //ADD_B3 is defined
'else
    bus_master b4(); //b4 is instantiate by default
'endif

'ifndef IGNORE_B5
    bus_master b5(); //b5 is instantiated conditionally if text macro
                     //IGNORE_B5 is not defined
'endif
endmodule
```

The `ifdef and `ifndef directives can appear anywhere in the design. A designer can conditionally compile statements, modules, blocks, declarations, and other compiler directives. The `else directive is optional. A maximum of one `else directive can accompany an `ifdef or `ifndef. Any number of `elsif directives can

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## 9.4 Time Scales

Often, in a single simulation, delay values in one module need to be defined by using certain time unit, e.g., 1 s, and delay values in another module need to be defined by using a different time unit, e.g. 100 ns. Verilog HDL allows the reference time unit for modules to be specified with the `timescale compiler directive.

Usage: `timescale <reference\_time\_unit> / <time\_precision>

The <reference\_time\_unit> specifies the unit of measurement for times and delays. The <time\_precision> specifies the precision to which the delays are rounded off during simulation. Only 1, 10, and 100 are valid integers for specifying time unit and time precision. Consider the two modules, dummy1 and dummy2, in [Example 9-8](#).

### Example 9-8 Time Scales

```
//Define a time scale for the module dummy1
//Reference time unit is 100 nanoseconds and precision is 1 ns
`timescale 100 ns / 1 ns

module dummy1;

reg toggle;

//initialize toggle
initial
  toggle = 1'b0;

//Flip the toggle register every 5 time units
//In this module 5 time units = 500 ns = .5 ?s
always #5
  begin
    toggle = ~toggle;
    $display("%d , In %m toggle = %b ", $time, toggle);
  end

endmodule

//Define a time scale for the module dummy2
//Reference time unit is 1 microsecond and precision is 10 ns
`timescale 1 us / 10 ns

module dummy2;

reg toggle;

//initialize toggle
initial
  toggle = 1'b0;

//Flip the toggle register every 5 time units
//In this module 5 time units = 5 ?s = 5000 ns
always #5
  begin
    toggle = ~toggle;
    $display("%d , In %m toggle = %b ", $time, toggle);
  end
```

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## 9.5 Useful System Tasks

In this section, we discuss the system tasks that are useful for a variety of purposes in Verilog. We discuss system tasks [1] for file output, displaying hierarchy, strobing, random number generation, memory initialization, and value change dump.

[1] Other system tasks such as \$signed and \$unsigned used for sign conversion are not discussed in this book. For details, please refer to the "IEEE Standard Verilog Hardware Description Language" document.

### 9.5.1 File Output

Output from Verilog normally goes to the standard output and the file verilog.log. It is possible to redirect the output of Verilog to a chosen file.

#### Opening a file

A file can be opened with the system task \$fopen.

Usage: \$fopen("<name\_of\_file>"); [2]

[2] The "IEEE Standard Verilog Hardware Description Language" document provides additional capabilities for \$fopen. The \$fopen syntax mentioned in this book is adequate for most purposes. However, if you need additional capabilities, please refer to the "IEEE Standard Verilog Hardware Description Language" document.

Usage: <file\_handle> = \$fopen("<name\_of\_file>");

The task \$fopen returns a 32-bit value called a multichannel descriptor.[3] Only one bit is set in a multichannel descriptor. The standard output has a multichannel descriptor with the least significant bit (bit 0) set. Standard output is also called channel 0. The standard output is always open. Each successive call to \$fopen opens a new channel and returns a 32-bit descriptor with bit 1 set, bit 2 set, and so on, up to bit 30 set. Bit 31 is reserved. The channel number corresponds to the individual bit set in the multichannel descriptor. [Example 9-9](#) illustrates the use of file descriptors.

[3] The "IEEE Standard Verilog Hardware Description Language" document provides a method for opening up to 230 files by using a single-channel file descriptor. Please refer to it for details.

#### Example 9-9 File Descriptors

```
//Multichannel descriptor
integer handle1, handle2, handle3; //integers are 32-bit values

//standard output is open; descriptor = 32'h0000_0001 (bit 0 set)
initial
begin
    handle1 = $fopen("file1.out"); //handle1 = 32'h0000_0002 (bit 1 set)
    handle2 = $fopen("file2.out"); //handle2 = 32'h0000_0004 (bit 2 set)
    handle3 = $fopen("file3.out"); //handle3 = 32'h0000_0008 (bit 3 set)
```

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## 9.6 Summary

In this chapter, we discussed the following aspects of Verilog:

- 
- Procedural continuous assignments can be used to override the assignments on registers and nets. assign and deassign can override assignments on registers. force and release can override assignments on registers and nets. assign and deassign are used in the actual design. force and release are used for debugging.
- 
- Parameters defined in a module can be overridden with the defparam statement or by passing a new value during module instantiation. During module instantiation, parameter values can be assigned by ordered list or by name. It is recommended to use parameter assignment by name.
- 
- Compilation of parts of the design can be made conditional by using the 'ifdef, 'ifndef, 'elsif, 'else, and 'endif directives. Compilation flags are defined at compile time by using the `define statement.
- 
- Execution is made conditional in Verilog simulators by means of the \$test\$plusargs system task. The execution flags are defined at run time by +<flag\_name>.
- 
- Up to 30 files can be opened for writing in Verilog. Each file is assigned a bit in the multichannel descriptor. The multichannel descriptor concept can be used to write to multiple files. The IEEE Standard Verilog Hardware Description Language document describes more advanced ways of doing file I/O.
- 
- Hierarchy can be displayed with the %m option in any display statement.
- 
- Strobing is a way to display values at a certain time or event after all other statements in that time unit have executed.
- 
- Random numbers can be generated with the system task \$random. They are used for random test vector generation. \$random task can generate both positive and negative numbers.
- 
- Memory can be initialized from a data file. The data file contains addresses and data. Addresses can also be specified in memory initialization tasks.
- 
- Value Change Dump is a popular format used by many designers for debugging with postprocessing tools. Verilog allows all or selected module variables to be dumped to the VCD file. Various system tasks are available for this purpose.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 9.7 Exercises

**1:**

Using assign and deassign statements, design a positive edge-triggered D-flipflop with asynchronous clear ( $q=0$ ) and preset ( $q=1$ ).

**2:**

Using primitive gates, design a 1-bit full adder FA. Instantiate the full adder inside a stimulus module. Force the sum output to  $a \& b \& c_{in}$  for the time between 15 and 35 units.

**3:**

A 1-bit full adder FA is defined with gates and with delay parameters as shown below.

```
// Define a 1-bit full adder
module fulladd(sum, c_out, a, b, c_in);
parameter d_sum = 0, d_cout = 0;

// I/O port declarations
output sum, c_out;
input a, b, c_in;

// Internal nets
wire s1, c1, c2;

// Instantiate logic gate primitives
xor (s1, a, b);
and (c1, a, b);

xor #(d_sum) (sum, s1, c_in); //delay on
output sum is d_sum
and (c2, s1, c_in);

or #(d_cout) (c_out, c2, c1); //delay
on output c_out is d_cout

endmodule
```

Define a 4-bit full adder fulladd4 as shown in [Example 5-8](#) on page 77, but pass the following parameter values to the instances, using the two methods discussed in the book:

Instance	Delay Values
fa0	d_sum=1, d_cout=1
fa1	d_sum=2, d_cout=2

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

# Part 2: Advanced Verilog Topics

## [10 Timing and Delays](#)

Distributed, lumped and pin-to-pin delays, specify blocks, parallel and full connection, timing checks, delay back-annotation.

## [11 Switch-Level Modeling](#)

MOS and CMOS switches, bidirectional switches, modeling of power and ground, resistive switches, delay specification on switches.

## [12 User-Defined Primitives](#)

Parts of UDP, UDP rules, combinational UDPs, sequential UDPs, shorthand symbols.

## [13 Programming Language Interface](#)

Introduction to PLI, uses of PLI, linking and invocation of PLI tasks, conceptual representation of design, PLI access and utility routines.

## [14 Logic Synthesis with Verilog HDL](#)

Introduction to logic synthesis, impact of logic synthesis, Verilog HDL constructs and operators for logic synthesis, synthesis design flow, verification of synthesized circuits, modeling tips, design partitioning.

## [15 Advanced Verification Techniques](#)

Introduction to a simple verification flow, architectural modeling, test vectors/testbenches, simulation acceleration, emulation, analysis/coverage, assertion checking, formal verification, semi-formal verification, equivalence checking.

[ Team LiB ]

◀ PREVIOUS    NEXT ▶

[ Team LiB ]

◀ PREVIOUS    NEXT ▶



# Chapter 10. Timing and Delays

Functional verification of hardware is used to verify functionality of the designed circuit. However, blocks in real hardware have delays associated with the logic elements and paths in them. Therefore, we must also check whether the circuit meets the timing requirements, given the delay specifications for the blocks. Checking timing requirements has become increasingly important as circuits have become smaller and faster. One of the ways to check timing is to do a timing simulation that accounts for the delays associated with the block during the simulation.

Techniques other than timing simulation to verify timing have also emerged in design automation industry. The most popular technique is static timing verification. Designers first do a pure functional verification and then verify timing separately with a static timing verification tool. The main advantage of static verification is that it can verify timing in orders of magnitude more quickly than timing simulation. Static timing verification is a separate field of study and is not discussed in this book.

In this chapter, we discuss in detail how timing and delays are controlled and specified in Verilog modules. Thus, by using timing simulation, the designer can verify both functionality and timing of the circuit with Verilog.

## Learning Objectives

- 
- Identify types of delay models, distributed, lumped, and pin-to-pin (path) delays used in Verilog simulation.
- 
- Understand how to set path delays in a simulation by using specify blocks.
- 
- Explain parallel connection and full connection between input and output pins.
- 
- Understand how to define parameters inside specify blocks by using specparam statements.
- 
- Describe state-dependent path delays.
- 
- Explain rise, fall, and turn-off delays. Understand how to set min, max, and typ values.
- 
- Define system tasks for timing checks \$setup, \$hold, and \$width.
- 
- Understand delay back-annotation.

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

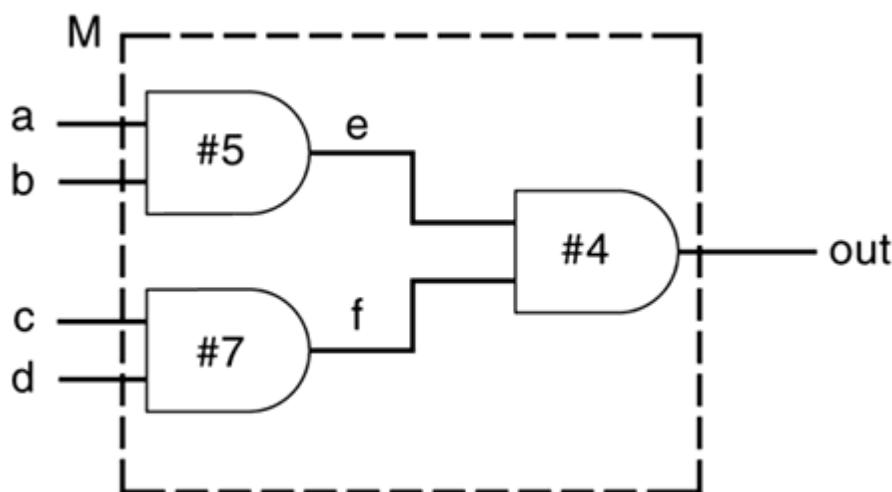
## 10.1 Types of Delay Models

There are three types of delay models used in Verilog: distributed, lumped, and pin-to-pin (path) delays.

### 10.1.1 Distributed Delay

Distributed delays are specified on a per element basis. Delay values are assigned to individual elements in the circuit. An example of distributed delays in module M is shown in [Figure 10-1](#).

**Figure 10-1. Distributed Delay**



Distributed delays can be modeled by assigning delay values to individual gates or by using delay values in individual assign statements. When inputs of any gate change, the output of the gate changes after the delay value specified. [Example 10-1](#) shows how distributed delays are specified in gates and dataflow description.

#### Example 10-1 Distributed Delays

```

//Distributed delays in gate-level modules
module M (out, a, b, c, d);
output out;
input a, b, c, d;

wire e, f;

//Delay is distributed to each gate.
and #5 a1(e, a, b);
and #7 a2(f, c, d);
and #4 a3(out, e, f);
endmodule

//Distributed delays in data flow definition of a module
module M (out, a, b, c, d);
output out;
input a, b, c, d;

wire e, f;

//Distributed delay in each expression
  
```

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## 10.2 Path Delay Modeling

In this section, we discuss various aspects of path delay modeling. In this section, the terms pin and port are used interchangeably.

### 10.2.1 Specify Blocks

A delay between a source (input or inout) pin and a destination (output or inout) pin of a module is called a module path delay. Path delays are assigned in Verilog within the keywords `specify` and `endspecify`. The statements within these keywords constitute a `specify` block.

Specify blocks contain statements to do the following:

- Assign pin-to-pin timing delays across module paths
  - Set up timing checks in the circuits
  - Define specparam constants

For the example in [Figure 10-3](#), we can write the module M with pin-to-pin delays, using specify blocks as follows:

### **Example 10-3 Pin-to-Pin Delay**

```

//Pin-to-pin delays
module M (out, a, b, c, d);
output out;
input a, b, c, d;

wire e, f;

//Specify block with path delay statements
specify
    (a => out) = 9;
    (b => out) = 9;
    (c => out) = 11;
    (d => out) = 11;
endspecify

//gate instantiations
and a1(e, a, b);
and a2(f, c, d);
and a3(out, e, f);
endmodule

```

The specify block is a separate block in the module and does not appear under any other block, such as initial or always. The meaning of the statements within specify blocks needs to be clarified. In the

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 10.3 Timing Checks

In the earlier sections of this chapter, we discussed how to specify path delays. The purpose of specifying path delays is to simulate the timing of the actual digital circuit with greater accuracy than gate delays. In this section, we describe how to set up timing checks to see if any timing constraints are violated during simulation. Timing verification is particularly important for timing critical, high-speed sequential circuits such as microprocessors.

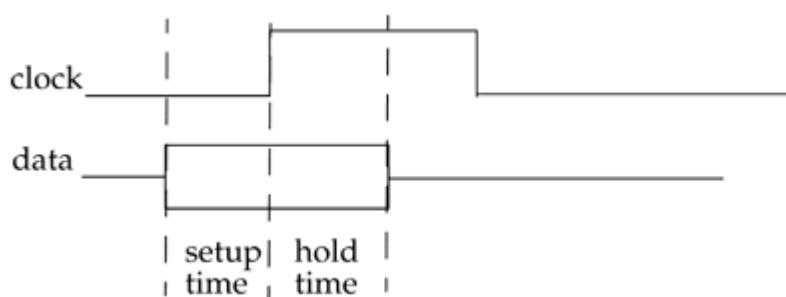
System tasks are provided to do timing checks in Verilog. There are many timing check system tasks available in Verilog. We will discuss the three most common timing checks [1] tasks: \$setup, \$hold, and \$width. All timing checks must be inside the specify blocks only. Optional notifier arguments used in these timing check system tasks are omitted to simplify the discussion.

[1] The IEEE Standard Verilog Hardware Description Language document provides additional constraint checks, \$removal, \$recrem, \$timeskew, \$fullskew. Please refer to it for details. Negative input timing constraints can also be specified.

### 10.3.1 \$setup and \$hold Checks

\$setup and \$hold tasks are used to check the setup and hold constraints for a sequential element in the design. In a sequential element such as an edge-triggered flip-flop, the setup time is the minimum time the data must arrive before the active clock edge. The hold time is the minimum time the data cannot change after the active clock edge. Setup and hold times are shown in [Figure 10-6](#).

**Figure 10-6. Setup and Hold Times**



#### \$setup task

Setup checks can be specified with the system task \$setup.

Usage:

`$setup(data_event, reference_event, limit);`

data\_event

Signal that is monitored for violations

reference\_event

Signal that establishes a

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## 10.4 Delay Back-Annotation

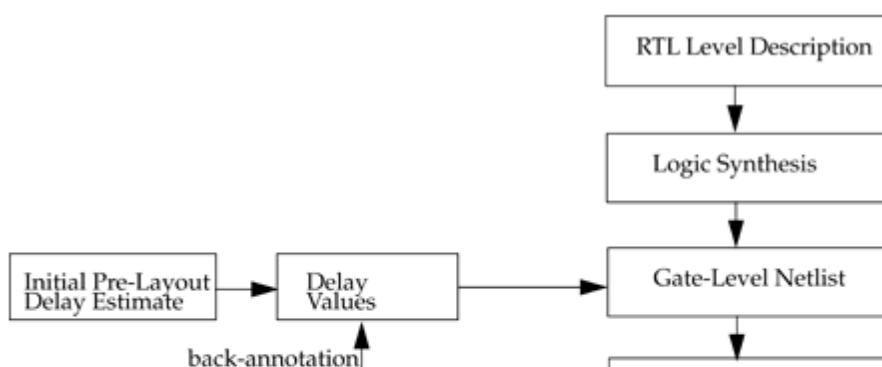
Delay back-annotation is an important and vast topic in timing simulation. An entire book could be devoted to that subject. However, in this section, we introduce the designer to the concept of back-annotation of delays in a simulation. Detailed coverage of this topic is outside the scope of this book. For details, refer to the IEEE Standard Verilog Hardware Description Language document.

The various steps in the flow that use delay back-annotation are as follows:

- 1.
2. The designer writes the RTL description and then performs functional simulation.
3. The RTL description is converted to a gate-level netlist by a logic synthesis tool.
4. The designer obtains pre-layout estimates of delays in the chip by using a delay calculator and information about the IC fabrication process. Then, the designer does timing simulation or static timing verification of the gate-level netlist, using these preliminary values to check that the gate-level netlist meets timing constraints.
- 5.
6. The gate-level netlist is then converted to layout by a place and route tool. The post-layout delay values are computed from the resistance ( $R$ ) and capacitance ( $C$ ) information in the layout. The  $R$  and  $C$  information is extracted from factors such as geometry and IC fabrication process.
- 7.
8. The post-layout delay values are back-annotated to modify the delay estimates for the gate-level netlist. Timing simulation or static timing verification is run again on the gate-level netlist to check if timing constraints are still satisfied.
- 9.
10. If design changes are required to meet the timing constraints, the designer has to go back to the RTL level, optimize the design for timing, and then repeat Step 2 through Step 5.

[Figure 10-7](#) shows the flow of delay back annotation.

**Figure 10-7. Delay Back-Annotation**



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 10.5 Summary

In this chapter, we discussed the following aspects of Verilog:

- 
- There are three types of delay models: lumped, distributed, and path delays. Distributed delays are more accurate than lumped delays but difficult to model for large designs. Lumped delays are relatively simpler to model.
- 
- Path delays, also known as pin-to-pin delays, specify delays from input or inout pins to output or inout pins. Path delays provide the most accuracy for modeling delays within a module.
- 
- Specify blocks are the basic blocks for expressing path delay information. In modules, specify blocks appear separately from initial or always blocks.
- 
- Parallel connection and full connection are two methods to describe path delays.
- 
- Parameters can be defined inside the specify blocks by specparam statements.
- 
- Path delays can be conditional or dependent on the values of signals in the circuit. They are known as State Dependent Path Delays (SDPD).
- 
- Rise, fall, and turn-off delays can be described in a path delay. Min, max, and typical values can also be specified. Transitions to x are handled by the pessimistic method.
- 
- Setup, hold, and width are timing checks that check timing integrity of the digital circuit. Other timing checks are also available but are not discussed in the book.
- 
- Delay back-annotation is used to resimulate the digital design with path delays extracted from layout information. This process is used repeatedly to obtain a final circuit that meets all timing requirements.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

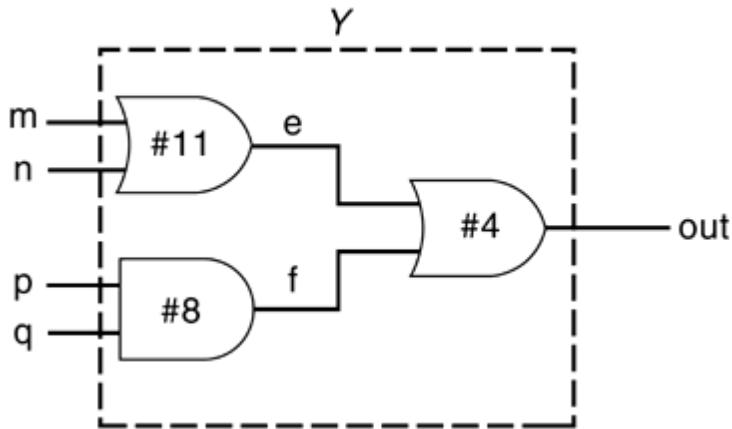
[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## 10.6 Exercises

1:

What type of delay model is used in the following circuit? Write the Verilog description for the module Y.



2:

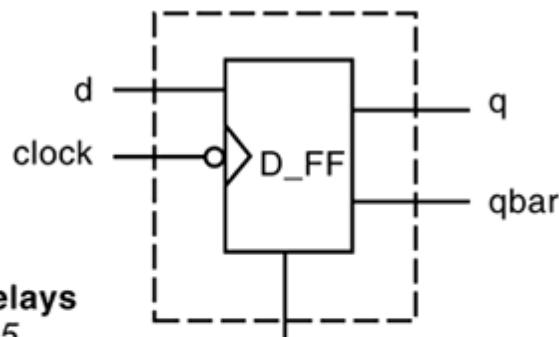
Use the largest delay in the module to convert the circuit to a lumped delay model. Using a lumped delay model, write the Verilog description for the module Y.

3:

Compute the delays along each path from input to output for the circuit in Exercise 1. Write the Verilog description, using the path delay model. Use specify blocks.

4:

Consider the negative edge-triggered with the asynchronous reset D-flipflop shown in the figure below. Write the Verilog description for the module D\_FF. Show only the I/O ports and path delay specification. Describe path delays, using parallel connection.



**Path Delays**

$d \rightarrow q = 5$   
 $d \rightarrow qbar = 5$   
 $clock \rightarrow q = 6$   
 $clock \rightarrow qbar = 7$   
 $reset \rightarrow q = 2$   
 $reset \rightarrow qbar = 3$

5:

Modify the D-flipflop in Exercise 4 if all path delays are 5 units. Describe the modified D-flipflop.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

# Chapter 11. Switch-Level Modeling

In [Part 1](#) of this book, we explained digital design and simulation at a higher level of abstraction such as gates, data flow, and behavior. However, in rare cases designers will choose to design the leaf-level modules, using transistors. Verilog provides the ability to design at a MOS-transistor level. Design at this level is becoming rare with the increasing complexity of circuits (millions of transistors) and with the availability of sophisticated CAD tools. Verilog HDL currently provides only digital design capability with logic values 0, 1, x, z, and the drive strengths associated with them. There is no analog capability. Thus, in Verilog HDL, transistors are also known switches that either conduct or are open. In this chapter, we discuss the basic principles of switch-level modeling. For most designers, it is adequate to know only the basics. Detailed information on signal strengths and advanced net definitions is provided in [Appendix A](#), Strength Modeling and Advanced Net Definitions. Refer to the IEEE Standard Verilog Hardware Description Language document for complete details on switch-level modeling.

## Learning Objectives

- 
- Describe basic MOS switches nmos, pmos, and cmos.
- 
- Understand modeling of bidirectional pass switches, power, and ground.
- 
- Identify resistive MOS switches.
- 
- Explain the method to specify delays on basic MOS switches and bidirectional pass switches.
- 
- Build basic switch-level circuits in Verilog, using available switches.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



**ABC Amber CHM Converter Trial version**

Please register to remove this banner.

<http://www.thebeatlesforever.com/processtext/abcchm.html>

[ [Team LiB](#) ]

[PREVIOUS](#) [NEXT](#)

## 11.1 Switch-Modeling Elements

Verilog provides various constructs to model switch-level circuits. Digital circuits at MOS-transistor level are described using these elements.[\[1\]](#)

[1] Array of instances can be defined for switches. Array of instances is described in [Section 5.1.3, Array of Instances](#).

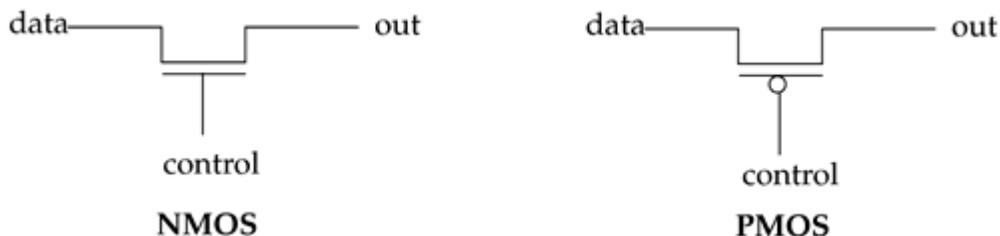
### 11.1.1 MOS Switches

Two types of MOS switches can be defined with the keywords nmos and pmos.

```
//MOS switch keywords  
nmos          pmos
```

Keyword nmos is used to model NMOS transistors; keyword pmos is used to model PMOS transistors. The symbols for nmos and pmos switches are shown in [Figure 11-1](#).

**Figure 11-1. NMOS and PMOS Switches**



In Verilog, nmos and pmos switches are instantiated as shown in [Example 11-1](#).

#### Example 11-1 Instantiation of NMOS and PMOS Switches

```
nmos n1(out, data, control); //instantiate a nmos switch  
pmos p1(out, data, control); //instantiate a pmos switch
```

Since switches are Verilog primitives, like logic gates, the name of the instance is optional. Therefore, it is acceptable to instantiate a switch without assigning an instance name.

```
nmos (out, data, control); //instantiate an nmos switch; no instance name  
pmos (out, data, control); //instantiate a pmos switch; no instance name
```

The value of the out signal is determined from the values of data and control signals. Logic tables for out are shown in [Table 11-1](#). Some combinations of data and control signals cause the gates to output to either a 1 or 0, or to an z value without a preference for either value. The symbol L stands for 0 or z; H stands for 1 or z.

**Table 11-1. Logic Tables for NMOS and PMOS**

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

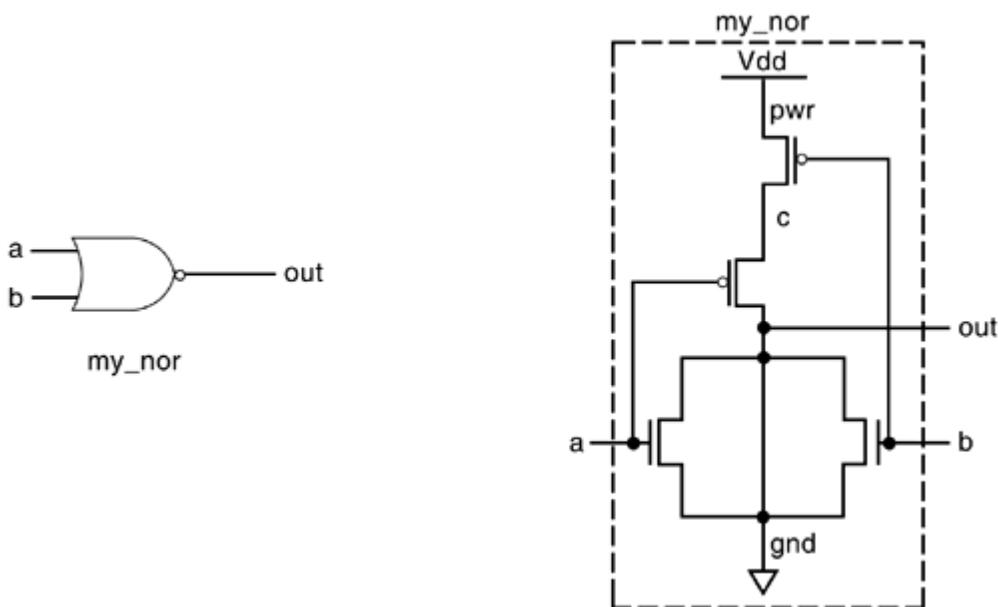
## 11.2 Examples

In this section, we discuss how to build practical digital circuits, using switch-level constructs.

### 11.2.1 CMOS Nor Gate

Though Verilog has a nor gate primitive, let us design our own nor gate, using CMOS switches. The gate and the switch-level circuit diagram for the nor gate are shown in [Figure 11-4](#).

**Figure 11-4. Gate and Switch Diagram for Nor Gate**



Using the switch primitives discussed in [Section 11.1](#), Switch-Modeling Elements, the Verilog description of the circuit is shown in [Example 11-4](#) below.

#### Example 11-4 Switch-Level Verilog for Nor Gate

```
//Define our own nor gate, my_nor
module my_nor(out, a, b);

output out;
input a, b;

//internal wires
wire c;

//set up power and ground lines
supply1 pwr;      //pwr is connected to Vdd (power supply)
supply0 gnd ;     //gnd is connected to Vss(ground)

//instantiate pmos switches
pmos  (c, pwr, b);
pmos  (out, c, a);

//instantiate nmos switches
nmos  (out, gnd, a);
nmos  (out, gnd, b);
```

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 11.3 Summary

We discussed the following aspects of Verilog in this chapter:

- 
- Switch-level modeling is at a very low level of design abstraction. Designers use switch modeling in rare cases when they need to customize a leaf cell. Verilog design at this level is becoming less popular with increasing complexity of circuits.
- 
- MOS, CMOS, bidirectional switches, and supply1 and supply0 sources can be used to design any switch-level circuit. CMOS switches are a combination of MOS switches.
- 
- Delays can be optionally specified for switch elements. Delays are interpreted differently for bidirectional devices.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 11.4 Exercises

1:

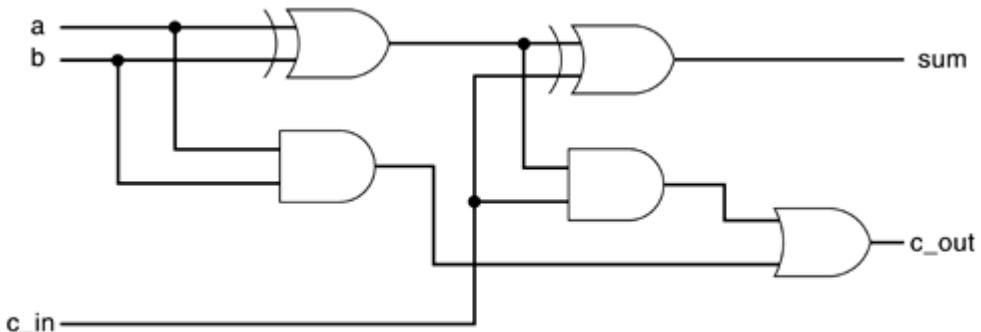
Draw the circuit diagram for an xor gate, using nmos and pmos switches. Write the Verilog description for the circuit. Apply stimulus and test the design.

2:

Draw the circuit diagram for and and or gates, using nmos and pmos switches. Write the Verilog description for the circuits. Apply stimulus and test the design.

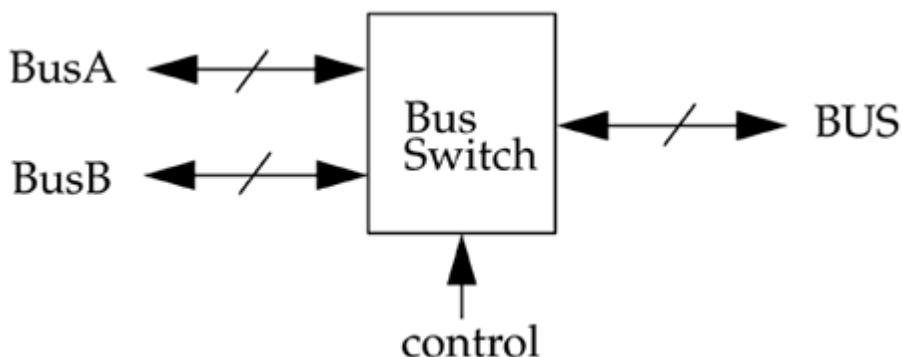
3:

Design the 1-bit full-adder shown below using the xor, and, and or gates built in Exercise 1 and Exercise 2 above. Apply stimulus and test the design.



4:

Design a 4-bit bidirectional bus switch that has two buses, BusA and BusB, on one side and a single bus, BUS, on the other side. A 1-bit control signal is used for switching. BusA and BUS are connected if control = 1. BusB and BUS are connected if control = 0. (Hint: Use the switches tranif0 and tranif1.) Apply stimulus and test the design.



5:

Instantiate switches with the following delay specifications. Use your own input/output port names.

a.

- A pmos switch with rise = 2 and fall = 3.
- b.

- b. An nmos switch with rise = 4, fall = 6, turn-off = 5
- c.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## Chapter 12. User-Defined Primitives

Verilog provides a standard set of primitives, such as and, nand, or, nor, and not, as a part of the language. These are also commonly known as built-in primitives. However, designers occasionally like to use their own custom-built primitives when developing a design. Verilog provides the ability to define User-Defined Primitives (UDP). These primitives are self-contained and do not instantiate other modules or primitives. UDPs are instantiated exactly like gate-level primitives.

There are two types of UDPs: combinational and sequential.

- 
- Combinational UDPs are defined where the output is solely determined by a logical combination of the inputs. A good example is a 4-to-1 multiplexer.
- 
- Sequential UDPs take the value of the current inputs and the current output to determine the value of the next output. The value of the output is also the internal state of the UDP. Good examples of sequential UDPs are latches and flipflops.

### Learning Objectives

- 
- Understand UDP definition rules and parts of a UDP definition.
- 
- Define sequential and combinational UDPs.
- 
- Explain instantiation of UDPs.
- 
- Identify UDP shorthand symbols for more conciseness and better readability.
- 
- Describe the guidelines for UDP design.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



ABC Amber CHM Converter Trial version

Please register to remove this banner.

<http://www.thebeatlesforever.com/processtext/abcchm.html>

[ Team LiB ]

[PREVIOUS](#)  [NEXT](#)

## 12.1 UDP basics

In this section, we describe parts of a UDP definition and rules for UDPs.

### 12.1.1 Parts of UDP Definition

[Figure 12-1](#) shows the distinct parts of a basic UDP definition in pseudo syntax form. For details, see the formal syntax definition described in Appendix , Formal Syntax Definition.

#### Figure 12-1 Parts of UDP Definition

```
//UDP name and terminal list
primitive <udp_name> (
<output_terminal_name>(only one allowed)
<input_terminal_names> );

//Terminal declarations
output <output_terminal_name>;
input <input_terminal_names>;
reg <output_terminal_name>;(optional; only for sequential
                           UDP)

// UDP initialization (optional; only for sequential UDP
initial <output_terminal_name> = <value>;

//UDP state table
table
  <table entries>
endtable

//End of UDP definition
endprimitive
```

A UDP definition starts with the keyword primitive. The primitive name, output terminal, and input terminals are specified. Terminals are declared as output or input in the terminal declarations section. For a sequential UDP, the output terminal is declared as a reg. For sequential UDPs, there is an optional initial statement that initializes the output terminal of the UDP. The UDP state table is most important part of the UDP. It begins with the keyword table and ends with the keyword endtable. The table defines how the output will be computed from the inputs and current state. The table is modeled as a lookup table. and the table entries resemble entries in a logic truth table. Primitive definition is completed with the keyword endprimitive.

### 12.1.2 UDP Rules

UDP definitions follow certain rules:

1.

1. UDPs can take only scalar input terminals (1 bit). Multiple input terminals are permitted

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## 12.2 Combinational UDPs

Combinational UDPs take the inputs and produce the output value by looking up the corresponding entry in the state table.

### 12.2.1 Combinational UDP Definition

The state table is the most important part of the UDP definition. The best way to explain a state table is to take the example of an and gate modeled as a UDP. Instead of using the and gate provided by Verilog, let us define our own and gate primitive and call it `udp_and`.

#### Example 12-1 Primitive `udp_and`

```
//Primitive name and terminal list
primitive udp_and(out, a, b);

//Declarations
output out; //must not be declared as reg for combinational UDP
input a, b; //declarations for inputs.

//State table definition; starts with keyword table
table
    //The following comment is for readability only
    //Input entries of the state table must be in the
    //same order as the input terminal list.
    // a   b   :   out;
    0   0   :   0;
    0   1   :   0;
    1   0   :   0;
    1   1   :   1;

endtable //end state table definition

endprimitive //end of udp_and definition
```

Compare parts of `udp_and` defined above with the parts discussed in [Figure 12-1](#). The missing parts are that the output is not declared as `reg` and the initial statement is absent. Note that these missing parts are used only for sequential UDPs, which are discussed later in the chapter.

ANSI C style declarations for UDPs are also supported. This style allows the declarations of a primitive port to be combined with the port list. [Example 12-2](#) shows an example of an ANSI C style UDP declaration.

#### Example 12-2 ANSI C Style UDP Declaration

```
//Primitive name and terminal list
primitive udp_and(output out,
                  input a,
                  input b);
-- 
-- 
endprimitive //end of udp_and definition
```

### 12.2.2 State Table Entries

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## 12.3 Sequential UDPs

Sequential UDPs differ from combinational UDPs in their definition and behavior. Sequential UDPs have the following differences:

- 
- The output of a sequential UDP is always declared as a reg.
- 
- An initial statement can be used to initialize output of sequential UDPs.
- 
- The format of a state table entry is slightly different.
- `<input1> <input2> .... <inputN> : <current_state> : <next_state>;`
- 
- There are three sections in a state table entry: inputs, current state, and next state. The three sections are separated by a colon (:) symbol.
- 
- The input specification of state table entries can be in terms of input levels or edge transitions.
- 
- The current state is the current value of the output register.
- 
- The next state is computed based on inputs and the current state. The next state becomes the new value of the output register.
- 
- All possible combinations of inputs must be specified to avoid unknown output values.

If a sequential UDP is sensitive to input levels, it is called a level-sensitive sequential UDP. If a sequential UDP is sensitive to edge transitions on inputs, it is called an edge-sensitive sequential UDP.

### 12.3.1 Level-Sensitive Sequential UDPs

Level-sensitive UDPs change state based on input levels. Latches are the most common example of level-sensitive UDPs. A simple latch with clear is shown in [Figure 12-3](#).

**Figure 12-3. Level-Sensitive Latch with clear**



[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## 12.4 UDP Table Shorthand Symbols

Shorthand symbols for levels and edge transitions are provided so UDP tables can be written in a concise manner. We already discussed the symbols ? and -. A summary of all shorthand symbols and their meaning is shown in [Table 12-1](#).

Table 12-1. UDP Table Shorthand Symbols

<b>Shorthand Symbols</b>	<b>Meaning</b>	<b>Explanation</b>
?	0, 1, x	Cannot be specified in an output field
b	0, 1	Cannot be specified in an output field
-	No change in state value	Can be specified only in output field of a sequential UDP
r	(01)	Rising edge of signal
f	(10)	Falling edge of signal
p	(01), (0x) or (x1)	Potential rising edge of signal
n	(10), (1x) or (x0)	Potential falling edge of signal
*	(??)	Any value change in signal

Using the shorthand symbols, we can rewrite the table entries in Example 12-9 on page 263 as follows.

```

table
// d clock clear : q : q+  ;

? ? 1 : ? : 0 ; //output = 0 if clear = 1
? ? f : ? : - ; //ignore negative transition of clear

1 f 0 : ? : 1 ; //latch data on negative transition of
0 f 0 : ? : 0 ; //clock

? (1x) 0 : ? : - ; //hold q if clock transitions to unknown
                     //state

? p 0 : ? : - ; //ignore positive transitions of clock

* ? 0 : ? : - ; //ignore any change in d when
                  //clock is steady

endtable

```

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## 12.5 Guidelines for UDP Design

When designing a functional block, it is important to decide whether to model it as a module or as a user-defined primitive. Here are some guidelines used to make that decision.

- 
- UDPs model functionality only. They do not model timing or process technology (such as CMOS, TTL, ECL). The primary purpose of a UDP is to define in a simple and concise form the functional portion of a block. A module is always used to model a complete block that has timing and process technology.
- 
- A block can modeled as a UDP only if it has exactly one output terminal. If the block to be designed has more than one output, it has to be modeled as a module.
- 
- The limit on the maximum number of inputs of a UDP is specific to the Verilog simulator being used. However, Verilog simulators are required to allow a minimum of 9 inputs for sequential UDPs and 10 for combinational UDPs.
- 
- A UDP is typically implemented as a lookup table in memory. As the number of inputs increases, the number of table entries grows exponentially. Thus, the memory requirement for a UDP grows exponentially in relation to the number of inputs. It is not advisable to design a block with a large number of inputs as a UDP.
- 
- UDPs are not always the appropriate method to design a block. Sometimes it is easier to design blocks as a module. For example, it is not advisable to design an 8-to-1 multiplexer as a UDP because of the large number of table entries. Instead, the data flow or behavioral representation would be much simpler. It is important to consider complexity trade-offs to decide whether to use UDP to represent a block.

There are also some guidelines for writing the UDP state table.

- 
- The UDP state table should be specified as completely as possible. All possible input combinations for which the output is known should be covered. If a certain combination of inputs is not specified, the default output value for that combination will be x. This feature is used frequently in commercial libraries to reduce the number of table entries.
- 
- Shorthand symbols should be used to combine table entries wherever possible. Shorthand symbols make the UDP description more concise. However, the Verilog simulator may internally expand the table entries. Thus, there is no memory requirement reduction by using shorthand symbols.
- 
- Level-sensitive entries take precedence over edge sensitive entries. If edge-sensitive and level-sensitive entries both occur in the same row, the output is determined by the level sensitivity.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 12.6 Summary

We discussed the following aspects of Verilog in this chapter:

- 
- User-defined primitives (UDP) are used to define custom Verilog primitives by the use of lookup tables. UDPs offer a convenient way to design certain functional blocks.
- 
- UDPs can have only one output terminal. UDPs are defined at the same level as modules. UDPs are instantiated exactly like gate primitives. A state table is the most important component of UDP specification.
- 
- UDPs can be combinational or sequential. Sequential UDPs can be edge- or level-sensitive.
- 
- Combinational UDPs are used to describe combinational circuits where the output is purely a logical combination of the inputs.
- 
- Sequential UDPs are used to define blocks with timing controls. Blocks such as latches or flipflops can be described with sequential UDPs. Sequential UDPs are modeled like state machines. There is a present state and a next state. The next state is also the output of the UDP. Edge- and level-sensitive descriptions can be mixed.
- 
- Shorthand symbols are provided to make UDP state table entries more concise. Shorthand notation should be used wherever possible.
- 
- It is important to decide whether a functional block should be described as a UDP or as a module. Memory requirements and complexity trade-offs must be considered.

[ Team LiB ]

PREVIOUS | NEXT

[ Team LiB ]

PREVIOUS | NEXT

## 12.7 Exercises

1:

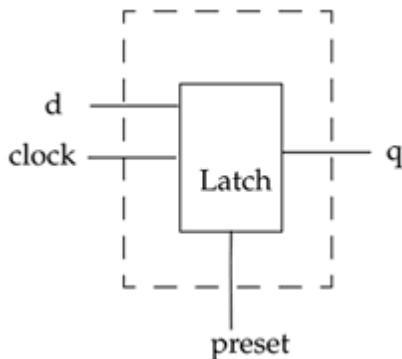
Design a 2-to-1 multiplexer by using UDP. The select signal is s, inputs are i0, i1, and the output is out. If the select signal s = x, the output out is always 0. If s = 0, then out = i0. If s = 1, then out = i1.

2:

Write the truth table for the boolean function  $Y = (A \& B) | (C \wedge D)$ . Define a UDP that implements this boolean function. Assume that the inputs will never take the value x.

3:

Define a level-sensitive latch with a preset signal. Inputs are d, clock, and preset. Output is q. If clock = 0, then q = d. If clock = 1 or x, then q is unchanged. If preset = 1, then q = 1. If preset = 0, then q is decided by clock and d signals. If preset = x, then q = x.

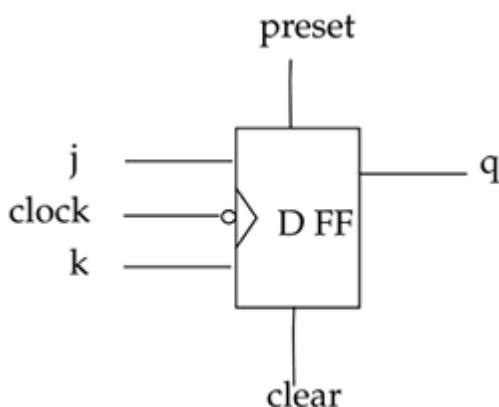


4:

Define a positive edge-triggered D-flipflop with clear as a UDP. Signal clear is active low. Use [Example 12-9](#) on page 263 as a guideline. Use shorthand notation wherever possible.

5:

Define a negative edge-triggered JK flipflop, jk\_ff with asynchronous preset and clear as a UDP. q = 1 when preset = 1 and q = 0 when clear = 1.



The table for a JK flipflop is shown below.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

# Chapter 13. Programming Language Interface

Verilog provides the set of standard system tasks and functions defined in [Appendix C](#), List of Keywords, System Tasks, and Compiler Directives. However, designers frequently need to customize the capability of the Verilog language by defining their own system tasks and functions. To do this, the designers need to interact with the internal representation of the design and the simulation environment in the Verilog simulator. The Programming Language Interface (PLI) provides a set of interface routines to read internal data representation, write to internal data representation, and extract information about the simulation environment. User-defined system tasks and functions can be created with this predefined set of PLI interface routines.

Verilog Programming Language Interface is a very broad area of study. Thus, only the basics of Verilog PLI are covered in this chapter. Designers should consult the IEEE Standard Verilog Hardware Description Language document for complete details of the PLI.

There are three generations of the Verilog PLI.

- 1.
  - 2.
  - 3.
1. Task/Function (tf\_) routines make up the first generation PLI. These routines are primarily used for operations involving user-defined task/function arguments, utility functions, callback mechanism, and writing data to output devices.
  2. Access (acc\_) routines make up the second-generation PLI. These routines provide object-oriented access directly into a Verilog HDL structural description. These routines can be used to access and modify a wide variety of objects in the Verilog HDL description.
  3. Verilog Procedural Interface (vpi\_) routines make up the third-generation PLI. These routines are a superset of the functionality of acc\_ and tf\_ routines.

For the sake of simplicity, we will discuss only acc\_ and tf\_ routines in this chapter.

## Learning Objectives

- Explain how PLI routines are used in a Verilog simulation.
- Describe the uses of the PLI.
- Define user-defined system tasks and functions and user-defined C routines.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 13.1 Uses of PLI

PLI provides a powerful capability to extend the Verilog language by allowing users to define their own utilities to access the internal design representation. PLI has various applications.

- 
- PLI can be used to define additional system tasks and functions. Typical examples are monitoring tasks, stimulus tasks, debugging tasks, and complex operations that cannot be implemented with standard Verilog constructs.
- 
- Application software like translators and delay calculators can be written with PLI.
- 
- PLI can be used to extract design information such as hierarchy, connectivity, fanout, and number of logic elements of a certain type.
- 
- PLI can be used to write special-purpose or customized output display routines. Waveform viewers can use this file to generate waveforms, logic connectivity, source level browsers, and hierarchy information.
- 
- Routines that provide stimulus to the simulation can be written with PLI. The stimulus could be automatically generated or translated from some other form of stimulus.
- 
- General Verilog-based application software can be written with PLI routines. This software will work with all Verilog simulators because of the uniform access provided by the PLI interface.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 13.2 Linking and Invocation of PLI Tasks

Designers can write their own user-defined system tasks by using PLI library routines. However, the Verilog simulator must know about the existence of the user-defined system task and its corresponding user-defined C function. This is done by linking the user-defined system task into the Verilog simulator.

To understand the process, let us consider the example of a simple system task \$hello\_verilog. When invoked, the task simply prints out a message "Hello Verilog World". First, the C routine that implements the task must be defined with PLI library routines. The C routine hello\_verilog in the file hello\_verilog.c is shown below.

```
#include "veriuser.h" /*include the file provided in release dir */

int hello_verilog()
{
    io_printf("Hello Verilog World\n");
}
```

The hello\_verilog routine is fairly straightforward. The io\_printf is a PLI library routine that works exactly like printf.

The following sections show the steps involved in defining and using the new \$hello\_verilog system task.

### 13.2.1 Linking PLI Tasks

Whenever the task \$hello\_verilog is invoked in the Verilog code, the C routine hello\_verilog must be executed. The simulator needs to be aware that a new system task called \$hello\_verilog exists and is linked to the C routine hello\_verilog. This process is called linking the PLI routines into the Verilog simulator. Different simulators provide different mechanisms to link PLI routines. Also, though the exact mechanics of the linking process might be different for simulators, the fundamentals of the linking process remain the same. For details, refer to the latest reference manuals available with your simulator.

At the end of the linking step, a special binary executable containing the new \$hello\_verilog system task is created. For example, instead of the usual simulator binary executable, a new binary executable hverilog is produced. To simulate, run hverilog instead of your usual simulator executable file.

### 13.2.2 Invoking PLI Tasks

Once the user-defined task has been linked into the Verilog simulator, it can be invoked like any Verilog system task by the keyword \$hello\_verilog. A Verilog module hello\_top, which calls the task \$hello\_verilog, is defined in file hello.v as shown below.

```
module hello_top;

initial
    $hello_verilog; //Invoke the user-defined task $hello_verilog

endmodule
```

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

### 13.3 Internal Data Representation

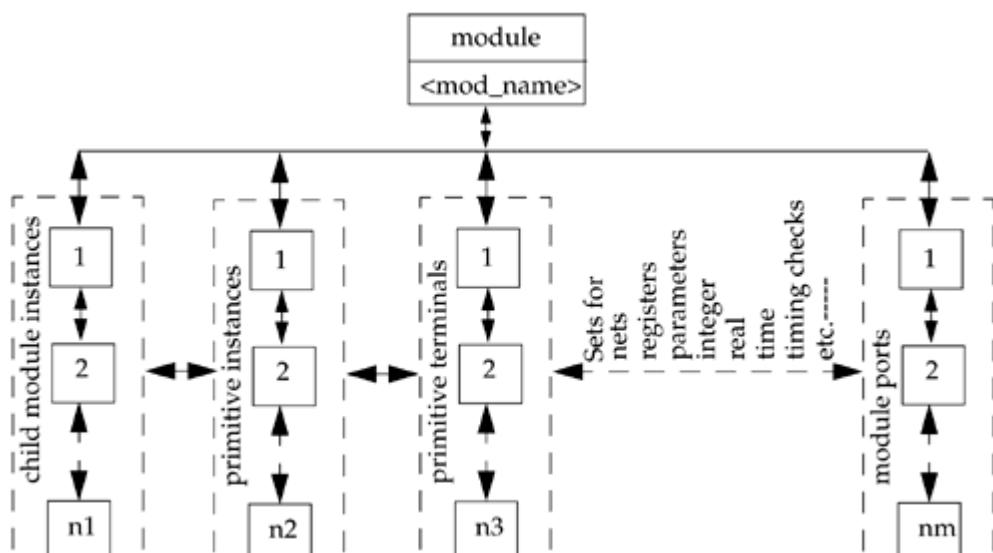
Before we understand how to use PLI library routines, it is first necessary to describe how a design is viewed internally in the simulator. Each module is viewed as a collection of object types. Object types are elements defined in Verilog, such as:

- 
- Module instances, module ports, module pin-to-pin paths, and intermodule paths
- 
- Top-level modules
- 
- Primitive instances, primitive terminals
- 
- Nets, registers, parameters, specparams
- 
- Integer, time, and real variables
- 
- Timing checks
- 
- Named events

Each object type has a corresponding set that identifies all objects of that type in the module. Sets of all object types are interconnected.

A conceptual internal representation of a module is shown in [Figure 13-3](#).

**Figure 13-3. Conceptual Internal Representation a Module**



[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

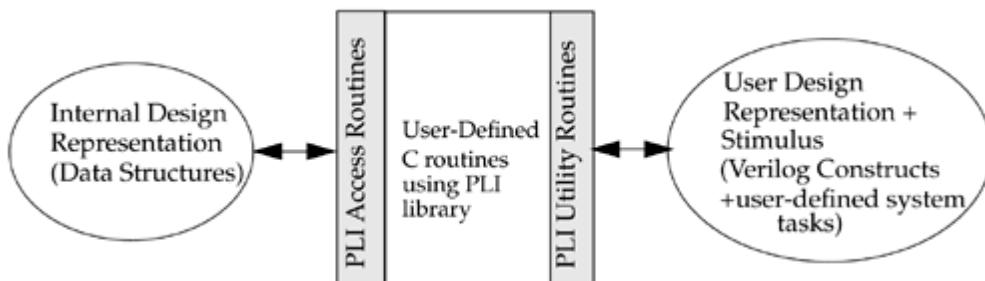
## 13.4 PLI Library Routines

PLI library routines provide a standard interface to the internal data representation of the design. The user-defined C routines for user-defined system tasks are written by using PLI library routines. In the example in [Section 13.2](#), Linking and Invocation of PLI Tasks, \$hello\_verilog is the user-defined system task, hello\_verilog is the user-defined C routine, and io\_printf is a PLI library routine.

There are two broad classes of PLI library routines: access routines and utility routines. (Note that vpi\_routines are a superset of access and utility routines and are not discussed in this book.)

Access routines provide access to information about the internal data representation; they allow the user C routine to traverse the data structure and extract information about the design. Utility routines are mainly used for passing data across the Verilog/Programming Language Boundary and for miscellaneous housekeeping functions. [Figure 13-6](#) shows the role of access and utility routines in PLI.

**Figure 13-6. Role of Access and Utility Routines**



A complete list of PLI library routines is provided in [Appendix B](#), List of PLI Routines. The function and usage of each routine are also specified.

### 13.4.1 Access Routines

Access routines are also popularly called acc routines. Access routines can do the following:

- 
- Read information about a particular object from the internal data representation
- 
- Write information about a particular object into the internal data representation

We will discuss only reading of information from the design. Information about modifying internal design representation can be found in the Programming Language Interface (PLI) Manual. However, reading of information is adequate for most practical purposes.

Access routines can read information about objects in the design. Objects can be one of the following types:

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## 13.5 Summary

In this chapter, we described the Programming Language Interface (PLI) for Verilog. The following aspects were discussed:

- 
- PLI Interface provides a set of C interface routines to read, write, and extract information about the internal data structures of the design. Designers can write their own system tasks to do various useful functions.
- 
- PLI Interface can be used for monitors, debuggers, translators, delay calculators, automatic stimulus generators, dump file generators, and other useful utilities.
- 
- A user-defined system task is implemented with a corresponding user-defined C routine. The C routine uses PLI library calls.
- 
- The process of informing the simulator that a new user-defined system task is attached to a corresponding user C routine is called linking. Different simulators handle the linking process differently.
- 
- User-defined system tasks are invoked like standard Verilog system tasks, e.g., \$hello\_verilog(); . The corresponding user C routine hello\_verilog is executed whenever the task is invoked.
- 
- A design is represented internally in a Verilog simulator as a big data structure with sets for objects. PLI library routines allow access to the internal data structures.
- 
- Access (acc) routines and utility (tf) routines are two types of PLI library routines.
- 
- Utility routines represent the first generation of Verilog PLI. Utility routines are used to pass data back and forth across the boundary of user C routines and the original Verilog design. Utility routines start with the prefix tf\_. Utility routines do not interact with object handles.
- 
- Access routines represent the second generation of Verilog PLI. Access routines can read and write information about a particular object from/to the design. Access routines start with the prefix acc\_. Access routines are used primarily across the boundary of user C routines and internal data representation. Access routines interact with object handles.
- 
- Value change link (VCL) is a special category of access routines that allow monitoring of objects in a design. A consumer routine is executed whenever the monitored object value changes.
-

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 13.6 Exercises

Refer to [Appendix B](#), List of PLI Routines and IEEE Standard Verilog Hardware Description Language document, for a list of PLI access and utility routines, their function, and usage. You will need to use some PLI library calls that were not discussed in this chapter.

**1:**

Write a user-defined system task, \$get\_in\_ports, that gets full hierarchical names of only the input ports of a module instance. Hierarchical module instance name is the input to the task (Hint: Use the C routine in [Example 13-2](#) as a reference). Link the task into the Verilog simulator. Find the input ports of the 1-bit full adder defined in [Example 5-7](#) on page 75.

**2:**

Write a user-defined system task, \$count\_and\_gates, which counts the number of and gate primitives in a module instance. Hierarchical module instance name is the input to the task. Use this task to count the number of and gates in the 4-to-1 multiplexer in [Example 5-5](#).

**3:**

Create a user-defined system task, \$monitor\_mod\_output, that finds out all the output signals of a module instance and adds them to a monitoring list. The line "Output signal has changed" should appear whenever any output signal of the module changes value. (Hint: Use VCL routines.) Use the 2-to-1 multiplexer in [Example 13-1](#). Add output signals to the monitoring list by using \$monitor\_mod\_output. Check results by applying stimulus.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[  PREVIOUS | NEXT  ]

# Chapter 14. Logic Synthesis with Verilog HDL

Advances in logic synthesis have pushed HDLs into the forefront of digital design technology. Logic synthesis tools have cut design cycle times significantly. Designers can design at a high level of abstraction and thus reduce design time. In this chapter, we discuss logic synthesis with Verilog HDL. Synopsys synthesis products were used for the examples in this chapter, and results for individual examples may vary with synthesis tools. However, the concepts discussed in this chapter are general enough to be applied to any logic synthesis tool.<sup>[1]</sup> This chapter is intended to give the reader a basic understanding of the mechanics and issues involved in logic synthesis. It is not intended to be comprehensive material on logic synthesis. Detailed knowledge of logic synthesis can be obtained from reference manuals, logic synthesis books, and by attending training classes.

[1] Many EDA vendors now offer logic synthesis tools. Please see the reference documentation provided with your logic synthesis tool for details on how to synthesize RTL to gates. There may be minor variations from the material presented in this chapter.

## Learning Objectives

- 
- Define logic synthesis and explain the benefits of logic synthesis.
- 
- Identify Verilog HDL constructs and operators accepted in logic synthesis. Understand how the logic synthesis tool interprets these constructs.
- 
- Explain a typical design flow, using logic synthesis. Describe the components in the logic synthesis-based design flow.
- 
- Describe verification of the gate-level netlist produced by logic synthesis.
- 
- Understand techniques for writing efficient RTL descriptions.
- 
- Describe partitioning techniques to help logic synthesis provide the optimal gate-level netlist.
- 
- Design combinational and sequential circuits, using logic synthesis.



**ABC Amber CHM Converter Trial version**

Please register to remove this banner.

<http://www.thebeatlesforever.com/processtext/abcchm.html>

[ [Team LiB](#) ]

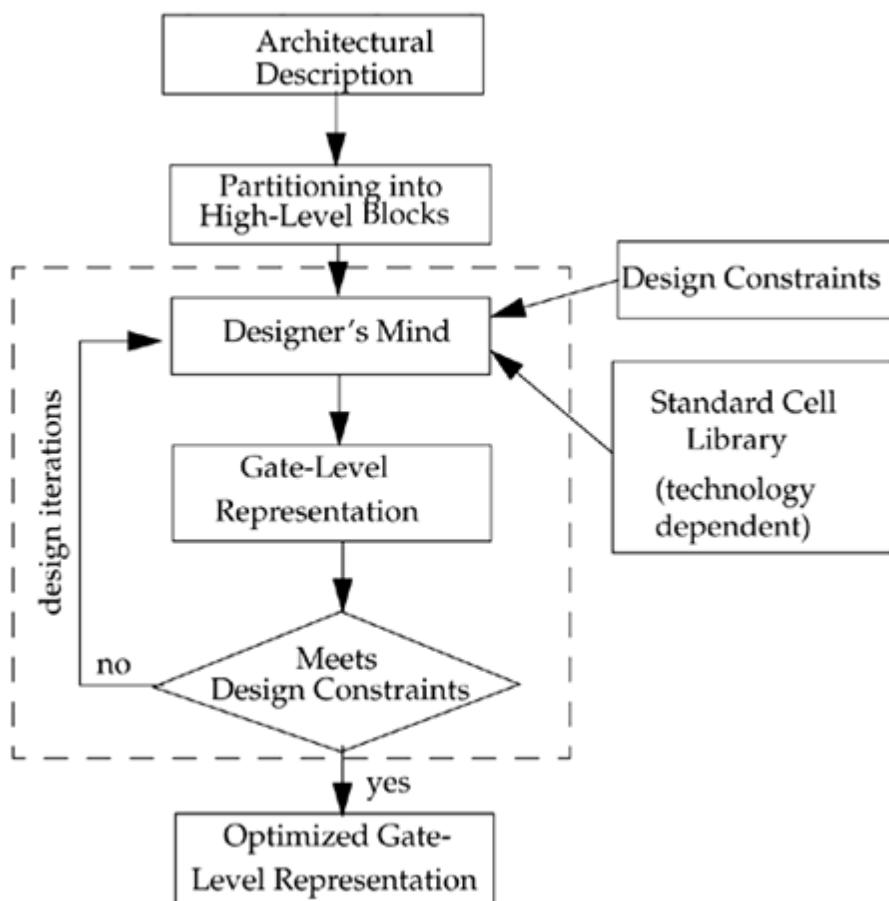
[ [◀ PREVIOUS](#) ] [NEXT ▶](#) ]

## 14.1 What Is Logic Synthesis?

Simply speaking, logic synthesis is the process of converting a high-level description of the design into an optimized gate-level representation, given a standard cell library and certain design constraints. A standard cell library can have simple cells, such as basic logic gates like and, or, and nor, or macro cells, such as adders, muxes, and special flip-flops. A standard cell library is also known as the technology library. It is discussed in detail later in this chapter.

Logic synthesis always existed even in the days of schematic gate-level design, but it was always done inside the designer's mind. The designer would first understand the architectural description. Then he would consider design constraints such as timing, area, testability, and power. The designer would partition the design into high-level blocks, draw them on a piece of paper or a computer terminal, and describe the functionality of the circuit. This was the high-level description. Finally, each block would be implemented on a hand-drawn schematic, using the cells available in the standard cell library. The last step was the most complex process in the design flow and required several time-consuming design iterations before an optimized gate-level representation that met all design constraints was obtained. Thus, the designer's mind was used as the logic synthesis tool, as illustrated in [Figure 14-1](#).

**Figure 14-1. Designer's Mind as the Logic Synthesis Tool**



The advent of computer-aided logic synthesis tools has automated the process of converting the high-level description to logic gates. Instead of trying to perform logic synthesis in their minds, designers can now concentrate on the architectural trade-offs, high-level description of the design, accurate design constraints, and optimization of cells in the standard cell library. These are fed to the computer-aided logic synthesis tool, which performs several iterations internally and generates the optimized gate-level description. All this is done using the high-level description, which is a major improvement over the traditional manual process.

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## 14.2 Impact of Logic Synthesis

Logic synthesis has revolutionized the digital design industry by significantly improving productivity and by reducing design cycle time. Before the days of automated logic synthesis, when designs were converted to gates manually, the design process had the following limitations:

- 
- For large designs, manual conversion was prone to human error. A small gate missed somewhere could mean redesign of entire blocks.
- 
- The designer could never be sure that the design constraints were going to be met until the gate-level implementation was completed and tested.
- 
- A significant portion of the design cycle was dominated by the time taken to convert a high-level design into gates.
- 
- If the gate-level design did not meet requirements, the turnaround time for redesign of blocks was very high.
- 
- What-if scenarios were hard to verify. For example, the designer designed a block in gates that could run at a cycle time of 20 ns. If the designer wanted to find out whether the circuit could be optimized to run faster at 15 ns, the entire block had to be redesigned. Thus, redesign was needed to verify what-if scenarios.
- 
- Each designer would implement design blocks differently. There was little consistency in design styles. For large designs, this could mean that smaller blocks were optimized, but the overall design was not optimal.
- 
- If a bug was found in the final, gate-level design, this would sometimes require redesign of thousands of gates.
- 
- Timing, area, and power dissipation in library cells are fabrication-technology specific. Thus if the company changed the IC fabrication vendor after the gate-level design was complete, this would mean redesign of the entire circuit and a possible change in design methodology.
- 
- Design reuse was not possible. Designs were technology-specific, hard to port, and very difficult to reuse.

Automated logic synthesis tools addressed these problems as follows:

- 
-

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## 14.3 Verilog HDL Synthesis

For the purpose of logic synthesis, designs are currently written in an HDL at a register transfer level (RTL). The term RTL is used for an HDL description style that utilizes a combination of data flow and behavioral constructs. Logic synthesis tools take the register transfer-level HDL description and convert it to an optimized gate-level netlist. Verilog and VHDL are the two most popular HDLs used to describe the functionality at the RTL level. In this chapter, we discuss RTL-based logic synthesis with Verilog HDL. Behavioral synthesis tools that convert a behavioral description into an RTL description are slowly evolving, but RTL-based synthesis is currently the most popular design method. Thus, we will address only RTL-based synthesis in this chapter.

### 14.3.1 Verilog Constructs

Not all constructs can be used when writing a description for a logic synthesis tool. In general, any construct that is used to define a cycle-by-cycle RTL description is acceptable to the logic synthesis tool. A list of constructs that are typically accepted by logic synthesis tools is given in [Table 14-1](#). The capabilities of individual logic synthesis tools may vary. The constructs that are typically acceptable to logic synthesis tools are also shown.

Table 14-1. Verilog HDL Constructs for Logic Synthesis

Construct Type	Keyword or Description	Notes
ports	input, inout, output	
parameters	parameter	
module definition	module	
signals and variables	wire, reg, tri	Vectors are allowed
instantiation	module instances, primitive gate instances	E.g., mymux m1(out, i0, i1, s); E.g., nand (out, a, b);
functions and tasks	function, task	Timing constructs ignored
procedural	always, if, then, else, case, casex, casez	initial is not supported
procedural blocks	begin, end, named blocks, disable	Disabling of named blocks allowed
data flow	assign	Delay information is ignored
loops	for, while, forever	while and forever loops must

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

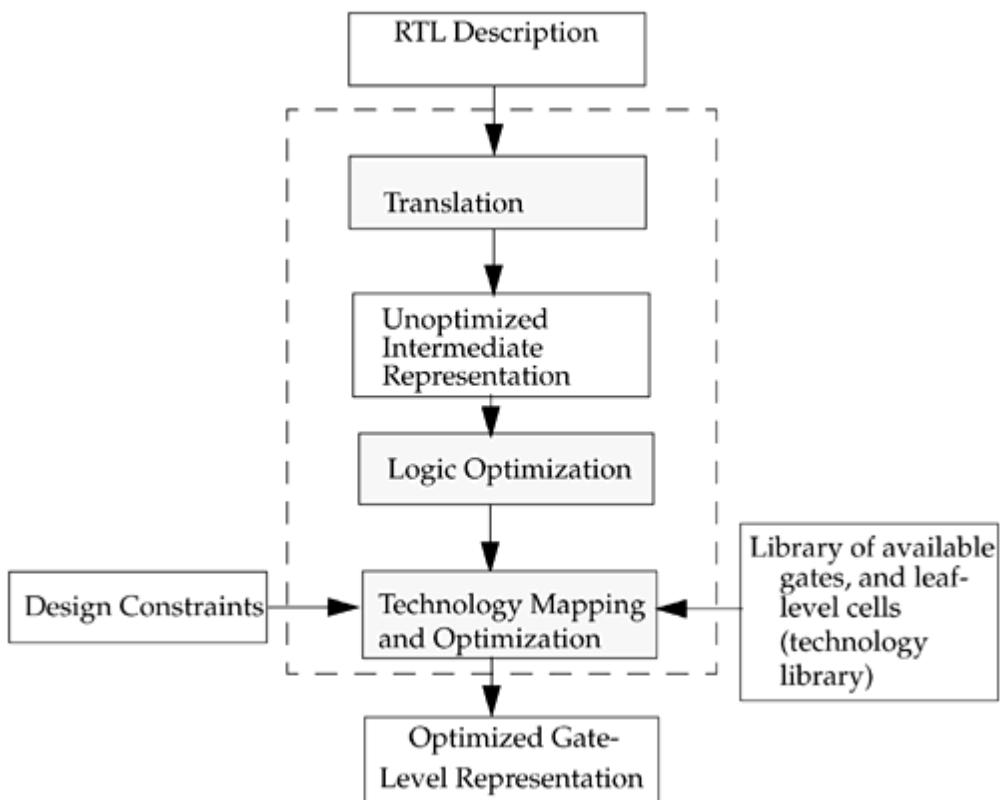
## 14.4 Synthesis Design Flow

Having understood how basic Verilog constructs are interpreted by the logic synthesis tool, let us now discuss the synthesis design flow from an RTL description to an optimized gate-level description.

### 14.4.1 RTL to Gates

To fully utilize the benefits of logic synthesis, the designer must first understand the flow from the high-level RTL description to a gate-level netlist. [Figure 14-4](#) explains that flow.

**Figure 14-4. Logic Synthesis Flow from RTL to Gates**



Let us discuss each component of the flow in detail.

#### RTL description

The designer describes the design at a high level by using RTL constructs. The designer spends time in functional verification to ensure that the RTL description functions correctly. After the functionality is verified, the RTL description is input to the logic synthesis tool.

#### Translation

The RTL description is converted by the logic synthesis tool to an unoptimized, intermediate, internal representation. This process is called translation. Translation is relatively simple and uses techniques similar to those discussed in [Section 14.3.3](#), Interpretation of a Few Verilog Constructs. The translator understands the basic primitives and operators in the Verilog RTL description. Design constraints such

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## 14.5 Verification of Gate-Level Netlist

The optimized gate-level netlist produced by the logic synthesis tool must be verified for functionality. Also, the synthesis tool may not always be able to meet both timing and area requirements if they are too stringent. Thus, a separate timing verification can be done on the gate-level netlist.

### 14.5.1 Functional Verification

Identical stimulus is run with the original RTL and synthesized gate-level descriptions of the design. The output is compared to find any mismatches. For the magnitude comparator, a sample stimulus file is shown below.

#### Example 14-3 Stimulus for Magnitude Comparator

```
module stimulus;

reg [3:0] A, B;
wire A_GT_B, A_LT_B, A_EQ_B;

//Instantiate the magnitude comparator
magnitude_comparator MC(A_GT_B, A_LT_B, A_EQ_B, A, B);

initial
$monitor($time, " A = %b, B = %b, A_GT_B = %b, A_LT_B = %b, A_EQ_B = %b",
A, B, A_GT_B, A_LT_B, A_EQ_B);

//stimulate the magnitude comparator.
initial
begin
  A = 4'b1010; B = 4'b1001;
  # 10 A = 4'b1110; B = 4'b1111;
  # 10 A = 4'b0000; B = 4'b0000;
  # 10 A = 4'b1000; B = 4'b1100;
  # 10 A = 4'b0110; B = 4'b1110;
  # 10 A = 4'b1110; B = 4'b1110;
end

endmodule
```

The same stimulus is applied to both the RTL description in [Example 14-1](#) and the synthesized gate-level description in [Example 14-2](#), and the simulation output is compared for mismatches. However, there is an additional consideration. The gate-level description is in terms of library cells VAND, VNAND, etc. Verilog simulators do not understand the meaning of these cells. Thus, to simulate the gate-level description, a simulation library, abc\_100.v, must be provided by ABC Inc. The simulation library must describe cells VAND, VNAND, etc., in terms of Verilog HDL primitives and, nand, etc. For example, the VAND cell will be defined in the simulation library as shown in [Example 14-4](#).

#### Example 14-4 Simulation Library

```
//Simulation Library abc_100.v. Extremely simple. No timing checks.

module VAND (out, in0, in1);
input in0;
input in1;
output out;
```

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 14.6 Modeling Tips for Logic Synthesis

The Verilog RTL design style used by the designer affects the final gate-level netlist produced by logic synthesis. Logic synthesis can produce efficient or inefficient gate-level netlists, based on the style of RTL descriptions. Hence, the designer must be aware of techniques used to write efficient circuit descriptions. In this section, we provide tips about modeling trade-offs, for the designer to write efficient, synthesizable Verilog descriptions.

### 14.6.1 Verilog Coding Style[2]

[2] Verilog coding style suggestions may vary slightly based on your logic synthesis tool. However, the suggestions included in this chapter are applicable to most cases. The IEEE Standard Verilog Hardware Description Language document also adds a new language construct called attribute. Attributes such as full\_case, parallel\_case, state\_variable, and optimize can be included in the Verilog HDL specification of the design. These attributes are used by synthesis tools to guide the synthesis process.

The style of the Verilog description greatly affects the final design. For logic synthesis, it is important to consider actual hardware implementation issues. The RTL specification should be as close to the desired structure as possible without sacrificing the benefits of a high level of abstraction. There is a trade-off between level of design abstraction and control over the structure of the logic synthesis output. Designing at a very high level of abstraction can cause logic with undesirable structure to be generated by the synthesis tool. Designing at a very low level (e.g., hand instantiation of each cell) causes the designer to lose the benefits of high-level design and technology independence. Also, a "good" style will vary among logic synthesis tools. However, many principles are common across logic synthesis tools. Listed below are some guidelines that the designer should consider while designing at the RTL level.

#### Use meaningful names for signals and variables

Names of signals and variables should be meaningful so that the code becomes self-commented and readable.

#### Avoid mixing positive and negative edge-triggered flipflops

Mixing positive and negative edge-triggered flipflops may introduce inverters and buffers into the clock tree. This is often undesirable because clock skews are introduced in the circuit.

#### Use basic building blocks vs. use continuous assign statements

Trade-offs exist between using basic building blocks versus using continuous assign statements in the RTL description. Continuous assign statements are a very concise way of representing the functionality and they generally do a good job of generating random logic. However, the final logic structure is not necessarily symmetrical. Instantiation of basic building blocks creates symmetric designs, and the logic synthesis tool is able to optimize smaller modules more effectively. However, instantiation of building blocks is not a concise way to describe the design; it inhibits retargeting to alternate technologies, and generally there is a degradation in simulator performance.

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## 14.7 Example of Sequential Circuit Synthesis

In [Section 14.4.2](#), An Example of RTL-to-Gates, we synthesized a combinational circuit. Let us now consider an example of sequential circuit synthesis. Specifically, we will design finite state machines.

### 14.7.1 Design Specification

A simple digital circuit is to be designed for the coin acceptor of an electronic newspaper vending machine.

- 
- Assume that the newspaper cost 15 cents. (Wow! Who gives that kind of a price any more? Well, let us assume that it is a special student edition!!)
- 
- The coin acceptor takes only nickels and dimes.
- 
- Exact change must be provided. The acceptor does not return extra money.
- 
- Valid combinations including order of coins are one nickel and one dime, three nickels, or one dime and one nickel. Two dimes are valid, but the acceptor does not return money.

This digital circuit can be designed by using the finite state machine approach.

### 14.7.2 Circuit Requirements

We must set some requirements for the digital circuit.

- 
- When each coin is inserted, a 2-bit signal  $\text{coin}[1:0]$  is sent to the digital circuit. The signal is asserted at the next negative edge of a global clock signal and stays up for exactly 1 clock cycle.
- 
- The output of the digital circuit is a single bit. Each time the total amount inserted is 15 cents or more, an output signal  $\text{newspaper}$  goes high for exactly one clock cycle and the vending machine door is released.
- 
- A reset signal can be used to reset the finite state machine. We assume synchronous reset.

### 14.7.3 Finite State Machine (FSM)

We can represent the functionality of the digital circuit with a finite state machine.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## 14.9 Exercises

1:

A 4-bit full adder with carry lookahead was defined in [Example 6-5](#) on page 109, using an RTL description. Synthesize the full adder, using a technology library available to you. Optimize for fastest timing. Apply identical stimulus to the RTL and the gate-level netlist and compare the output.

2:

A 1-bit full subtractor has three inputs x, y, and z (previous borrow) and two outputs D(difference) and B(borrow). The logic equations for D and B are as follows:

$$\begin{aligned}D &= x'y'z + x'yz' + xy'z' + xyz \\B &= x'y + x'z + yz\end{aligned}$$

Write the Verilog RTL description for the full subtractor. Synthesize the full subtractor, using any technology library available to you. Optimize for fastest timing. Apply identical stimulus to the RTL and the gate-level netlist and compare the output.

3:

Design a 3-to-8 decoder, using a Verilog RTL description. A 3-bit input a[2:0] is provided to the decoder. The output of the decoder is out[7:0]. The output bit indexed by a[2:0] gets the value 1, the other bits are 0. Synthesize the decoder, using any technology library available to you. Optimize for smallest area. Apply identical stimulus to the RTL and the gate-level netlist and compare the outputs.

4:

Write the Verilog RTL description for a 4-bit binary counter with synchronous reset that is active high. (Hint: Use always loop with the @(posedge clock) statement.) Synthesize the counter, using any technology library available to you. Optimize for smallest area. Apply identical stimulus to the RTL and the gate-level netlist and compare the outputs.

5:

Using a synchronous finite state machine approach, design a circuit that takes a single bit stream as an input at the pin in. An output pin

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

# Chapter 15. Advanced Verification Techniques

Verilog HDL was traditionally used both as a simulation modeling language and as a hardware description language. Verilog HDL was heavily used in verification and simulation for testbenches, test environments, simulation models, and architectural models. This approach worked well for smaller designs and simpler test environments.

As the average gate count for designs began to approach or exceed one million, verification soon became the main bottleneck in the design process. Design teams started spending 50-70% of their time in verifying designs rather than creating new ones.

Designers quickly realized that to verify complex designs, they needed to use tools that contained enhanced verification capabilities. They needed tools that could automate some of the tedious processes. Moreover, it was important to find bugs the very first time to avoid expensive chip re-spins.

To address these needs, a variety of verification methodologies and tools has emerged over the past few years. The latest addition to verification methodology is assertion-based verification. However, Verilog HDL remains the focal point in the design process. These new developments enhance the productivity of verifying Verilog HDL-based designs. This chapter gives the reader a basic understanding of these verification concepts that complement Verilog HDL.

## Learning Objectives

- 
- Define the components of a traditional verification flow.
- 
- Understand architectural modeling concepts.
- 
- Explain the use of high-level verification languages (HVLs).
- 
- Describe different techniques for effective simulation.
- 
- Explain the methods for analysis of simulation results.
- 
- Describe coverage techniques.
- 
- Understand assertion checking techniques.
-

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

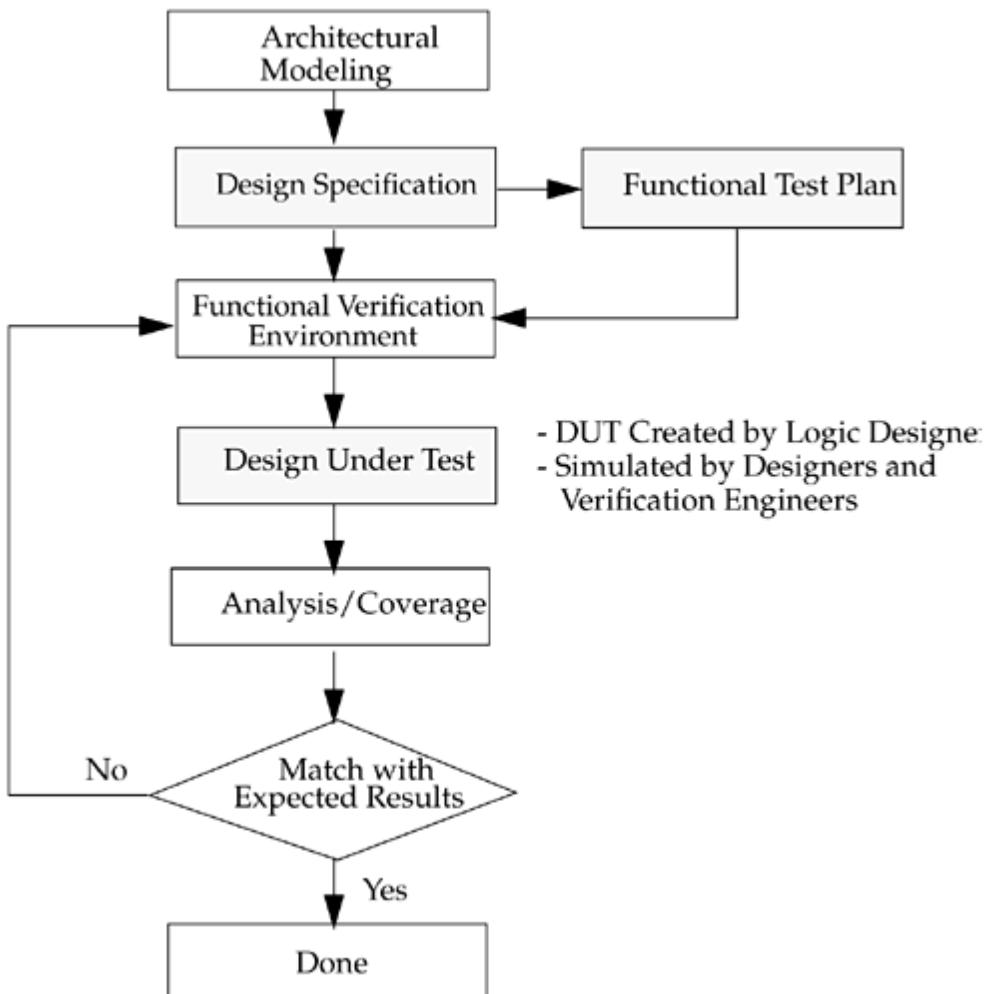
[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## 15.1 Traditional Verification Flow

A traditional verification flow consisting of certain standard components is illustrated in [Figure 15-1](#). This flow addresses only the verification perspective. It assumes that logic design is done separately.

**Figure 15-1. Traditional Verification Flow**



As shown in [Figure 15-1](#), the traditional verification flow consists of the following steps:

1.
  1. The chip architect first needs to create a design specification. In order to create a good specification, an analysis of architectural trade-offs has to be performed so that the best possible architecture can be chosen. This is usually done by simulating architectural models of the design. At the end of this step, the design specification is complete.
2.
  2. When the specification is ready, a functional test plan is created based on the design specification. This test plan forms the fundamental framework of the functional verification environment. Based on the test plan, test vectors are applied to the design-under-test (DUT), which is written in Verilog HDL. Functional test environments are needed to apply these test vectors. There are many tools available for generating and applying test vectors. These tools also allow the efficient creation of test environments.
- 3.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## 15.2 Assertion Checking

The traditional verification flow discussed in the previous section is a black box approach, i.e., verification relies only on the knowledge of the input and output behavior of the system.

Many other verification methodologies have evolved over the past few years to complement the traditional verification flow discussed in the previous section. In this section and the following sections, we explain some of these new verification methodologies that use the white box verification approach, i.e., knowledge of the internal structure of the design is needed for verification.

Assertion checking is a form of white box verification. It requires knowledge of internal structures of the design. The main purpose of assertion checkers is to improve observability.

Assertions are statements about a design's intended behavior. There are two types of assertions:

- 
- Temporal assertions ?they describe the timing relationship between signals.
- 
- Static assertions ?they describe a property of a signal that is always true or false.

Assertions may be used in the RTL code to describe the intended behavior of a piece of Verilog HDL code. The following are examples of such behavior:

- 
- An FSM state register should always be one-hot.
- 
- The full and empty flags of a FIFO should never be asserted at the same time.

Assertions can also be used to describe the behavior of the internal or external interface of a chip. For example, the acknowledge signal should always be asserted within five cycles of the request signal. Assertions may be verified in simulation or by using formal methods.

Assertions do not contribute to the element being designed; they are usually treated as comments for logic synthesis. Their sole purpose is to ensure consistency between the designer's intention and the design that is created. [Figure 15-7](#) shows the interfaces at which assertions could be placed in a FIFO-based design.

**Figure 15-7. Assertion Checks**



[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## 15.3 Formal Verification

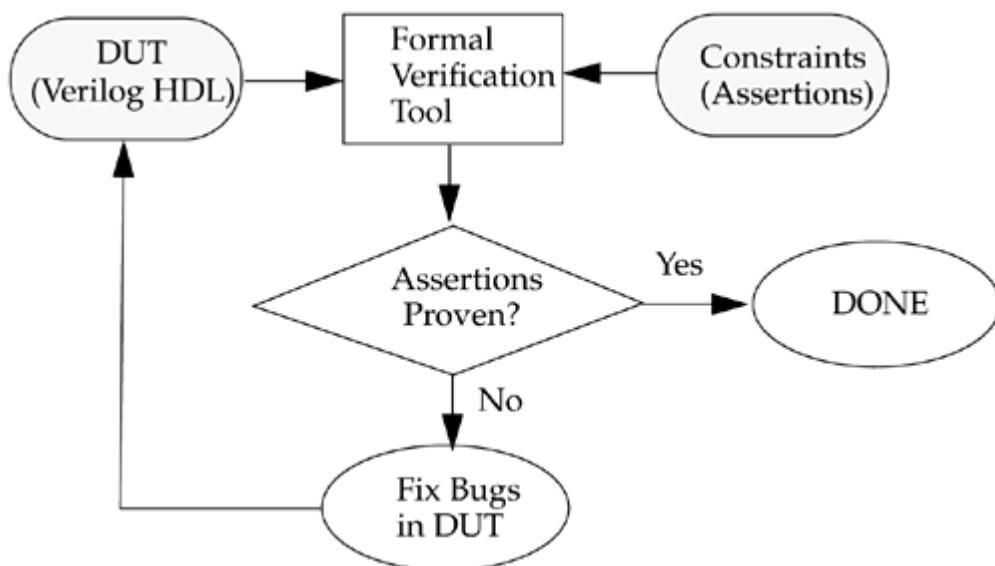
A well-known white-box approach is formal verification, in which mathematical techniques are used to prove an assertion or a property of the design. The property to be proven may be related to the chip's overall functional specification, or it may represent internal design behavior. Detailed knowledge of the behavior of design structures is often required to specify useful properties that are worth proving. Thus, one can prove the correctness of a design without doing simulations. Another application of formal verification is to prove that the architectural specifications of a design are sound before starting with the RTL implementation.

A formal verification tool proves a design property by exploring all possible ways to manipulate a design. All input changes must conform to the constraints for legal behavior. Assertions on interfaces act as constraints to the formal tool to constrain what is legal behavior on the inputs. Attempts are then made to prove the assertions in the RTL code to be true or false. If the constraints on the inputs are too loose, then the formal verification tool can generate counter-examples that rely on illegal input sequences that would not occur in the design. If the constraints are too tight, then the tool will not explore all possible behavior and will wrongly report the design as "proven."

[Figure 15-8](#) shows the verification flow with a formal verification tool. In the best case, the tool either proves a particular assertion absolutely or provides a counter-example to show the circumstances under which the assertion[\[4\]](#) is not met.

[4] Assertions are not used simply to increase observability. In formal verification, they are used as constraints. The formal verification tool explores the state space such that it proves the assertion absolutely or produces a counter-example. Thus, assertions also increase controllability, i.e., they control how the formal verification tool explores the state space to prove a property.

**Figure 15-8. Formal Verification Flow**



Since formal verification tools explore a design exhaustively, they can run only on designs that are limited in size. Typically, beyond 10,000 gates, absolute formal proofs become too hard and the tool blows up in terms of computation time and memory usage.

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## 15.4 Summary

- 
- A traditional verification flow contains a test-vector-based approach. An architectural model is developed to analyze design trade-offs. Once the design is finalized, it is verified using test vectors and simulation. Then the results are analyzed and coverage is measured. If the analysis meets the verification goals, the design is deemed verified.
- 
- Architectural modeling is used by architects for design exploration. The initial model of the design typically does not capture exact design behavior, except to the extent required for the initial design decisions. Architectural modeling languages are suitable for building architectural models.
- 
- Functional verification environments often contain test generators, input drivers, output receivers, data checkers, protocol checkers and coverage analyzers. High level verification languages (HVLs) can be used to effectively create and maintain these environments.
- 
- Software simulators are the most popular tools for simulating Verilog HDL designs. Hardware accelerators are used to accelerate simulation by a few orders of magnitude. Hardware emulators run in the megahertz range and are used to run software applications as if they were running on the real chip.
- 
- Waveforms and log files are the most common methods to analyze the output from a simulation. For effective analysis, it is important to build automatic data checker and protocol checker modules. If there is a violation of data value or protocol, the simulation is stopped immediately and an error message is displayed. A self-checking methodology allows the designer to run thousands of tests without having to analyze each test for correctness.
- 
- Toggle coverage, code coverage, and branch coverage are three types of structural coverage techniques. Functional coverage perceives the design from a system point of view. Functional coverage also provides finite state machine coverage, including states and state transitions. A combination of functional coverage and other coverage techniques is recommended.
- 
- Assertion checking is a form of white-box verification. It requires knowledge of the internal structures of the design. Assertion checking improves observability and verification efficiency. Assertion checks are placed by the designer at critical points in the design. If there is a failure at that point, the designer is notified.
- 
- Formal verification is a white-box approach in which mathematical techniques are used to exhaustively prove an assertion or a property of the design. Semi-formal verification combines the traditional verification flow using test vectors with the power and thoroughness of formal verification. Equivalence checking is an application of formal verification that examines the RTL representation of the design and checks to see if it matches the gate level and physical implementations of the design.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

# Part 3: Appendices

## [A Strength Modeling and Advanced Net Definitions](#)

Strength levels, signal contention, advanced net definitions.

## [B List of PLI Routines](#)

[A list of all access \(acc\) and utility \(tf\) PLI routines.](#)

## [C List of Keywords, System Tasks, and Compiler Directives](#)

A list of keywords, system tasks, and compiler directives in Verilog HDL.

## [D Formal Syntax Definition](#)

Formal syntax definition of the Verilog Hardware Description Language.

## [E Verilog Tidbits](#)

Origins of Verilog HDL, interpreted, compiled and native simulators, event-driven and oblivious simulation, cycle simulation, fault simulation, Verilog newsgroup, Verilog simulators, and Verilog-related Web sites.

## [F Verilog Examples](#)

Synthesizable model of a FIFO, behavioral model of a 256K X 16 DRAM.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

# Appendix A. Strength Modeling and Advanced Net Definitions

- [Section A.1. Strength Levels](#)
- [Section A.2. Signal Contention](#)
- [Section A.3. Advanced Net Types](#)

[ Team LiB ]



[ Team LiB ]

## A.1 Strength Levels

Verilog allows signals to have logic values and strength values. Logic values are 0, 1, x, and z. Logic strength values are used to resolve combinations of multiple signals and to represent behavior of actual hardware elements as accurately as possible. Several logic strengths are available. [Table A-1](#) shows the strength levels for signals. Driving strengths are used for signal values that are driven on a net. Storage strengths are used to model charge storage in trireg type nets, which are discussed later in this appendix.

Table A-1. Strength Levels

Strength Level	Abbreviation	Degree	Strength Type
supply1	Su1	strongest 1	driving
strong1	St1		driving
pull1	Pu1		driving
large1	La1		storage
weak1	We1		driving
medium1	Me1		storage
small1	Sm1		storage
highz1	HiZ1	weakest1	high impedance
highz	HiZ0	weakest0	high impedance
small0	Sm0		storage
medium0	Me0		storage
weak0	We0		driving
large0	La0		storage
pull0	Pu0		driving
strong0	St0		driving
supply0	Su0	strongest0	driving



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

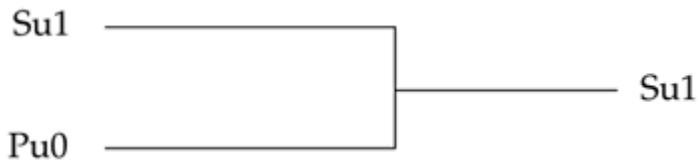
[ PREVIOUS ] [ NEXT ]

## A.2 Signal Contention

Logic strength values can be used to resolve signal contention on nets that have multiple drivers. There are many rules applicable to resolution of contention. However, two cases of interest that are most commonly used are described below.

### A.2.1 Multiple Signals with Same Value and Different Strength

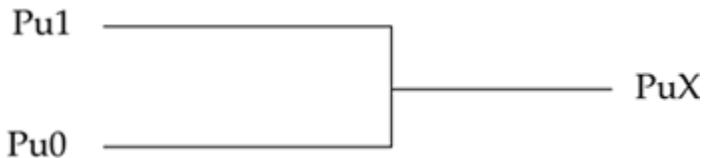
If two signals with same known value and different strength drive the same net, the signal with the higher strength wins.



In the example shown, supply strength is greater than pull. Hence, Su1 wins.

### A.2.2 Multiple Signals with Opposite Value and Same Strength

When two signals with opposite value and same strength combine, the resulting value is x.



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## A.3 Advanced Net Types

We discussed resolution of signal contention by using strength levels. There are other methods to resolve contention without using strength levels. Verilog provides advanced net declarations to model logic contention.

### A.3.1 tri

The keywords wire and tri have identical syntax and function. However, separate names are provided to indicate the purpose of the net. Keyword wire denotes nets with single drivers, and tri is denotes nets that have multiple drivers. A multiplexer, as defined below, uses the tri declaration.

```
module mux(out, a, b, control);
output out;
input a, b, control;
tri out;
wire a, b, control;

bufif0 b1(out, a, control); //drives a when control = 0; z otherwise
bufif1 b2(out, b, control); //drives b when control = 1; z otherwise

endmodule
```

The net is driven by b1 and b2 in a complementary manner. When b1 drives a, b2 is tristated; when b2 drives b, b1 is tristated. Thus, there is no logic contention. If there is contention on a tri net, it is resolved by using strength levels. If there are two signals of opposite values and same strength, the resulting value of the tri net is x.

### A.3.2 trireg

Keyword trireg is used to model nets having capacitance that stores values. The default strength for trireg nets is medium. Nets of type trireg are in one of two states:

- 
- Driven state?At least one driver drives a 0, 1, or x value on the net. The value is continuously stored in the trireg net. It takes the strength of the driver.
- 
- Capacitive state?All drivers on the net have high impedance (z) value. The net holds the last driven value. The strength is small, medium, or large (default is medium).

```
trireg (large) out;
wire a, control;

bufif1 (out, a, control); // net out gets value of a when control = 1;
//when control = 0, out retains last value of a
//instead of going to z. strength is large.
```

### A.3.3 tri0 and tri1

[\[ Team LiB \]](#)

PREVIOUS  NEXT

[\[ Team LiB \]](#)

PREVIOUS  NEXT

## Appendix B. List of PLI Routines

A list of PLI acc\_ and tf\_ routines is provided. VPI routines are not listed.[\[1\]](#) Names, the argument list, and a brief description of the routine are shown for each PLI routine. For details regarding the use of each PLI routine, refer to the IEEE Standard Verilog Hardware Description Language document.

[1] See the "IEEE Standard Verilog Hardware Description Language" document for details on VPI routines.

[\[ Team LiB \]](#)

PREVIOUS  NEXT

[\[ Team LiB \]](#)

PREVIOUS  NEXT

### B.1 Conventions

Conventions to be used for arguments are shown below.

Convention	Meaning
char *format	Pass formatted string
char *	Pass name of object as a string
underlined arguments	Arguments are optional
*	Pointer to the data type
.....	More arguments of the same type

[\[ Team LiB \]](#)

PREVIOUS  NEXT



**ABC Amber CHM Converter Trial version**

Please register to remove this banner.

<http://www.thebeatlesforever.com/processtext/abcchm.html>

[ [Team LiB](#) ]

[PREVIOUS](#)  [NEXT](#)

## B.2 Access Routines

Access routines are classified into five categories: handle, next, value change link, fetch, and modify routines.

### B.2.1 Handle Routines

Handle routines return handles to objects in the design. The names of handle routines always starts with the prefix acc\_handle\_. See [Table B-1](#).

Table B-1. Handle Routines

Return Type	Name	Argument List	Description
handle	acc_handle_by_name	(char *name, handle scope)	Object from name relative to scope.
handle	acc_handle_condition	(handle object)	Conditional expression for module path or timing check handle.
handle	acc_handle_conn	(handle terminal);	Get net connected to a primitive, module path, or timing check terminal.
handle	acc_handle_datapath	(handle modpath);	Get the handle to data path for an edge-sensitive module path.
handle	acc_handle_hiconn	(handle port);	Get hierarchically higher net connection to a module port.
handle	acc_handle_interactive_scope	( );	Get the handle to the current simulation interactive scope.
handle	acc_handle_loconn	(handle port);	Get hierarchically lower net connection to a module port.
handle	acc_handle_modpath	(handle module, char *src, char *dest); or	Get the handle to module path whose source and destination are specified. Module

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## B.3 Utility (tf\_) Routines

Utility (tf\_) routines are used to pass data in both directions across the Verilog/user C routine boundary. All the tf\_ routines assume that operations are being performed on current instances. Each tf\_ routine has a tf\_i counterpart in which the instance pointer where the operations take place has to be passed as an additional argument at the end of the argument list. See [Table B-7](#) through [B-16](#).

### B.3.1 Get Calling Task/Function Information

Table B-7. Get Calling Task/Function Information

Return Type	Name	Argument List	Description
char *	tf_getinstance	();	Get the pointer to the current instance of the simulation task or function that called the user's PLI application program.
char *	tf_mipname	();	Get the Verilog hierarchical path name of the simulation module containing the call to the user's PI application program.
char *	tf_ispname	()	Get the Verilog hierarchical path name of the scope containing the call to the user's PLI application program.

### B.3.2 Get Argument List Information

Table B-8. Get Argument List Information

Return Type	Name	Argument List	Description
int	tf_nump	();	Get the number of parameters in the argument list.
int	tf_typep	(int param_index#);	Get the type of a particular parameter in

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## Appendix C. List of Keywords, System Tasks, and Compiler Directives

- [Section C.1. Keywords](#)
- [Section C.2. System Tasks and Functions](#)
- [Section C.3. Compiler Directives](#)

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## C.1 Keywords

Keywords [1] are predefined, nonescaped identifiers that define the language constructs. An escaped identifier is never treated as a keyword. All keywords are defined in lowercase.

[1] From IEEE Std. 1364-2001. Copyright 2001 IEEE. All rights reserved.

The list is sorted in alphabetical order.

always	ifnone	rmmos
and	inmdir	rpmos
assign	include	rtran
automatic	initial	rtranif0
begin	inout	rtranif1
buf	input	scalared
bufif0	instance	showcancelled
bufif1	integer	signed
case	join	small
casex	large	specify
casez	liblist	specparam
cell	library	strong0
cmos	localparam	strong1
config	macromodule	supply0
deassign	medium	supply1
default	module	table

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## C.2 System Tasks and Functions

The following is a list of keywords frequently used by Verilog simulators for names of system tasks and functions. Not all system tasks and functions are explained in this book. For details, refer to the IEEE Standard Verilog Hardware Description Language document. This list is sorted in alphabetical order.

\$bitstoreal	\$countdrivers	\$display	\$fclose
\$fdisplay	\$fmonitor	\$fopen	\$fstrobe
\$fwrite	\$finish	\$getpattern	\$history
\$incsave	\$input	\$itor	\$key
\$list	\$log	\$monitor	\$monitoroff
\$monitoron	\$nokey		

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## C.3 Compiler Directives

The following is a list of keywords frequently used by Verilog simulators for specifying compiler directives. Only the most frequently used directives are discussed in the book. For details, refer to the IEEE Standard Verilog Hardware Description Language document. This list is sorted in alphabetical order.

'accelerate	'autoexpand_vectornets	'celldefine
'default_nettype	'define	'define
'else	'elsif	'endcelldefine
'endif	'endprotect	'endprotected
'expand_vectornets	'ifdef	'ifndef
'include	'noaccelerate	'noexpand_vectornets
'noremove_gatenames	'nounconnected_drive	'protect
'protected	'remove_gatenames	'remove_netnames
'resetall	'timescale	'unconnected_drive

[ Team LiB ]

PREVIOUS | NEXT

[ Team LiB ]

PREVIOUS | NEXT

## Appendix D. Formal Syntax Definition

This appendix contains the formal definition [1] of the Verilog-2001 standard in Backus-Naur Form (BNF). The formal definition contains a description of every possible usage of Verilog HDL. Therefore, it is very useful if there is a doubt on the usage of certain Verilog HDL syntax.

[1] From IEEE Std. 1364-2001. Copyright 2001 IEEE. All rights reserved.

Though the BNF may be hard to understand initially, the following summary may help the reader better understand the formal syntax definition:

1. Bold text represents literal words themselves (these are called terminals). Example: module.
- 2.
2. Non-bold text (possibly with underscores) represents syntactic categories (these are called non terminals). Example: port\_identifier.
- 3.
3. Syntactic categories are defined using the form: syntactic\_category ::= definition
- 4.
4. [ ] square brackets (non-bold) surround optional items.
- 5.
5. { } curly brackets (non-bold) surrounds items that can repeat zero or more times.
- 6.
6. | vertical line (non-bold) separates alternatives.

[\[ Team LiB \]](#)



[\[ Team LiB \]](#)



## D.1 Source Text

### D.1.1 Library Source Text

```
library_text ::= { library_descriptions }
library_descriptions ::=  
    library_declarati  
on  
    | include_statement  
    | config_declaration  
library_declarati  
on ::=  
    library library_identifier file_path_spec [ { , file_path_spec } ]  
    [ -incdir file_path_spec [ { , file_path_spec } ] ];  
file_path_spec ::= file_path  
include_statement ::= include <file_path_spec>;
```

### D.1.2 Configuration Source Text

```
config_declaration ::=  
    config config_identifier ;  
    design_statement  
    {config_rule_statement}  
    endconfig  
design_statement ::= design { [library_identifier.]cell_identifier } ;  
config_rule_statement ::=  
    default_clause liblist_clause  
    | inst_clause liblist_clause  
    | inst_clause use_clause  
    | cell_clause liblist_clause  
    | cell_clause use_clause  
default_clause ::= default  
inst_clause ::= instance inst_name  
inst_name ::= topmodule_identifier{ .instance_identifier }  
cell_clause ::= cell [ library_identifier.]cell_identifier  
liblist_clause ::= liblist [{library_identifier}]  
use_clause ::= use [library_identifier.]cell_identifier[:config]
```

### D.1.3 Module and Primitive Source Text

```
source_text ::= { description }
description ::=  
    module_declaration  
    | udp_declaration
module_declaration ::=  
    { attribute_instance } module_keyword module_identifier [ module_parameter_port_list  
        ]  
        [ list_of_ports ] ; { module_item }  
        endmodule
```

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## D.2 Declarations

### D.2.1 Declaration Types

#### Module parameter declarations

```
local_parameter_declaration ::=  
    localparam [ signed ] [ range ] list_of_param_assignments ;  
    | localparam integer list_of_param_assignments ;  
    | localparam real list_of_param_assignments ;  
    | localparam realtime list_of_param_assignments ;  
    | localparam time list_of_param_assignments ;  
parameter_declaration ::=  
    parameter [ signed ] [ range ] list_of_param_assignments ;  
    | parameter integer list_of_param_assignments ;  
    | parameter real list_of_param_assignments ;  
    | parameter realtime list_of_param_assignments ;  
    | parameter time list_of_param_assignments ;  
specparam_declaration ::= specparam [ range ] list_of_specparam_assignments ;
```

#### Port declarations

```
inout_declaration ::= inout [ net_type ] [ signed ] [ range ]  
    list_of_port_identifiers  
input_declaration ::= input [ net_type ] [ signed ] [ range ]  
    list_of_port_identifiers  
output_declaration ::=  
    output [ net_type ] [ signed ] [ range ]  
    list_of_port_identifiers  
    | output [ reg ] [ signed ] [ range ]  
    list_of_port_identifiers  
    | output reg [ signed ] [ range ]  
    list_of_variable_port_identifiers  
    | output [ output_variable_type ]  
    list_of_port_identifiers  
    | output output_variable_type  
    list_of_variable_port_identifiers
```

#### Type declarations

```
event_declaration ::= event list_of_event_identifiers ;  
genvar_declaration ::= genvar list_of_genvar_identifiers ;  
integer_declaration ::= integer list_of_variable_identifiers ;  
net_declaration ::=  
    net_type [ signed ]  
    [ delay3 ] list_of_net_identifiers ;  
    | net_type [ drive_strength ] [ signed ]
```

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## D.3 Primitive Instances

### D.3.1 Primitive Instantiation and Instances

```
gate_instantiation ::=  
    cmos_switchtype [delay3]  
        cmos_switch_instance { , cmos_switch_instance } ;  
    | enable_gatetype [drive_strength] [delay3]  
        enable_gate_instance { , enable_gate_instance } ;  
    | mos_switchtype [delay3]  
        mos_switch_instance { , mos_switch_instance } ;  
    | n_input_gatetype [drive_strength] [delay2]  
        n_input_gate_instance { , n_input_gate_instance } ;  
    | n_output_gatetype [drive_strength] [delay2]  
        n_output_gate_instance { , n_output_gate_instance } ;  
  
    | pass_en_switchtype [delay2]  
        pass_enable_switch_instance { , pass_enable_switch_instance } ;  
    | pass_switchtype  
        pass_switch_instance { , pass_switch_instance } ;  
    | pulldown [pulldown_strength]  
        pull_gate_instance { , pull_gate_instance } ;  
    | pullup [pullup_strength]  
        pull_gate_instance { , pull_gate_instance } ;  
cmos_switch_instance ::= [ name_of_gate_instance ] ( output_terminal , input_terminal ,  
    ncontrol_terminal , pcontrol_terminal )  
enable_gate_instance ::= [ name_of_gate_instance ] ( output_terminal , input_terminal ,  
    enable_terminal )  
mos_switch_instance ::= [ name_of_gate_instance ] ( output_terminal , input_terminal ,  
    enable_terminal )  
n_input_gate_instance ::= [ name_of_gate_instance ] ( output_terminal , input_terminal { ,  
    input_terminal } )  
n_output_gate_instance ::= [ name_of_gate_instance ] ( output_terminal { , output_terminal }  
    , input_terminal )  
pass_switch_instance ::= [ name_of_gate_instance ] ( inout_terminal , inout_terminal )  
pass_enable_switch_instance ::= [ name_of_gate_instance ] ( inout_terminal , inout_terminal  
    , enable_terminal )  
pull_gate_instance ::= [ name_of_gate_instance ] ( output_terminal )  
name_of_gate_instance ::= gate_instance_identifier [ range ]
```

### D.3.2 Primitive Strengths

```
pulldown_strength ::=  
    ( strength0 , strength1 )  
    | ( strength1 , strength0 )  
    | ( strength0 )  
pullup_strength ::=  
    ( strength0 , strength1 )  
    | ( strength1 , strength0 )
```

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## D.4 Module and Generated Instantiation

### D.4.1 Module Instantiation

```

module_instantiation ::=

  module_identifier [ parameter_value_assignment ]
    module_instance { , module_instance } ;
parameter_value_assignment ::= #( list_of_parameter_assignments )
list_of_parameter_assignments ::=

  ordered_parameter_assignment { , ordered_parameter_assignment } |
  named_parameter_assignment { , named_parameter_assignment }

ordered_parameter_assignment ::= expression
named_parameter_assignment ::= . parameter_identifier ( [ expression ] )
module_instance ::= name_of_instance ( [ list_of_port_connections ] )
name_of_instance ::= module_instance_identifier [ range ]
list_of_port_connections ::=

  ordered_port_connection { , ordered_port_connection } |
  | named_port_connection { , named_port_connection }

ordered_port_connection ::= { attribute_instance } [ expression ]
named_port_connection ::= { attribute_instance } .port_identifier ( [ expression ] )

```

### D.4.2 Generated Instantiation

```

generated_instantiation ::= generate { generate_item } endgenerate
generate_item_or_null ::= generate_item | ;
generate_item ::=

  generate_conditional_statement
  | generate_case_statement
  | generate_loop_statement
  | generate_block
  | module_or_generate_item
generate_conditional_statement ::=

  if ( constant_expression ) generate_item_or_null [ else generate_item_or_null ]
generate_case_statement ::= case ( constant_expression )
  genvar_case_item { genvar_case_item } endcase
genvar_case_item ::= constant_expression { , constant_expression } :
  generate_item_or_null | default [ : ] generate_item_or_null
generate_loop_statement ::= for ( genvar_assignment ; constant_expression ;
  genvar_assignment )
  begin : generate_block_identifier { generate_item } end
genvar_assignment ::= genvar_identifier = constant_expression
generate_block ::= begin [ : generate_block_identifier ] { generate_item } end

```

[ Team LiB ]

[◀ PREVIOUS](#) [NEXT ▶](#)

## D.5 UDP Declaration and Instantiation

### D.5.1 UDP Declaration

```
udp_declaration ::=  
  { attribute_instance } primitive udp_identifier ( udp_port_list );  
  udp_port_declaration { udp_port_declaration }  
  udp_body  
  endprimitive  
| { attribute_instance } primitive udp_identifier ( udp_declaration_port_list );  
  udp_body  
  endprimitive
```

### D.5.2 UDP Ports

```
udp_port_list ::= output_port_identifier , input_port_identifier { , input_port_identifier }  
udp_declaration_port_list ::=  
  udp_output_declaration , udp_input_declaration { , udp_input_declaration }  
udp_port_declaration ::=  
  udp_output_declaration ;  
| udp_input_declaration ;  
| udp_reg_declaration ;  
udp_output_declaration ::=  
  { attribute_instance } output port_identifier  
  | { attribute_instance } output reg port_identifier [ = constant_expression ]  
udp_input_declaration ::= { attribute_instance } input list_of_port_identifiers  
udp_reg_declaration ::= { attribute_instance } reg variable_identifier
```

### D.5.3 UDP Body

```
udp_body ::= combinational_body | sequential_body  
combinational_body ::= table combinational_entry { combinational_entry } endtable  
combinational_entry ::= level_input_list : output_symbol ;  
sequential_body ::= [ udp_initial_statement ] table sequential_entry { sequential_entry }  
  endtable  
udp_initial_statement ::= initial output_port_identifier = init_val ;  
init_val ::= 1'b0 | 1'b1 | 1'bx | 1'bX | 1'B0 | 1'B1 | 1'Bx | 1BX | 1 | 0  
sequential_entry ::= seq_input_list : current_state : next_state ;  
seq_input_list ::= level_input_list | edge_input_list  
level_input_list ::= level_symbol { level_symbol }  
edge_input_list ::= { level_symbol } edge_indicator { level_symbol }  
edge_indicator ::= ( level_symbol level_symbol ) | edge_symbol  
current_state ::= level_symbol  
next_state ::= output_symbol | -  
output_symbol ::= 0 | 1 | x | X  
level_symbol ::= 0 | 1 | x | X | ? | b | B  
edge_symbol ::= n | P | f | E | p | D | n | N | *
```

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

[ Team LiB ]

◀ PREVIOUS | NEXT ▶

## D.6 Behavioral Statements

### D.6.1 Continuous Assignment Statements

```
continuous_assign ::= assign [ drive_strength ] [ delay3 ] list_of_net_assignments ;
list_of_net_assignments ::= net_assignment { , net_assignment }
net_assignment ::= net_lvalue = expression
```

### D.6.2 Procedural Blocks and Assignments

```
initial_construct ::= initial statement
always_construct ::= always statement
blocking_assignment ::= variable_lvalue = [ delay_or_event_control ] expression
nonblocking_assignment ::= variable_lvalue <= [ delay_or_event_control ] expression
procedural_continuous_assignments ::=
    assign variable_assignment
    | deassign variable_lvalue
    | force variable_assignment
    | force net_assignment
    | release variable_lvalue
    | release net_lvalue
function_blocking_assignment ::= variable_lvalue = expression
function_statement_or_null ::=
    function_statement
    | { attribute_instance } ;
```

### D.6.3 Parallel and Sequential Blocks

```
function_seq_block ::= begin [ : block_identifier
    { block_item_declaration } ] { function_statement } end
variable_assignment ::= variable_lvalue = expression
par_block ::= fork [ : block_identifier
    { block_item_declaration } ] { statement } join
seq_block ::= begin [ : block_identifier
    { block_item_declaration } ] { statement } end
```

### D.6.4 Statements

```
statement ::=
    { attribute_instance } blocking_assignment ;
    | { attribute_instance } case_statement
    | { attribute_instance } conditional_statement
    | { attribute_instance } disable_statement
    | { attribute_instance } event_trigger
```

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## D.7 Specify Section

### D.7.1 Specify Block Declaration

```
specify_block ::= specify { specify_item } endspecify
specify_item ::=  
    specparam_declaration  
    | pulstyle_declaration  
    | showcancelled_declaration  
    | path_declaration  
    | system_timing_check
pulstyle_declaration ::=  
    pulstyle_onevent list_of_path_outputs ;  
    | pulstyle_ondetect list_of_path_outputs ;
showcancelled_declaration ::=  
    showcancelled list_of_path_outputs ;  
    | noshowcancelled list_of_path_outputs ;
```

### D.7.2 Specify Path Declarations

```
path_declaration ::=  
    simple_path_declaration ;  
    | edge_sensitive_path_declaration ;  
    | state_dependent_path_declaration ;  
simple_path_declaration ::=  
    parallel_path_description = path_delay_value  
    | full_path_description = path_delay_value
parallel_path_description ::=  
    ( specify_input_terminal_descriptor [ polarity_operator ] =>  
        specify_output_terminal_descriptor )
full_path_description ::=  
    ( list_of_path_inputs [ polarity_operator ] *-> list_of_path_outputs )
list_of_path_inputs ::=  
    specify_input_terminal_descriptor { , specify_input_terminal_descriptor }
list_of_path_outputs ::=  
    specify_output_terminal_descriptor { , specify_output_terminal_descriptor }
```

### D.7.3 Specify Block Terminals

```
specify_input_terminal_descriptor ::=  
    input_identifier  
    | input_identifier [ constant_expression ]  
    | input_identifier [ range_expression ]
specify_output_terminal_descriptor ::=  
    output_identifier  
    | output_identifier [ constant_expression ]
```

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## D.8 Expressions

### D.8.1 Concatenations

```
concatenation ::= { expression { , expression } }

constant_concatenation ::= { constant_expression { , constant_expression } }

constant_multiple_concatenation ::= { constant_expression constant_concatenation }

module_path_concatenation ::= { module_path_expression { , module_path_expression } }

module_path_multiple_concatenation ::= { constant_expression module_path_concatenation }

multiple_concatenation ::= { constant_expression concatenation }

net_concatenation ::= { net_concatenation_value { , net_concatenation_value } }

net_concatenation_value ::=

    hierarchical_net_identifier

    | hierarchical_net_identifier [ expression ] { [ expression ] }

    | hierarchical_net_identifier [ expression ] { [ expression ] } [ range_expression ]

    | hierarchical_net_identifier [ range_expression ]

    | net_concatenation

variable_concatenation ::= { variable_concatenation_value { , variable_concatenation_value }

    }

variable_concatenation_value ::=

    hierarchical_variable_identifier

    | hierarchical_variable_identifier [ expression ] { [ expression ] }

    | hierarchical_variable_identifier [ expression ] { [ expression ] } [ range_expression ]

    | hierarchical_variable_identifier [ range_expression ]

    | variable_concatenation
```

### D.8.2 Function calls

```
constant_function_call ::= function_identifier { attribute_instance }

    ( constant_expression { , constant_expression } )

function_call ::= hierarchical_function_identifier{ attribute_instance }

    ( expression { , expression } )

genvar_function_call ::= genvar_function_identifier { attribute_instance }

    ( constant_expression { , constant_expression } )

system_function_call ::= system_function_identifier

    [ ( expression { , expression } ) ]
```

### D.8.3 Expressions

```
base_expression ::= expression

conditional_expression ::= expression1 ? { attribute_instance } expression2 : expression3

constant_base_expression ::= constant_expression

constant_expression ::=

    constant_primary

    | unary_operator { attribute_instance } constant_primary

    | constant_expression binary_operator { attribute_instance } constant_expression
```

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## D.9 General

### D.9.1 Attributes

```
attribute_instance ::= (* attr_spec { , attr_spec } *)
attr_spec ::=  
    attr_name = constant_expression  
    | attr_name
attr_name ::= identifier
```

### D.9.2 Comments

```
comment ::=  
    one_line_comment  
    | block_comment
one_line_comment ::= // comment_text \n
block_comment ::= /* comment_text */
comment_text ::= { Any_ASCII_character }
```

### D.9.3 Identifiers

```
arrayed_identifier ::=  
    simple_arrayed_identifier  
    | escaped_arrayed_identifier
block_identifier ::= identifier
cell_identifier ::= identifier
config_identifier ::= identifier
escaped_arrayed_identifier ::= escaped_identifier [ range ]
escaped_hierarchical_identifier[4] ::=  
    escaped_hierarchical_branch  
    { .simple_hierarchical_branch | .escaped_hierarchical_branch }
escaped_identifier ::= \ {Any_ASCII_character_except_white_space} white_space
event_identifier ::= identifier
function_identifier ::= identifier
gate_instance_identifier ::= arrayed_identifier
generate_block_identifier ::= identifier
genvar_function_identifier ::= identifier /* Hierarchy disallowed */
genvar_identifier ::= identifier
hierarchical_block_identifier ::= hierarchical_identifier
hierarchical_event_identifier ::= hierarchical_identifier
hierarchical_function_identifier ::= hierarchical_identifier
hierarchical_identifier ::=  
    simple_hierarchical_identifier  
    | escaped_hierarchical_identifier
hierarchical_net_identifier ::= hierarchical_identifier
hierarchical_variable_identifier ::= hierarchical_identifier
hierarchical_task_identifier ::= hierarchical_identifier
```

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## Endnotes

1.

1. Embedded spaces are illegal.

2.

2. A simple\_identifier and arrayed\_reference shall start with an alpha or underscore (\_) character, shall have at least one character, and shall not have any spaces.

3.

3. The period (.) in simple\_hierarchical\_identifier and simple\_hierarchical\_branch shall not be preceded or followed by white\_space.

4.

4. The period in escaped\_hierarchical\_identifier and escaped\_hierarchical\_branch shall be preceded by white\_space, but shall not be followed by white\_space.

5.

5. The \$ character in a system\_function\_identifier or system\_task\_identifier shall not be followed by white\_space. A system\_function\_identifier or system\_task\_identifier shall not be escaped.

6.

6. End of file.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## Appendix E. Verilog Tidbits

Answers to common Verilog questions are provided in this appendix.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## Origins of Verilog HDL

Verilog HDL originated around 1983 at Gateway Design Automation, which was then located in Acton, Massachusetts. The language that most influenced Verilog HDL was HILO-2, which was developed at Brunel University in England under contract to produce a test generation system for the British Ministry of Defense. HILO-2 successfully combined the gate and register transfer levels of abstraction and supported verification simulation, timing analysis, fault simulation, and test generation.

Gateway Design Automation was privately held at that time and was headed by Dr. Prabhu Goel, the inventor of the PODEM test generation algorithm. Verilog HDL was introduced into the EDA market in 1985 as a simulator product. Verilog HDL was designed by Phil Moorby, who was later to become the Chief Designer for Verilog-XL and the first Corporate Fellow at Cadence Design Systems. Gateway Design Automation grew rapidly with the success of Verilog-XL and was finally acquired by Cadence Design Systems, San Jose, CA, in 1989.

Verilog HDL was opened to the public by Cadence Design Systems in 1990. Open Verilog International(OVI) was formed to standardize and promote Verilog HDL and related design automation products.

In 1992, the Board of Directors of OVI began an effort to establish Verilog HDL as an IEEE standard. In 1993, the first IEEE Working Group was formed and, after 18 months of focused efforts, Verilog became the IEEE Standard 1364-1995.

After the standardization process was complete, the 1364 Working Group started looking for feedback from 1364 users worldwide so that the standard could be enhanced and modified accordingly. This led to a five-year effort to create a much better Verilog standard IEEE 1364-2001.

[\[ Team LiB \]](#)

[!\[\]\(56ce2fb78c8e1d4dffbde1b4d295a85e\_img.jpg\) PREVIOUS](#) [!\[\]\(abf138848e9a9d86b8f09b7c7431c1bb\_img.jpg\) NEXT](#)

[\[ Team LiB \]](#)

[!\[\]\(514079e65433539949516a1b33bfc7c5\_img.jpg\) PREVIOUS](#) [!\[\]\(a8223eebed5110b38dd3dc5703c554f5\_img.jpg\) NEXT](#)

## Interpreted, Compiled, Native Compiled Simulators

Verilog simulators come in three flavors, based on the way they perform the simulation.

Interpreted simulators read in the Verilog HDL design, create data structures in memory, and run the simulation interpretively. A compile is performed each time the simulation is run, but the compile is usually very fast. An example of an interpreted simulator is Cadence Verilog-XL simulator.

Compiled code simulators read in the Verilog HDL design and convert it to equivalent C code (or some other programming language). The C code is then compiled by a standard C compiler to get the binary executable. The binary is executed to run the simulation. Compile time is usually long for compiled code simulators, but, in general, the execution speed is faster compared to interpreted simulators. An example of compiled code simulator is Synopsys VCS simulator.

Native compiled code simulators read in the Verilog HDL design and convert it directly to binary code for a specific machine platform. The compilation is optimized and tuned separately for each machine platform. Of course, that means that a native compiled code simulator for a Sun workstation will not run on an HP workstation, and vice versa. Because of fine tuning, native compiled code simulators can yield significant performance benefits. An example of a native compiled code simulator is Cadence Verilog-NC simulator.

[\[ Team LiB \]](#)



[\[ Team LiB \]](#)

## Event-Driven Simulation, Oblivious Simulation

Verilog simulators typically use an event-driven or an oblivious simulation algorithm. An event-driven algorithm processes elements in the design only when signals at the inputs of these elements change. Intelligent scheduling is required to process elements. Oblivious algorithms process all elements in the design, irrespective of changes in signals. Little or no scheduling is required to process elements.

[ Team LiB ]

PREVIOUS | NEXT

[ Team LiB ]

PREVIOUS | NEXT

## Cycle-Based Simulation

Cycle-based simulation is useful for synchronous designs where operations happen only at active clock edges. Cycle simulators work on a cycle-by-cycle basis. Timing information between two clock edges is lost. Significant performance advantages can be obtained by using cycle simulation.

[ Team LiB ]

PREVIOUS | NEXT

[ Team LiB ]

PREVIOUS | NEXT

## Fault Simulation

Fault simulation is used to deliberately insert stuck-at or bridging faults in the reference circuit. Then, a test pattern is applied and the outputs of the faulty circuit and the reference circuit are compared. The fault is said to be detected if the outputs mismatch. A set of test patterns is developed for testing the circuit.

[ Team LiB ]

PREVIOUS | NEXT

[ Team LiB ]

PREVIOUS | NEXT

## General Verilog Web sites

The following sites provide interesting information related to Verilog HDL.

- 1.
1. Verilog? <http://www.verilog.com>
- 2.
2. Cadence? <http://www.cadence.com/>
- 3.
3. EE Times? <http://www.eetimes.com>
- 4.
4. Synopsys? <http://www.synopsys.com/>
- 5.
5. DVCon (Conference for HDL and HVL Users)? <http://www.dvcon.org>
- 6.
6. Verification Guild? <http://www.janick.bergeron.com/guild/default.htm>
- 7.
7. Deep Chip? <http://www.deepchip.com>

[\[ Team LiB \]](#)

[PREVIOUS](#) [NEXT](#)

[\[ Team LiB \]](#)

[PREVIOUS](#) [NEXT](#)

## Architectural Modeling Tools

- 1.
1. For details on System C, see <http://www.systemc.org>

[\[ Team LiB \]](#)

[PREVIOUS](#) [NEXT](#)



[\[ Team LiB \]](#)

PREVIOUS  NEXT

## High-Level Verification Languages

1.

1. Information on e is available at <http://www.verisity.com>
- 2.
2. Information on Vera is available at <http://www.open-vera.com>
- 3.
3. Information on SuperLog is available at <http://www.synopsys.com>
- 4.
4. Information on SystemVerilog is available at <http://www.accellera.org>

[\[ Team LiB \]](#)

PREVIOUS  NEXT

[\[ Team LiB \]](#)

PREVIOUS  NEXT

## Simulation Tools

1.

1. Information on Verilog-XL and Verilog-NC is available at <http://www.cadence.com>
- 2.
2. Information on VCS is available at <http://www.synopsys.com>

[\[ Team LiB \]](#)

PREVIOUS  NEXT

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## Hardware Acceleration Tools

Information on hardware acceleration tools is available at the Web sites of the following companies:

- 1.
1. <http://www.cadence.com>
- 2.
2. <http://www.aptix.com>
- 3.
3. <http://www.mentorg.com>
- 4.
4. <http://www.axiscorp.com>
- 5.
5. <http://www.tharas.com>

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## In-Circuit Emulation Tools

Information on in-circuit emulation tools is available at the Web sites of the following companies.

- 1.
1. <http://www.cadence.com>
- 2.
2. <http://www.mentorg.com>

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## Coverage Tools

Information on coverage tools is available at the Web sites of the following companies:

- 1.
1. <http://www.verisity.com>
- 2.
2. <http://www.synopsys.com>

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## **Assertion Checking Tools**

Information on assertion checking tools is available at the Web sites of the following companies:

- 1.
1. Information on e is available at <http://www.verisity.com>
- 2.
2. Information on Vera is available at <http://www.open-vera.com>
- 3.
3. Information on SystemVerilog is available at <http://www.accellera.org>
- 4.
4. <http://www.0-in.com>
- 5.
5. <http://www.verplex.com>
- 6.
6. Information on Open Verification Library is available at <http://www.accellera.org>

[\[ Team LiB \]](#)



[\[ Team LiB \]](#)



## **Equivalence Checking Tools**

- 1.
1. Information on equivalence checking tools is available at <http://www.verplex.com>
- 2.
2. Information on equivalence checking tools is available at <http://www.synopsys.com>

[\[ Team LiB \]](#)



[\[ Team LiB \]](#)



## Formal Verification Tools

Information on formal verification tools is available at the Web sites of the following companies:

- 1.
1. <http://www.verplex.com>
- 2.
2. <http://www.realintent.com>
- 3.
3. <http://www.synopsys.com>
- 4.
4. <http://www.athdl.com>
- 5.
5. <http://www.0-in.com>

[ Team LiB ]



[ Team LiB ]



## Appendix F. Verilog Examples

This appendix contains the source code for two examples.

- 
- The first example is a synthesizable model of a FIFO implementation.
- 
- The second example is a behavioral model of a 256K x 16 DRAM.

These examples are provided to give the reader a flavor of real-life Verilog HDL usage. The reader is encouraged to look through the source code to understand coding style and the usage of Verilog HDL constructs.

[ Team LiB ]



[ Team LiB ]

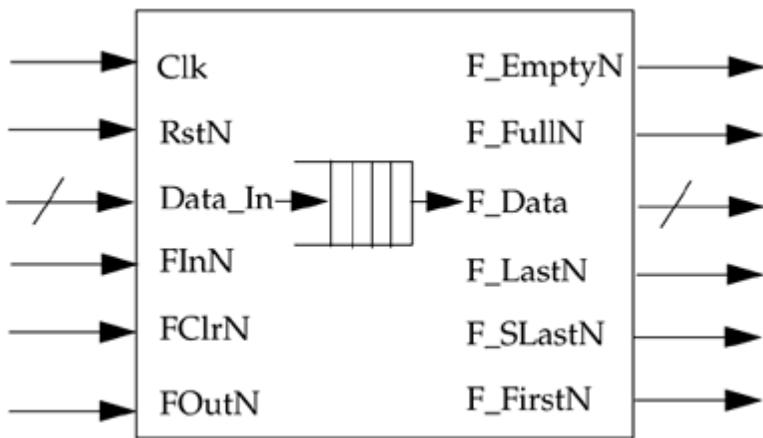




## F.1 Synthesizable FIFO Model

This example describes a synthesizable implementation of a FIFO. The FIFO depth and FIFO width in bits can be modified by simply changing the value of two parameters, `FWIDTH and `FDEPTH. For this example, the FIFO depth is 4 and the FIFO width is 32 bits. The input/output ports of the FIFO are shown in [Figure F-1](#).

**Figure F-1. FIFO Input/Output Ports**



### Input ports

All ports with a suffix "N" are low asserted.

Clk?Clock signal

RstN?Reset signal

Data\_In?32-bit data into the FIFO

FInN?Write into FIFO signal

FClrN?Clear signal to FIFO

FOutN?Read from FIFO signal

### Output ports

F\_Data?32-bit output data from FIFO

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



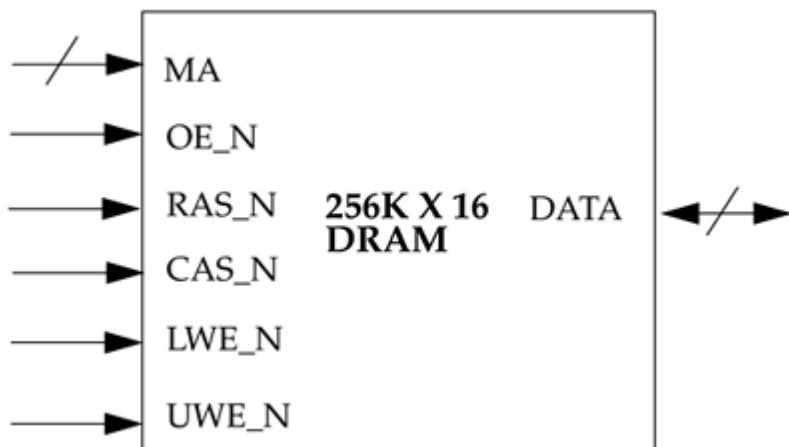
[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## F.2 Behavioral DRAM Model

This example describes a behavioral implementation of a 256K x 16 DRAM. The DRAM has 256K 16-bit memory locations. The input/output ports of the DRAM are shown in [Figure F-2](#).

**Figure F-2. DRAM Input/Output Ports**



### Input ports

All ports with a suffix "N" are low asserted.

MA?10-bit memory address

OE\_N?Output enable for reading data

RAS\_N?Row address strobe for asserting row address

CAS\_N?Column address strobe for asserting column address

LWE\_N?Lower write enable to write lower 8 bits of DATA into memory

UWE\_N?Upper write enable to write upper 8 bits of DATA into memory

### Inout ports

DATA?16-bit data as input or output. Write input if LWE\_N

or UWE\_N is asserted. Read output if OE\_N is asserted.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## Bibliography

- [Manuals](#)
- [Books](#)
- [Quick Reference Guides](#)

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## Manuals

IEEE 1364-2001 Standard, IEEE Standard Verilog Hardware Description Language, 2001 ( <http://www.ieee.org> ).

Accellera Standard, System Verilog 3.0: Accellera's Extensions to Verilog, 2002. ( <http://www.accellera.org> ).

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ► ]

## Books

E. Sternheim, Rajvir Singh, Rajeev Madhavan, Yatin Trivedi, Digital Design and Synthesis with Verilog HDL, Automata Publishing Company, 1993. ISBN 0-9627488-2-X.

Donald Thomas and Phil Moorby, The Verilog Hardware Description Language, Fourth Edition, Kluwer Academic Publishers, 1998. ISBN 0-7923-8166-1.

Stuart Sutherland, Verilog 2001? Guide to the New Features of the Verilog Hardware Description Language, Kluwer Academic Publishers, 2002. ISBN 0-7923-7568-8.

Stuart Sutherland, The Verilog Pli Handbook: A User's Guide and Comprehensive Reference on the Verilog Programming Language Interface, Second Edition, Kluwer Academic Publishers, 2002. ISBN: 0-7923-7658-7.

Douglas Smith, HDL Chip Design : A Practical Guide for Designing, Synthesizing and Simulating ASICs and FPGAs Using VHDL or Verilog, Doone Publications, TX, 1996. ISBN: 0-9651934-3-8.

Ben Cohen, Real Chip Design and Verification Using Verilog and VHDL, VhdlCohen Publishing, 2001. ISBN 0-9705394-2-8.

J. Bhasker, Verilog HDL Synthesis: A Practical Primer, Star Galaxy Publishing, 1998. ISBN 0-9650391-5-3.

J. Bhasker, A Verilog HDL Primer, Star Galaxy Publishing, 1999. ISBN 0-9650391-7-X.

James M. Lee, Verilog Quickstart, Kluwer Academic Publishers, 1997. ISBN 0-7923992-7-7.

Bob Zeidman, Verilog Designer's Library, Prentice Hall, 1999. ISBN 0-1308115-4-8.

Michael D. Ciletti, Modeling, Synthesis, and Rapid Prototyping with the Verilog HDL, Prentice Hall, 1999. ISBN 0-1397-7398-3.

Janick Bergeron, Writing Testbenches: Functional Verification of HDL Models, Kluwer Academic Publishers, 2000. ISBN 0-7923-7766-4.

Lionel Bening and Harry Foster, Principles of Verifiable RTL Design, Second Edition, Kluwer Academic Publishers, 2001. ISBN: 0-7923-7368-5.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## Quick Reference Guides

Stuart Sutherland, Verilog HDL Quick Reference Guide, Sutherland Consulting, OR, 2001. ISBN: 1-930368-03-8.

Rajeev Madhavan, Verilog HDL Reference Guide, Automata Publishing Company, CA, 1993. ISBN 0-9627488-4-6.

Stuart Sutherland, Verilog PLI Quick Reference Guide, Sutherland Consulting, OR, 2001. ISBN: 1-930368-02-X.

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]



[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## About the CD-ROM

- [Using the CD-ROM](#)
- [Technical Support](#)

[ Team LiB ]

[ PREVIOUS ] [ NEXT ]

## Using the CD-ROM

The CD that accompanies this book contains a demonstration version of the SILOS 2001 Verilog HDL Toolbox for Microsoft Windows (98/98SE/ME/NT/2000/XP). To run it, please follow these

[ Team LiB ]

 PREVIOUS

## Technical Support

Prentice Hall does not offer technical support for the material on the CD-ROM. If you have any problems with the Verilog simulator, please visit <http://www.simucad.com/>. If the CD is physically damaged, you may obtain a replacement copy by sending an email to [disc\\_exchange@prenhall.com](mailto:disc_exchange@prenhall.com).

[ Team LiB ]

 PREVIOUS

## Embedded Secure Document

The file *file:///C/Program%20Files/eMule/Incoming/Prentice%20Hall%20-%20Verilog%20HDL%20-%20A%20Guide%20To%20Digital%20Design%20And%20Synthesis,%202nd%20Edition%20(2003).pdf* is a secure document that has been embedded in this document. Double click the pushpin to view.



# safariexamples.informit.com - /0130449113/

---

[\[To Parent Directory\]](#)

Tuesday, November 20, 2001 4:41 PM	3673 <a href="#">0x0409.ini</a>
Thursday, July 02, 1998 1:07 PM	41 <a href="#">Autorun.inf</a>
Friday, September 26, 2003 11:04 AM	<dir> <a href="#">doc</a>
Friday, September 26, 2003 11:04 AM	<dir> <a href="#">Examples</a>
Friday, September 26, 2003 11:04 AM	<dir> <a href="#">Help</a>
Wednesday, September 26, 2001 4:56 PM	1707856 <a href="#">instmsia.exe</a>
Monday, August 27, 2001 2:13 PM	1821008 <a href="#">instmsiw.exe</a>
Friday, November 30, 2001 12:43 PM	477696 <a href="#">isscript.msi</a>
Friday, September 26, 2003 11:04 AM	<dir> <a href="#">PLI</a>
Friday, September 26, 2003 11:03 AM	<dir> <a href="#">program_files</a>
Wednesday, November 06, 2002 8:02 PM	5565 <a href="#">README.TXT</a>
Tuesday, November 06, 2001 1:03 PM	81758 <a href="#">setup.bmp</a>
Friday, November 30, 2001 12:35 PM	200704 <a href="#">setup.exe</a>
Monday, June 24, 2002 10:49 PM	1031 <a href="#">Setup.ini</a>
Monday, June 24, 2002 10:49 PM	1381004 <a href="#">Silos2001.msi</a>
Friday, September 26, 2003 11:03 AM	<dir>

[VERILOG BOOK EXAMPLES](#)

---

# safariexamples.informit.com - /0130449113/ Examples/

---

[\[To Parent Directory\]](#)

Sunday, February 02, 1997	5:46 PM	3972	<a href="#">abc_100.v</a>
Thursday, March 14, 2002	2:54 AM	782	<a href="#">analog.spj</a>
Wednesday, July 26, 2000	11:13 PM	6691	<a href="#">analog.v</a>
Thursday, March 14, 2002	2:54 AM	809	<a href="#">code_coverage.spj</a>
Wednesday, April 11, 2001	9:40 PM	437	<a href="#">code_coverage.v</a>
Thursday, March 14, 2002	2:54 AM	819	<a href="#">code_coverage2.spj</a>
Wednesday, July 29, 1998	1:17 AM	10133	<a href="#">design.lib</a>
Thursday, July 23, 1998	8:50 PM	15308	<a href="#">design.sdf</a>
Monday, July 28, 1997	11:59 AM	238	<a href="#">design.sym</a>
Thursday, July 23, 1998	8:59 PM	4538	<a href="#">design.v</a>
Saturday, July 25, 1998	1:08 PM	204	<a href="#">design1</a>
Saturday, July 25, 1998	1:08 PM	205	<a href="#">design2</a>
Saturday, August 19, 2000	4:05 AM	542	<a href="#">faulttst.v</a>
Thursday, March 14, 2002	2:54 AM	1743	<a href="#">fltsim.spj</a>
Thursday, March 14, 2002	2:55 AM	960	<a href="#">gate.spj</a>
Sunday, March 16, 1997	1:17 PM	194	<a href="#">rtl.sym</a>
Thursday, March 14, 2002	2:53 AM	1613	<a href="#">rtl.spj</a>
Thursday, March 14, 2002	2:55 AM	677	<a href="#">rtl_err.spj</a>
Saturday, August 19, 2000	4:01 AM	2063	<a href="#">stimulus.v</a>
Saturday, August 19, 2000	4:03 AM	1911	<a href="#">stimulus.v1</a>
Wednesday, April 11, 2001	7:59 PM	339	<a href="#">testbench.v</a>
Wednesday, April 11, 2001	9:40 PM	339	<a href="#">testbench2.v</a>
Thursday, March 27, 1997	1:19 AM	2156	<a href="#">vend.gv</a>
Thursday, March 27, 1997	1:19 AM	2459	<a href="#">vend.v</a>
Thursday, March 27, 1997	1:19 AM	1446	<a href="#">venderr.v</a>
Monday, July 24, 2000	9:03 PM	10182	<a href="#">vending fsm</a>
Thursday, March 14, 2002	2:56 AM	1028	<a href="#">vending.spj</a>
Tuesday, July 11, 2000	2:52 PM	194	<a href="#">vending.sym</a>
Monday, July 24, 2000	9:03 PM	3916	<a href="#">vending.v</a>
Tuesday, July 11, 2000	3:07 PM	1428	<a href="#">vending_testbench.</a>

V

Tuesday, September 19, 2000 4:24 PM

1690 [vendtest.v](#)

---

## Embedded Secure Document

The file *file:///G/Program%20Files/eMule/Incoming/Prentice%20Hall%20-%20Verilog%20HDL%20-%20A%20Guide%20To%20Digital%20Design%20And%20Synthesis,%202nd%20Edition%20(2003).pdf* is a secure document that has been embedded in this document. Double click the pushpin to view.



# safariexamples.informit.com - /0130449113/doc/

---

[\[To Parent Directory\]](#)

Wednesday, May 23, 2001 3:05 PM

2732769 [sse.pdf](#)

---

# Silos User's Manual

## Version 2001.1

By Simucad, Inc.

# Contents

<b>1. Overview / Installation</b>	<b>1-1</b>
1.1 Silos Overview .....	1-1
1.2 Installation .....	1-2
1.2.1 How to Install Silos for Microsoft Windows .....	1-2
1.2.2 FLEXIm Installation Problems.....	1-6
1.2.3 Solaris Installation Instructions .....	1-8
1.2.4 Linux Installation Instructions.....	1-11
1.3 Accessing Technical Support .....	1-14
<b>2. Tutorial</b>	<b>2-1</b>
2.1 Overview for Debugging FPGA and ASIC Designs .....	2-1
2.1.1 General Items for Tutorial.....	2-4
2.2 RTL (Behavioral) Debugging.....	2-5
2.2.1 Starting Silos .....	2-6
2.2.2 Accessing On-line Help.....	2-6
2.2.3 Example Projects .....	2-8
2.2.4 Creating a Project .....	2-9
2.2.5 Simulating a Design .....	2-11
2.2.6 Selecting Signals for Waveforms .....	2-12
2.2.7 Waveform Colors .....	2-15
2.2.8 Resizing the Name List Box .....	2-18
2.2.9 Copying Waveforms.....	2-19
2.2.10 Rearranging the Signal Names .....	2-20
2.2.11 Expanding/Hiding Bits to Vectors .....	2-21
2.2.12 Displaying Vectors using Symbolic Names .....	2-22
2.2.13 Creating an Annotated Timeline for the Waveform Display.....	2-26
2.2.14 Organizing Signals Into Groups .....	2-28
2.2.15 Buses .....	2-30
2.2.16 Search on Condition .....	2-33
2.2.17 Scan-to-Edge, Scan-to-Value and Panning .....	2-36
2.2.18 Zoom-Buttons.....	2-40
2.2.19 Bookmark, Timescale, Goto Timepoint .....	2-42
2.2.20 Data Tips .....	2-45
2.2.21 Single Stepping .....	2-47
2.2.22 Setting and Forcing Values .....	2-51
2.2.23 Breakpoints.....	2-53
2.3 Code Coverage .....	2-55
2.3.1 Generating Line Coverage Reports .....	2-55
2.3.2 Generating Operator Coverage Reports .....	2-60
2.3.3 Merging Code Coverage Reports .....	2-64
2.4 Finite State Machine (FSM) Entry.....	2-66
2.4.1 Creating a Simple FSM .....	2-66
2.4.2 Debugging using a Finite State Machine.....	2-75

2.5 Gate level Debugging .....	2-81
2.5.1 Trace Signal Inputs.....	2-81
2.6 Creating Smaller Save Files for Logic Simulation .....	2-90
2.6.1 Saving Across a Time Interval for a Constant Save File Size.....	2-90
2.6.2 Saving Selected Wires and Instances .....	2-92
2.7 Example for Batch Logic Simulation .....	2-95
2.8 Error reporting .....	2-97
2.9 Analog Behavioral Modeling (AHDL).....	2-100
2.9.1 Specifying the Analog Behavioral Modeling Project.....	2-100
2.9.2 Running the Analog Behavioral Modeling Simulation .....	2-101
2.10 Exiting Silos .....	2-103

**3. Menus****3-1**

3.1 Menus Overview.....	3-1
3.1.1 Pull-Down Menu Bar .....	3-1
3.1.2 Context Menus .....	3-1
3.1.3 Dialog Box Conventions .....	3-2
3.2 File Menu.....	3-3
3.2.1 New .....	3-3
3.2.2 Open Menu.....	3-3
3.2.3 Save .....	3-3
3.2.4 Save As.....	3-4
3.2.5 Export.....	3-4
3.2.6 Print menu .....	3-4
3.2.7 Print Preview.....	3-5
3.2.8 Print Setup .....	3-5
3.2.9 Exit Menu.....	3-5
3.3 Edit Menu .....	3-6
3.3.1 Undo menu selection .....	3-6
3.3.2 Cut menu selection .....	3-6
3.3.3 Copy menu selection .....	3-6
3.3.4 Paste menu selection .....	3-7
3.3.5 Clear menu selection .....	3-7
3.3.6 Select All menu selection .....	3-7
3.3.7 Find menu selection.....	3-7
3.3.8 Find Next menu selection.....	3-7
3.3.9 Replace menu selection .....	3-7
3.3.10 Goto Line menu selection.....	3-7
3.3.11 Delete menu selection.....	3-8
3.3.12 Copy Image to Clipboard menu selection .....	3-8
3.4 View Menu .....	3-9
3.4.1 Zoom selections.....	3-9
3.4.2 Main Toolbar menu selections .....	3-10
3.4.3 Analyzer Toolbar menu selections .....	3-10
3.4.4 FSM Toolbar menu selections.....	3-10
3.4.5 Status Bar .....	3-11
3.5 Project Menu.....	3-12
3.5.1 New Menu Selection .....	3-12
3.5.2 Open Menu Selection .....	3-13
3.5.3 Files Menu Selection.....	3-13
3.5.4 Save As Menu Selection .....	3-15
3.5.5 Close Menu Selection.....	3-15
3.5.6 Save Project State Menu Selection.....	3-15
3.5.7 Restore Project State Menu Selection .....	3-15

3.5.8 Load/Reload Input Files Menu Selection .....	3-16
3.5.9 Reload and Go Menu Selection.....	3-16
3.5.10 Project Settings Menu Selection.....	3-17
3.5.11 Filters.....	3-20
3.5.12 Project List Size.....	3-20
3.6 Code Coverage Menu .....	3-21
3.6.1 Enable Code Coverage Menu Selection .....	3-21
3.6.2 Enable Operator Coverage Menu Selection .....	3-22
3.6.3 Select Files to Merge Menu Selection .....	3-22
3.6.4 Line Report Menu Selection.....	3-23
3.6.5 Exporting Code Coverage Analysis .....	3-24
3.6.6 Restricting Code Coverage Analysis.....	3-24
3.6.7 Operator Report Menu Selection.....	3-25
3.7 State Machine Menu .....	3-27
3.7.1 New Menu Selection .....	3-28
3.7.2 Open Menu Selection .....	3-28
3.7.3 Save Menu Selection .....	3-28
3.7.4 Save As Menu Selection.....	3-28
3.7.5 Select Mode Menu Selection.....	3-28
3.7.6 State Mode Menu Selection.....	3-29
3.7.7 Expression Mode Menu Selection.....	3-29
3.7.8 Note Mode Menu Selection.....	3-29
3.7.9 Transition Mode Menu Selection .....	3-29
3.8 Reports Menu .....	3-30
3.8.1 Activity Menu Selection .....	3-30
3.8.2 Errors Menu Selection.....	3-31
3.8.3 Fault Menu Selection.....	3-31
3.8.4 Iteration Menu Selection .....	3-31
3.8.5 Nonconvergence Menu Selection.....	3-32
3.8.6 Size Menu Selection .....	3-36
3.9 Explorer Menu .....	3-37
3.9.1 Open Explorer Menu Selection .....	3-37
3.9.2 Go to Module Source Menu Selection .....	3-38
3.10 Debug Menu .....	3-39
3.10.1 Enable Single Step/Breakpoints Menu Selection .....	3-39
3.10.2 Go Menu Selection.....	3-39
3.10.3 Break Simulation Menu Selection.....	3-40
3.10.4 Finish Current Timepoint Menu Selection .....	3-41
3.10.5 Restart Simulation Menu Selection .....	3-41
3.10.6 Step Menu Selection.....	3-41
3.10.7 Breakpoints Menu Selection .....	3-42
3.11 Options Menu .....	3-44
3.11.1 Fonts menu selection .....	3-44
3.11.2 Tabs menu selection .....	3-44
3.11.3 Snap to Edges .....	3-44
3.11.4 Title Tips menu selection .....	3-45
3.11.5 Analog Integer Display .....	3-45
3.11.6 Strength Color Coding .....	3-45
3.11.7 Trace Color Coding .....	3-45
3.11.8 White Background.....	3-45
3.11.9 Black Background .....	3-46
3.11.10 Full Path Title.....	3-46
3.11.11 Data Tips .....	3-46
3.11.12 Syntax Color Coding .....	3-46
3.12 Window Menu .....	3-47

3.12.1 Cascade Menu Selection .....	3-47
3.12.2 Tile Menu Selection .....	3-47
3.12.3 Arrange Icons Menu Selection .....	3-47
3.12.4 Open Explorer Menu Selection .....	3-48
3.12.5 Open Watch Menu Selection.....	3-48
3.12.6 Open New Data Analyzer Menu Selection .....	3-49
3.13 Help Menu .....	3-55
3.13.1 Contents menu selection.....	3-55
3.13.2 Using Help menu selection.....	3-55
3.13.3 Quick Start Guide menu selection.....	3-55
3.13.4 User' Manual menu selection.....	3-55
3.13.5 Verilog LRM menu selection .....	3-55
3.13.6 SDF Manual menu selection .....	3-55
3.13.7 About Silos.....	3-55
3.13.8 User Registration.....	3-56
3.14 Context Menus.....	3-57
3.14.1 Explorer Window .....	3-58
3.14.2 Watch Window.....	3-65
3.14.3 Data Analyzer Context menus.....	3-67
3.14.4 Source Window Context menus .....	3-76
3.14.5 Output Window Context menus .....	3-78

## 4. Verilog HDL Extensions

4-1

4.1 Silos PLI Interface .....	4-1
4.1.1 Silos PLI Interface on the PC .....	4-1
4.1.2 Silos PLI Interface on the Workstation .....	4-2
4.1.3 List of Implemented PLI Routines .....	4-3
4.2 Standard Delay Format .....	4-4
4.3 Expected Values and Stimulustable.....	4-6
4.3.1 BNF .....	4-6
4.3.2 Stimulustable .....	4-7
4.3.3 Radix .....	4-9
4.3.4 Delay Time .....	4-9
4.3.5 Memory Utilization .....	4-11
4.3.6 Strobe .....	4-11
4.3.7 I/O Pad.....	4-13
4.3.8 Expected Value Error .....	4-14
4.3.9 Expected Value Error Storage.....	4-16
4.3.10 Incremental Update .....	4-16
4.3.11 Changing Behavioral Stimulus to a "stimulustable" Format.....	4-18
4.4 Analog Extensions .....	4-20
4.4.1 Real and Integer Data Types .....	4-20
4.4.2 Utility Transcendental Functions .....	4-21
4.4.3 Examples for Transcendental Math Functions .....	4-22
4.5 "silos" and "sse" keywords.....	4-23
4.6 Extensions to Turn-off, Reset, and Turn-on Saving .....	4-23
4.7 Silos Extensions to Verilog HDL .....	4-24
4.7.1 Global Variables:.....	4-24
4.7.2 Global tasks and functions: .....	4-24
4.7.3 Functions with multiple outputs: .....	4-25
4.7.4 Functions without any inputs: .....	4-25
4.7.5 Tasks and functions with ports declared like a module:.....	4-25
4.7.6 Procedural assignment to wires: .....	4-26
4.7.7 Continuous assignments to register and memory variables:.....	4-26

4.7.8 Continuous assignments using intra-assignment/non-blocking delays: .....	4-26
4.7.9 Default state value for UDP: .....	4-27
4.7.10 UDP additional states for High-Z on inputs or output: .....	4-27
4.7.11 UDP edge for High-Z: .....	4-28
4.7.12 UDP Multiple Edges in a Row: .....	4-28
4.7.13 Non-Constant Specify Block Delays: .....	4-28
4.7.14 Parameter for Specify Block Delays: .....	4-28
4.7.15 Stimulustable Extension: .....	4-29
4.7.16 “input/output/inout” declarations after the variable’s declaration: .....	4-29
4.7.17 Using registers as module inputs: .....	4-29
4.7.18 Duplicate variable definitions: .....	4-30
4.7.19 Parameter used for sizing numbers: .....	4-30
4.7.20 Null statements: .....	4-30
4.7.21 Timing checks without edge specifications for selected variables: .....	4-31
4.7.22 More precision in “\$timeformat” than “timescale”: .....	4-31
4.7.23 Missing port connections for VCS compatibility: .....	4-31
4.7.24 VCS compatibility extension: .....	4-31

**5. Silos Commands****5-1**

5.1 Commands Overview .....	5-1
5.1.1 Command Syntax .....	5-1
5.1.2 Inputting SILOS Commands .....	5-1
5.1.3 Stopping Processes .....	5-1
5.2 Activity Report For Nodes .....	5-2
5.3 Bus Contention Report .....	5-10
5.4 Exporting Code Coverage Results .....	5-12
5.5 Exclude Code Coverage for Module Instances .....	5-13
5.6 Keeping Code Coverage for Module Instances .....	5-14
5.7 Encrypting Library Files .....	5-15
5.8 Control Parameters For Logic Simulation .....	5-16
5.9 Default Device Delay Times .....	5-21
5.10 Disk File Name Reassignment .....	5-22
5.11 Error Summary .....	5-23
5.12 Exclude Saving Simulation Node States .....	5-24
5.13 Exiting The Program .....	5-25
5.14 File Name Specification .....	5-26
5.15 Keeping Simulation Node States .....	5-27
5.16 Exclude Saving Module Instance Variable Values .....	5-28
5.17 Keeping Module Instance Simulation Variable Values .....	5-29
5.18 Nonconvergence Summary .....	5-30
5.19 Narrow Storing Outputs .....	5-33
5.20 Preprocessing Data .....	5-34
5.21 Probing Node States .....	5-35
5.22 Quitting Execution .....	5-37
5.23 Resetting Selected Data .....	5-38
5.24 Scope For Printing Module Variables .....	5-40
5.25 Logic Simulation Specification .....	5-41
5.26 Size-Of-Data Reprint .....	5-43
5.27 Spike Summary Output .....	5-44
5.28 Storing Outputs .....	5-45
5.29 Symbol Modification For Output .....	5-48

**6. Appendix A: Libraries****6-1**

6.1 Overview .....	6-1
6.2 Library Command.....	6-2
6.3 TTL LS Parts List .....	6-4
6.4 TTL BCT Parts List.....	6-13

**7. Appendix B: Command Line Arguments** **7-1**

7.1 Batch Execution Overview .....	7-1
7.1.1 Commands in Files .....	7-2
7.1.2 Command-line Options .....	7-4
7.1.3 Windows Batch Execution .....	7-10
7.1.4 Unix Batch Execution .....	7-12

**8. Index** **8-1**

# 1. Overview / Installation

## 1.1 Silos Overview

Silos is a logic simulation environment developed for use in the design and verification of electronic circuits and systems. Silos can simulate designs at the behavioral, gate and switch levels that are modeled with the Verilog Hardware Description Language (HDL). Silos can also back annotate delays specified using the Standard Delay Format (SDF).

During logic simulation, Silos saves every changed event for every variable in a very compact, binary save file. The save file allows the Silos to display waveforms for any signal and to traceback on any gate. The save file is random access so results are quickly displayed no matter how large the simulation. A tutorial example is provided for demonstrating the debugging features (see “Overview for Debugging FPGA and ASIC Design” on page 2-1). The debugging capabilities include:

- Single stepping through source code and setting breakpoints assists with the quick discovery of design flaws.
- Drag and drop from the hierarchical Explorer to the Data Analyzer and Watch windows provides easy access to the simulation results.
- Unlimited traceback for gate designs quickly isolates the cause of Unknown levels.
- A graphical Finite State Machine entry, source code generation, documentation, and debugging tool.
- Code Coverage reporting for Line reports and Operator reports display a purple dot beside any line or operator that was not executed by the testbench.

# Installation

## 1.2 Installation

### 1.2.1 How to Install Silos for Microsoft Windows

#### 1.2.1.1 Minimum System Requirements for the PC:

- 1) P5, or P6 Personal Computer.
- 2) Microsoft Windows 98, Windows 2000, or Windows NT (version 4.0 or higher).
- 3) At least 16 MB available memory recommended.
- 4) The Silos installation requires less than 40 Mb of disk space. The amount of disk needed during simulation will vary based on the size of the circuit and its activity.

#### 1.2.1.2 Materials Required:

- 1) Silos CD-ROM.
- 2) Silos security block with serial number. (The security block is for the parallel port.).
- 3) Silos license file.

#### 1.2.1.3 Windows 98, Windows 2000, and Windows NT Installation:

To install Silos, do the following steps:

- 1) Insert the Silos CD-ROM into the CD-ROM drive.
- 2) Using the Windows Explorer, double-click on file “SETUP.EXE” on the CD-ROM.
- 3) Answer the questions when prompted by the Silos installation program. For the installation directory you can use any valid name. The installation will attempt to create the directory if it does not already exist.

(continued on next page)

## Installation

### (Windows 98, Windows 2000, and Windows NT Installation)

4) To install the new security code that you obtained from Simucad:

- During installation of Silos from the CD-ROM, the installation will ask you if you would like the installation script to set the SIMUCAD\_LICENSE\_FILE environmental variable path to the “silos.lic” file in the installation directory. Click on “yes”. For Windows 98, the SIMUCAD\_LICENSE\_FILE environmental variable path is set in the autoexec.bat file, i.e.:

```
SET SIMUCAD_LICENSE_FILE=C:\Silos\SILOS.LIC
```

For Windows NT, the SIMUCAD\_LICENSE\_FILE environmental variable path is set in the Environment tab for the System applet in the Control Panel. For Windows 2000, the SIMUCAD\_LICENSE\_FILE environmental variable path is set in the Environment button for the Advanced tab for the System applet in the Control Panel.

- If the security code that you received from Simucad on the "SIMUCAD Silos User Certificate" does not start with the keyword “SERVER”, then this is a node locked license. You can enter the security code for a floating license into any file (the suggested file is “silos.lic” in the installation directory for Silos). You do not need to start a server. An example security code would be:

```
FEATURE Sse simucad 2000.1 11-dec-1999 uncounted A5E6FAFF504E \
HOSTID=SIMUCAD=701024 ck=192
```

During installation, the environmental variable SIMUCAD\_LICENSE\_FILE is set to point to the “silos.lic” file in the installation directory for Silos. If you put the security code in any other file or directory, then you will need to set the environmental variable SIMUCAD\_LICENSE\_FILE to point to the file with the security code.

Note: If you have more than one security block and you want to freely move the security blocks between machines, you can concatenate the security codes for all of the licenses into a single file. The license file can be anywhere as long as it is visible to each user’s machine. Set the SIMUCAD\_LICENSE\_FILE environmental variable on each machine to point to the license file.

Note: You can put the security license for more than one version of Silos in the same “silos.lic” file, and FLEXlm is smart enough to search all of the FEATURE lines to find the correct license.

**(continued on next page)**

## Installation

### (Windows 98, Windows 2000, and Windows NT Installation)

(To install the new security code that you obtained from Simucad: continued)

- If the security code that you received from Simucad does start with the keyword “SERVER”, then this is a floating license. You can enter the security code for a floating license into the file “silos.lic” in the installation directory for Silos (or any other file and point the SIMUCAD\_LICENSE\_FILE environmental variable to the file). An example security code would be:

```
 SERVER yourhost 0020af0c4e31 27009  
 VENDOR simucad  
 FEATURE Sse simucad 2000.1 01-jul-2000 3 6639804D2588 ck=125
```

To start a floating license, the license server must be running Windows NT or Windows 2000. To start the floating license, open a DOS command style window. Change directories to the installation directory for Silos. Then enter the following command at the DOS prompt:

```
 lmgrd -c path_to_Silos_license\silos.lic -l flexlm.log
```

Where “path\_to\_Silos\_license” is the path to the silos.lic security license from Simucad. If you have problems, you can review the flexlm.log file for diagnostic messages, and you can use the “lmdiag” program to run diagnostics on the license.

(continued on next page)

# Installation

## (Windows 98, Windows 2000, and Windows NT Installation)

### 1.2.1.4 Finding the Computer Name and HOSTID for PC Computers

To obtain the FLEXlm hostid for a security code that uses the hostid for your PC computer instead of a security block, open a DOS style Command Prompt window on your PC, "cd" to the installation directory for Silos, and enter "lmutil lmhostid" at the system prompt for your PC.

One method to obtain the computer (network) name for a machine is to open the Control Panel by selecting "Start/Settings/Control-Panel". In the Control Panel, double-click on the Network applet. Then read the "-Computer Name" on the Identification-tab in the Network dialog box.

### 1.2.1.5 Finding the Size of RAM Memory for PC Computers

One method to obtain the size of RAM memory for a PC computer is to open the Control Panel by selecting "Start/Settings/Control-Panel". In the Control Panel, double-click on the System applet. For the "General" tab, read the RAM memory listed under the "Computer" heading.

### 1.2.1.6 Accessing Unix Floating Licenses from PC Computers

If you have purchased a floating license for your Unix computer and you wish to checkout a license from your PC computer that is on the same network:

- Install Silos on your PC computer (if you have not already installed Silos). The "SIMUCAD\_LICENSE\_FILE" environmental variable should be pointing to the silos.lic file in the Silos installation directory.
- In the Silos installation directory on your PC computer, rename the silos.lic file to silos.lic.old, just in case you have a node locked license on your PC and you want to keep a copy of it.
- Copy the silos.lic floating license file from your Silos installation directory on Unix to the Silos installation directory on the PC computer.
- When you run the sse.exe or the silos.exe on the PC computer, it will automatically use the information in the silos.lic file to access across the network the floating license on the Unix computer. If there is a problem, verify if your PC can access the Unix machine using another method (ftp, ping, etc.), and check that the "HOSTS" file is correct in the \winnt\system32\drivers\etc directory.

(continued on next page)

# Installation

## (Windows 98, Windows 2000, and Windows NT Installation)

### 1.2.1.7 Running Silos

To start the interactive version of Silos, use the Windows “Start” menu, then select Programs to access the Silos group, and select Silos.

To put the Silos icon on the desktop, use the Windows Explorer to navigate over to the Silos installation directory. Highlight file “Sse.exe” using the Windows Explorer. From the File menu for the Windows Explorer, select “Create Shortcut”. This will create a shortcut to the Sse.exe in the Windows Explorer, which you can then drag and drop onto the desktop.

To run logic simulation in the batch mode, open a DOS style window and enter “silos” at the command prompt. For information on running Silos in the batch mode, see “Windows Batch Execution” on page 7-10.

### 1.2.2 FLEXlm Installation Problems

If you need to correct a problem with the installation, the error message returned from FLEXlm is important:

- If the error message states “License Error: Invalid host”, click on the OK button for the error message. The User Registration dialog box will then appear. If the Serial # in the User Registration dialog box is “0”, then the problem is FLEXlm cannot access the security block. Try the following to correct the problem:
  - Install the security drivers for FLEXlm from the Drivers directory on the Silos CD-ROM.
  - Ensure the parallel port is working by printing from it, or trying another program that uses a security block.
  - Ensure the Silos security block works by trying it with Silos on another computer, or trying another Silos security block on your computer.

(continued on next page)

# Installation

## (Windows 98, Windows 2000, and Windows NT Installation)

### (FLEXlm Installation Problems)

- Ensure that nothing else on the parallel port is interfering with the Silos security block, such as the security block from another program that is also attached to the parallel port. If this is the problem, you can resolve it by installing another parallel port (Silos automatically searches all of the parallel ports for the security block), or installing a straight through cable or switch box to make it easier to switch security blocks.
- If the error message states “License Error: No such feature exists”, notice the “License path:” entry for the path to the license file. Try the following to correct the problem:
  - Ensure the path for the license file is correct and the license file exists. For Windows 98, the SIMUCAD\_LICENSE\_FILE environmental variable path for the “silos.lic” file is set in the autoexec.bat file. For Windows NT, the SIMUCAD\_LICENSE\_FILE environmental variable path for the “silos.lic” file is set in the Environment tab for the System applet in the Control Panel. For Windows 2000, the SIMUCAD\_LICENSE\_FILE environmental variable path is set in the Environment button for the Advanced tab for the System applet in the Control Panel. You may need to check that the SIMUCAD\_LICENSE\_FILE environmental variable is set in both the System environment and the User environment for the System Applet.
  - Ensure that the requested feature, such as Feature: Sse”, is in the license file.
- If the error message states “License Error: Invalid (inconsistent) license key”, try the following to correct the problem:
  - Ensure that the security code from the License Certificate from Simucad was correctly entered into the license file.
  - For a floating license, you may have used the lmgrd.exe program from an older release. Make sure you are using the lmgrd.exe program from the current release directory.
  - If the error message states “the ordinal “xyz” could not be located in the dynamic link library dll\_file\_name.dll”, try the following to correct the problem:
    - Make sure that you have administrator privileges when installing Silos so that you have permission to overwrite the .dlls. You will also have to reboot the system to have the .dlls updated.

# Solaris

## 1.2.3 Solaris Installation Instructions

### 1.2.3.1 Material Required:

- 1) Silos CD-ROM.
- 2) Silos security code.

### 1.2.3.2 Silos CD-ROM Loading

Place the CD-ROM into the CD-ROM holder for your machine and close the CD-ROM door. For Solaris, the CD-ROM is automatically mounted.

To run the installation script on Solaris, you will need to enter the following commands at the Unix prompt:

```
cd /cdrom/unix  
./install.
```

The installation script will guide you through the process of installing Silos. For the installation directory you can use any valid name without embedded spaces. The installation will attempt to create the directory if it does not already exist.

(continued on next page)

## Solaris

### 1.2.3.3 Silos Security File

After the installation from the install script is complete, you may need to do the following:

- Change directories to the installation directory for Silos.
- Use the security information provided by Simucad to create file “silos.lic” in the installation directory.

If the security information does have “SERVER” lines, then this is a floating license. To test to see if FLEXlm is running, enter “bin/lmstat” at the Unix prompt from the Silos installation directory. If FLEXlm is not running, then you will need to run file “bin/startflexlm” in the installation directory for Silos as a root user.

If you try to start FLEXlm and it is not running when you enter “bin/lmstat”, then review the bin/flexlm.log file to see if there are any problems.

If security information is not provided by Simucad, then you may have to contact Simucad, and provide your hostid and machine name to have a new file created for you.

### 1.2.3.4 Hostid and Hostname for Unix

To find out your host id (in hex) on the Solaris or Linux computer, enter the following command at the Unix prompt:

hostid

To find out your host name on the Sun or Linux computer, enter the following command at the Unix prompt:

hostname

To find out your host id (in decimal) on the HP computer, enter the following command at the Unix prompt:

uname -l

(continued on next page)

### **1.2.3.5 Available Memory and CPU Speed for Solaris**

To find out the amount of memory available on a Sun computer running Solaris, enter the following command at the Unix prompt:

```
dmesg
```

The available memory will be reported along with other statistics. For example, the below entry shows this Sun has 3 gig of available memory:

```
avail mem = 3183763456
```

Memory usage during logic simulation is different for every circuit and set of vectors. Contact your local distributor or Simucad for benchmark data on circuit size and memory usage.

To see how much memory Silos is using during preprocessing, the easiest way would be to enter "ps -el" periodically at the Unix prompt. On the "ps" output from Unix, there will be a line with "silos" in the "COMD" column. On that line, write down the number under the "SZ" column. For some Solaris computers, every 6000 units in the "SZ" column corresponds to about 50Meg, i.e.  $\text{memory-in-use} = \text{SZ} * 50 / 6000$ , where the resulting number is in units of MB. For example, if "SZ" is 109958 then  $\text{memory-in-use} = 109958 * 50 / 6000 = 916\text{MB}$ .

You can find out the speed of your Sun computer by entering the following command at the Unix prompt:

```
/usr/sbin/psrinfo -v
```

### **1.2.3.6 Running Silos**

To run the command line version of Silos for logic simulation, enter:

```
runsilos
```

The Silos copyright notice, version number, etc. will appear on your screen. You can then enter Silos commands at the "Ready:" prompt for inputting files, simulating, etc. (see "Inputting SILOS Commands" on page 5-1, or "Unix Batch Execution" on page 7-12).

To run the Graphical User Interface (GUI) version of Silos for logic simulation, enter:

```
runsse
```

# Linux

## 1.2.4 Linux Installation Instructions

### 1.2.4.1 Material Required:

- 3) Silos CD-ROM.
- 4) Silos security code.

### 1.2.4.2 Silos CD-ROM Loading

Place the CD-ROM into the CD-ROM holder for your machine and close the CD-ROM door.

To run the installation script on Linux, you will need to be the “**root**” user. One method of installing Silos is to bring up the Linux installation GUI by entering the following command at the Unix prompt:

```
gnormp
```

In the “Gnome RPM” dialog box select the Packages/Applications/Engineering menu. Next click on the “Install” button to bring up the Install dialog box. Click on the “Add” button in the “Install” dialog box to bring up the “Add Packages” dialog box. Navigate in the Directories box for the “Add Packages” dialog box until you get to the “/mnt/cdrom/Unix” directory on the Silos CDROM. Select “silos.rpm” in the right hand window of the “Add Packages” dialog box. Click on the “Add” button in the “Add Packages” dialog box, and click on the “Close” button in the “Add Packages” dialog box after the Silos application icon is added beneath the Packages/Applications/Engineering menu in the “Install” dialog box. Click on the “Install” button in the “Install” dialog box. An “Installing” dialog box will appear that has progress bars for the installation. When the installation is complete, you will see the Silos application icon in the main window for the “Gnome RPM” dialog box. Now you can exit “Gnome RPM” by clicking on the “x” in the upper right hand corner. Silos has now been installed in the “/usr/local/simucad” directory on your computer.

(continued on next page)

# Linux

### 1.2.4.3 Silos Security File

After the installation from the install script is complete, you may need to do the following:

- Change directories to the installation directory for Silos.
- Use the security information provided by Simucad to create file “silos.lic” in the installation directory.

If the security information does have “SERVER” or “DEAMON” lines, then this is a floating license. To test to see if FLEXlm is running, enter “bin/lmstat” at the Unix prompt. If FLEXlm is not running, then you will need to run file “bin/startflexlm” in the installation directory for Silos as a root user.

If you try to start FLEXlm and it is not running when you enter “bin/lmstat”, then review the bin/flexlm.log file to see if there are any problems.

If security information is not provided by Simucad, then you may have to contact Simucad, and provide your hostid and machine name to have a new file created for you.

### 1.2.4.4 Hostid and Hostname for Linux

To find out your host id (in hex) on the Linux computer, change directories to the “bin” subdirectory of the Silos installation, and enter the following command at the Unix prompt:

`lmutil lmhostid`

To find out your host name on the Linux computer, enter the following command at the Unix prompt:

`hostname`

To find out which version of Linux you are running on a Linux computer, enter the following command at the Unix prompt:

`uname -a`

(continued on next page)

# Linux

## 1.2.4.5 Running Silos

To run the command line version of Silos for logic simulation, enter:

```
runsilos
```

The Silos copyright notice, version number, etc. will appear on your screen. You can then enter Silos commands at the “Ready:” prompt for inputting files, simulating, etc. (see “Example for Batch Logic Simulation” on page 2-95. See also “Inputting SILOS Commands” on page 5-1, or “Unix Batch Execution” on page 7-12).

To run the Graphical User Interface (GUI) version of Silos for logic simulation, enter:

```
runsse
```

# Technical Support

## 1.3 Accessing Technical Support

Simucad offers technical support to customers on maintenance. It is most efficient to contact Simucad's technical support via e-mail:

**support@simucad.com**

When sending problem circuits to Simucad, please send the complete circuit that can be run to duplicate the problem, including any libraries and the “.spj” file. Also enclose a few sentences that precisely state the problem and how to run the circuit, for example:

At time=200, node “top.out1” should be high instead of unknown. To run the circuit, use project “problem.spj”.

From a PC system, the following PKZIP commands may be the simplest way to e-mail a circuit:

```
pkzip -Pr foo.zip directory_name\*.*
```

Then e-mail support@simucad.com and attach the file using uuencode.

From a Unix system, the following Unix commands may be the simplest way to e-mail a circuit:

```
tar cvf foo file1 file2 ...
compress foo
uuencode foo.Z < foo.Z > foo
```

Edit file “**foo**” to briefly describe the problem, then enter:

```
mail support@simucad.com < foo
```

Simucad's mailing address, phone number, and fax number are:

Simucad, Inc.  
32970 Alvarado-Niles Road  
Union City, CA. 94587  
Phone: (510) 487-9700x206  
Fax: (510) 487-9721

## 2. Tutorial

### 2.1 Overview for Debugging FPGA and ASIC Designs

#### Benefits

- The Silos logic simulation plug-ins provide a low cost solution for quickly debugging FPGA or ASIC designs that are written using Verilog HDL:
- Silos's ability to display any variable without re-simulating makes the user interface easy to learn and saves invaluable development time by not interrupting the designer's concentration during debugging.
- The intuitive debugging environment is made possible because the simulation algorithms are optimized to save information quickly to disk. Other simulators may run fast only when they save nothing.
- The superior debugging capabilities of Silos can save valuable design time even though designers may already have access to regression style Unix simulators, because debugging the RTL code can take 60% to 70% of the total design time.
- Silos is available on Windows so that the designer can use the faster and more cost effective PC platforms. Silos is also available on Unix.

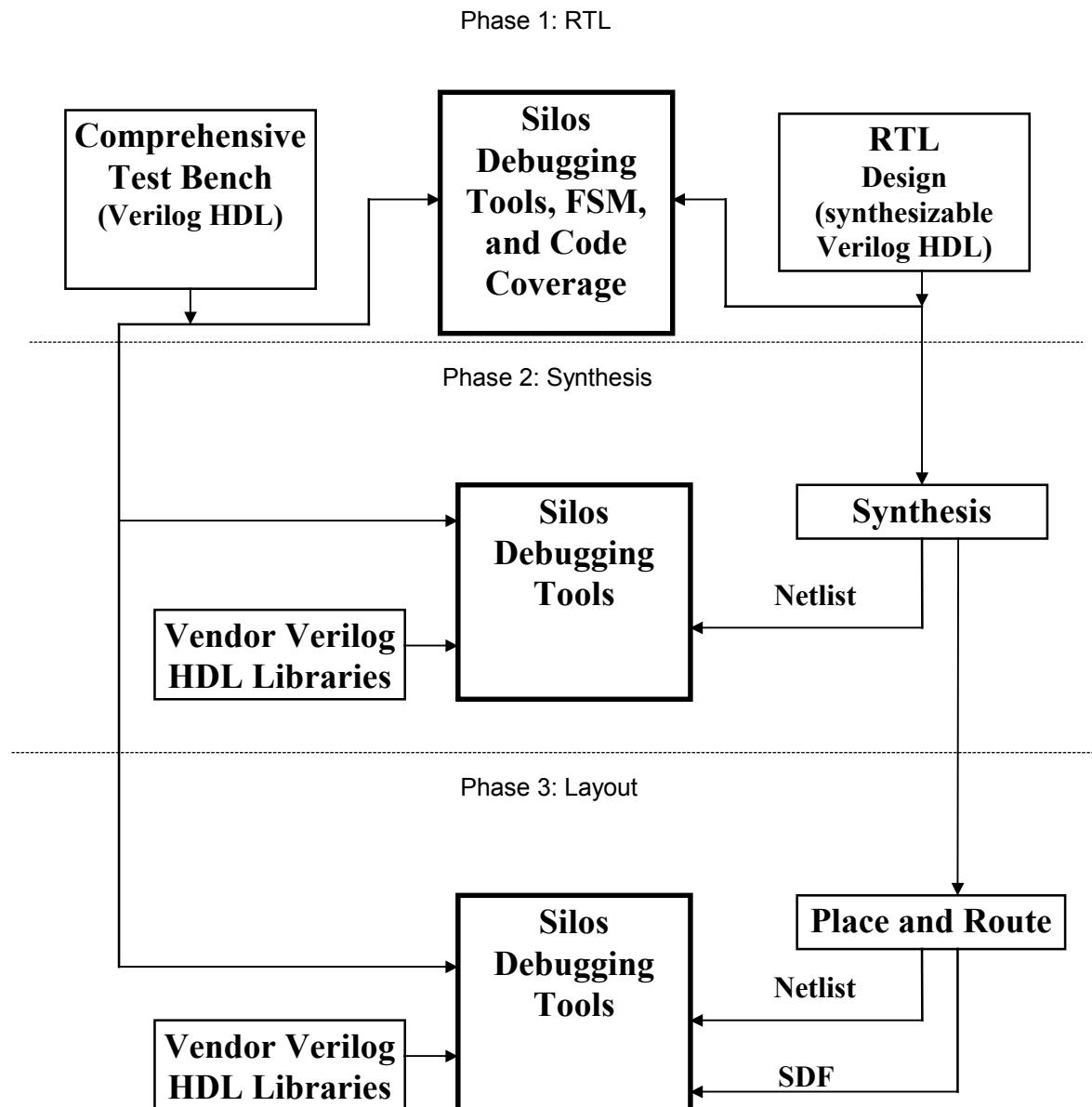
Silos has a variety of highly integrated tools to assist with debugging behavioral designs:

- Single stepping through your source code.
- Drag and drop variables directly from your source code into the Data Analyzer or a Watch window.
- Breakpoints so that you can conveniently skip over uninteresting source code.
- Explorer Window which displays the design's hierarchy in a convenient tree structure.
- Code Coverage reporting for Line reports and Operator reports. Double clicking on an entry in these reports opens the source window and displays a purple dot beside any line or operator that failed to execute.
- Silos provides a built in Finite State Machine entry tool that can be used to enter finite state machines, generate the Verilog HDL source code, and simulate and graphically debug the finite state machines.

#### Procedure

The purpose of this Tutorial is to demonstrate the features of Silos that can be used to debug a FPGA or an ASIC design. The general design flow for designing a FPGA or an ASIC is described [on the pages that follow](#). The Verilog HDL libraries for the FPGA or ASIC design can be obtained from the respective vendor.

## Example FPGA / ASIC Design Flow



## Example FPGA / ASIC Design Flow

### Phase 1:

- Create a Register Transfer Level (RTL) behavioral description using Verilog HDL to model the structure of the design. Depending on how complex the total design is, you may want to first create a Verilog HDL behavioral design to model the general functionality for your design (optional). You can use the Finite State Machine (FSM) entry tool for Silos to create and document finite state machines.
- Create a test bench that models the interface between your design and the total system.
- Run exhaustive tests using Silos to debug and verify that the RTL design is correct. The RTL description and test bench are usually created by using a text editor or a high level tool. For many designers, 60% to 70% of the total project time is spent developing an exhaustive test bench and debugging the RTL design. Code coverage can be used to verify the testbench fully exercises the behavioral design.

### Phase 2:

- Synthesize the RTL design into a Verilog netlist. The synthesis tool will also require constraint files for area, timing, etc.
- Generate a Verilog HDL netlist from the synthesis tool.
- Simulate the netlist, test bench, and the vendor's Verilog HDL libraries with Silos. Use the debugging tools provided with Silos to debug the netlist. Ensure that the synthesized design gives the same results as the RTL design.

### Phase 3:

- Input the synthesis tool's output into the vendor's tool kit to perform the place and route on the design. The vendor's tool kit will generate a Verilog HDL netlist, and an SDF file to back annotate the delays for post place and route simulation with Silos.
- Use Silos to simulate and debug the routed gate level design with the test bench for the RTL design. If the synthesized or routed design does not perform as expected, you can use the trace back feature for Silos to trace the problem back through the topology to its cause.

# Tutorial

## 2.1.1 General Items for Tutorial

The flow for running the Tutorial assumes that you will follow the topics in the order they are listed. If you skip some topics then you may not have performed the necessary steps for each topic.

The files used for the Tutorial examples can be found in the “examples” subdirectory for the Silos installation directory. **For information on additional examples provided in the “examples” subdirectory, see the README file in the installation directory.**

### Please note:

- Actions that you should do to run this tutorial are in **bold**.
- “**Click-on**” means place the mouse cursor on the appropriate item and click and release on the item using the left mouse button.
- You can see the tool tips (text labels) for each of the buttons on the Toolbars for the Silos by **placing** the mouse cursor over a button for a few seconds until the text label for the button appears. Then **move** the mouse along the Toolbar and stop at each button to see the text label. There can be more than one Toolbar (see “Main Toolbar menu selections” on page 3-10, “Analyzer Toolbar menu selections” on page 3-10, and “FSM Toolbar menu selections” on page 3-10). The location of each Toolbar on the screen can be changed by using the mouse to grab an edge of a toolbar and dragging the toolbar to the desired location.
- The menus for the Silos change depending on which window has the focus, such as the Data Analyzer Window has different menus from the Output Window.

# Debugging

## 2.2 RTL (Behavioral) Debugging

Skills presented in this topic are:

- Setting up a project.
- Starting Silos and accessing the on-line help.
- Running a simulation and viewing the waveforms with the Data Analyzer Window.
- Viewing bits to a vector, and using symbolic names to display the vector's value.
- Setting up ASCII vectors to create a text timeline of the simulation progress.
- Organizing signals into groups.
- Scan to value for a waveform.
- Search on condition using a Verilog HDL expression, and viewing the search condition as a waveform.

# Starting Silos

## 2.2.1 Starting Silos

Use the Windows Start menu to start Silos on the PC. If you made a shortcut to the “sse.exe” on the PC (see “Running Silos” on page 1-6), you can **double-click** on the Silos icon to start Silos.

## 2.2.2 Accessing On-line Help

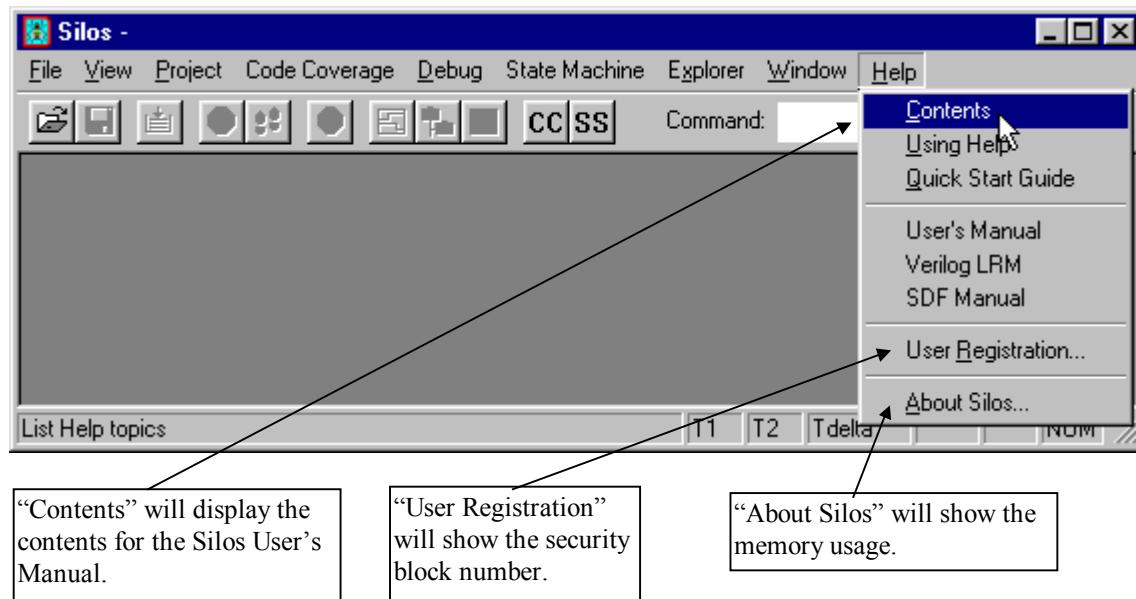
### Benefit

- The complete Silos, Verilog HDL, and SDF manuals available as on-line help files. These manuals are also in the “doc” subdirectory in the Silos installation as “.pdf” files for viewing and printing. This provides easy access and lookup for answers to reference questions.

### Procedure

To access the on-line help for Silos:

- Select** the Help menu selection for Silos.



(continued on next page)

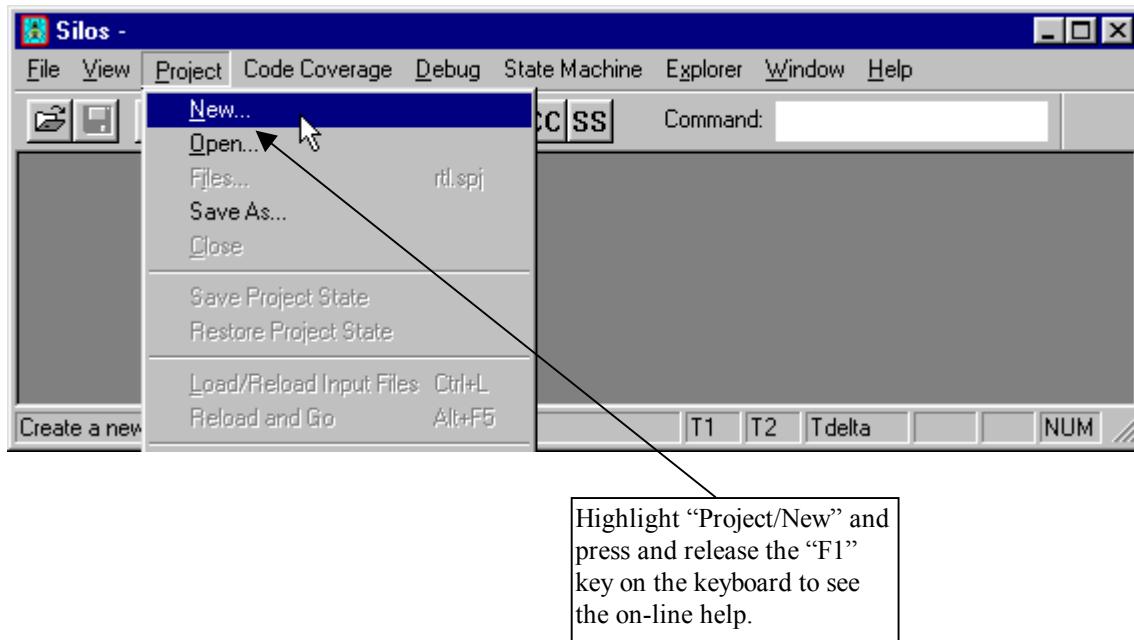
## Memory usage

The following manuals are provided as on-line help files:

- Silos User's Manual (Help/Contents will also select this);
- Open Verilog International (OVI) Verilog Language Reference Manual version 1.0;
- OVI Standard Delay Format (SDF) Manual version 2.0.

To provide an example of obtaining on-line help for a menu selection, the user could locate help on the “Project/New” menu selection in the Menus Chapter (the next item in the tutorial) with the procedure that follows:

- **Click-on** the “Projects” menu selection, and **scroll down and highlight** the “New” menu selection but **do not** release the left mouse button. With the left mouse button still held down, **press and release** the “F1” key and the on-line help entry for Project/New will pop up.



To see the memory usage for Silos:

- **Select** the “Help/About Silos” menu selection to open the “About Silos” dialog box. This box displays the Silos version number and the “Sim Engine Mem” value for the Silos logic simulation engine memory usage. The first number listed for the “Sim Engine Mem” is the memory allocated for the Silos logic simulation engine. The second number listed for the “Sim Engine Mem” is the total amount of memory that Silos thinks it can possibly allocate for your computer.
- **Click-on** the “OK” button to close the dialog box.

## Example Projects

### 2.2.3 Example Projects

The “examples” subdirectory for the Silos installation has example projects to assist the user:

- ANALOG.SPJ Project file for Analog to Digital converter circuit that models the analog portion at the behavioral level and the digital portion at the gate level. To run this example, see “Analog Behavioral Modeling (AHDL)” on page 2-100.
- CODE\_COVERAGE.SPJ Project file for demonstrating code coverage for the Line report and Operator report for Verilog HDL behavioral code. To run this example, see “Code Coverage” on page 2-55.
- CODE\_COVERAGE2.SPJ Project file for merging code coverage results from two different simulations using the same behavioral model and different testbenches. To run this example, see “Code Coverage” on page 2-55.
- FLTSIM.SPJ Project file for circuit that demonstrates the features for fault simulation. To run this example, see "Tutorial/Fault Simulation Examples Overview" in the on-line help for the HyperFault User's Manual.
- GATE.SPJ Project file for circuit that demonstrates traceback at the gate level. Traceback is useful for finding problems after synthesis. To run this example, see “Gate level Debugging” on page 2-81.
- RTL\_.SPJ Project file for circuit that demonstrates features for debugging an RTL design. To run this example, see “RTL (Behavioral) Debugging” on page 2-5.
- RTL\_ERR.SPJ Project file for circuit that shows how to automatically open the source file for displaying a syntax error. To run this example, see “Error reporting” on page 2-97.
- VENDING.SPJ Project file for circuit that demonstrates features for a finite state machine. To run this example, see “Finite State Machine (FSM) Entry” on page 2-66.

## New Project

### 2.2.4 Creating a Project

#### Benefit

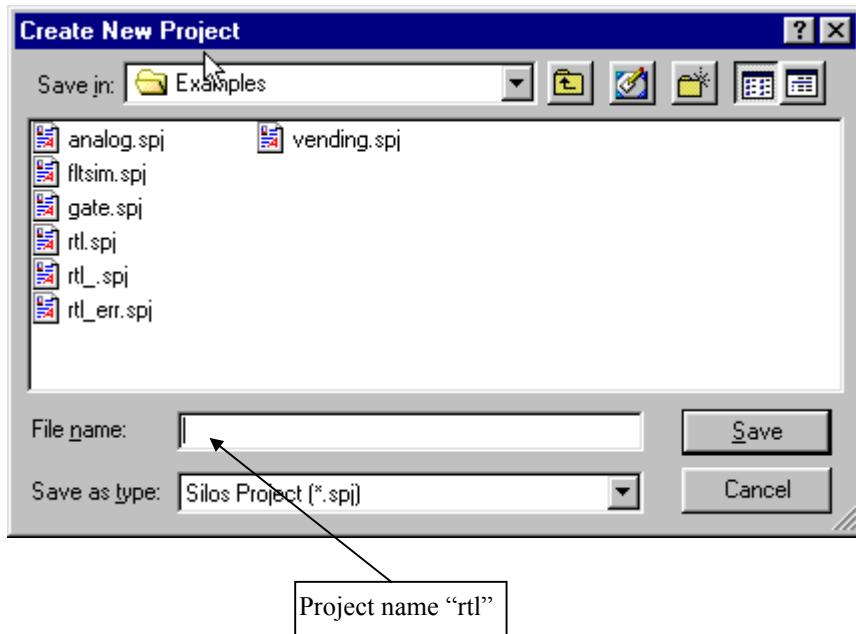
- Projects provide easy access to the input files and libraries for a design.
- Projects can be easily passed between designers on a team by simply providing the project file name.
- All of the settings to run the design and organize the signal names for viewing waveforms are saved by the project, so useful debugging can be immediately started after simulating.

#### Procedure

The circuit for this example is a RTL description of a newspaper vending machine.

The files used for this example are listed [in the README file](#).

This section shows you how to create a new project for inputting your source files. To specify library files, see the “Files Menu Selection” on page 3-13.



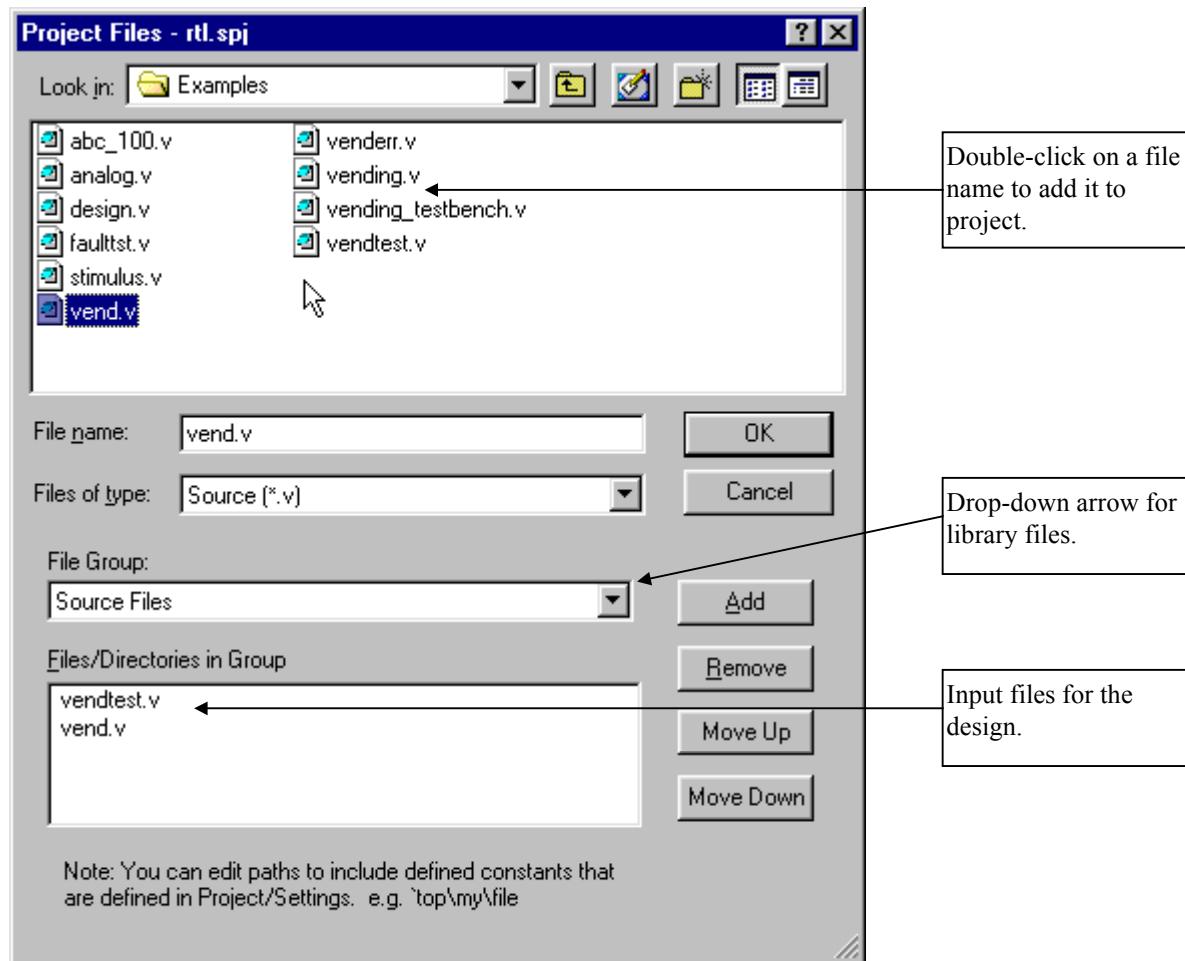
- Select the “Project/New” menu selection to open the “Create New Project” dialog box.
- To change to the “examples” subdirectory of the installation directory, use the drop-down arrow for the “Save in” box and the “Up One Level” button in the “Create New Project” dialog box.

[\(continued on next page\)](#)

## Project Files

- In the “File name” box, **enter “rtl”** and then **click-on** the “Save” button to close the dialog box. The Silos will automatically append the suffix “.spj” to the project name if you do not add a suffix to the project name.

The “Project Files” dialog box will be automatically opened so that you can specify the input files for the project.



To input the files for project “rtl.spj”:

- Double-click** on file “vendtest.v” in the list box for the examples subdirectory to add “vendtest.v” to the “Files in Group” list box .
- Next **select** file “vend.v” in the list box for the examples subdirectory and **click-on** the “Add” button to add it to the “Files in Group” list box..
- Now **click-on** the “Ok” button to close the “Project Files” dialog box.

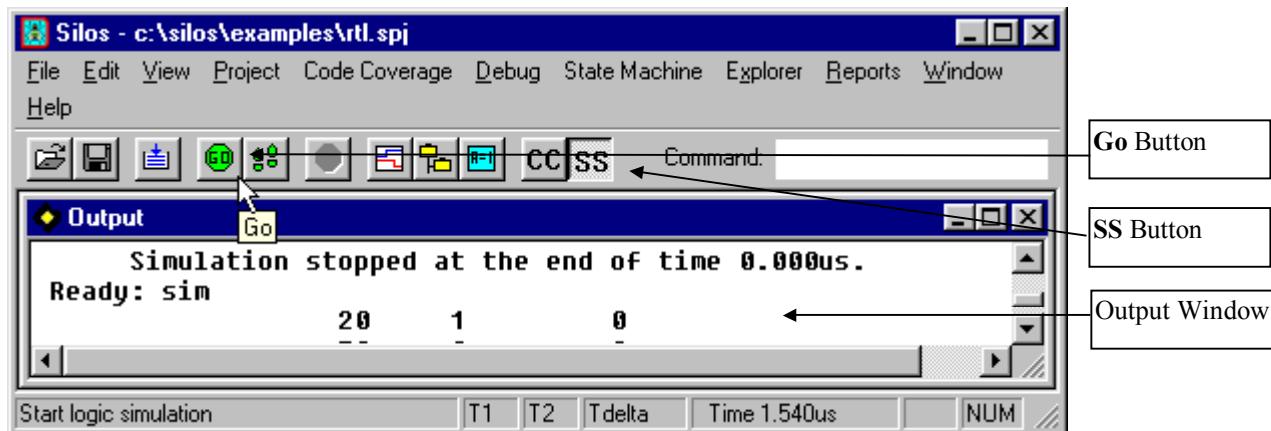
# Starting Simulation

## 2.2.5 Simulating a Design

### Benefit

- The Tool bar buttons have titles so that their function can be easily identified. This reduces training time.
- Clicking on the “Go” button is all that is required to simulate a design, making it simple and easy to start debugging.

### Procedure



- Before starting the simulation, ensure the “SS” button on the Main toolbar is enabled (depressed in) by **clicking on** it if necessary. When the “SS” debug button is enabled, then single stepping and setting breakpoint capabilities are available for this tutorial example. If you do not need to use single stepping and breakpoints for your design, then you should disable the “SS” button to increase the simulation speed for behavioral designs. Next **click-on** the “Go” button on the Toolbar to load the input files and run logic simulation (you can see the text labels for each of the buttons on the Toolbar for the Silos by **placing** the mouse cursor over a button for a few seconds until the text label for the button appears). The logic simulation will run until it encounters the \$finish system task in file “vendtest.v”. You could also have used the “Debug/Go” menu to run the logic simulation.

## 2.2.6 Selecting Signals for Waveforms

### Benefit

- Any variable can be viewed as a waveform without re-simulating, saving time and allowing the designer to debug without interruptions. This is possible because Silos's algorithms quickly and compactly save everything to disk when simulating.
- The design's hierarchy is displayed so that variables can be easily selected.

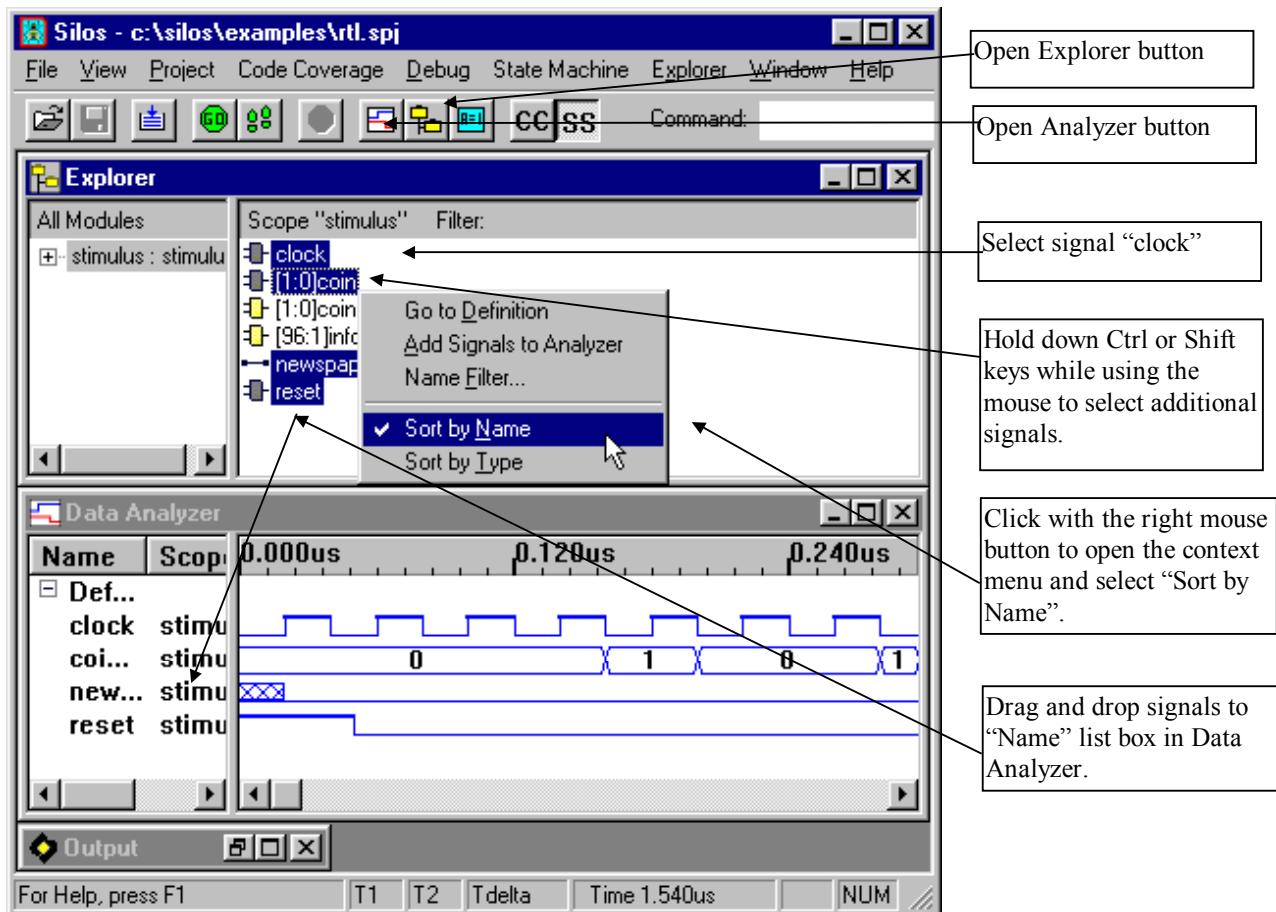
### Procedure

The Silos has the capability to drag and drop variable names from the Explorer Window (and also from any HDL source window) to the Data Analyzer Window. To demonstrate this:

- **Click-on** the “Open Analyzer” button on the Toolbar to open the Data Analyzer Window. For more information on the Data Analyzer Window, see “Open New Data Analyzer Menu Selection” on page 3-49.
- Next **click-on** the “Open Explorer” button on the Toolbar to open the Explorer Window. Use the Window/Tile menu to conveniently align the two windows. You may want to minimize the Output Window first. You may also want to list the variables by selecting the View>List menu for Silos. You can also change the sorting for the signal names by clicking with the right mouse button in the right side of the Explorer to open the context menu, and then selecting “Sort by Name”.

(continued on next page)

## Explorer

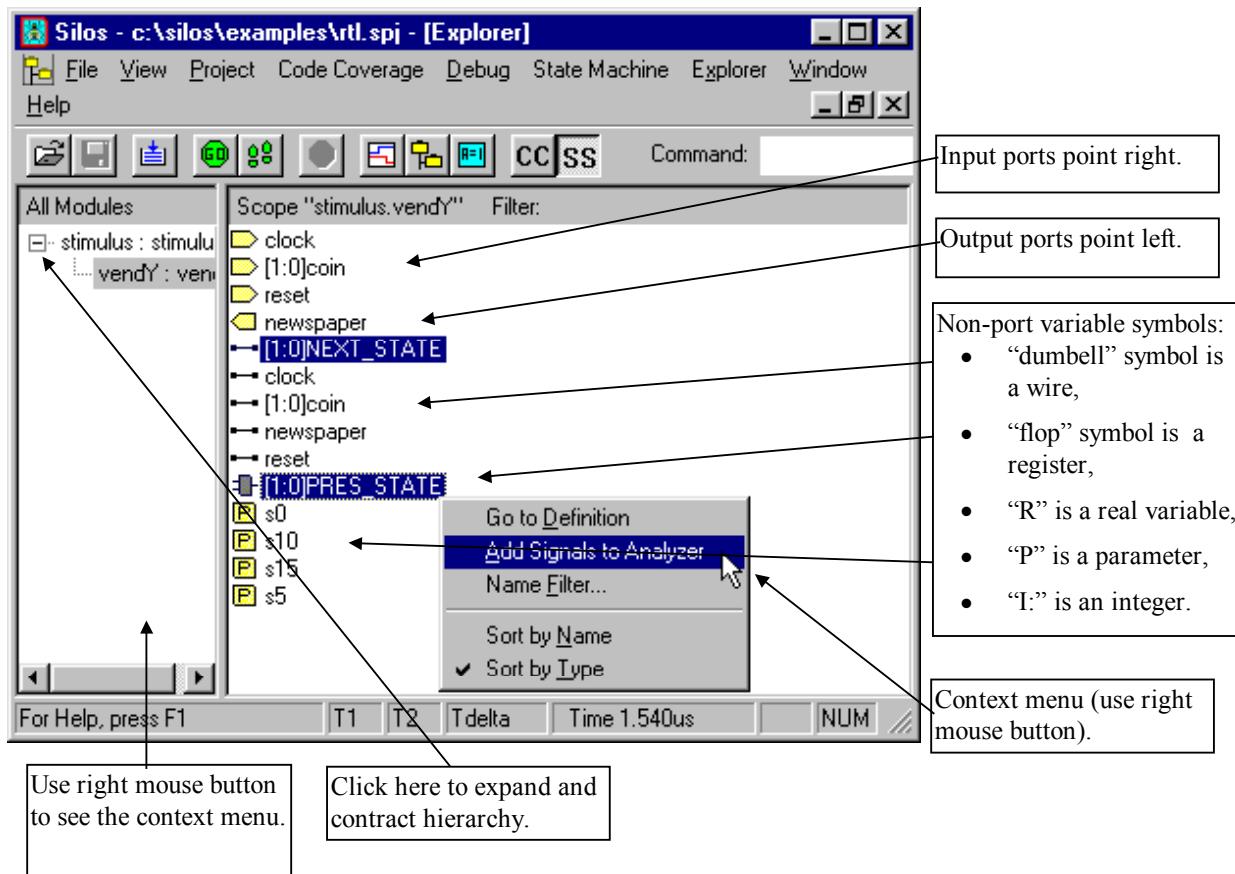


Instance name “stimulus”, in the left-hand box of the Explorer Window, is the top-level module for the design. To select the signal names for instance “stimulus”:

- Use your mouse to **select** instance “stimulus”.
- To highlight the variable names for instance “stimulus”, **click and release** on the first name, “clock”. Next **hold** down the Ctrl key on the keyboard, and then **click and release** on additional signal names, “coin[1:0]”, “newspaper”, and “reset”.
- To drag and drop signal names, **click-on** the highlighted variable names without releasing the left mouse button, and **drag** them from the “Explorer” Window and **drop** them in the “Name” list box in the Data Analyzer Window. The waveforms for each signal will appear.

(continued on next page)

# Explorer



To drag and drop signal names from further down in the hierarchy:

- **Click** on the “+” sign to the left of instance “stimulus” to show the hierarchy beneath it. You can also change the sorting for the signal names by clicking with the right mouse button in the right side of the Explorer to open the context menu, and then selecting “Sort by Type”.
- Then **select** instance “vendY” to show the variable names in the right hand side of the Explorer Window. The symbol to the left of each variable shows its function. Input ports have the pad symbol pointing to the right, output ports have the pad symbol pointing to the left, and inout ports have the pad symbol pointing in both directions. For non-port variables, the symbol for a wire is a wire connecting two points, the symbol for a register is a “flop” symbol, “P” is used for parameters, “R” for real variables and “I” for integer variables.
- **Select** signals “NEXT\_STATE” and “PRES\_STATE” for instance “vendY”, and use the “Add Signals to Analyzer” selection in the context menu for right hand window for the Explorer by **clicking** with the right mouse button in the right hand pane of the Explorer.

## Waveform Colors

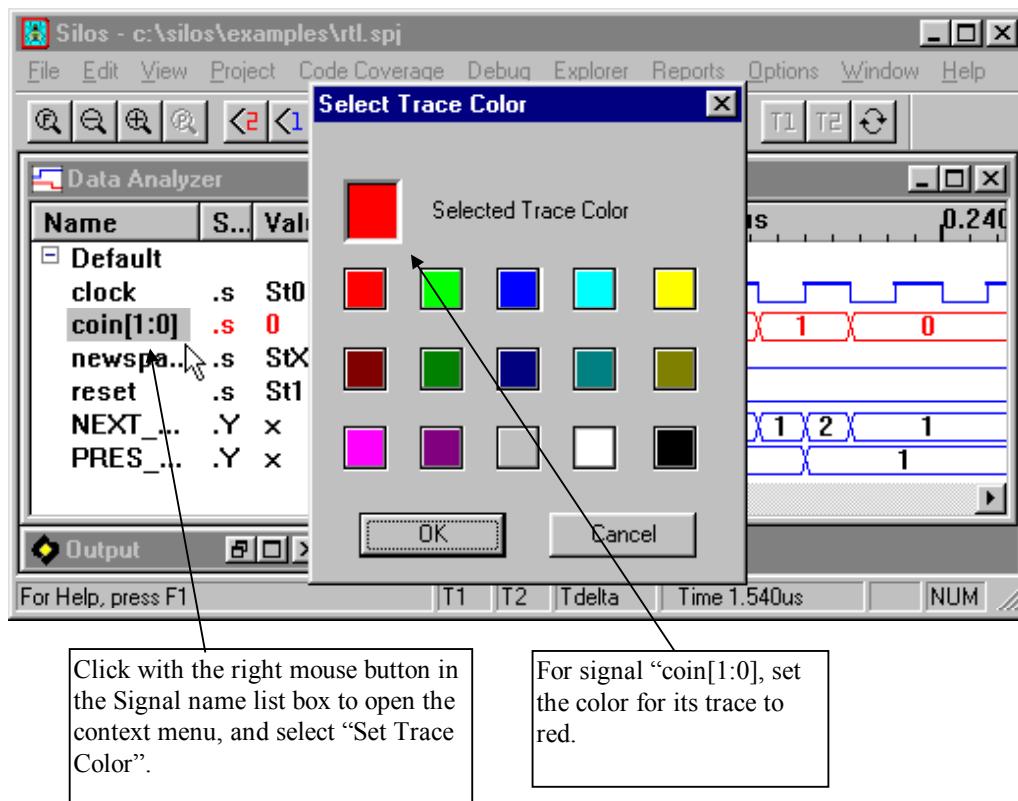
### 2.2.7 Waveform Colors

#### Benefit

- The color can be set for waveforms in the Data Analyzer window. This lets the user easily follow a waveform across the Data Analyzer.
- The background color and signal traces can be set for the Data Analyzer, which can make viewing waveforms easier for the user.

#### Procedure

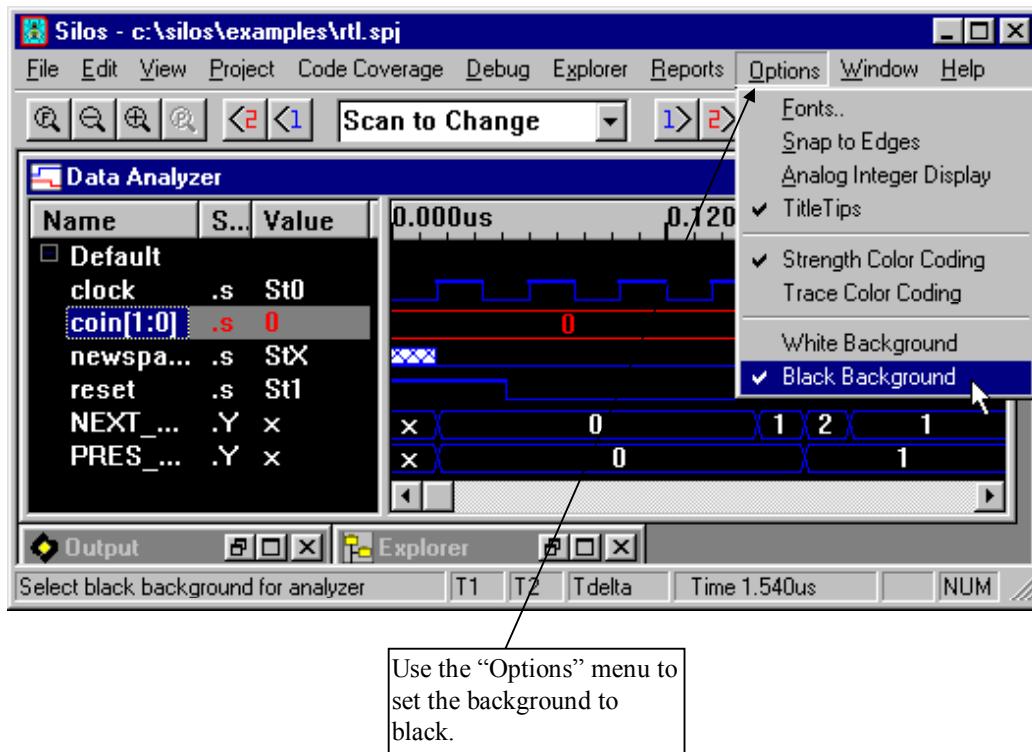
- Select signal “coin[1:0]”, and click with the right mouse button in “Signal Name” list box to open the context menu, and select “Set Trace Color” to open the “Select Trace Color” dialog box. Choose red as the waveform color, and click on the “OK” button to close the dialog box.



(continued on next page)

## Black Background

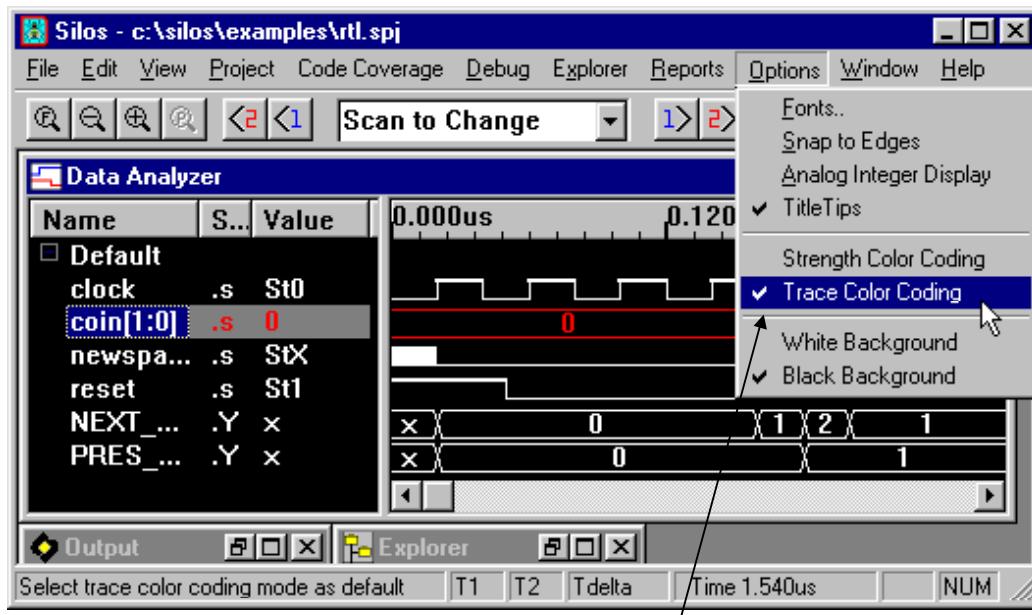
- To change the background color to black, select the “Options/Black Background” menu selection.



(continued on next page)

## Trace Colors

- To change the waveforms so they are simply traces and their color does not denote strength, **select** the “Options/Trace Color Coding” menu selection. For the remainder of the Tutorial, **select** the “Options/White Background” menu selection to set the background back to white, and **select** the “Options/Strength Color Coding” menu selection to reset the strength coloring.



Use the “Options” menu to  
set the waveform traces to  
a default color (white with  
black background).

## Name List Box

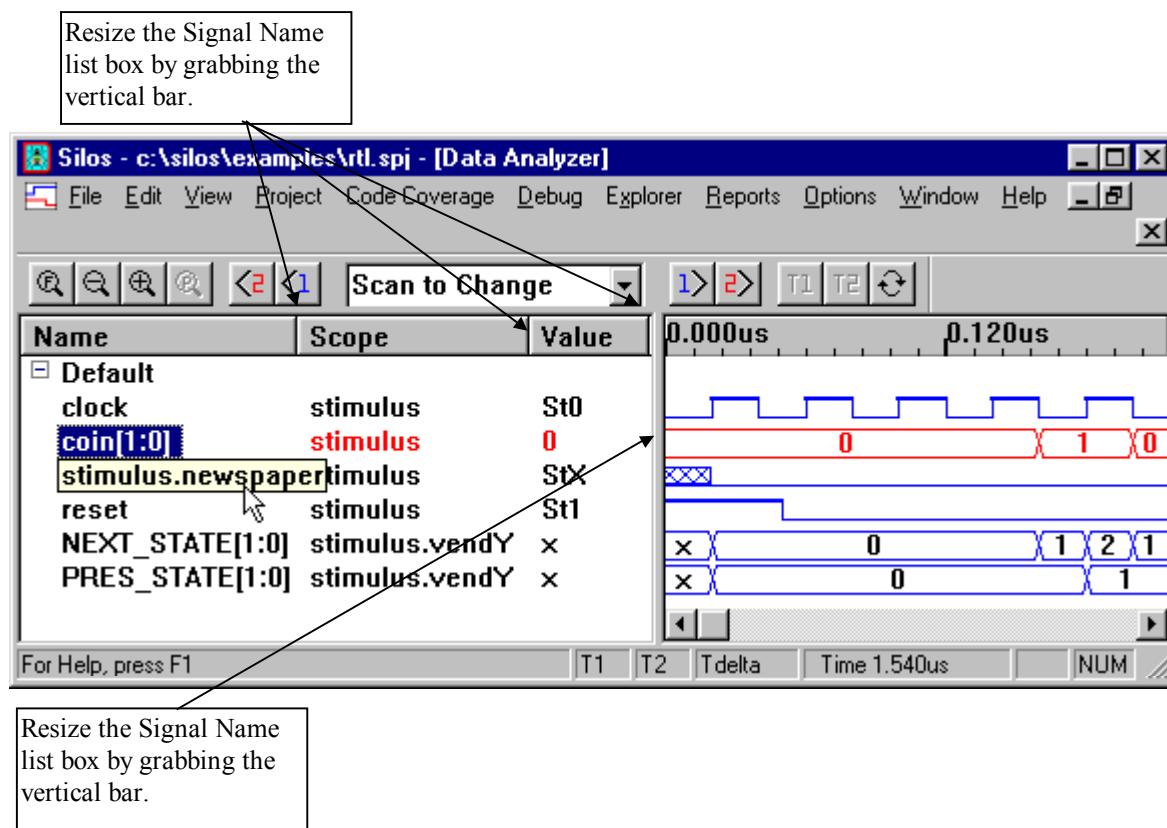
### 2.2.8 Resizing the Name List Box

#### Benefit

- Many designers prefer to maximize the display area for waveforms. Title tips let the designer still see the full path name for signals, making it easier to debug designs when other tools automatically create long path names and signal names.

#### Procedure

Resize the Name list box by using the mouse to **drag** the right vertical edge of the Name list box and **slide** it to the left or the right. The Name, Scope, and Value columns can be resized by grabbing the vertical separator between them. Each of the signal names can have its full hierarchical path (“Title Tips”) displayed when the mouse cursor is held over the name. To turn the “Title Tips” “on” or “off”, select “Options/Title Tips”.



## Copying Waveforms

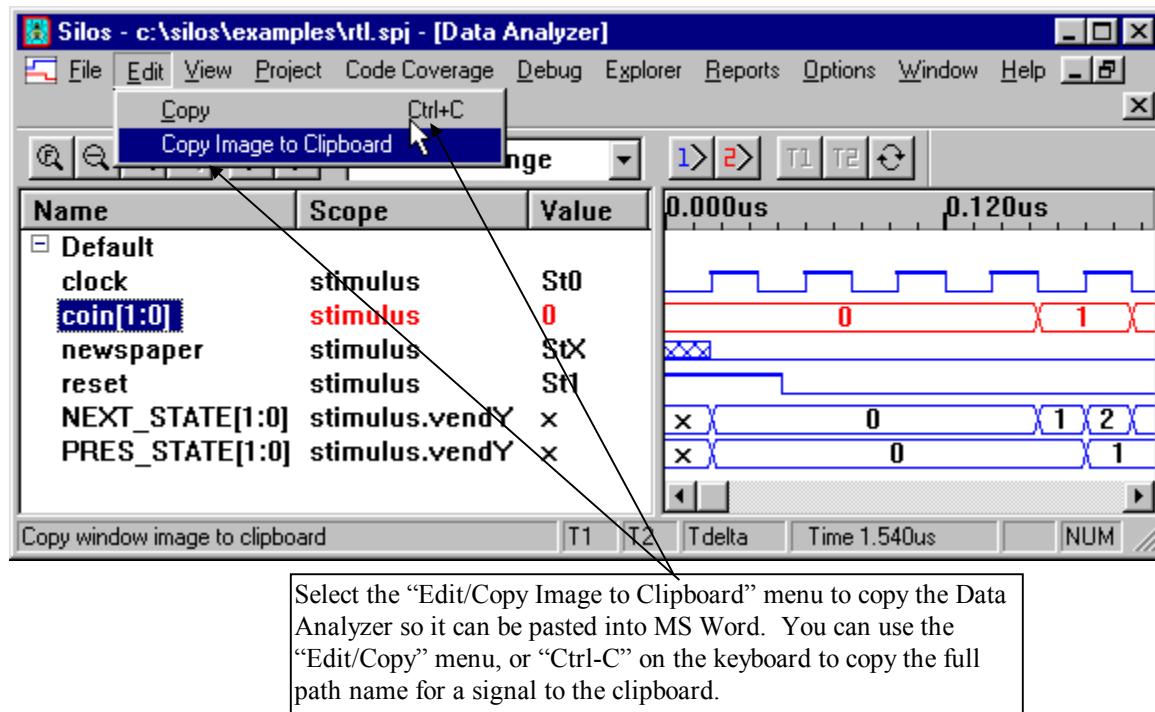
### 2.2.9 Copying Waveforms

#### Benefits

- The user can use the “Edit/Copy Image to Clipboard” menu to copy the Data Analyzer waveform display to the clipboard and paste it into Microsoft Word so the simulation results can be documented.
- The user can use the “Edit/Copy” menu to copy the full path name for a signal from the Data Analyzer to the clipboard and pasted into another document so the name do not have to be entered by hand in the other document.

#### Procedure

- **Ensure** that the Data Analyzer window has the focus.
- **Select** the “Edit/Copy Image to Clipboard” menu selection.
- Then open Microsoft Word and **paste** the Data Analyzer waveform display into a new or existing document.



## Name List Box

### 2.2.10 Rearranging the Signal Names

#### Benefit

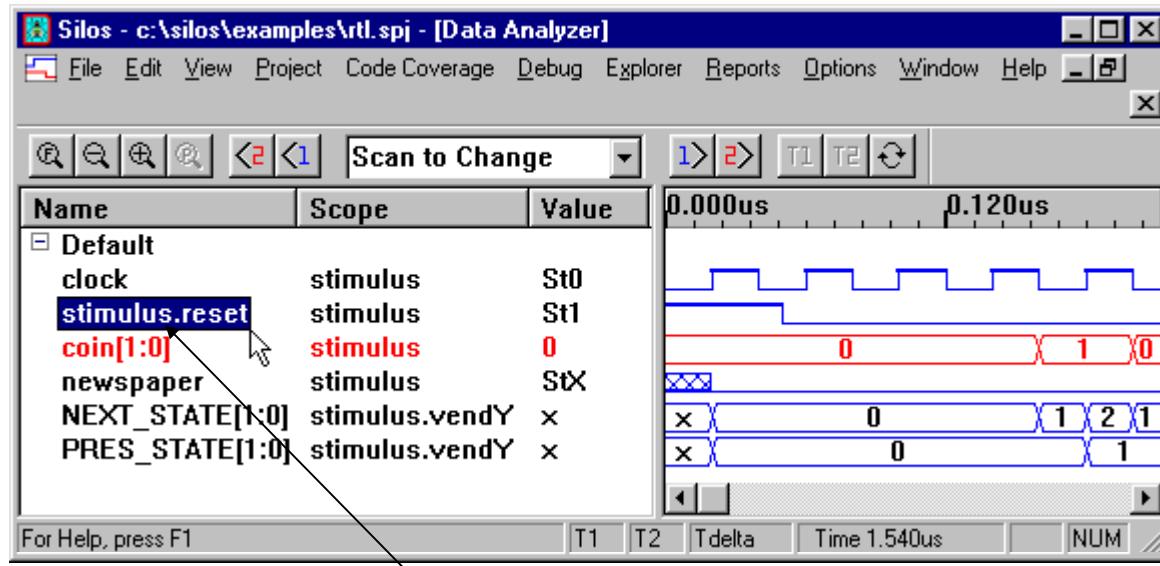
- Intuitive drag and drop when rearranging signals makes it easy to view relationships between signals.

#### Procedure

Signal names can be rearranged in the Name list box by using the mouse to drag the signal and then drop it just before the pointed end of the mouse cursor arrow. For example:

- Select signal “reset” with the mouse.
- Then drag and drop “reset” above signal “coin[1:0]” in the Name list box.

Signal names can be copied by holding down the “Ctrl” key as you drag them. Signal names can be deleted by highlighting a name and pressing the “Delete” key on the keyboard.



Rearrange the signal names by dragging and dropping them in the Signal Name list box.

## Displaying Vectors

### 2.2.11 Expanding/Hiding Bits to Vectors

#### Benefit

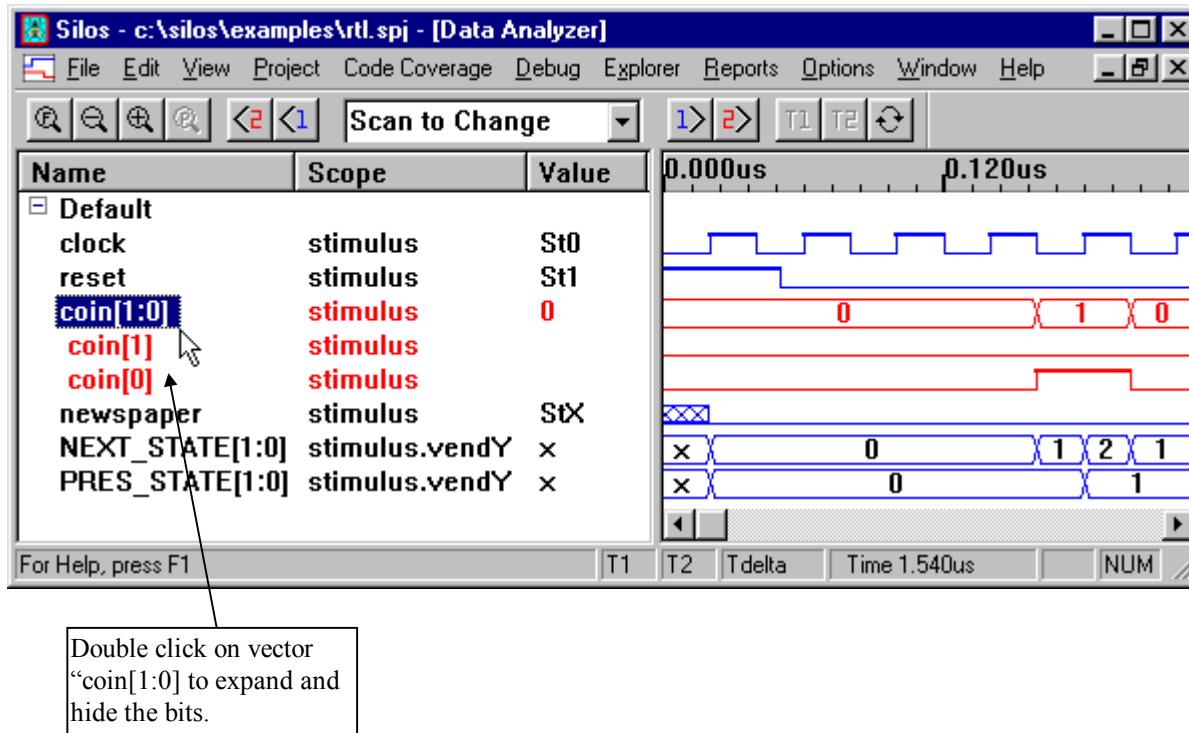
- The designer can display the individual bits for a vector without re-simulating, saving time and eliminating unnecessary interruption of the designer's concentration. This is possible because Silos saves everything quickly and compactly to disk when simulating.

#### Procedure

To expand the individual bits for a vector:

- Double-click** on vector name “coin[1:0]”.
- Double-clicking** on the vector name again will hide the vector signal bits.

To make the signals easier to view, you can add blank lines by using the “Add Blank Line” menu selection in the context menu for the Name list box. For more information on accessing context menus in the Data Analyzer, see “Open New Data Analyzer Menu Selection” on page 3-49 and “Data Analyzer Signal List Box” on page 3-70.



## State Machine Values

### 2.2.12 Displaying Vectors using Symbolic Names

#### Benefit

- Vectors and buses can be displayed with meaningful state machine names, such as “5 cents”. This eliminates constantly looking up the state machine symbol corresponding to a hex value for a vector.

#### Procedure

To display the state machine values for vectors coin[1:0], NEXT\_STATE[1:0], and PRES\_STATE[1:0]:

- Select the “Project/Project Settings” menu to open the “Project Settings” dialog box. Use the Browse button in the “Project Settings” dialog box to open the Select Symbol Table dialog box. Double-click on file “rtl.sym” to set the “Analyzer Symbol Table File:” box.

The format for the symbol table file is a hex value equals symbolic name, i.e:

001b=jump

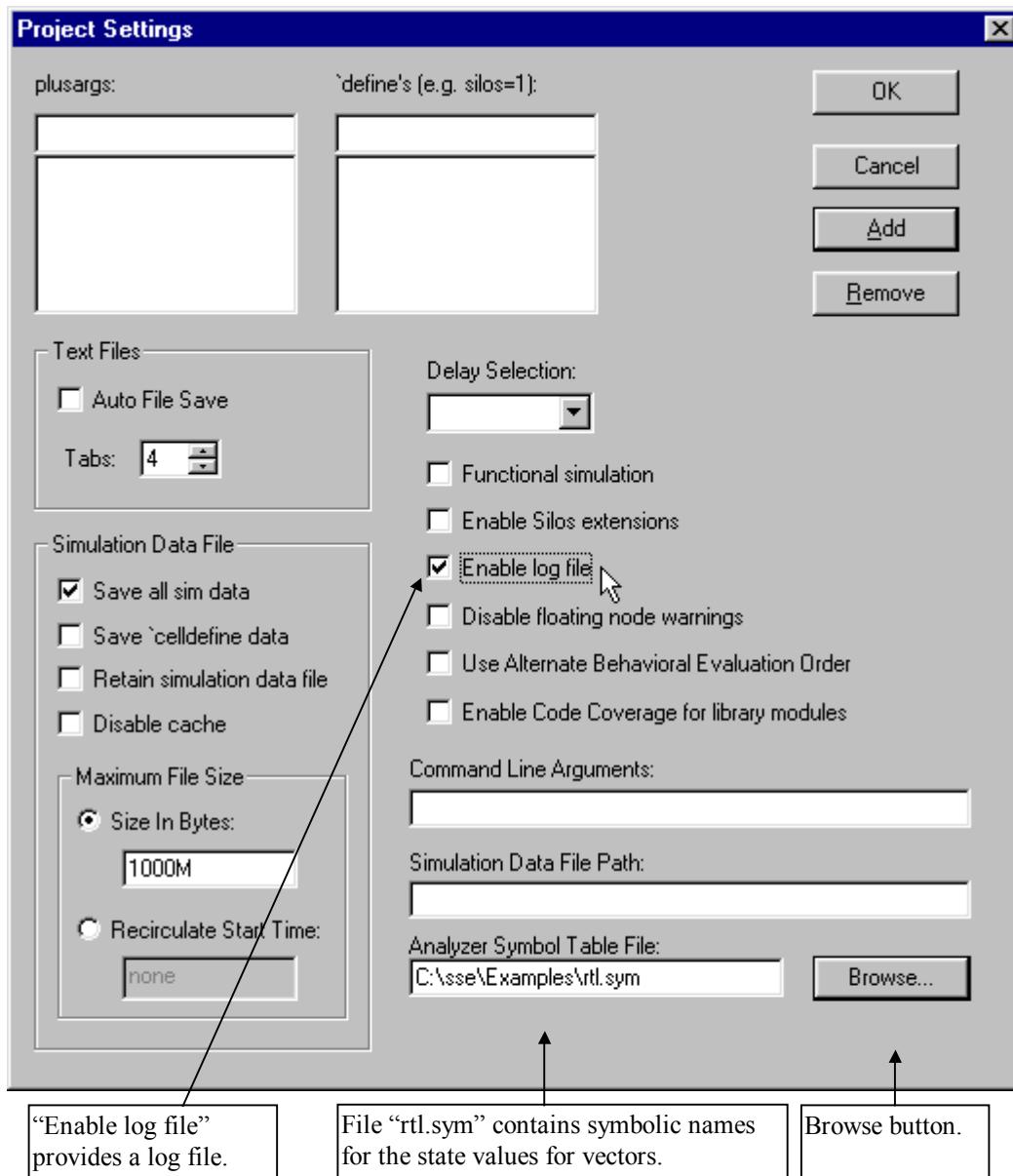
The hex values are listed as hex numbers, i.e. “1”, “8”, “b”, without any size or base specification. Note the hex value in the table must exactly match the hex value for the vector that is displayed by the Data Analyzer when the “Hex” radix is used. For example, if a vector was nineteen bits wide and its hex value when displayed in the Data Analyzer is “001b”, then the symbol table entry must be (also notice there are no blank spaces on either side of the equal sign):

001b=jump

To view an example of the format for a symbol file, use the “File/Open” menu to open file “rtl.sym” in the examples directory.

(continued on next page)

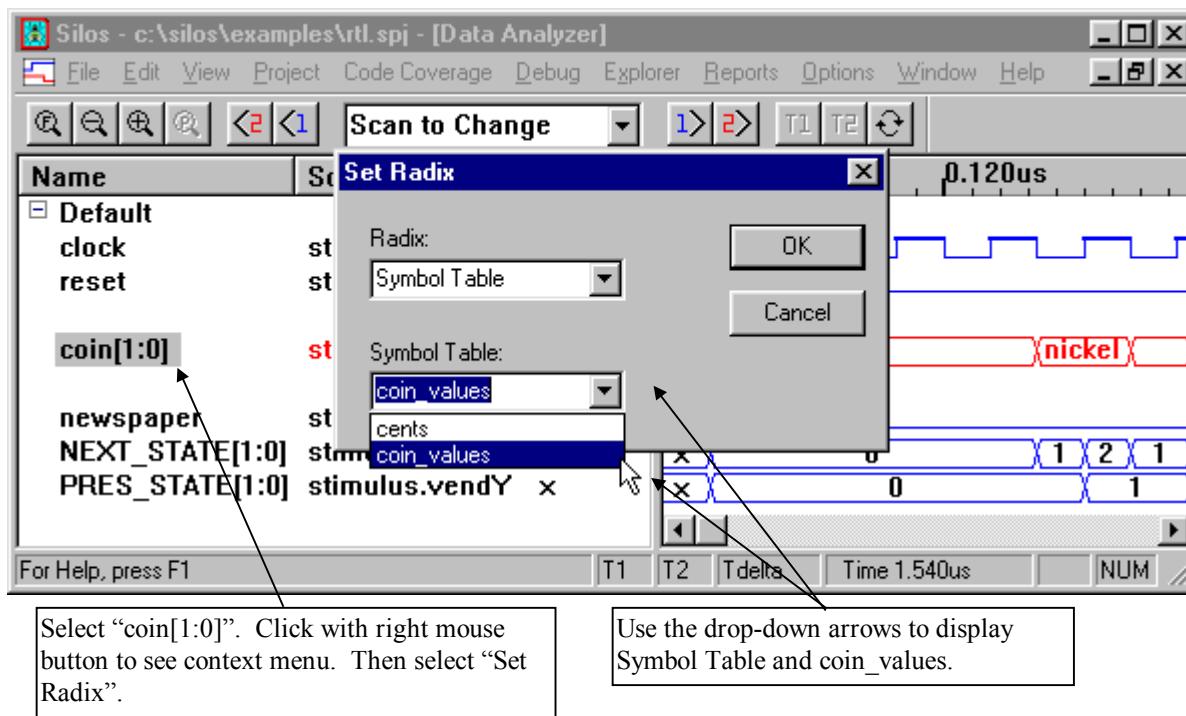
## State Machine Values



(continued on next page)

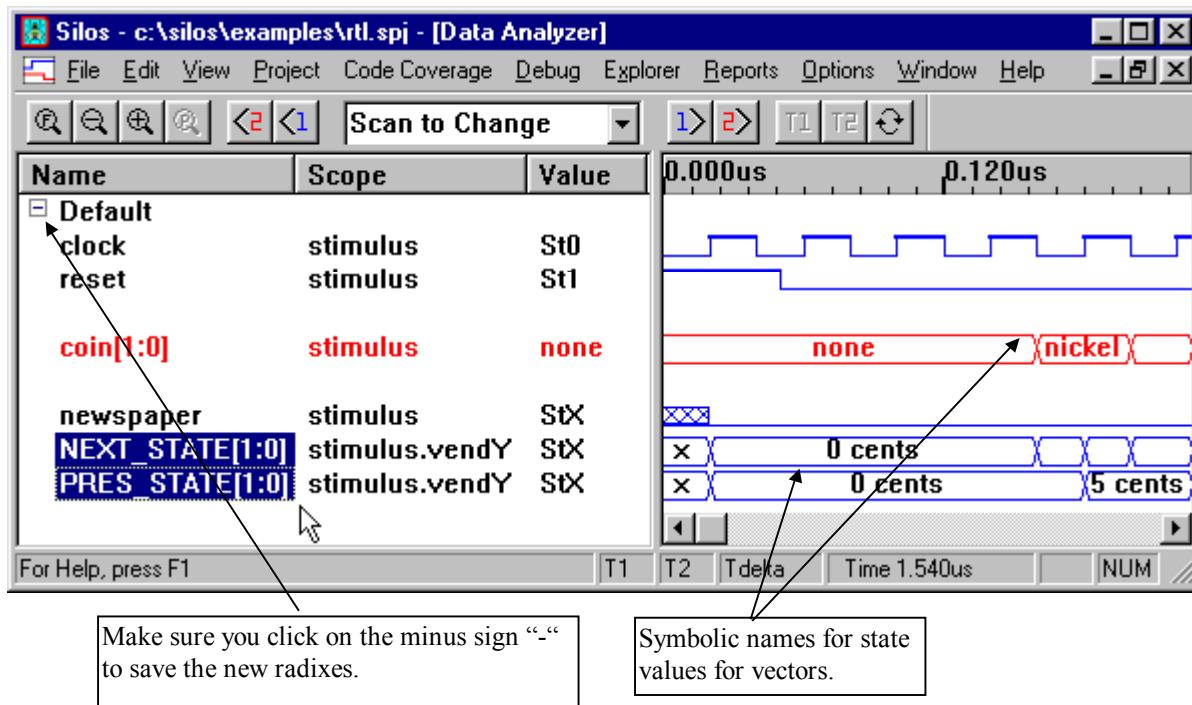
## State Machine Values

- Click on the “Ok” button to close the “Project Settings” dialog box. Click on the “Yes” button in the message box asking to reload the project.
- To open the context menu in the Name list box, click with the right mouse button over top of signal “coin[1:0]” in the Name list box for the Data Analyzer Window. Then select the “Set Radix” selection to open the “Set Radix” dialog box.
- In the “Set Radix” dialog box, click on the drop-down arrow for “Radix” and select “Symbol Table”. For the “Symbol Table” box in the “Set Radix” dialog box, click on the drop-down arrow and select “coin\_values”, and then click on the “Ok” button.



(continued on next page)

## State Machine Values



- Now use the “Set Radix” menu selection to set the radix for the vectors NEXT\_STATE[1:0] and PRES\_STATE[1:0]. To set the radix for both vectors at the same time, select both vectors by holding down the Ctrl key on the keyboard as you select them, and then use the “Set Radix” menu selection to change the radix. Make sure you **click on** the minus sign “-“ for the default group to save the new radices.

## Waveform Annotation

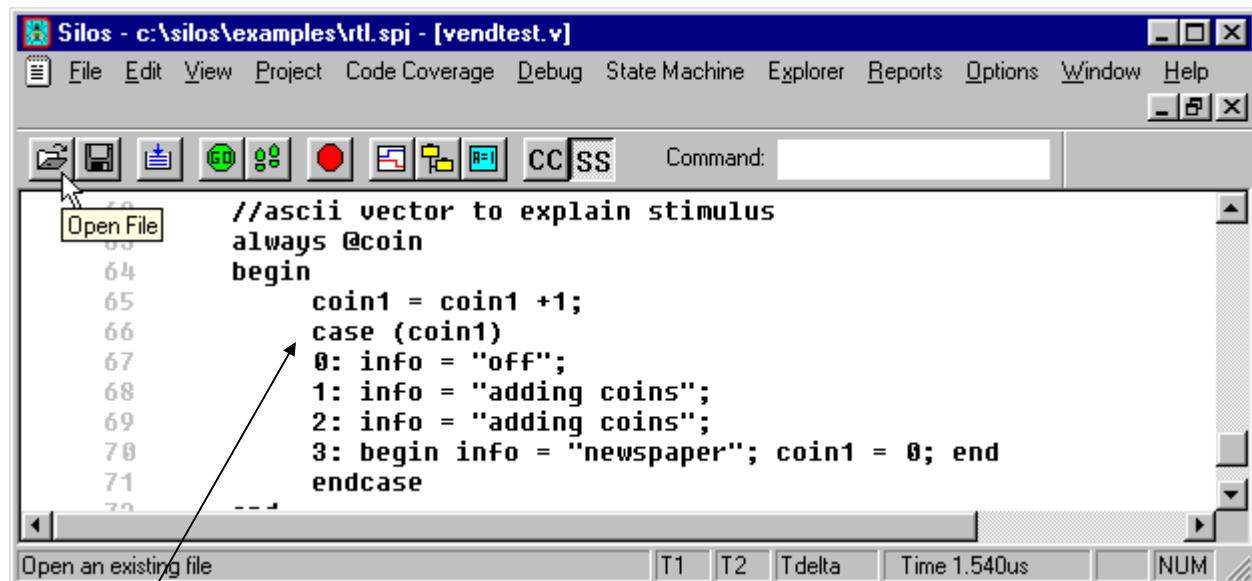
### 2.2.13 Creating an Annotated Timeline for the Waveform Display

#### Benefit

- An annotated timeline displays the design's operation in meaningful symbols as the designer reviews the waveforms, making it much easier to find and debug the design's operations.

#### Procedure

Whether you are debugging your own design or someone else's design, displaying a vector of ASCII text that explains what the simulation is doing can be very useful. To demonstrate this:



The screenshot shows the Silos software interface with the title bar "Silos - c:\silos\examples\rtl.spj - [vendtest.v]". The main window displays a Verilog source code file named "vendtest.v". The code includes a case statement for an ASCII vector "info". A callout box points to this section of the code with the text "case statement for ASCII vector "info"".

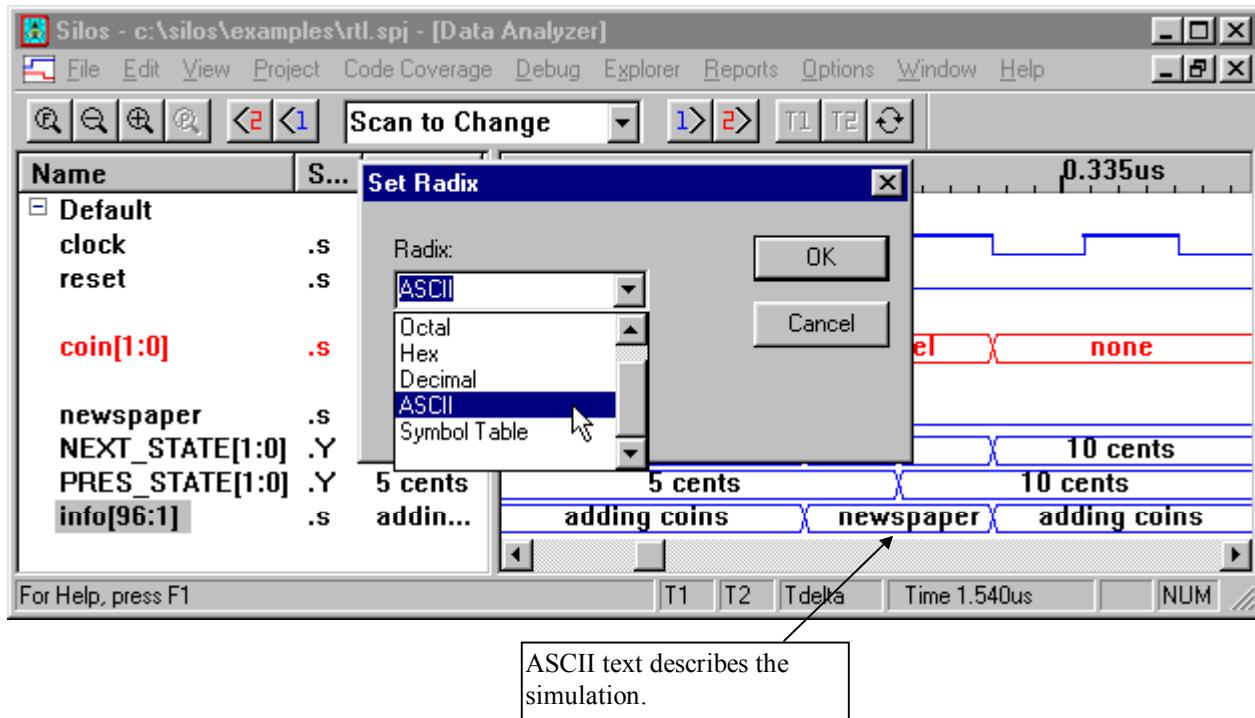
```
//ascii vector to explain stimulus
always @coin
begin
    coin1 = coin1 +1;
    case (coin1)
        0: info = "off";
        1: info = "adding coins";
        2: info = "adding coins";
        3: begin info = "newspaper"; coin1 = 0; end
    endcase

```

- Use the "Open File" button on the Main toolbar to open file "vendtest.v".
- Scroll to the bottom of file "vendtest.v", and view the "case" statement used to set the vector "info" to ASCII strings. This illustrates how to setup the ASCII vector in your source code so that you can use it to display a timeline.

(continued on next page)

## Waveform Annotation



- In the left hand side of the Explorer Window, **click on** instance “stimulus” with the left mouse button. Next in the right hand side of the Explorer Window, **click on** signal “info” with the right mouse button to open the context menu. **Select** the “Add Signals to Analyzer” menu selection in the context menu to add signal “stimulus.info” at the bottom of the Default group in the Name list box for the Data Analyzer Window.
- To change the radix for signal name “info” to ASCII, **click** with the right mouse button on signal “info[96:1]” in the Name list box for the Data Analyzer Window. Then **select** the “Set Radix” selection to open the “Set Radix” dialog box.
- In the “Set Radix” dialog box, **change** the radix to ASCII, and **click on** the “Ok” button. Because the initial value is an unknown level “x” for vector “info”, you can scroll to the right in the Data Analyzer to see meaningful descriptions. Make sure you **click on** the minus sign “-” for the default group to save the new radices.

# Groups

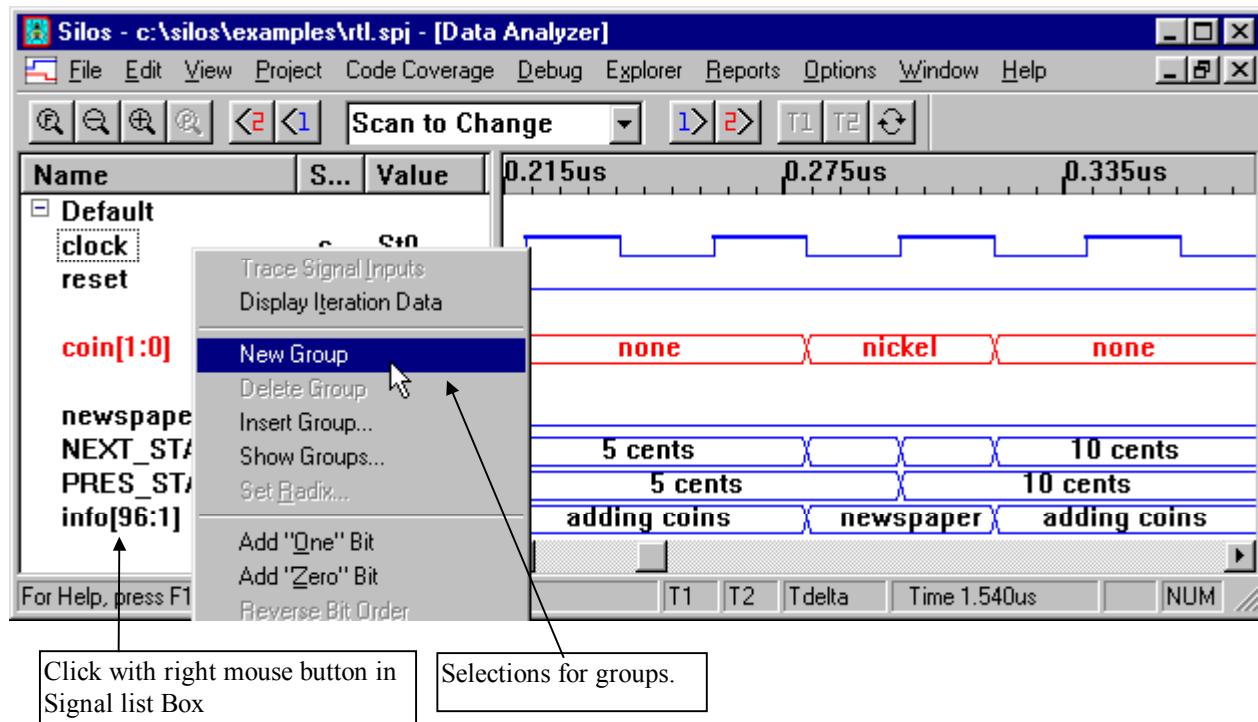
## 2.2.14 Organizing Signals Into Groups

### Benefit

- Allows for compact display of the relationships between signals as the designer opens and closes groups to debug the design's behavior.
- Remembers signal grouping for fast display of waveforms after simulation.
- Allows for easier understanding of the design by a team of engineers.
- Useful for record keeping if the design needs to be rerun.
- Immediately saving changes to groups prevents the loss of work if Silos is interrupted.

### Procedure

To see the following selections for groups, put the mouse cursor in the Name list box for the Data Analyzer, and click on the right mouse button to open the context menu:



(continued on next page)

## Groups

- New Group: This selection can be used to add a new group to the Name list box.
- Delete Group: This selection can be used to delete a group.
- Insert Group: This selection opens the “Add Group” dialog box. This dialog box will insert a group within a group. The inserted group is displayed as a bus, which can be expanded and hidden by double-clicking on it.
- Show Groups: This selection opens the “Select Signal Groups” dialog box. This dialog box can be used to select which groups are displayed in the Data Analyzer.

When the Data Analyzer Window is opened, the “Default” group is displayed. To save the signals that have been added to the Default group:

- **Click on** the minus sign “-” just to the left of the Default group in the Name list box.
- Silos will ask you if you want to save the changes to the Default group (unless they have already been saved). **Click on** the “Yes” button (if the signals have not already been saved).

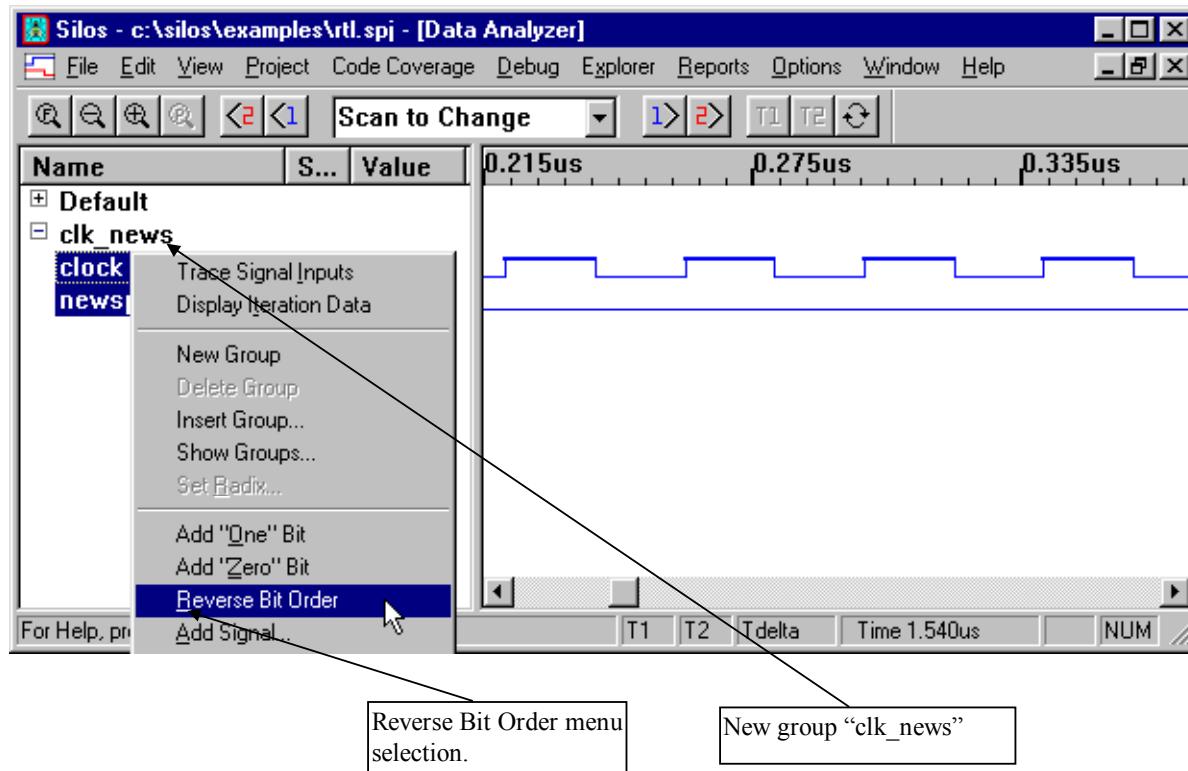
(continued on next page)

**Buses****2.2.15 Buses****Benefit**

- New buses can be easily created without re-simulating, saving time and eliminating unnecessary interruption of the designer's concentration.

**Procedure**

Groups can be used to represent a bus. For example, to create a bus that uses the signals clock and newspaper:

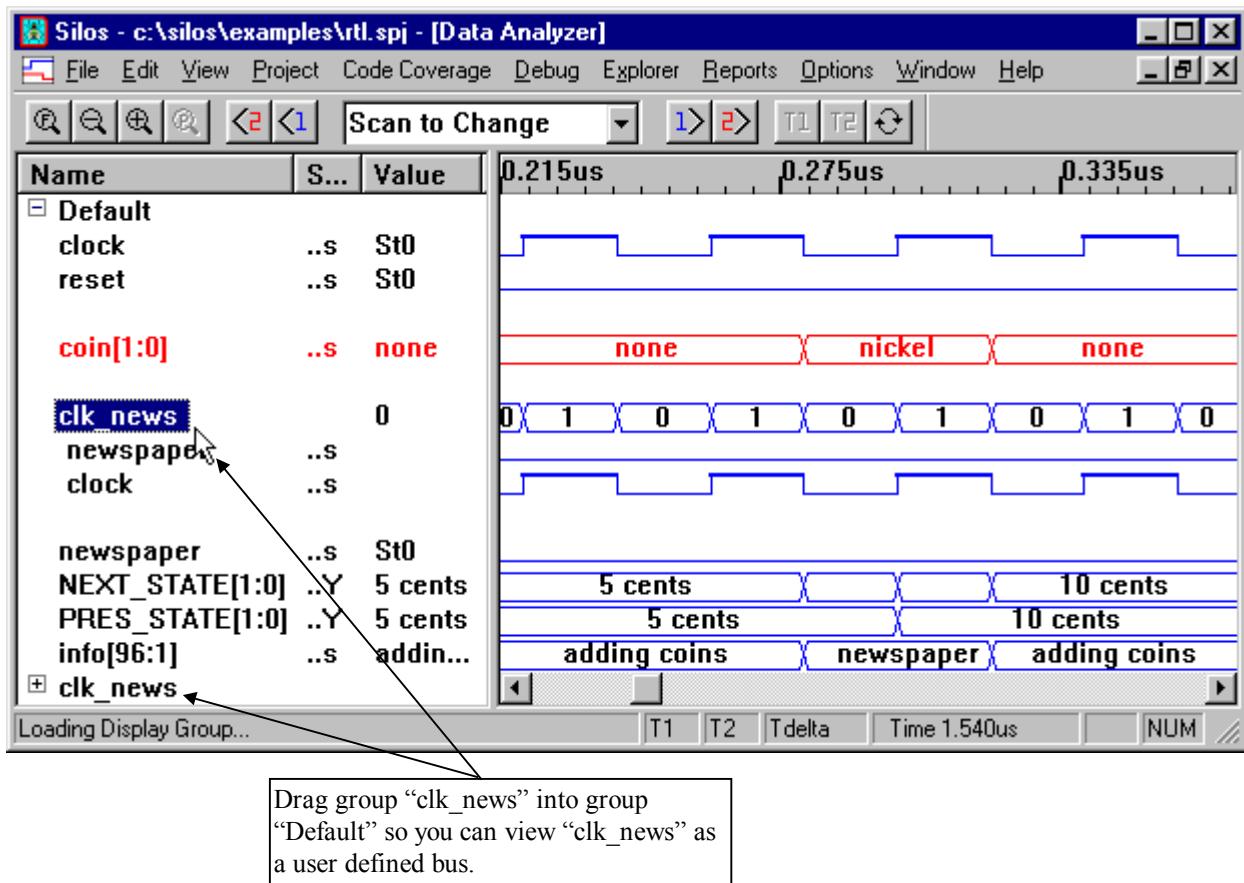


- Click with the right mouse button in the Name list box for the Data Analyzer to open the context menu.

(continued on next page)

## Buses

- Select “New Group” in the context menu. This will add the group labeled “New Group” to the Name list box.
- To change the name of the new group, with the left mouse button **click on** “New Group” once, **pause**, and if necessary, click on “New Group” a second time. The “New Group” label should allow you to **change** its name. Change the name to “clk\_news” and press and release the Enter key on the keyboard.
- **Select** signals “clock” and “newspaper” by holding down the “Ctrl” key, and **copy** them to the new group “clk\_news” by holding down the “Ctrl” key while you drag and drop them with the mouse.
- To reverse the bit order in group “clk\_news”, **highlight** signals “clock” and “newspaper” in group “clk\_news”. Then **open** the context menu in the Name list box and **select** “Reverse Bit Order”.



(continued on next page)

## Buses

- To save the group “clk\_news”, **click on** the minus sign to the left of the group “clk\_news”, and **answer** “Yes”.
- Now **drag and drop** group “clk\_news” into group “Default”. You can also add blank lines before and after “clk\_news”.

## Conditional Search

### 2.2.16 Search on Condition

#### Benefit

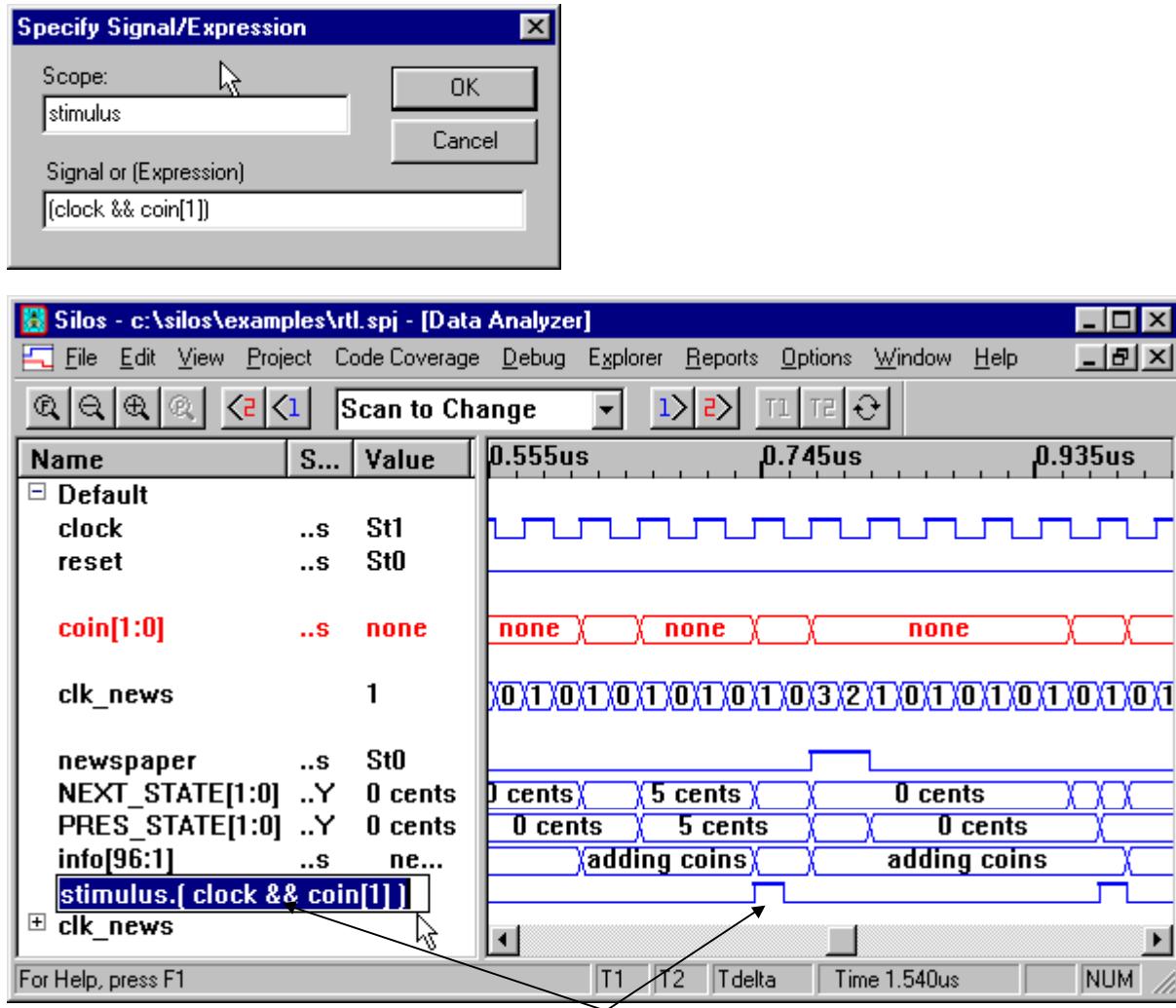
- The designer can define a search condition using any Verilog HDL expression without re-simulating. Scanning to the condition and viewing it as a waveform gives the designer quick access to important conditions of the design's operation. Searching on any Verilog HDL expression is possible because Silos saves everything quickly and compactly to disk when simulating.

#### Procedure

A unique feature of Silos is the ability to view a search condition as a waveform. You can use any valid Verilog HDL expression to create a search condition and then display it as a waveform.

(continued on next page)

## Search on Condition



(continued on next page)

## Search on Condition

For example, to create a search condition for when “stimulus.clock” and “stimulus.coin[1]” are both true:

- **Click** with the right mouse button in the Name list box for the Data Analyzer to open the context menu.
- **Select** the “Add Signal” menu selection to **open** the “Specify Signal/Expression” dialog box.
- Set the scope in the “Scope” edit box to “stimulus”.
- **Enter** the expression `(clock && coin[1])` in the “Signal or (Expression)” edit box, and **click-on** the “OK” button. The waveform for the expression `((clock && coin[1]))` will then be displayed in the Data Analyzer.
- If you single click on the expression that you added to the Data Analyzer, pause, then click on the expression again, you can modify the expression.

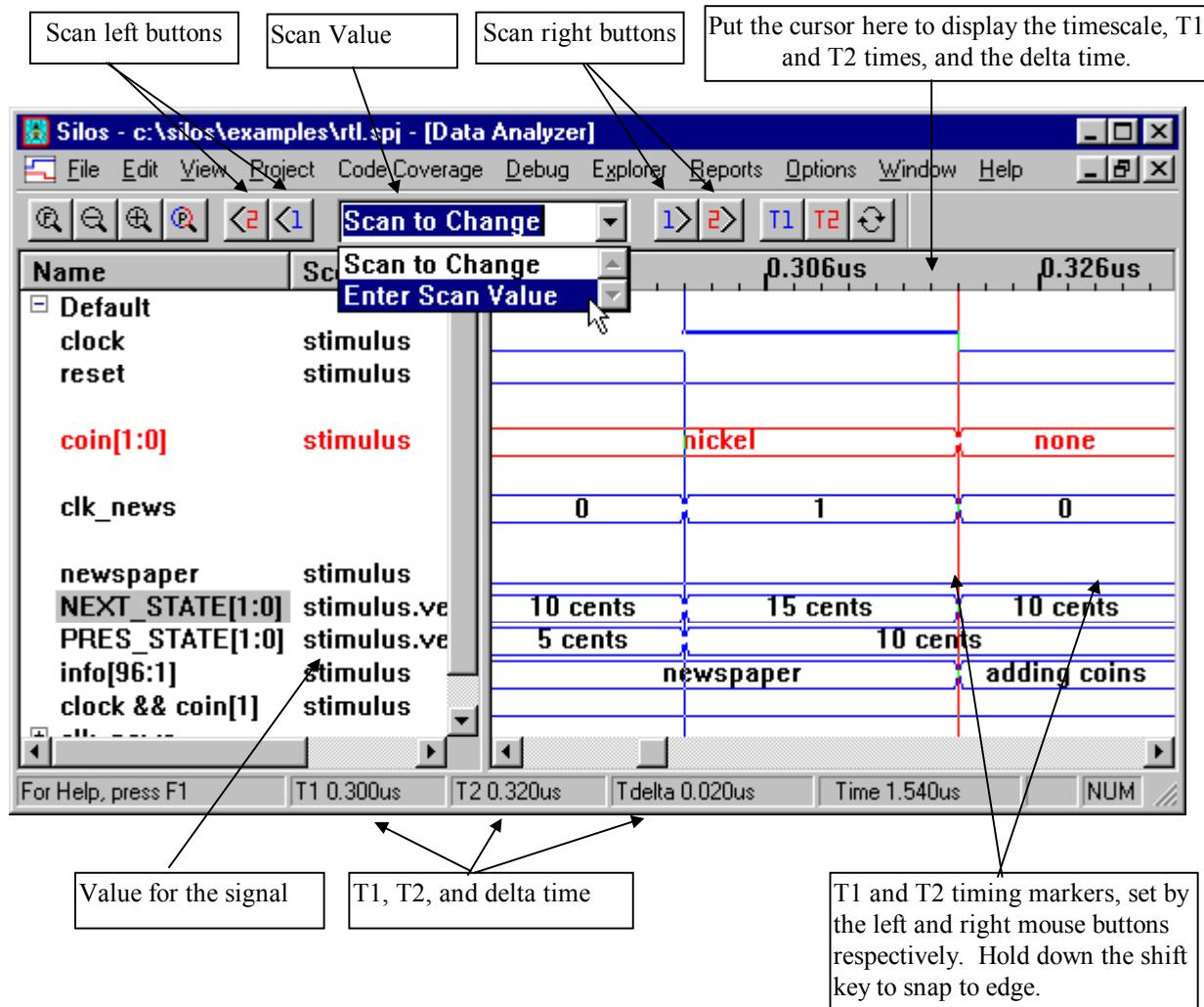
## Scan and Pan

### 2.2.17 Scan-to-Edge, Scan-to-Value and Panning

#### Benefit

- The user can scan to change or scan to value. Easy scanning and panning saves time when debugging.

#### Procedure



(continued on next page)

## Scan and Pan

The Waveform Display window allows you to set two timing markers, T1 and T2, which can be used to display the time that edges occur, and the delta time between edges. To set the timing markers:

- Place the mouse cursor in the “Waveform Display”.
- Click on the left and right mouse buttons to set the T1 (blue vertical line) and T2 (red vertical line) timing markers. Notice that whenever the mouse cursor is held next to a timing marker, a text box displays the time next to the marker.

If you place the mouse cursor in the gray timeline area, a text box will appear that displays the current timescale, the T1 and T2 time values, and the delta time between the T1 and T2 timing markers. This information is also displayed in the Status Bar at the bottom of the Silos window. The state value at the T1 timing marker for each waveform is displayed in the “Value” column for the Name list box for the Data Analyzer.

When setting the T1 and T2 timing markers for the Data Analyzer, they will snap to the nearest edge if the “Options/Snap to Edges” menu selection is active. When setting a timing marker, you can hold down the shift key to temporarily toggle the “Snap to Edges” selection to its opposite effect.

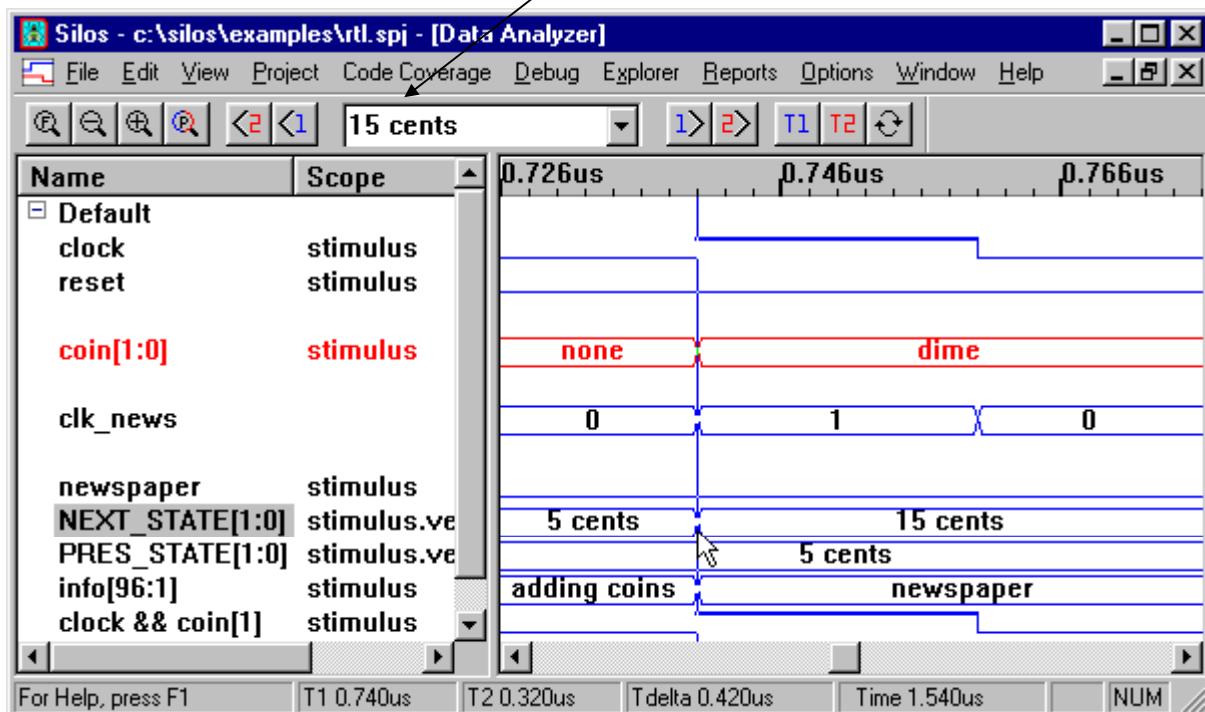
(continued on next page)

## Scan and Pan

### Scan to Edge and Value

- To demonstrate scanning to an edge, **highlight** vector “NEXT\_STATE[1:0]” in the “Name” list box and click on the “**1>**” toolbar button to scan the T1 marker to the right. The state value for each waveform in the “Value” column will change as you scan the T1 marker edge to edge. The T2 marker has no effect on the state values in the Value column.
- To demonstrate scanning to a value, **click on** the drop down arrow for the “Scan to Change” box on the Main toolbar, and **choose** the “Enter Scan Value” selection. **Enter** the value “15 cents” in the box on the Main toolbar, and click on the “**1>**” button on the Main toolbar to scan the T1 marker to the right as it matches each value of “15 cents” for the vector “NEXT\_STATE[1:0]”. “Scan to Value” can be used to scan to the value for a bit, a vector, or a symbolic name because “Scan to Value” does a character match. The “?” character can be used as a don’t care character, such as “?ff?” would match with “1ff3” and “ffff”.

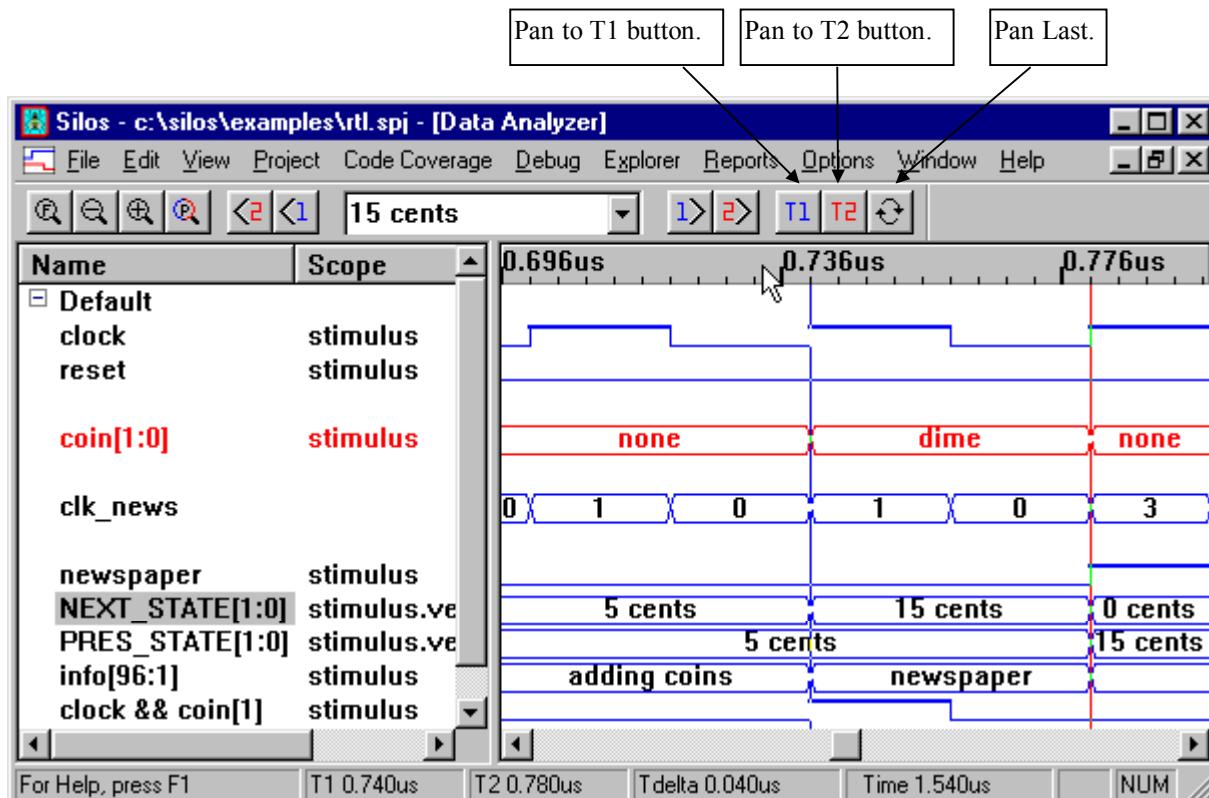
Scan to value does a character search, so you can scan a signal that is a single bit, a vector, or that uses symbolic names.



(continued on next page)

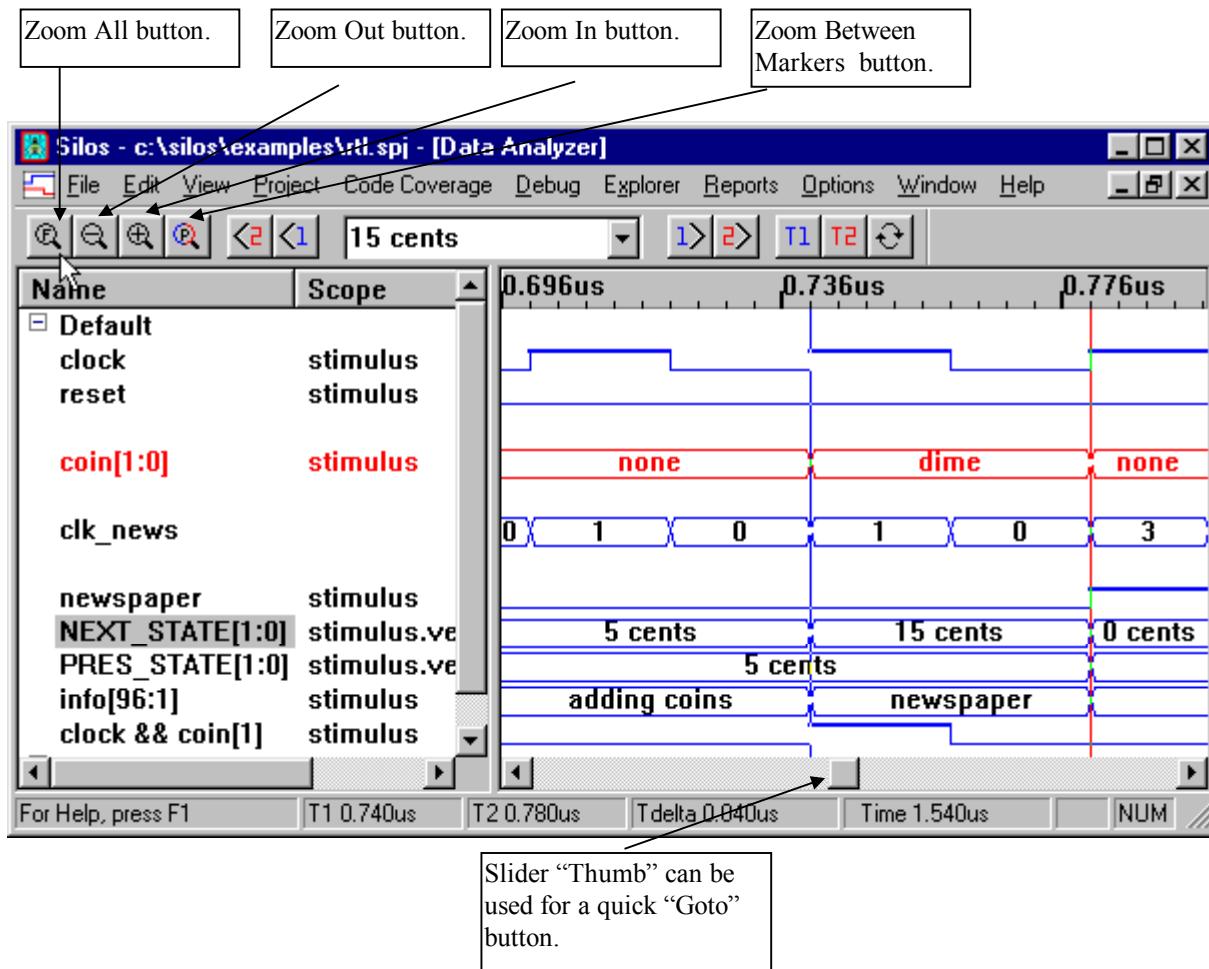
## Scan and Pan

The “Pan to T1”, “Pan to T2”, and “Pan Last” buttons on the Analyzer Toolbar let you pan to the T1, T2 timing markers, or pan to the last view in the Data Analyzer.



**Zoom****2.2.18 Zoom-Buttons****Benefit**

- Allows for easy viewing of waveforms across simulation time.

**Procedure**

The zoom buttons on the Toolbar allow you to zoom-in, zoom-out, zoom-all, and zoom-in-between markers.

(continued on next page)

## Zoom

To demonstrate the zoom capabilities:

- **Click-on** the Zoom-All button on the Toolbar to display the entire simulation range. You can stop the zoom-all by **striking** the escape key “Esc” on the keyboard.
- Next **set** the T1 and T2 timing markers near the end of the simulation range.
- **Use** the Zoom Markers button on the Toolbar to zoom in-between the timing markers.
- **Use** the Zoom Out button to zoom-out by a factor of two and the Zoom In button to zoom-in by a factor of two.
- The slider “thumb” at the bottom of the waveform display can be used as a “go to” button. **Use** the mouse to move the slider “thumb” to approximately time 1 us (micro seconds). The new time values are displayed along the horizontal time axis as you move the slider “thumb”.

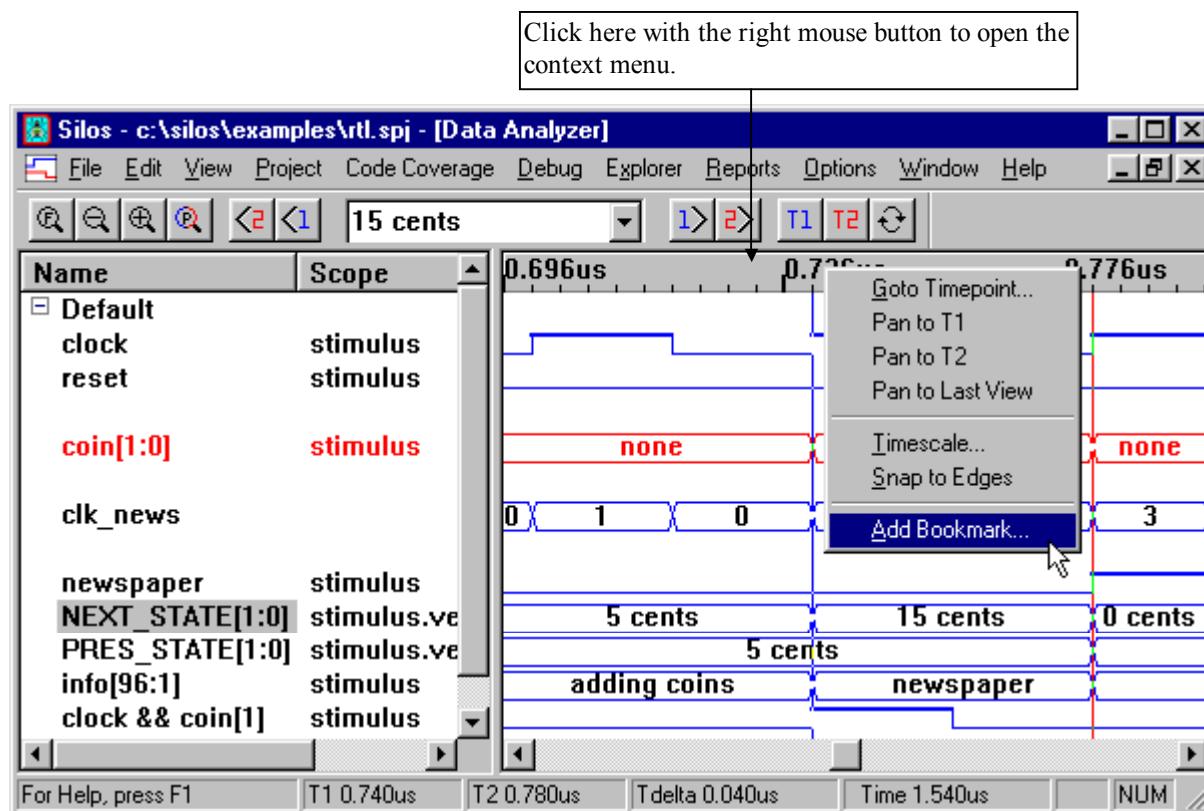
## Bookmarks

### 2.2.19 Bookmark, Timescale, Goto Timepoint

#### Benefit

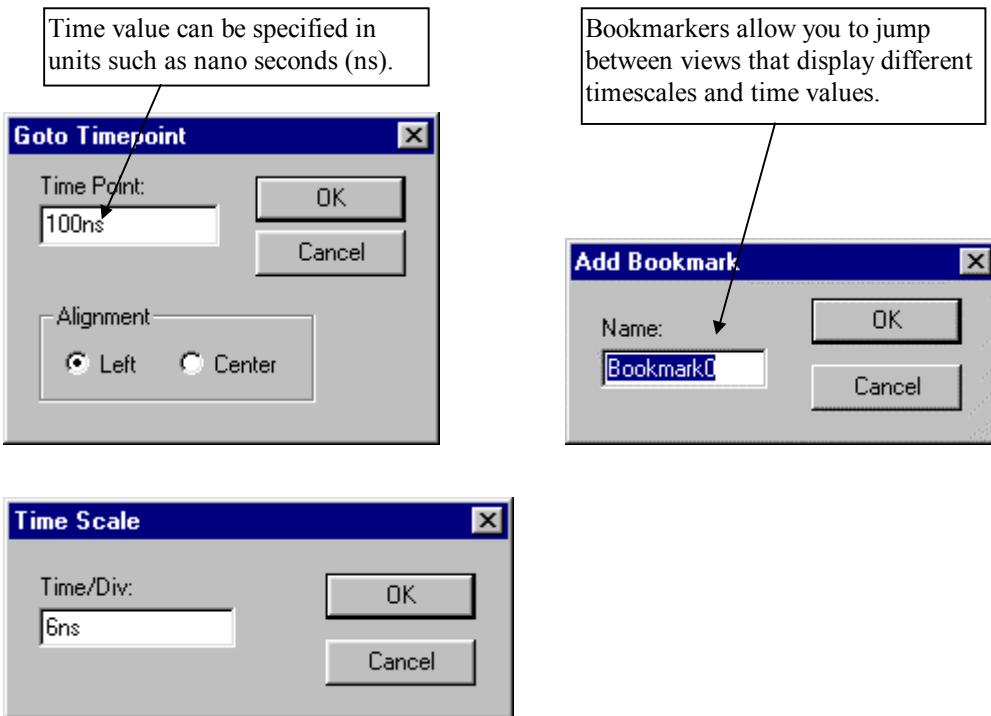
- Enables the designer to quickly jump between simulation times and resolutions so that he can debug a problem across simulation time.

#### Procedure



(continued on next page)

## Bookmarks



The Data Analyzer Window has a context menu that includes selections for “Goto Timepoint”, “Timescale” and “Add Bookmark” menu selections. To invoke the context menu, use the right mouse button to **click-on** the timescale just above the Waveform Display Window. The context menu will remain open while the left mouse button is used to select a menu item.

The “Goto Timepoint” menu selection opens the “Goto Timepoint” dialog box. The specified timepoint can be displayed either at the left or the center of the Waveform Display Window.

The “Timescale” menu selection opens the Time Scale dialog box. When specifying the time scale you can use any standard unit, such as 600ns (nanoseconds), 1000ps (pico seconds), etc.

The “Add Bookmark” menu selection places a virtual marker at the center of the Waveform Display Window for the current time and timescale resolution. After opening the “Add Bookmark” dialog box you will see the default bookmark name, i.e. “Bookmark1”, “Bookmark2”, etc. You can specify any string of characters for the bookmark and then click-on the OK button to set the bookmark. The bookmarks you have set are listed at the bottom of the context menu.

(continued on next page)

## Bookmarks

To demonstrate using a bookmarker:

- Activate the context menu by **clicking** with the right mouse button in the Data Analyzer timeline. **Select** the “Add Bookmark” menu selection to open the “Add Bookmark” dialog box.
- To **set** “Bookmark0”, click on the “OK” button.
- **Open** the context menu and **use** the “Goto Timepoint” menu selection to go to another simulation time, such as “0.5us”.
- Then **open** the context menu again and **select** “Bookmark0” to go back to where you were.

## Data Tips

### 2.2.20 Data Tips

#### Benefit

- The simulation value for any variable or expression can be viewed by opening the Verilog HDL source window and holding the mouse cursor over the variable, or highlighting an expression and holding the mouse cursor over the expression. This lets the designer debug directly in a Verilog HDL source code window.
- The “Go to Source” code menu selection can be used to quickly display the module definition for any instance in the hierarchy.

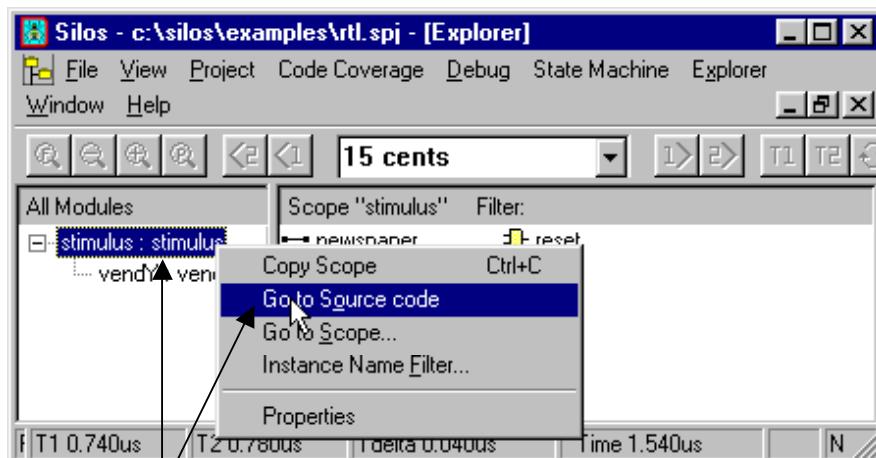
#### Procedure

The Explorer Window can be used to open the source file for a module definition to display the value for a variable or expression. To demonstrate this:

- To open the context menu in the Explorer window, **select** instance “stimulus” in the hierarchy in the left side of the Explorer window, and **click** with the right mouse button while over the instance “stimulus”. In the context menu that appears, **select** “Go to Source code”. This will open file “vendtest.v” with the cursor just to the left of the definition for “module stimulus”. Click on the text “module stimulus” so it is no longer highlighted. **Hold** the mouse cursor over the variable “clock”. The value, scope, radix, and simulation time for variable “clock” will be displayed. The simulation time for the value is determined by the following criteria:
  - If the simulation timepoint has not completed, such as during single stepping, then the current simulation time point is displayed.
  - If the Data Analyzer is not open, then the last simulation time point is used to display the data tip value.
  - If the Data Analyzer is open, then the time for the left axis of the Waveform window for the Data Analyzer is used to display the data tip value. If the “T1” timing marker is displayed in the Data Analyzer, then the time associated with the “T1” timing marker (blue color, set by left mouse button) is used to display the data tip value. Setting the “T1” timing marker lets the user display the data tips value at different time points.
- If you want to select the radix for the value displayed by the Data Tips, open the context menu in a source window by clicking in the source window with the right mouse button, and then select the “Data Tips Radix” menu selection to see the list of radices.
- If you want to turn the Data Tips “on” or “off”, use the Opens/Data Tips menu selection in the main menus, or the “Data Tips” selection in the context menu for a source window.

(continued on next page)

## Data Tips



Select instance "stimulus", open the context menu with the right mouse button, and then select "Go to Source code".

The screenshot shows the Silos software interface with the title bar "Silos - c:\silos\examples\rtl.spi - [vendtest.v]". The menu bar includes File, Edit, View, Project, Code Coverage, Debug, State Machine, Explorer, Reports, Options, Window, and Help. The toolbar includes icons for file operations and simulation controls. The main window displays Verilog source code:

```

6     `timescale 1ns / 1ns
7     module stimulus;
8     reg clock;
9     reg [1:0] scope = St1;
10    reg [1:0] radix = hex;
11    reg [1:0] time = 0.740us;

```

A Data Tip is shown for the variable "clock", providing information: "clock = St1", "scope = stimulus", "radix = hex", and "time = 0.740us". The status bar at the bottom shows simulation time points T1 0.740us, T2 0.780us, Delta 0.040us, and Time 1.540us.

Data Tips display value, scope, radix, and simulation time point for variable clock. The time point can be set by the "T1" timing marker in the Data Analyzer. The Data Tips can be turned on/off and the radix can be changed by using the context menu in the source window.

# Single Stepping

## 2.2.21 Single Stepping

### Benefits

- The designer can single step through the source code as it executes to understand, locate and fix problems.
- The value for any variable or expression in the source code can be displayed so that the user can trace iterations at a timepoint.
- Any variable or expression can be dragged and dropped to the Watch window to display its instantaneous value, or the Data Analyzer to view the waveform.
- Silos's unique ability to quickly and efficiently save every variable during simulation makes this possible. This intuitive approach to source code debugging reduces the time required to understand design flaws.
- The single stepping and breakpoints features can reduce simulation speed, so these features should be disabled before simulation if you do not intend to use them.

### Procedure

Skills presented in this topic are:

- Single stepping through a design
- Dragging and dropping variables and expressions from your source code into the Data Analyzer and Watch Windows.
- Viewing variables as they change value in the Data Analyzer Window and the Watch Window.
- Setting the value for a two bit register.
- Using breakpoints to skip over uninteresting behavioral code.

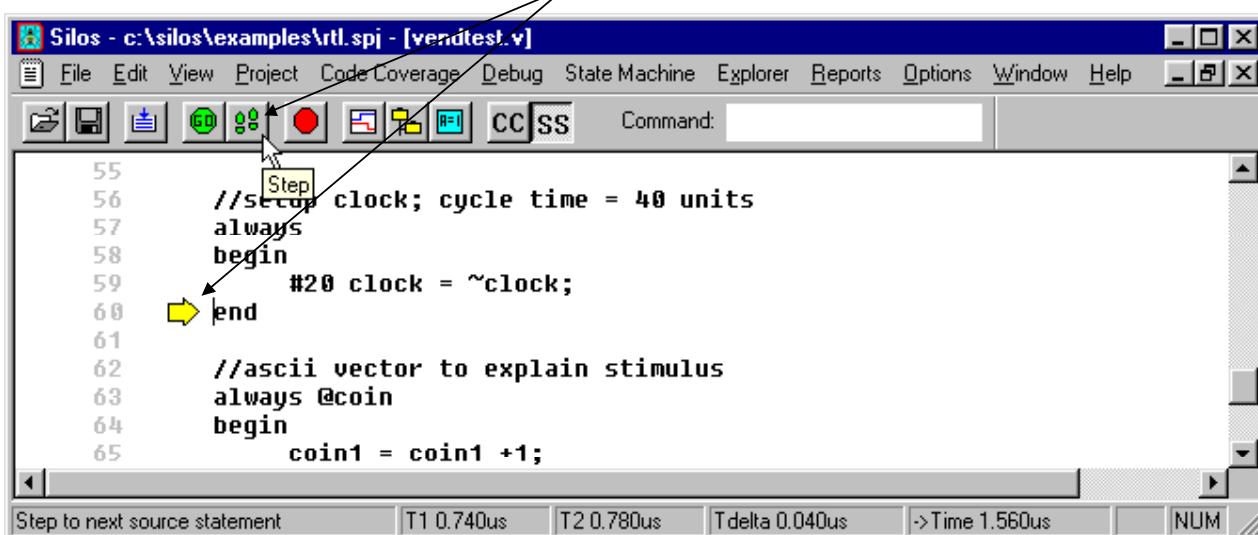
(continued on next page)

## Single Stepping

To demonstrate the capabilities of single-stepping for behavioral code:

- **Click-on** the “Load/Reload Input Files” button to restart the simulation to time = 0.
- **Click-on** the “Step” button on the Toolbar. Silos will automatically open the source window for file “vendtest.v” and put a yellow arrow to the left of the source code line that was just executed. Silos has a built-in editor for basic editing in the source windows (see “Edit Menu” on page 3-6).

Click on the “Step” button to open the source window to the line that is currently executing.



Silos - c:\silos\examples\rtl.spi - [vendtest.v]

```

55
56 //Setup clock; cycle time = 40 units
57 always
58 begin
59 #20 clock = ~clock;
60 end
61
62 //ascii vector to explain stimulus
63 always @coin
64 begin
65     coin1 = coin1 +1;

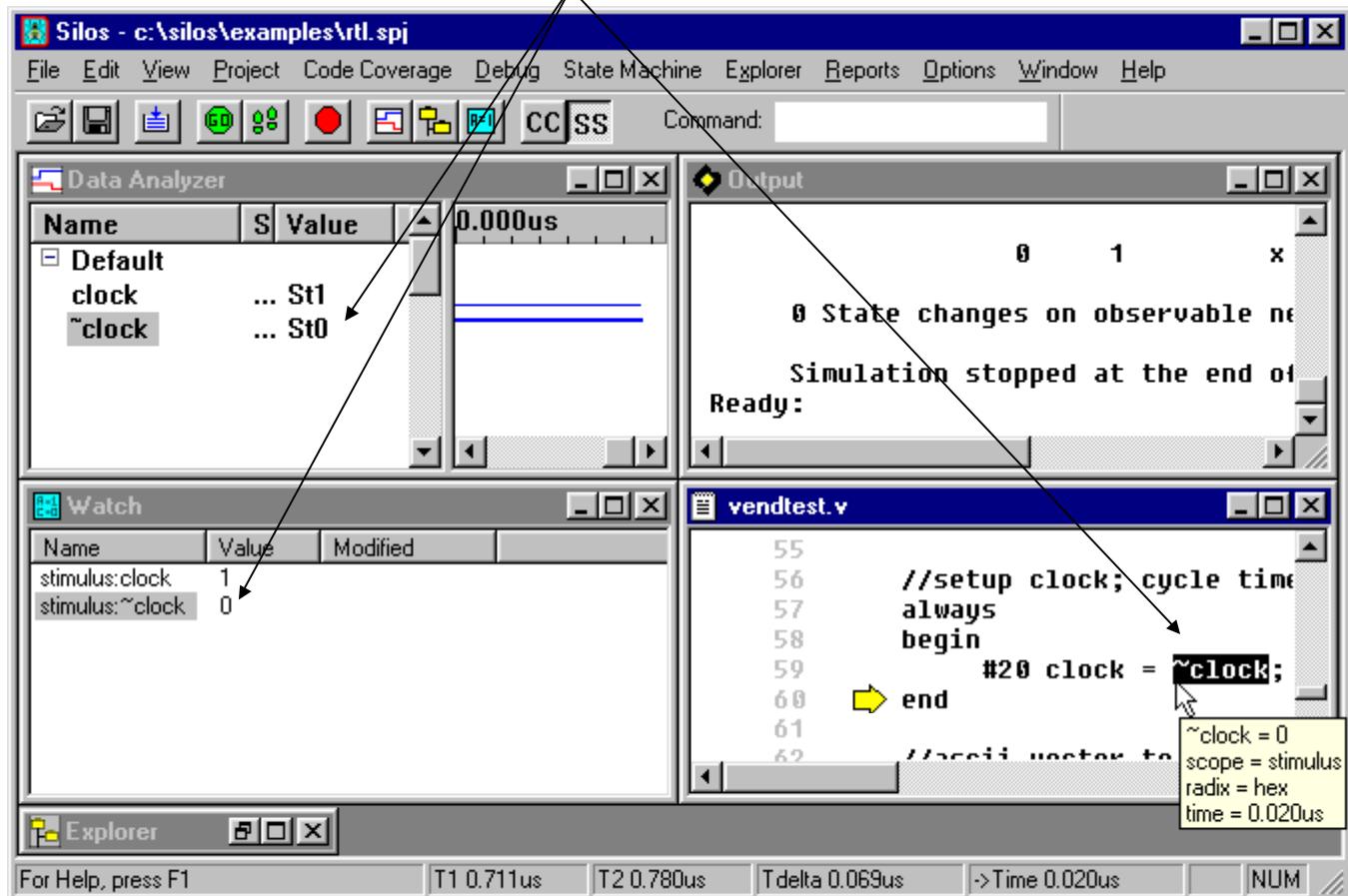
```

Step to next source statement    T1 0.740us    T2 0.780us    Tdelta 0.040us    ->Time 1.560us    NUM

(continued on next page)

## Single Stepping

Drag and drop expression “~clock” from file “vendtest.v” to the Data Analyzer and Watch Window. Notice the Data Tip shows the instantaneous value for expression “~clock” is “0” at the current timestep. However, the Data Analyzer is not updated until the end of the timestep, so its value for “~clock” is “1”.

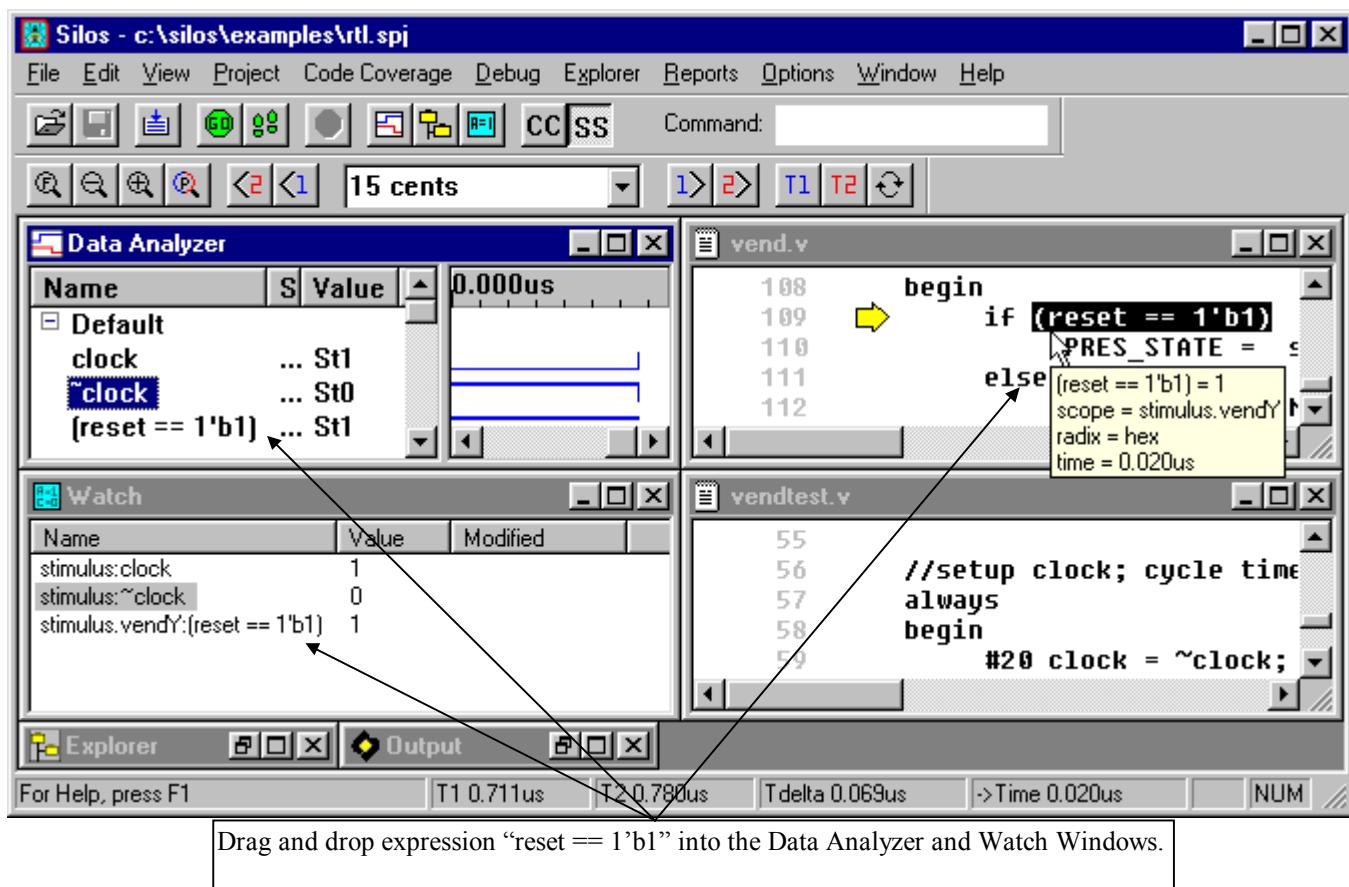


- Next **click-on** the “Open Watch Window” button on the Toolbar to open the Watch Window. Minimize the Explorer Window by clicking on its minimize button in its upper left hand corner.
- Select the Window/Tile menu to tile the windows that are currently open for Silos. To select a variable, **double-click** on variable “clock” in the source window. Make sure that the space after variable clock is not highlighted, as it would become “clock” with a space, instead of just “clock”. **Drag and drop** variable “clock” from the source window to the Watch Window and to the top of the “Default” group in the Data Analyzer Window. To create additional space for viewing the windows, **minimize** the Output Window.

(continued on next page)

## Single Stepping

- Single stepping can be used to determine the condition for an “if” test or a “case” statement before Silos branches into it. To demonstrate this, continue **clicking** on the “Step” button on the Toolbar until the source window opens for file “vend.v”. Use the mouse to **highlight** the expression “reset == ‘b1” in file “vend.v” and **drag and drop** the expression into the Data Analyzer Window and the Watch Window. You will see that the expression is true, so that the next step will go to the statement directly below the “if” test.
- Then continue to **click-on** the “Step” button. Notice that the values in the Watch Window and the Data Analyzer Window are updated as the single-stepping progresses.



## Set and Force

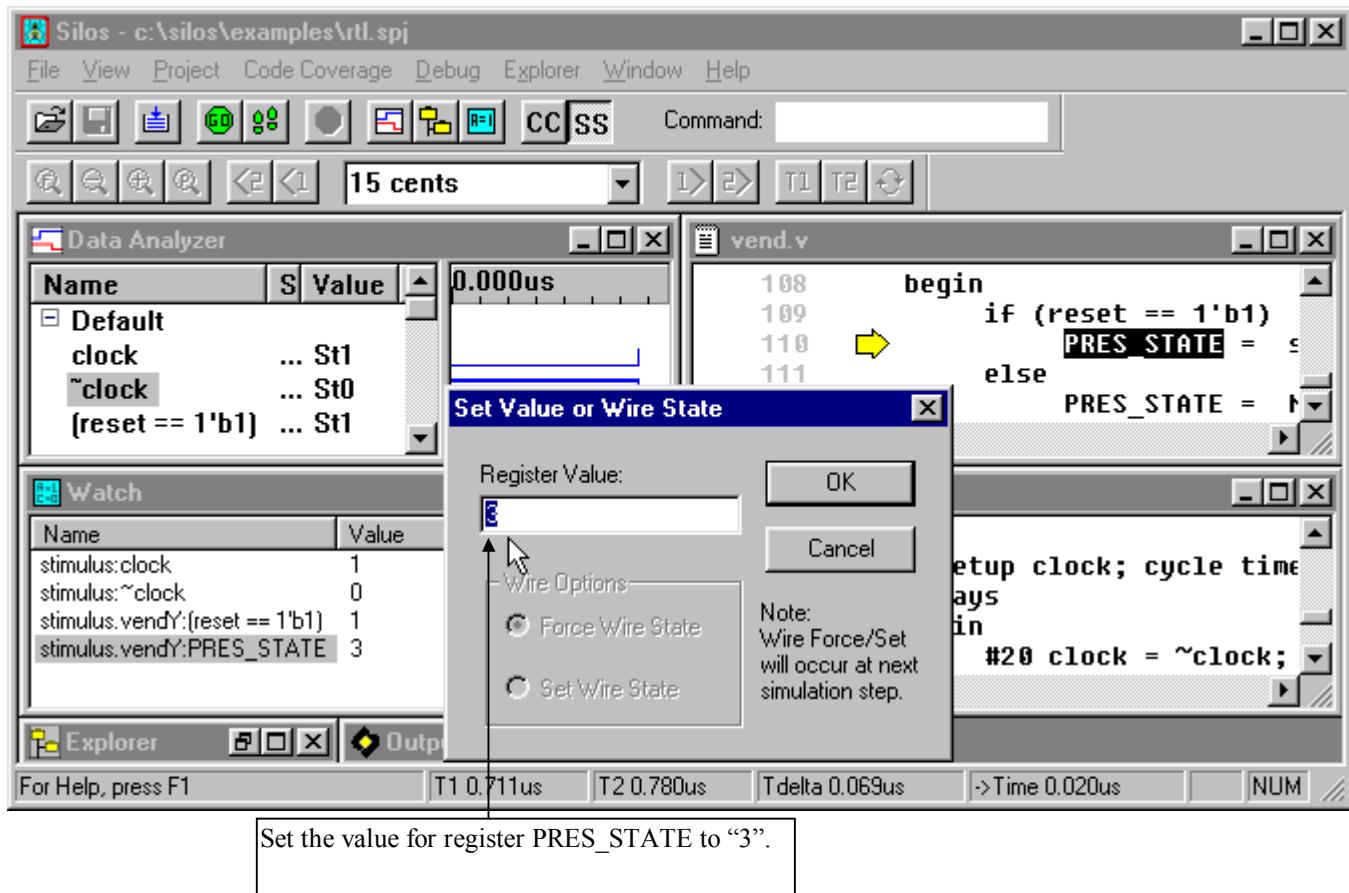
### 2.2.22 Setting and Forcing Values

#### Benefit

- A variable can be set or forced to a new value, saving time and allowing you to quickly try different conditions on your design.

#### Procedure

When debugging, you may need to set or force the value to a variable. To demonstrate this, we will set signal PRES\_STATE to “3” at time=0 so that it will cause a free newspaper to be ejected at the start of the program.



- To reset the simulation to time=0, click on the “Load/Reload Input Files” button on the Main Toolbar.

(continued on next page)

## Set and Force

- **Drag and drop** variable “PRES\_STATE” from file “vend.v” to the Watch window.
- In the Watch Window, **click on** signal “stimulus:vendY:PRES\_STATE” with the right mouse button to open the context menu.
- **Select** the “Set Value” menu to open the “Set Value or Wire State” dialog box.
- **Change** the Register Value of “x” to “3” in the “Set Value or Wire State” dialog box and **click on** the “OK” button. The “Modified” column in the Watch Window will state “Set Value 3”. When you **click on** the Step button, the “Set Value 3” will disappear as variable “PRES\_STATE” is set to 3. As you continue to single step, you will notice that signal “PRES\_STATE” in the Data Analyzer starts as “3” (or 15 cents if you use the symbolic name) instead of “x”.

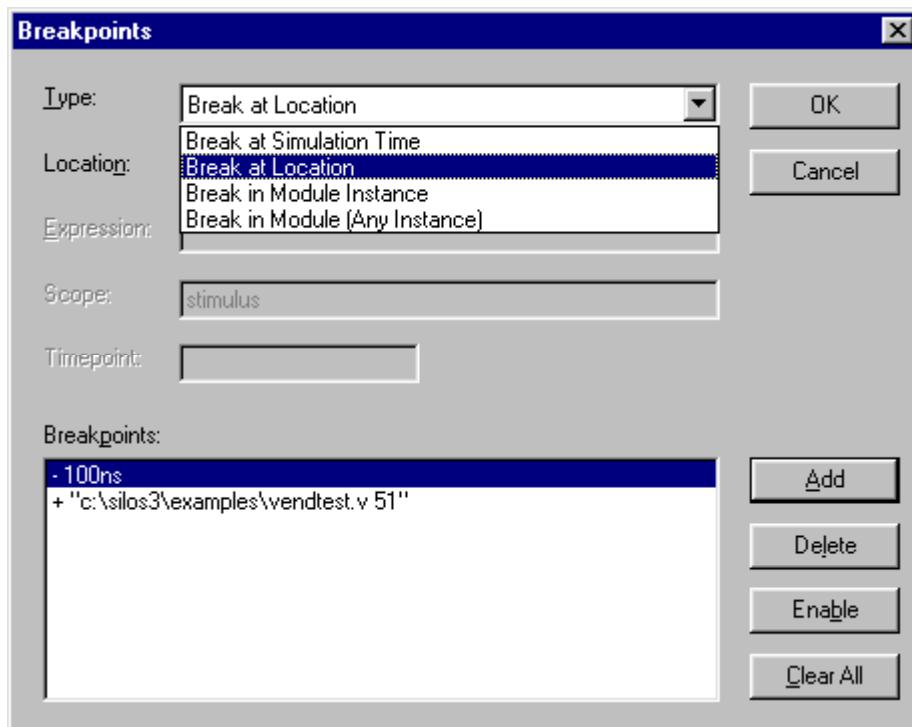
# Breakpoints

## 2.2.23 Breakpoints

### Benefit

- Breakpoints let you skip over uninteresting source code so that you can focus your debugging time on the code that is causing the problem.

### Procedure



Breakpoints can be set using the “Debug/Breakpoints” menu selection or the “Toggle Breakpoint” button on the Toolbar. Silos has the following breakpoints for debugging behavioral code:

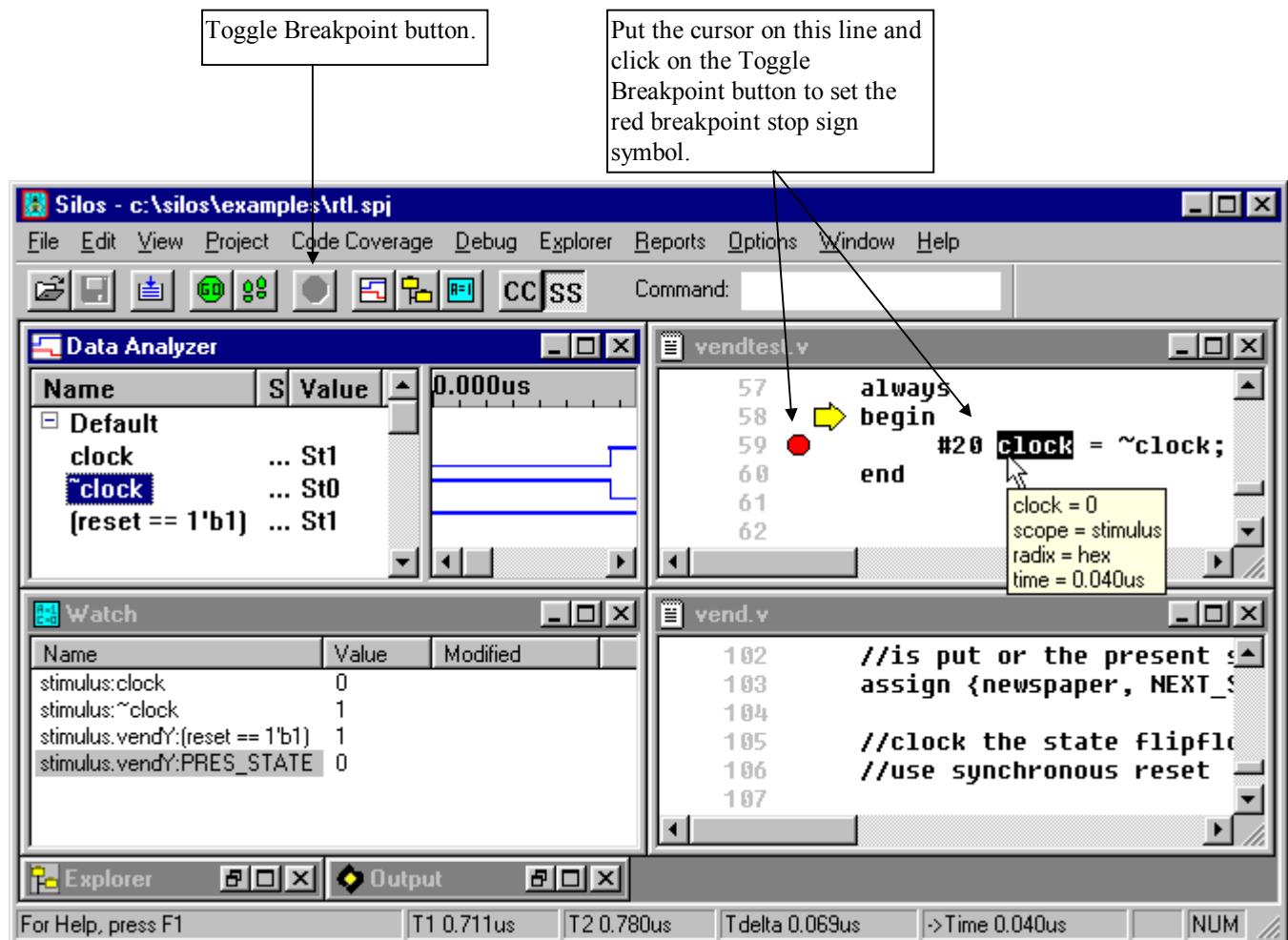
- Break at Simulation Time: stops logic simulation before the selected time is simulated.
- Break at Location: stops logic simulation before the selected source line is simulated.
- Break in Module Instance: allows you to select a module instance and then stop logic simulation each time any source line in the selected module instance is to be simulated.
- Break in Module (Any Instance): allows you to select a module instance and then stop logic simulation each time a source line in any instance of the module is to be simulated.

(continued on next page)

## Breakpoints

One of the uses for setting a breakpoint is to skip over uninteresting code so that you can continue to single step in the code of interest. For example, to keep single stepping in file “vendtest.v” in the “examples” subdirectory:

- Put the mouse cursor on the line (near the bottom of file “vendtest.v”):  
`#20 clock = ~clock;`
- Click-on the “Toggle Breakpoint” button on the Toolbar. A small, red stop sign symbol will be placed to the left of the line next to the line number.
- Next continue to click on the “Step” button on the Toolbar until you leave file “vendtest.v”.
- Then click-on the “Go” button on the Toolbar to finish the timepoint. Click on the “Go” button a second time and the logic simulation will stop on the breakpoint.
- Now you can continue to single step in module “vendtest.v”.



# Code Coverage

## 2.3 Code Coverage

The Code Coverage “plug-in” feature for Silos offers the following benefits:

- Code coverage reports behavioral lines that did not execute, and operators not fully exercised.
- Code Coverage plug-in is simple to use: flip a switch, simulate, and look for purple dots. No preprocessing of files is required, and the results are directly annotated into the source files.
- Exporting of coverage results.
- Sorting by execution count, file name, module name, line number for the tabular presentation of coverage.
- Double-clicking on an entry in the line coverage report or the operator coverage report directly opens the corresponding source code.
- Annotation of coverage results directly onto the source windows.
- Independent reporting of operands.
- Merging of coverage data from independent simulation runs.
- Ability to turn off coverage for specific lines and blocks of behavioral source code.
- Automatic elimination of spurious time 0 initialization from coverage results.
- Selecting module instances to be covered is graphical with the Explorer Plug-in.

### 2.3.1 Generating Line Coverage Reports

The skills presented in this topic are:

- Setting up the Code Coverage plug-in.
- Displaying a report for lines that are not executed.
- Revealing possible inefficiencies in the behavioral model.
- Sorting by execution count, file name, module name, line number for the tabular presentation of coverage.
- Exporting of coverage results.

(continued on next page)

## Line Code Coverage

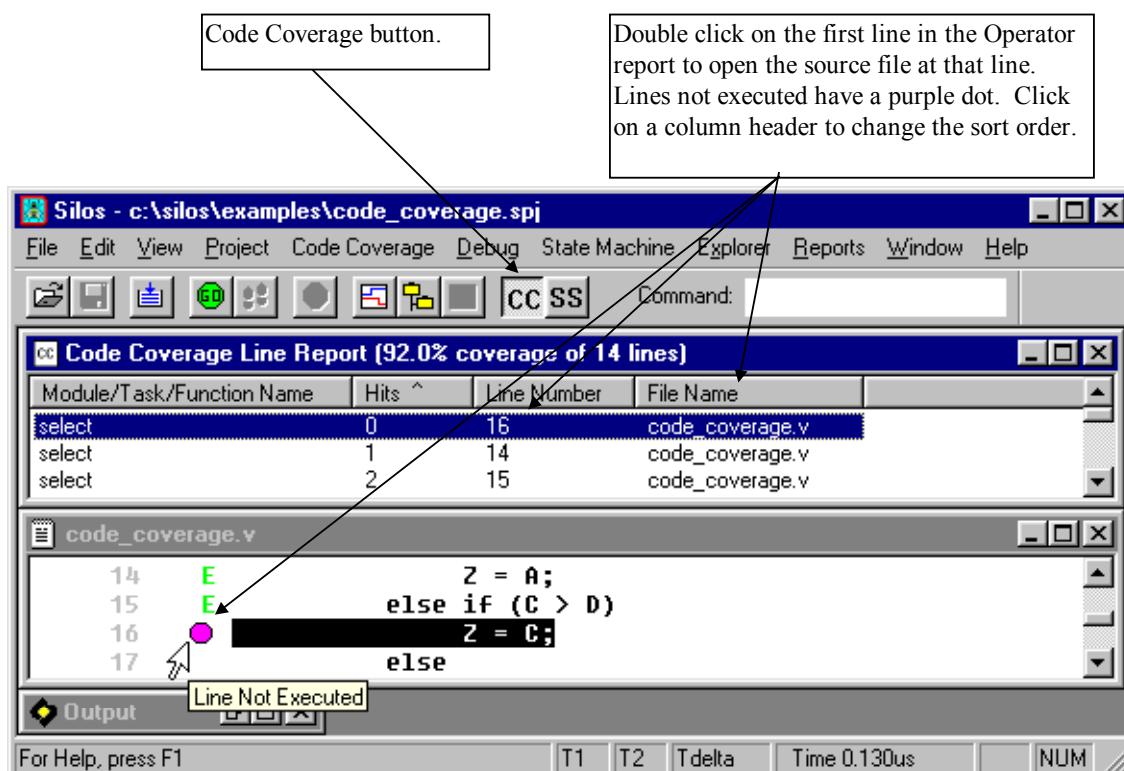
### Procedure

- Start Silos (if it is not already started).
- To open the example for the Code Coverage plug-in, select the “Project/Open” menu selection to open the “Open Project” dialog box. Select the project named “code\_coverage.spj” in the examples directory, and click on the “Open” button to close the dialog box.
- Since this example does not include single stepping or setting breakpoints, these features can be disabled to improve simulation speed by clicking on the “SS” button on the Main toolbar to push the button out (if it is not already pushed out).
- To setup the Code Coverage plug-in, click on the “CC” button if it is not already selected. If the “CC” button is selected after inputting the files for the design, Silos will query if you want to reload the design and setup the Code Coverage plug-in.
- Click on the “Go” button to simulate the design.

(continued on next page)

## Line Code Coverage

- Select the “Code Coverage/Line Report” menu selection to open the “Code Coverage Line Coverage” report. This report is sorted by the number of times each line is executed (“Hits”) in ascending order. Normally the user is interested in which lines are not executed (“0” Hits). However, the report can also be used to see which lines have a high number of executions, indicating a possible flaw in the user’s design. If you want to change the sort order, you can click on the column heading that you wish to sort by.

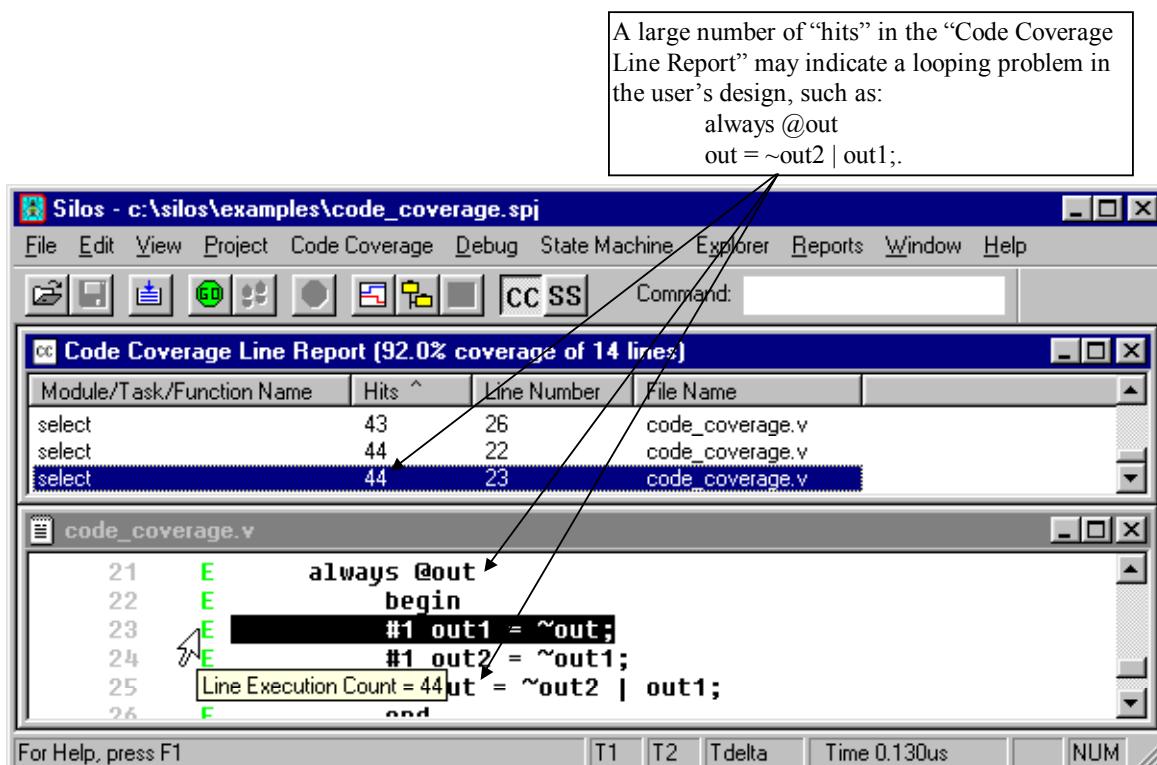


- When the “Code Coverage Line Report” comes up, **double click** on the first entry in the report that has zero “Hits”. This will open the “code\_coverage.v” file, and display the line that did not execute with a purple dot to the left of the line.
- Place the mouse cursor over the purple dot in file “code\_coverage.v”, and a yellow box will pop up stating “Line not executed”.

(continued on next page)

## Line Code Coverage

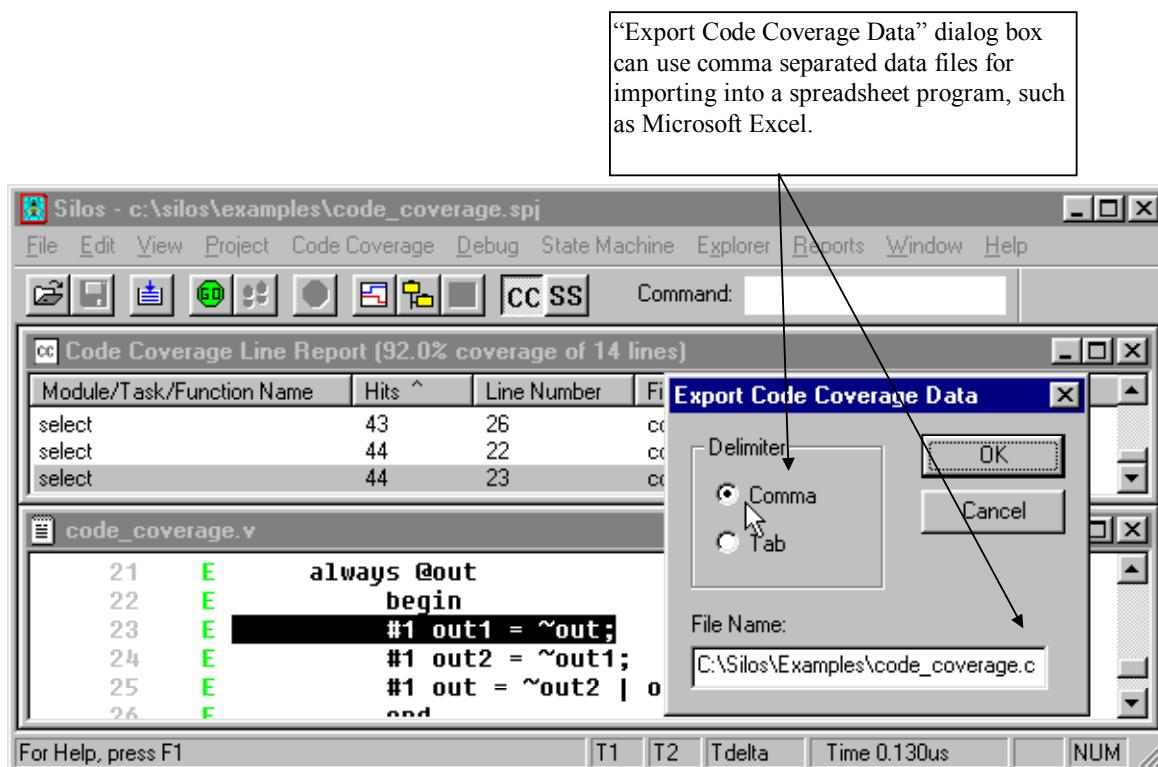
- A large number of hits in the “Code Coverage Line Report” may indicate a looping problem in the user’s design. To see an example of this, **double click** on the last entry in the “Code Coverage Line Report” to automatically go to that line of file “code\_coverage.v”. Notice that this line is in an “always” loop that triggers itself by recursively setting the variable in the sensitivity list. To see how many times the line was executed, **place** the mouse cursor over a green “E”, and a yellow box will pop up stating “Line Execution Count=44”.



(continued on next page)

## Line Code Coverage

- The user may want to display the Code Coverage plug-in results in another program, such as Microsoft Excel. To demonstrate the exporting of the Code Coverage plug-in results, select the “File/Export” menu selection to open the “Export Code Coverage Data” dialog box. Select a file name of your choice, click on the “OK” button to close the dialog box. You can edit the file you created to view the export format, or import the file into a program such as Microsoft Excel.



(continued on next page)

# Operator Code Coverage

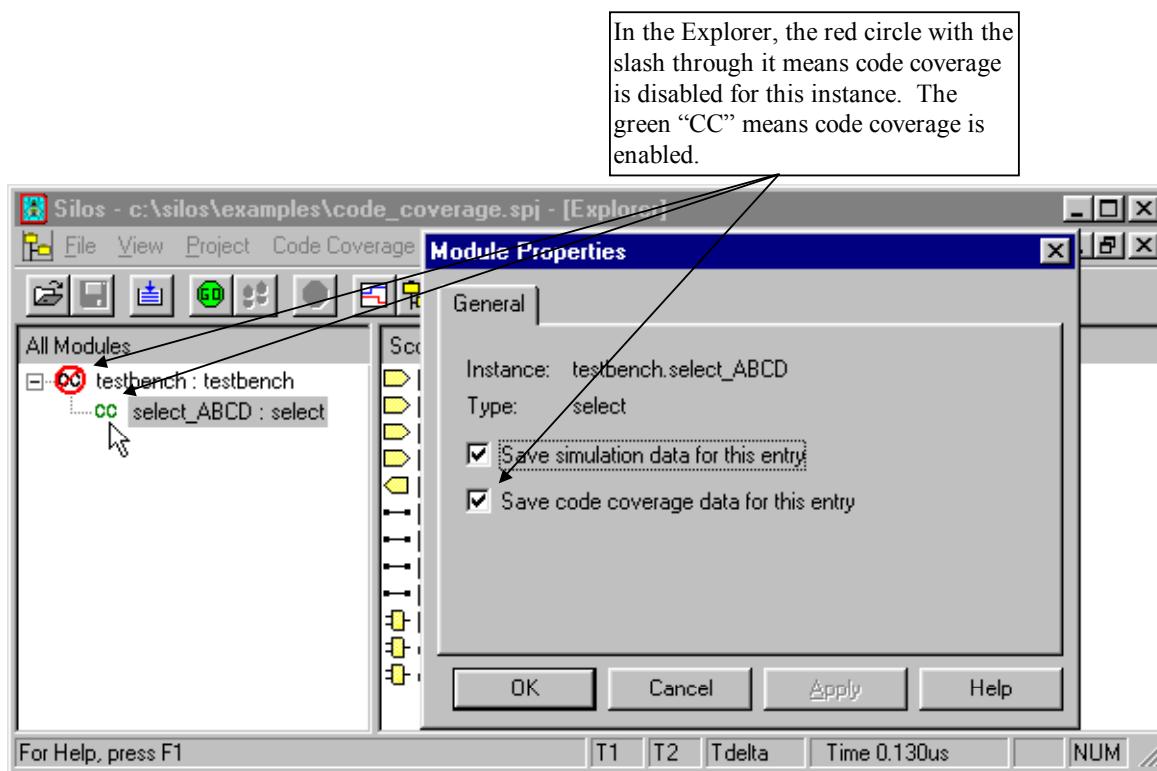
## 2.3.2 Generating Operator Coverage Reports

The skills presented in this topic are:

- Selecting module instances to be covered using the Explorer.
- Displaying a report for operands that fail to affect the corresponding operator.
- Displaying a report for bits in the operands that fail to affect the corresponding operator.

### Procedure

- To eliminate the testbench from code coverage, **click** on the “Open Explorer” button on the Main toolbar to open the Explorer window. **Click** with the right mouse button when the mouse cursor is over the top level instance “testbench”. In the context menu, **select** “Properties” to open the “Module Properties” dialog box. **Uncheck** the “Save code coverage data for this entry” box, and **click** “OK” to close the dialog box.
- To enable the design “select\_ABCD” for code coverage, **click** with the right mouse button when the mouse cursor is over the top level instance “select\_ABCD”. In the context menu, **select** “Properties” to open the “Module Properties” dialog box. **Check** the “Save code coverage data for this entry” box, and **click** “OK” to close the dialog box. The “Save code coverage data for this entry” will be grayed out if the module instance is outside of the device under test (“\$fs\_dut” system task), or if code coverage is not enabled.



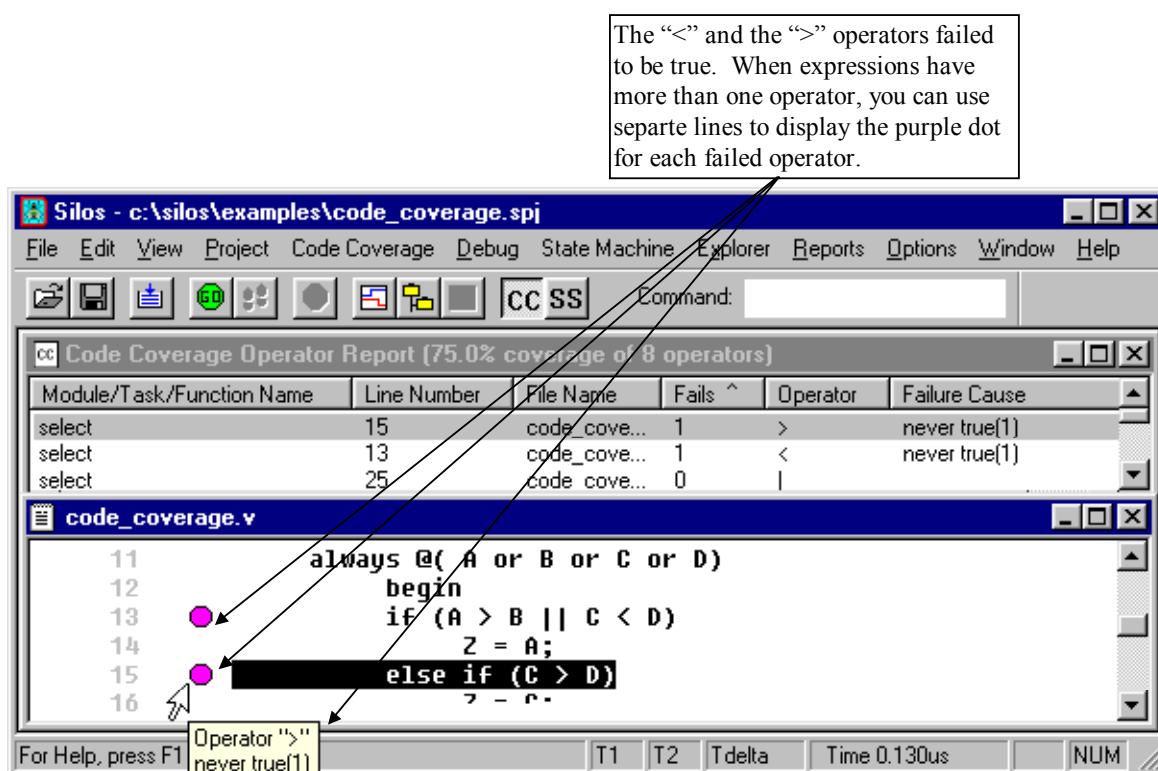
## Operator Code Coverage

- **Select** the “Code Coverage/Enable Operator Coverage” menu selection if it is not already selected. If the “Code Coverage/Enable Operator Coverage” menu is selected after inputting the files for the design, Silos will query if you want to reload the design and setup the Operator Coverage plug-in. **Answer** “Yes” to the query. Enabling the Operator Coverage plug-in can increase the simulation time, so normally the user would not enable this until he is ready to review operator coverage. If the files were reloaded, click on the “Go” button on the Main toolbar to run the simulation.
- **Select** the “Code Coverage/Operator Report” menu selection to open the “Select Operator Report Options” dialog box. The “DEFAULT Operator 1/0 Result” option is already selected, so **click** on the “OK” button to close the dialog box, and open the “Code Coverage Operator Report” list box. The “DEFAULT Operator 1/0 Result” option will report the result of each operator, except for math operators (+,-,/,\*,%).

(continued on next page)

## Operator Code Coverage

- Double click on the first entry in the “Code Coverage Operator Report” list box to go to the corresponding line of file “code\_coverage.v”. In the “code\_coverage.v” file, a purple dot shows that the result for the “>” operator for the expression “C > D” failed to be true. The two green “E’s in file “code\_coverage.v” show that the result for the “>” operator and the “||” operator were completely executed (both true and false). When there is more than one expression on a line, the Operator Report will list them separately. However, if you want to visually see a purple dot and a message in the Silos GUI for each operator, you can put each operator on a separate line in your Verilog HDL source code.

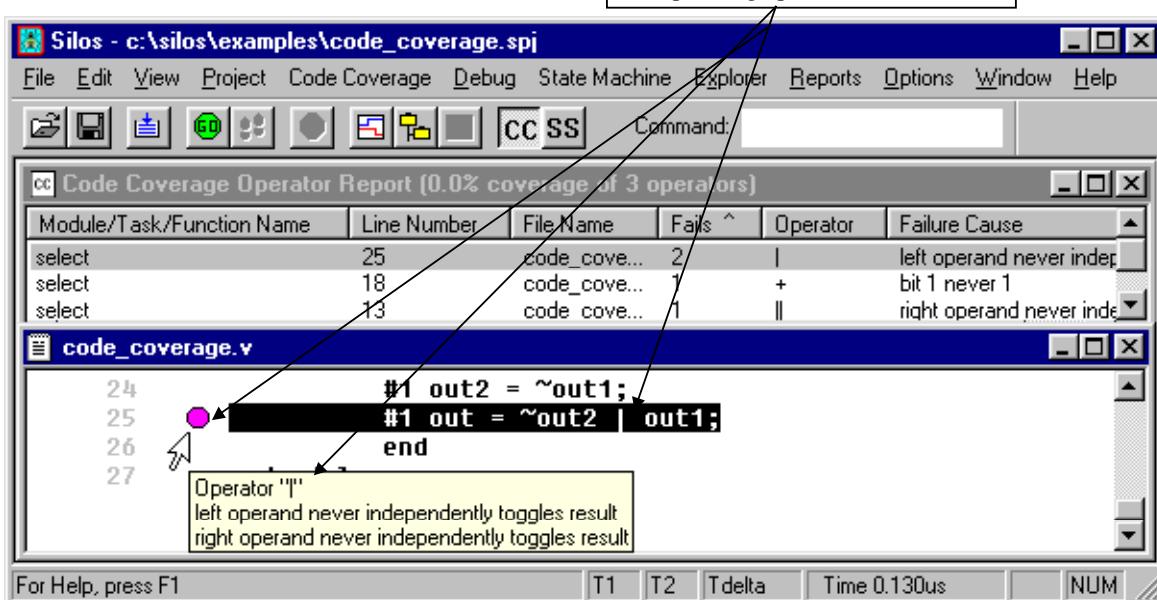


(continued on next page)

## Operator Code Coverage

- The Operator report can also be used to show which operands did not affect the corresponding operator. Select the “Code Coverage/Operator Report” menu selection again to open the “Select Operator Report Options” dialog box. Next deselect the “DEFAULT Operator 1/0 Result”, and select the remaining options in the “Select Operator Report Options” dialog box. This will display a purple dot on each line where an operand failed to affect an operator.

Deselecting the “DEFAULT” option, and selecting the other options for the Operator report will show which operands did not affect their corresponding operator.



(continued on next page)

## Merging Code Coverage

### 2.3.3 Merging Code Coverage Reports

The skill presented in this topic are:

- Merging the code coverage results from two simulations using different testbenches for the same design.

#### Procedure

The user may want to run different testbenches on the same design and merge the results. The user can generate the data for the Code Coverage plug-in reports by simulating with the Silos GUI (“sse.exe”), or by simulating with batch runs (“silos.exe”). The results for the different simulations can be combined and displayed with the Silos GUI.

- To demonstrate the merging for two code coverage runs with different testbenches, **select** the “Project/Open” menu selection to open the “Open Project” dialog box. **Select** project “code\_coverage2.spj”, and **click** on the “Open” button to open the project.
- **Click** on the “Go” button to simulate the design.
- When merging the Code Coverage plug-in results, it is important put the testbench in a separate file, so that changing the testbench does not change the code coverage results. It is also important to disable code coverage for the testbench, as there may be problems in trying to merge the testbench after it changes, and it is unlikely the user wishes to see code coverage for the testbench. To see how to disable code coverage for the testbench, **select** the “File/Open” menu selection to open the “Open” dialog box. **Select** the testbench for this run, file “testbench2.v”, and **click** on the “Open” button to open the file. **Notice** that file “testbench2.v” contains the following compiler directives for code coverage (file “testbench.v”, the testbench for the previous simulation, also contained these compiler directives):

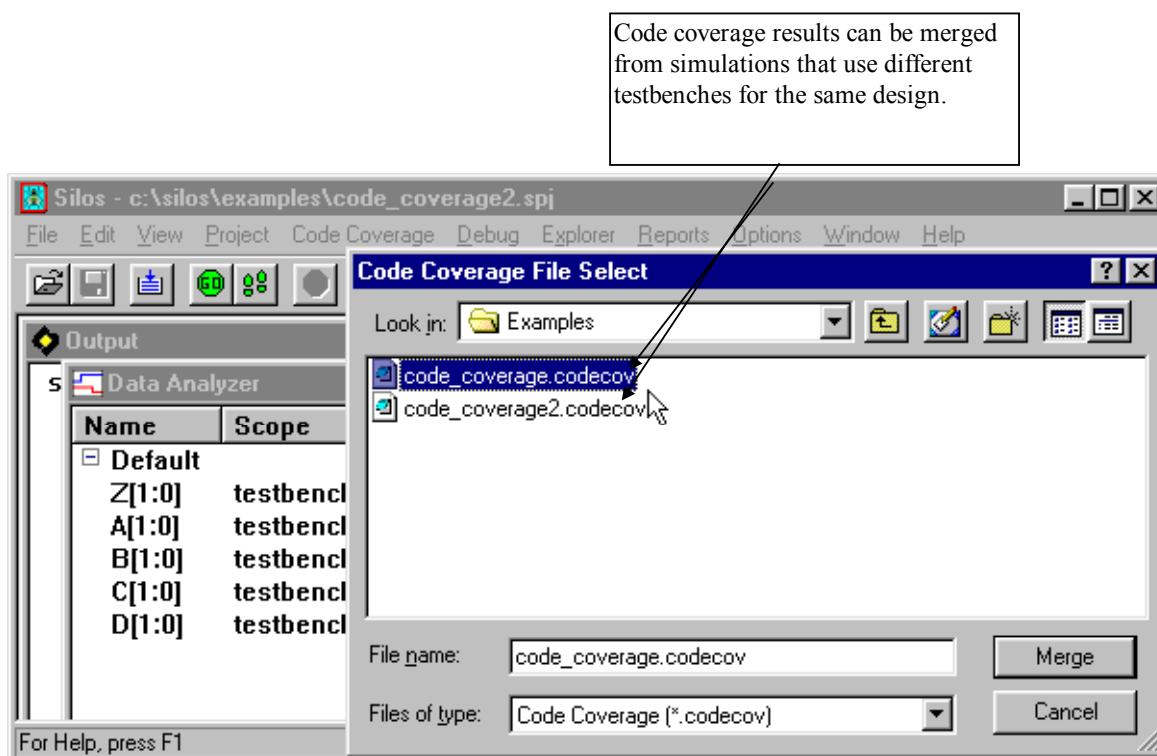
```
`disnable_codecoverage
module testbench;
...
endmodule
`enable_codecoverage
```

The “`disnable\_codecoverage” compiler directive turns off code coverage for the lines that follow it. The “`enable\_codecoverage” turns code coverage back on. These compiler directives can also be used to turn off code coverage for testbenches, libraries and other parts of the design that the user is not interested in. Enabling operator coverage may increase the runtime, so it can be an advantage to turn off those parts of the design that are not of interest.

(continued on next page)

## Merging Code Coverage

- To merge the results from this run and the previous run, select the “Code Coverage>Select Files to Merge” menu selection to open the “Code Coverage File Select” dialog box. Select file “code\_coverage.codecov”, and click on the “Merge” button to merge the code coverage results from the first run with those of the current run. This dialog box is additive, so you can select multiple files if you want, and you can use this dialog box again to add additional results. If you want to reset the code coverage results, you would need to click on the “Load/Reload Input Files” button on the Main toolbar to reload the design.



- Now that code coverage results have been merged, you can select the “Code Coverage/Line Report” menu selection or the “Code Coverage/Operator Report” menu selection to review the combined code coverage results.

## FSM Entry

### 2.4 Finite State Machine (FSM) Entry

Using a Finite State Machine (FSM) entry tool simplifies documentation and debugging.

- The animation between the Finite State Machine window and the Data Analyzer simplifies debugging.
- Scanning from state to state in the Finite State Machine window moves the T1 timing marker in the Data Analyzer, making it simple to view the waveforms for the Finite State Machine states.
- As the T1 marker is moved across the Data Analyzer, the Finite State Machine window shows which state is active.
- The Finite State Machine window can be minimized and still show which state is active, allowing the user to display more waveforms in the Data Analyzer.

#### 2.4.1 Creating a Simple FSM

The skills presented in this topic are:

- Creating a simple Finite State Machine (FSM).
- Documenting outputs for the FSM.
- Specifying clock and reset line.
- Saving the FSM and creating the Verilog HDL source code for the FSM.

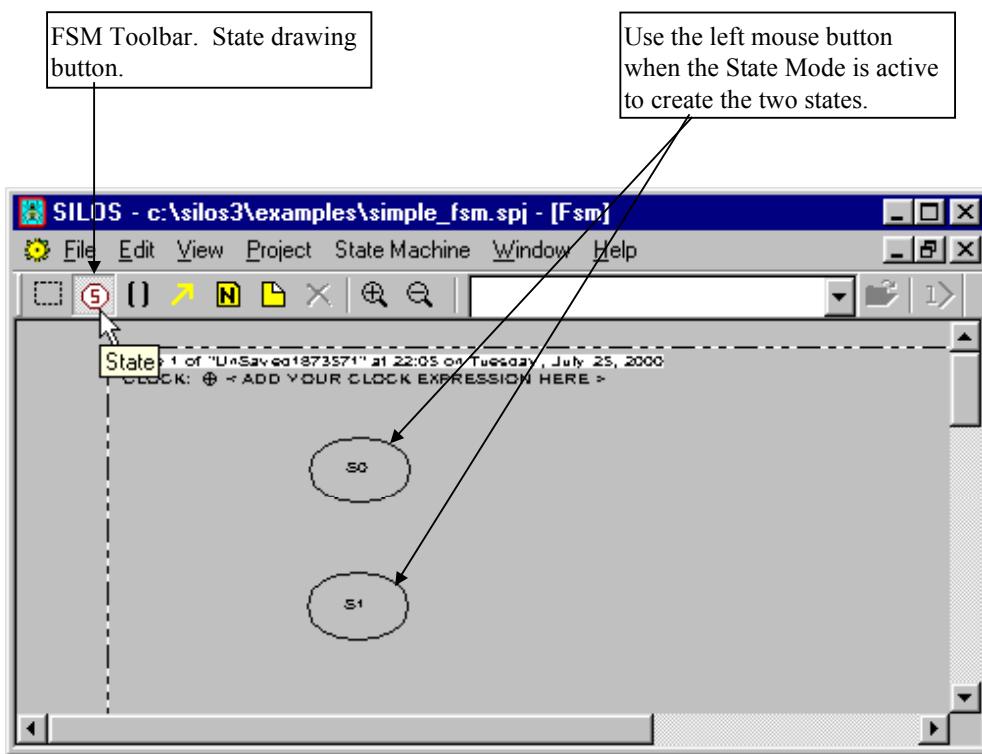
(continued on next page)

## FSM Entry

### Procedure

Creating a finite state machine (FSM) is simple using Silos.

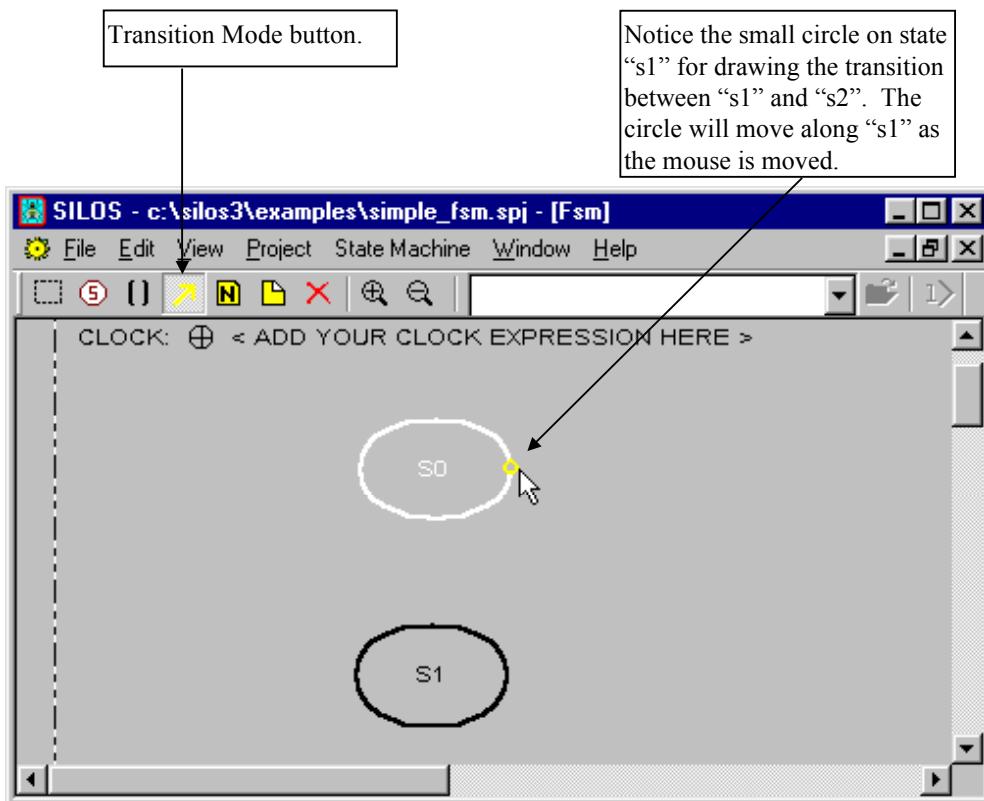
- For this example, **select** the Project/New menu selection to create a new project named “simple\_fsm” in the “Create New Project” dialog box. **Click** on the Save button to close the dialog box. When the Project Files dialog box appears, **click** on the Cancel button as you do not need to add any files at this point.
- **Select** the State Machine/New menu to open an “FSM” window. **Full screen** the FSM so that you have room to work.
- To draw two states, **click on** the State button on the FSM Toolbar. **Place** the mouse cursor in the FSM window where you want to draw the state symbol, and **click and release** the left mouse button to place the state. When the state symbol is created, it has a text box inside it with a highlighted question mark “?”. To name the state, use the computer’s keyboard to **enter** “s1” as the state name. **Create** a second state below the first state, naming it “s2”.



(continued on next page)

## FSM Entry

- To create a transition from state “s1” to state “s2”, **select** the “Transition” button on the FSM Toolbar. To select state “s1”, **click and release** with the left mouse button anywhere on the oval symbol for state “s1”. Once state “s1” is selected, a small circle will appear on the state symbol for “s1” whenever the mouse cursor is near it.

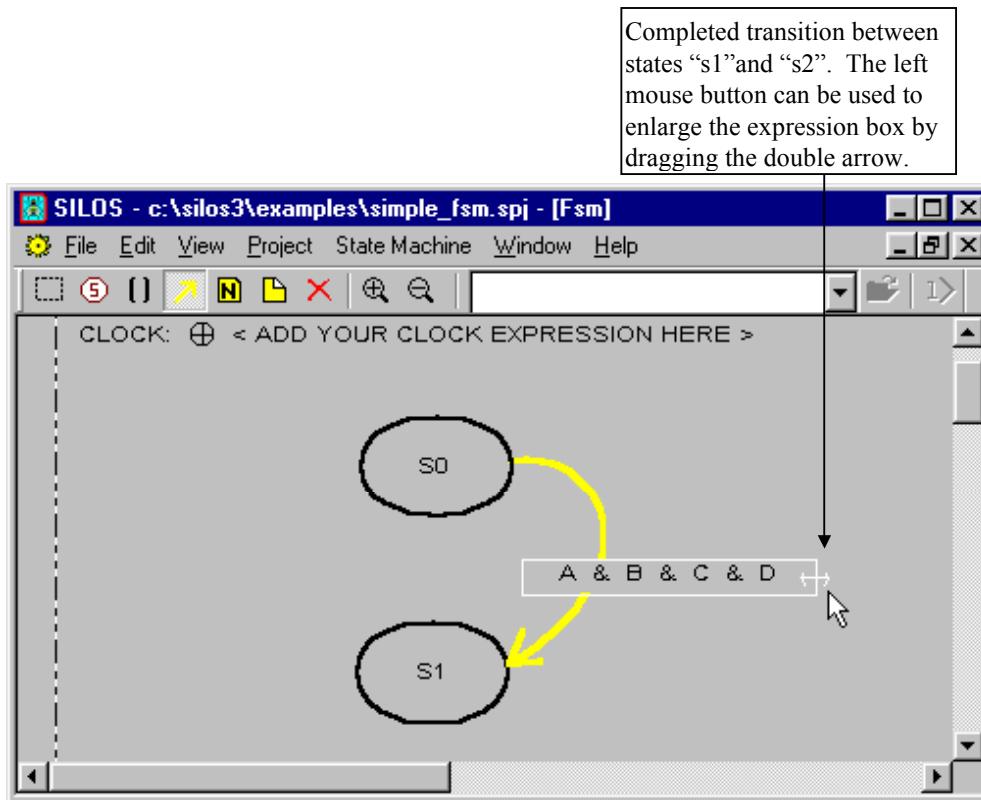


To draw the transition from state “s1” to state “s2”, **position** the mouse cursor on the oval symbol for state “s1”, **click and hold** down the left mouse button, and **drag** the mouse cursor from state “s1” to state “s2”. When the mouse cursor gets near state “s2”, an arrow will appear on the transition line. You can **release** the left mouse button after the arrow appears, and the transition between “s1” and “s2” is complete.

(continued on next page)

## FSM Entry

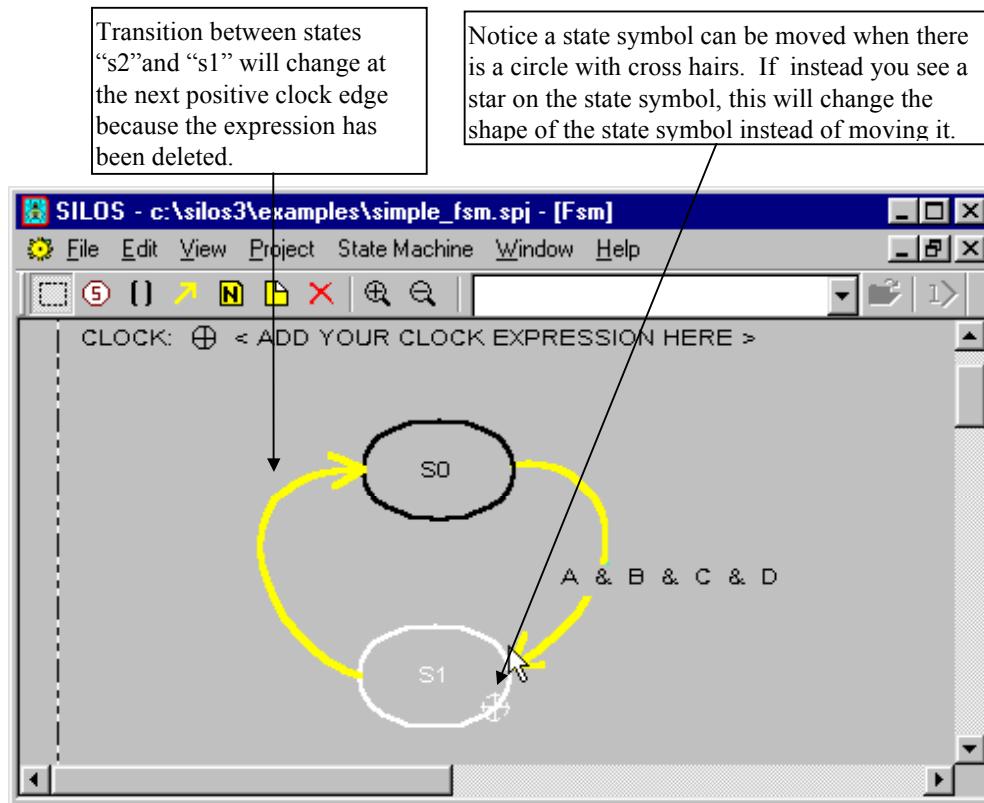
- Enter the expression “A & B & C & D” in the expression box between states “s1” and “s2”. To enlarge the expression box, put the mouse cursor on the left or right edge of the expression box, and drag the double arrow by holding down on the left mouse button and moving the mouse. Notice you can also enlarge the expression box vertically to allow additional lines for the expression.



(continued on next page)

## FSM Entry

- To create a transition that changes unconditionally at the next positive clock edge, draw a transition from state “s2” to state “s1”, and use the “Del” key on the keyboard to delete the expression box for the transition.

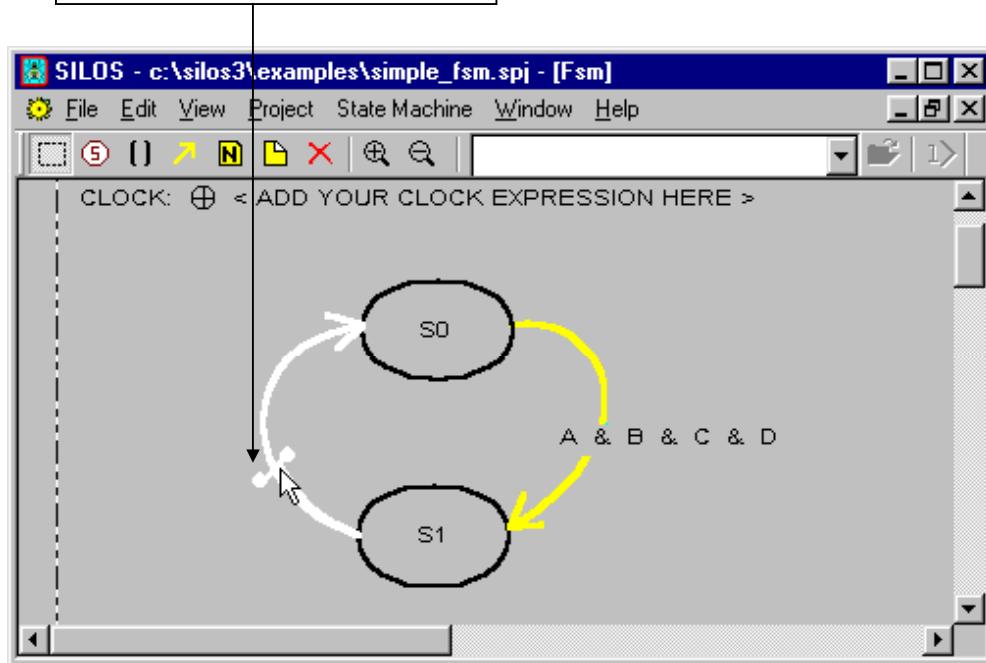


(continued on next page)

## FSM Entry

- You can also redraw a transition line by moving the mouse cursor tangentially across the transition line, and using the left mouse button to redraw the transition.

The transition can be redrawn by moving the mouse cursor tangentially across the transition line with the left mouse button held down.

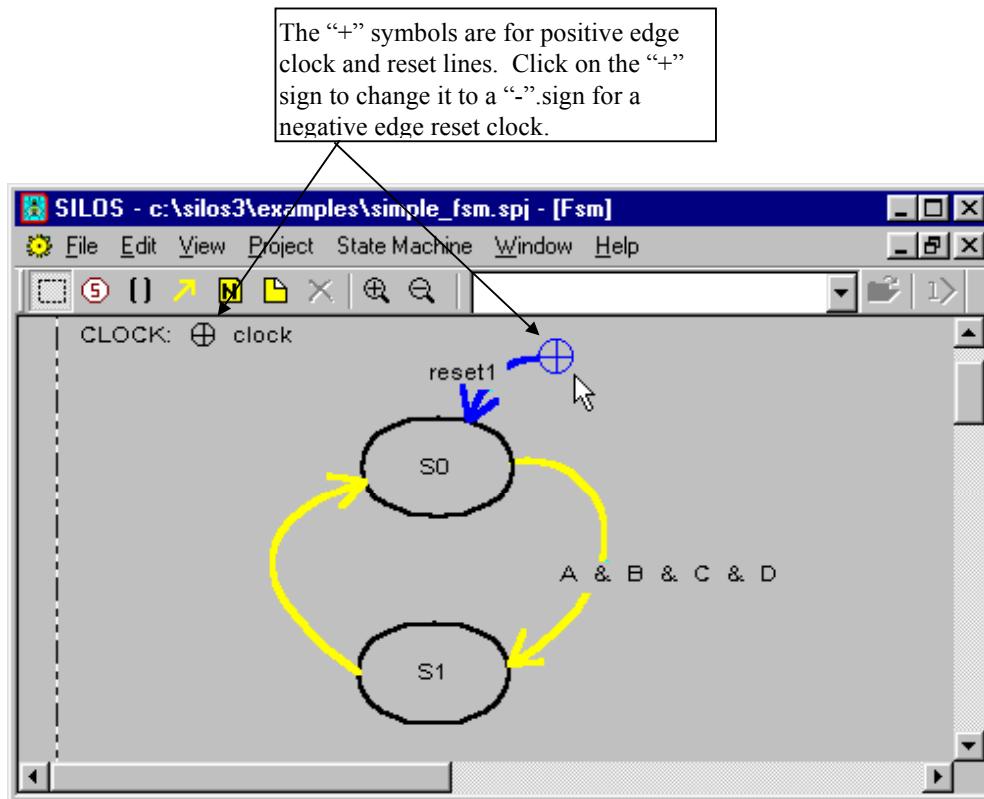


(continued on next page)

## FSM Entry

### Specifying the Clock and Reset Line:

- Each finite state machine needs to have at least one reset line and a clock line. You can specify a reset line by **clicking** on the Transition mode button on the FSM Toolbar. Then **click and hold down** the left mouse button in the background area of the FSM window, and **draw** a transition line that is only connected to the state “s0”, and not from any state. This will create a blue reset line, that you can give any name. The tail of the reset line has a “+” symbol for a positive edge reset line. For a negative edge reset line, click on the “+” symbol on the tail of the reset line to change it to a “-” symbol. The clock is specified in the upper left hand side of the first page for the finite state machine. To specify the clock name, **click** on the Select button on the FSM toolbar, and then **pass** the mouse cursor over the box that states “<ADD YOUR CLOCK EXPRESSION HERE>” to highlight the text, and **replace** the entire text string with the name of the clock.



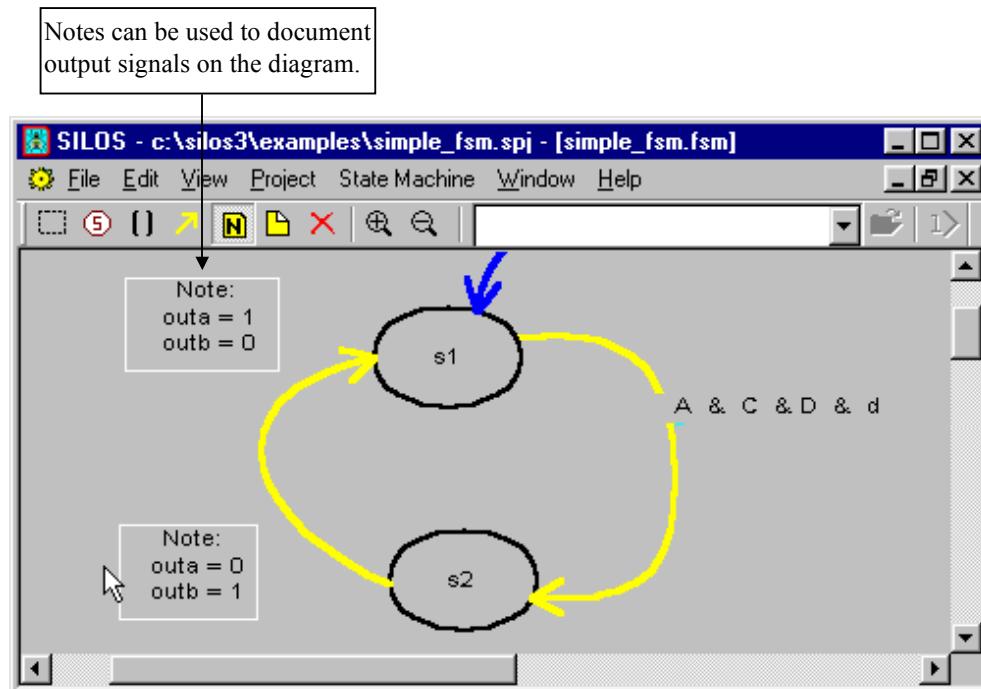
(continued on next page)

## FSM Entry

### Using Notes to Document Outputs for the FSM:

FSM outputs can be implemented manually in two steps.

- Step 1) Show the outputs on the FSM diagram using Notes (use the “Note” button on the FSM Toolbar).



- Step 2) Manually code the combinatorial output block for the outputs and insert it after the `include for the Finite State Machine (see “Debugging using a Finite State Machine” on page 2-75), e.g.:

```
always @($0 or $1) begin
    if ($0) begin
        outa = 1;
        outb = 0;
    end
    else if ($1) begin
        outa = 0;
        outb = 1;
    end
end
```

(continued on next page)

## FSM Entry

### Saving a finite state machine and creating Verilog HDL Source Code:

- To save the finite state machine, **select** the “State Machine/Save As” menu. The “Save Finite State Machine As” menu will appear, and you can **enter** a file name such as “simple\_fsm”. The finite state machine is saved, and the Verilog HDL source code is automatically written to file “simple\_fsm.v”. Inserting the Verilog HDL source code for a finite state machine into your design, and simulating it, is covered in the next topic on “Debugging using a Finite State Machine” on page 2-75

(continued on next page)

## FSM Debugging

### 2.4.2 Debugging using a Finite State Machine

The skills presented in this topic are:

- Inserting the finite state machine (FSM) source code into a Verilog HDL design.
- Simulating the FSM.
- Animating the FSM states and viewing the state changes in the Data Analyzer.

#### Procedure

For demonstrating the debugging capabilities of the Finite State Machine window, an existing project and Finite State Machine for a newspaper vending machine will be used.

Using `include to insert a finite state machine:

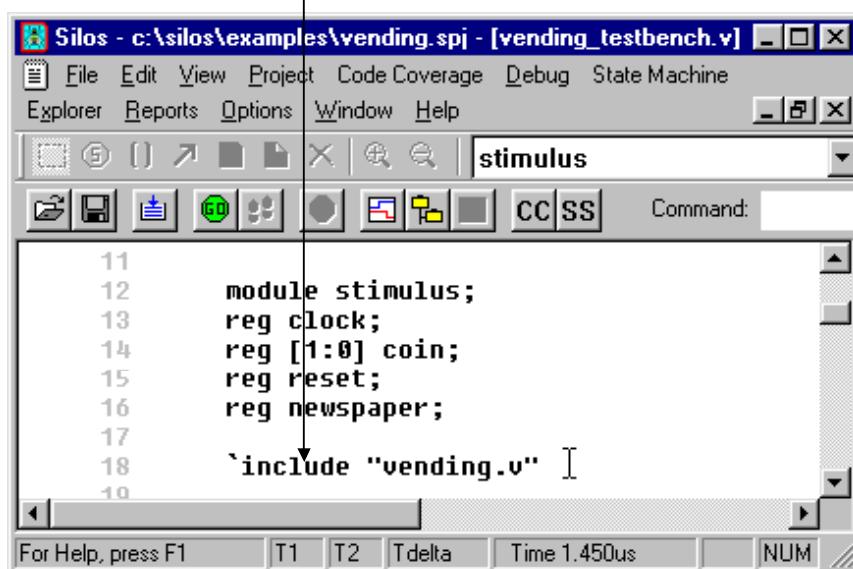
- **Select** the Project/Open menu selection to open the “Open Project” dialog box. **Double click** on the project named “vending.spj” in the “examples” subdirectory for the Silos installation.

(continued on next page)

## FSM Debugging

- The Verilog HDL source code for the finite state machine has already been written to a file named “vending.v” when the finite state machine “vending” was saved. To include the Verilog HDL source code for the finite state machine into this design, the ‘include compiler directive was used. To see an example of this, use the “Open File” button on the Main Toolbar to open the file “vending\_testbench.v”. When you scroll down in this file you can see the ‘include “vending.v” compiler directive to include the Verilog HDL source code for finite state machine “vending”, which models a newspaper vending machine.

The ‘include “vending.v” compiler directive is used to include the Verilog HDL source code for the newspaper vending FSM.



```

11
12     module stimulus;
13     reg clock;
14     reg [1:0] coin;
15     reg reset;
16     reg newspaper;
17
18     `include "vending.v"
19

```

### Simulating the finite state machine:

- To simulate the design for the newspaper vending machine, **click** on the “Go” button on the Main Toolbar. After simulation is complete, **minimize** the Output window and **close** the “vending\_testbench.v” window.

(continued on next page)

## FSM Debugging

### Animating the FSM states and viewing the state changes in the Data Analyzer:

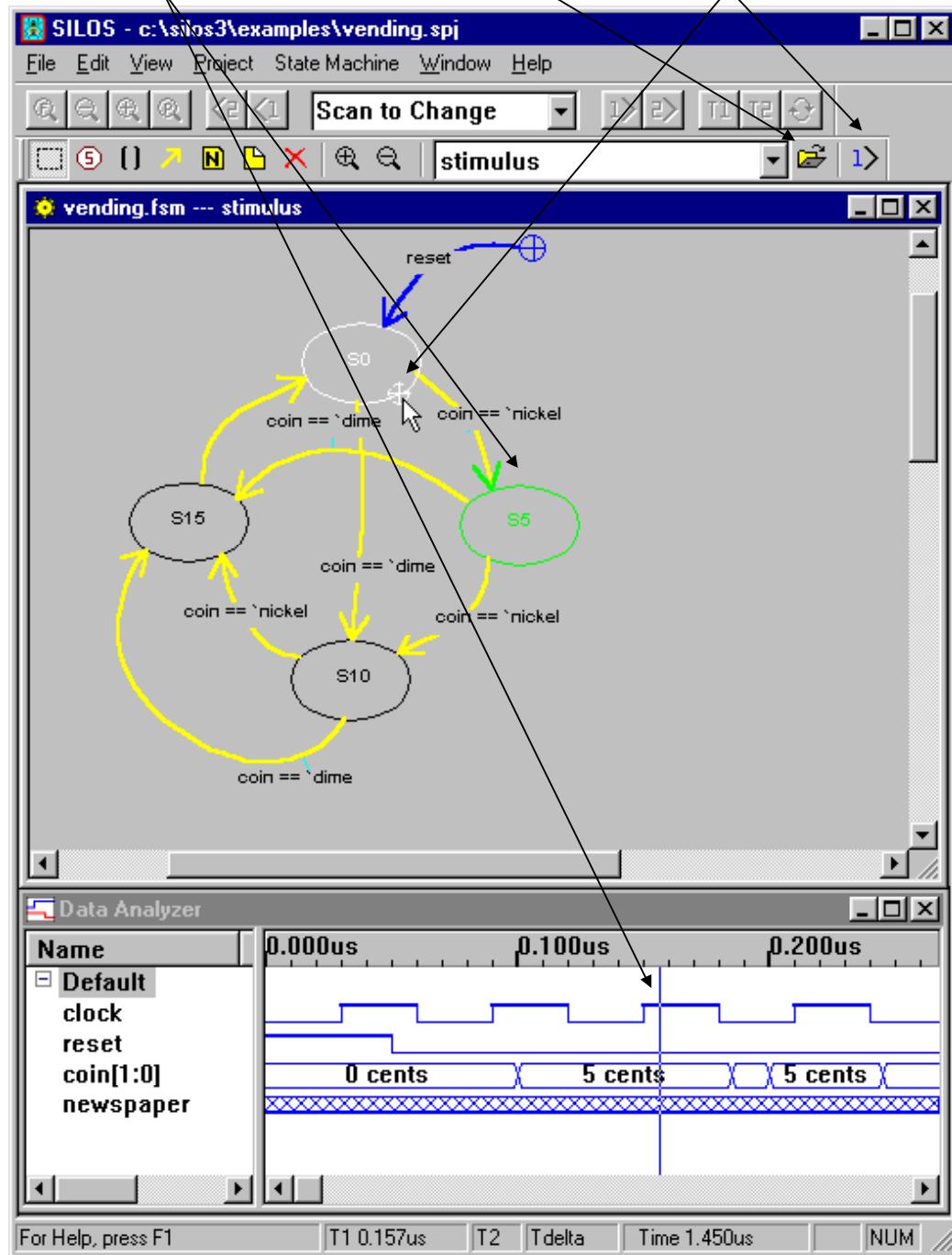
- To view the animation for the finite state machine, click on the “Open Analyzer” button on the Main Toolbar to open the Data Analyzer for viewing waveforms. Next open the single instance named “stimulus” for the “vending” finite state machine by **clicking** on the “Open Instance” button on the FSM Toolbar. Notice that the active state is colored green instead of black in the finite state machine window “stimulus”. Please note that only an instance of a finite state machine can be used to view the animation. The window that the finite state machine was created in is not an instance of the finite state machine, and will not work with the animation. Use the “Window/Tile” menu to tile the Data Analyzer and FSM windows.
- To display which state is active at a time point in the Data Analyzer, **hold down** the left mouse button to place a T1 timing marker in the waveform window. With the left mouse button **held down, drag** the T1 timing marker to the right in the waveform window. Notice that the highlighting for the states in the Finite State Machine window changes as each state becomes active.

(continued on next page)

As the “T1” timing marker is dragged in the Data Analyzer, the states change color as they become active.

Open Instance button .

“FSM Scan” button. To select states for scanning, click on the state with the left mouse button and hold down the “Ctrl” key.



(continued on next page)

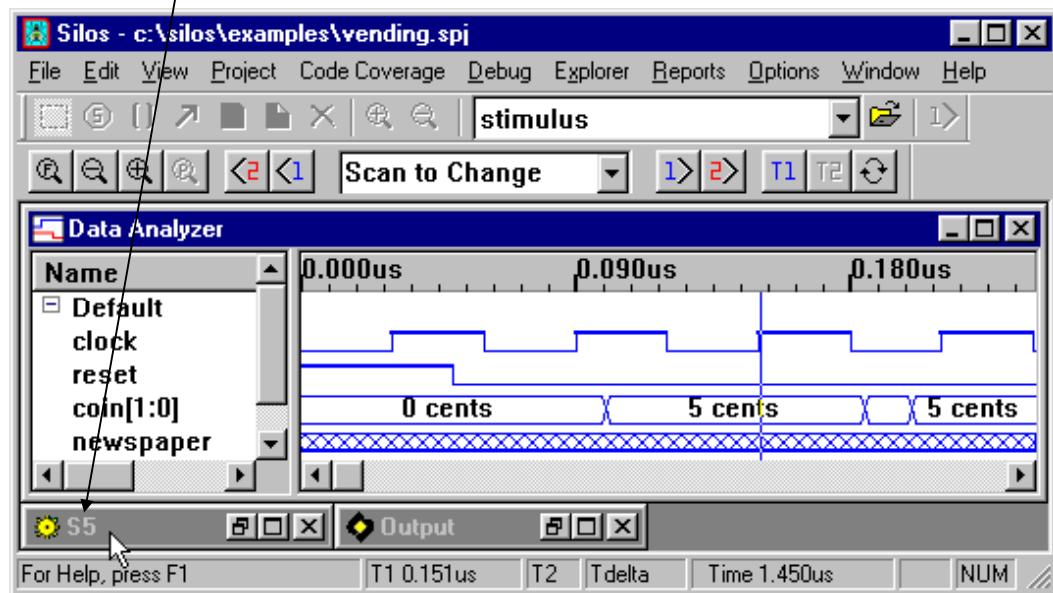
## FSM Debugging

- To select states for scanning with the “FSM Scan” button on the FSM Toolbar, **click and release** with the left mouse button on each state in the FSM windows. **Hold down** the “Ctrl” key on the keyboard to keep each state selected as you select the states. Each state will change to a white color when it is selected. Another method to select states is to click and hold down the left mouse button in the background of the FSM window, and as the mouse cursor is dragged a dotted rectangular box is created that will select anything within the boundary of the box. When scanning, change the focus to another window so the color for the active state will be green and the color for the inactive states will be black (instead of every selected state being white). For example, **click** on the Data Analyzer window so it has the focus. Now **click** on the “FSM Scan” button on the FSM toolbar and scan from state to state in the FSM window as the T1 timing marker is advanced in the Data Analyzer Window. Normally, the “FSM Scan” button will scan to the timepoint where a selected state becomes active. However, if you hold down the shift key when you click on the “FSM Scan” button, the “FSM Scan” button will scan to the timepoint where none of the states that are selected are active. For example, if you selected states “S0” and “S5”, when you hold down the shift key and click on the FSM scan button the scanning will stop at state “S10”.
- To see more waveforms in the Data Analyzer window, you can **minimize** the Finite State Machine window, and **select** the “Window/Tile” menu to tile the windows displayed by Silos. Notice the minimized icon for the Finite State Machine window displays the active state name as you **scan** from state to state using the “FSM Scan” button. This concludes the tutorial example for the finite state machine.

(continued on next page)

## FSM Debugging

The minimized Finite State Machine window shows the active state as you scan in the Data Analyzer.



# Gate Level Debugging

## 2.5 Gate level Debugging

The skill presented in this topic is:

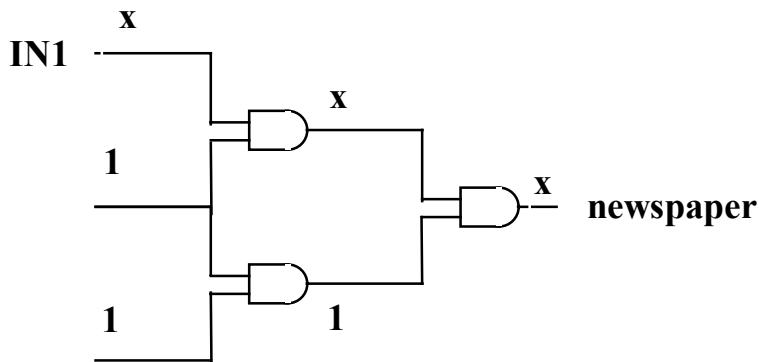
- Tracing back through a gate design to find the cause of an Unknown level.

### 2.5.1 Trace Signal Inputs

#### Benefit

When synthesizing an RTL design, the resulting gate design may not be what the designer intended and the simulation results may differ from the RTL design. Debugging the gate level design can be very difficult because there is no schematic. Also, the synthesis tool and the place and route tool may change existing names, or create many new names.

Silos has the ability to trace the signal backwards through the topology to find the cause of the undesired state value. Such as, for the simple schematic shown below, if signal “newspaper” is at an Unknown (“x”) level, you could use the “Trace Signal Inputs” feature to trace backwards to find that input “IN1” was the cause of the Unknown level.



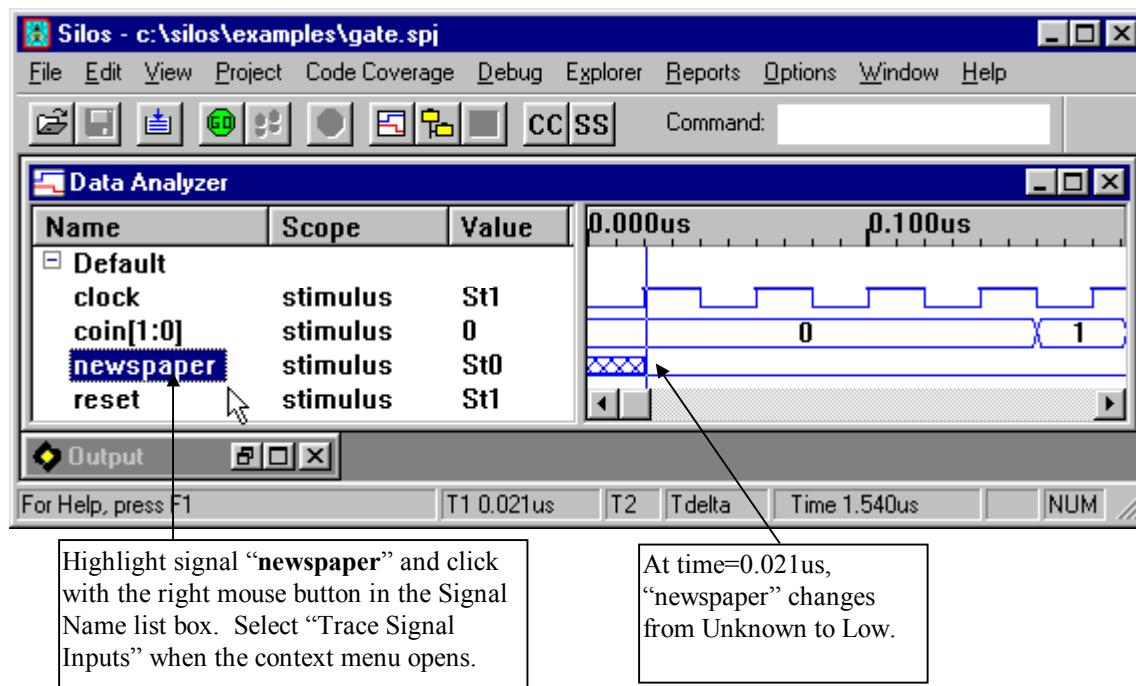
(continued on next page)

## Gate Level Debugging

### Procedure

For the Tutorial, let's use the “Trace Signal Inputs” menu to find out why signal “newspaper” went from an Unknown level to a Low level at time=0.021micro seconds (us) in the gate level synthesized design for the newspaper vending machine:

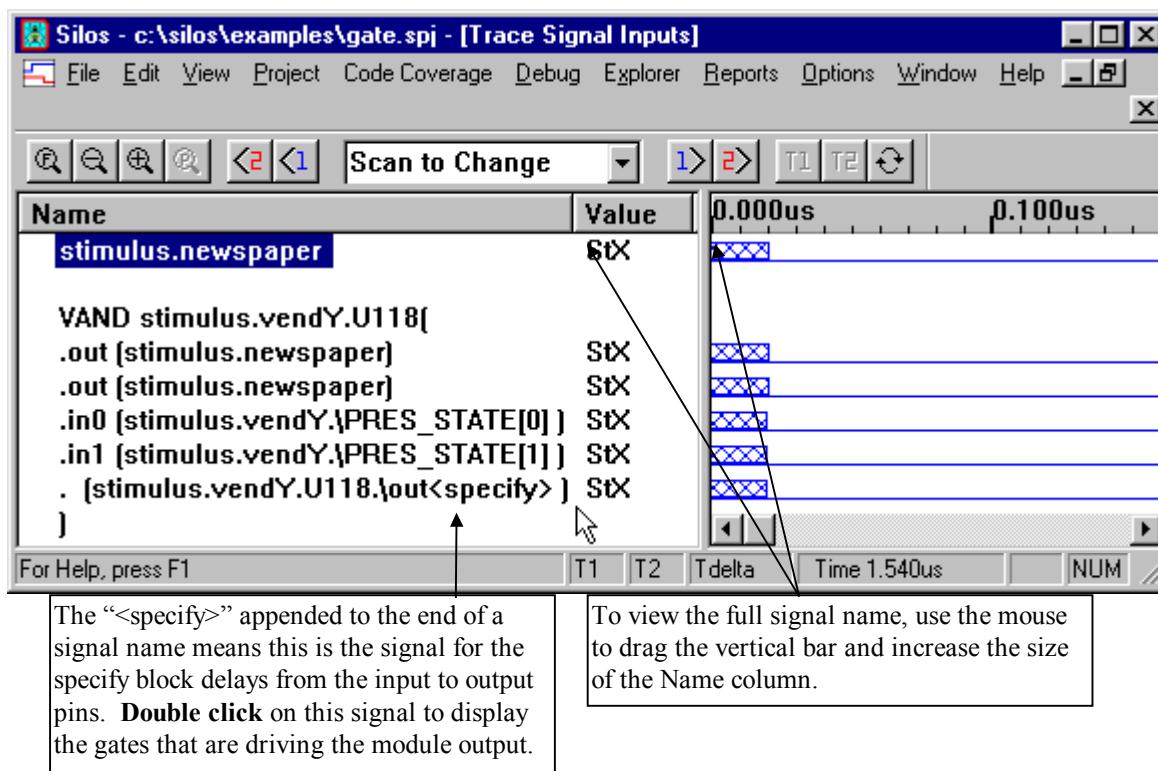
- **Select** the Project/Open menu to open the “Open Project” dialog box.
- **Highlight** the “gate.spj” project file (gate level design for the Newspaper Vending Machine FSM) and **click on** the Open button in the “Open Project” dialog box.
- To increase simulation speed, **disable** (pushed out) the Code Coverage plug-in “CC” button and the single stepping/breakpoints “SS” button on the Main toolbar. Code Coverage applies only to behavioral code, and Debugging enables single stepping and breakpoints for behavioral code, neither of which apply to a gate level simulation. **Click on** the “Go” button to simulate the design.
- **Click on** the “Open Analyzer” button to open the Data Analyzer (unless the Data Analyzer automatically opens).



- To open the context menu, **click** with the right mouse button on signal “newspaper” in the “Name” list box.
- Then **select** the “Trace Signal Inputs” menu to open the “Trace Signal Inputs” window.

(continued on next page)

## Gate Level Debugging

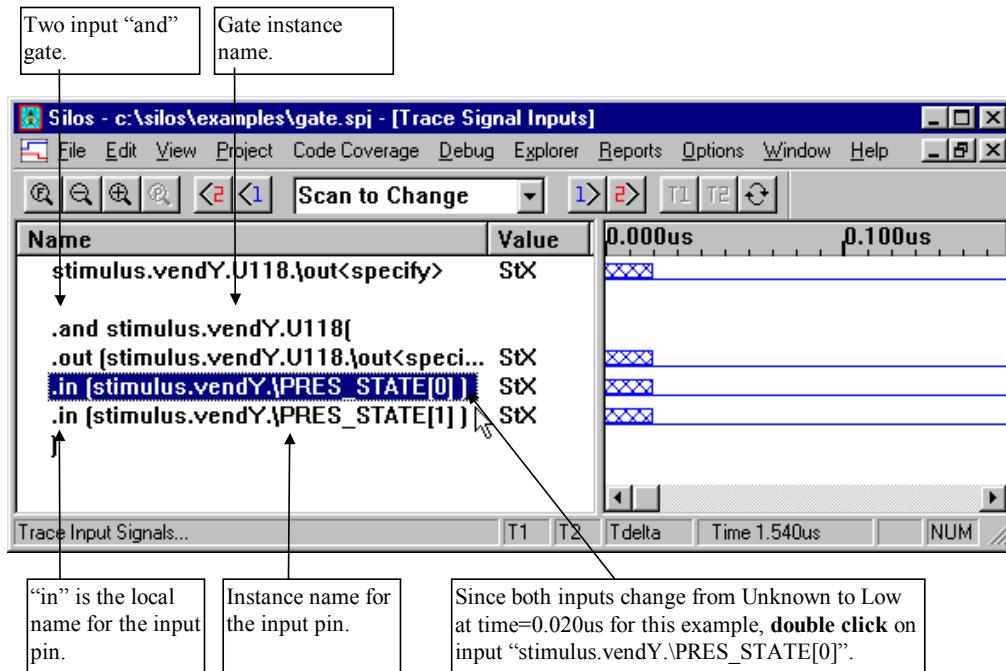


- Since signal “newspaper” is a module port, the “Trace Signal Inputs” window displays every variable for module “vend”. However, you can find the gate that is driving signal “newspaper” by double clicking on the signal that has the word “<specify>” appended to the end of the signal name (this is the specify block for the delay from the input pin to the output pin). For this example, **double click** on signal “stimulus.vendY.U118.\out<specify>”.

Please note: If you see “No Saved Data” instead of a waveform, this means the simulation history was not saved for the variable. A possible cause for not saving the simulation history is that the signal may be inside of a `celldefine boundary. To save data within `celldefine boundaries see the “Project Settings Menu Selection” on page 3-17.

(continued on next page)

## Gate Level Debugging

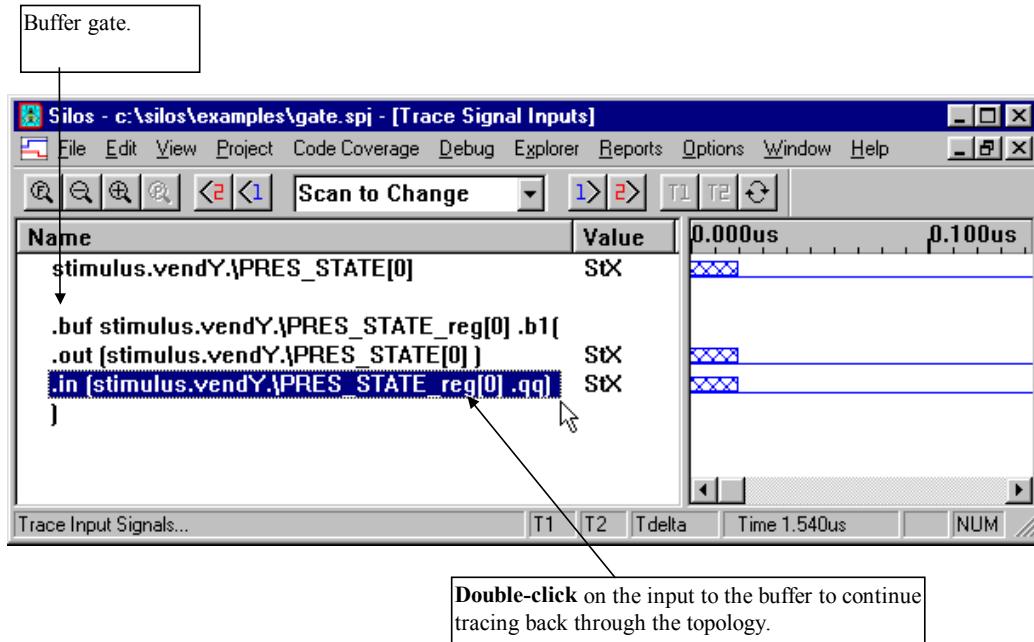


- Signal “stimulus.vendY.U118.\out<specify>” should now be at the top of the Trace Signal Inputs window. Listed below signal “stimulus.vendY.U118.\out<specify>” is the two input “and” gate whose output is driving the signal. Since both inputs for the “and” gate go from Unknown to Low at time=0.021us, you could trace backwards on either input. For the purposes of this example, **double click** on signal “stimulus.vendY.\PRES\_STATE[0]”.

Please note: if you incorrectly double click on another signal and trace back a wrong path, you can “undo” your signal tracing by double-clicking on the signal name that was traced (the signal name at the top of the Name list box).

(continued on next page)

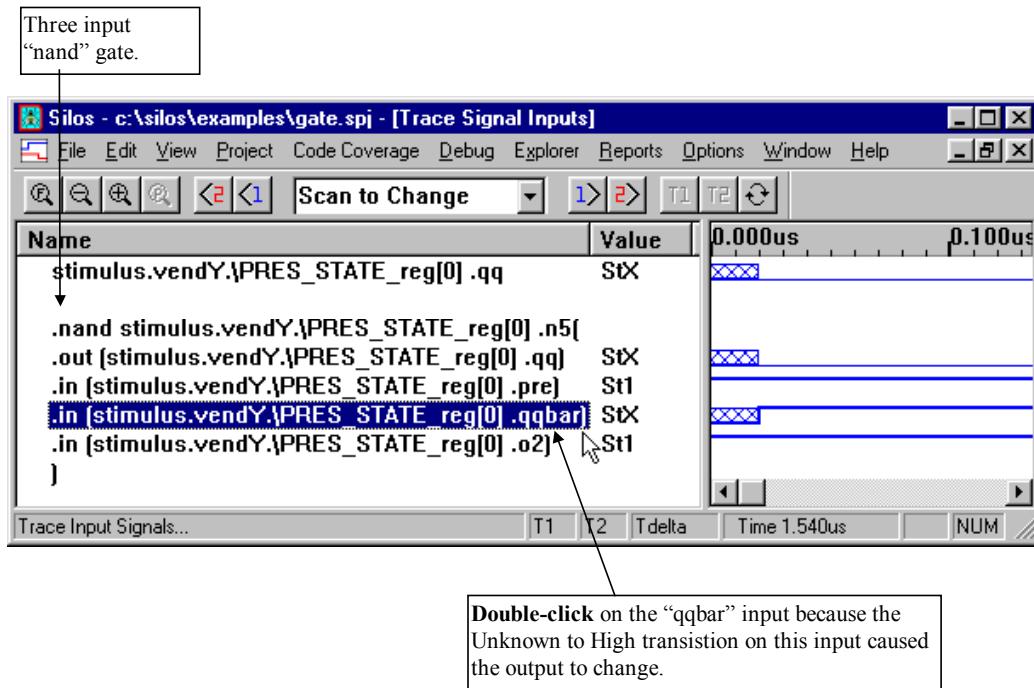
## Gate Level Debugging



- Signal “stimulus.vendY.\PRES\\_STATE[0]” is driven by a buffer gate, so **double click** on the buffers input, “stimulus.vendY.\PRES\\_STATE\\_reg[0].qq” to continue tracing backwards through the topology.

(continued on next page)

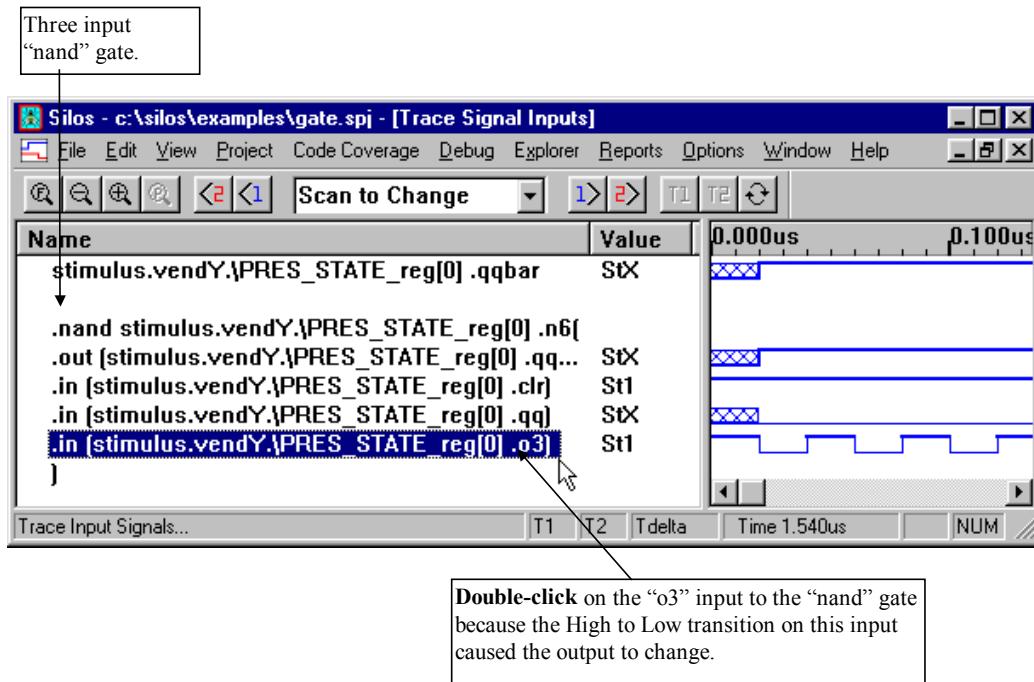
## Gate Level Debugging



- Signal “stimulus.vendY.\PRES\_STATE\_reg[0].qq” is driven by a three input “nand” gate, however, only signal “stimulus.vendY.\PRES\_STATE\_reg[0].qqbar” changes from Unknown to Low, so **double click** on signal “stimulus.vendY.\PRES\_STATE\_reg[0].qqbar” to continue tracing back.

(continued on next page)

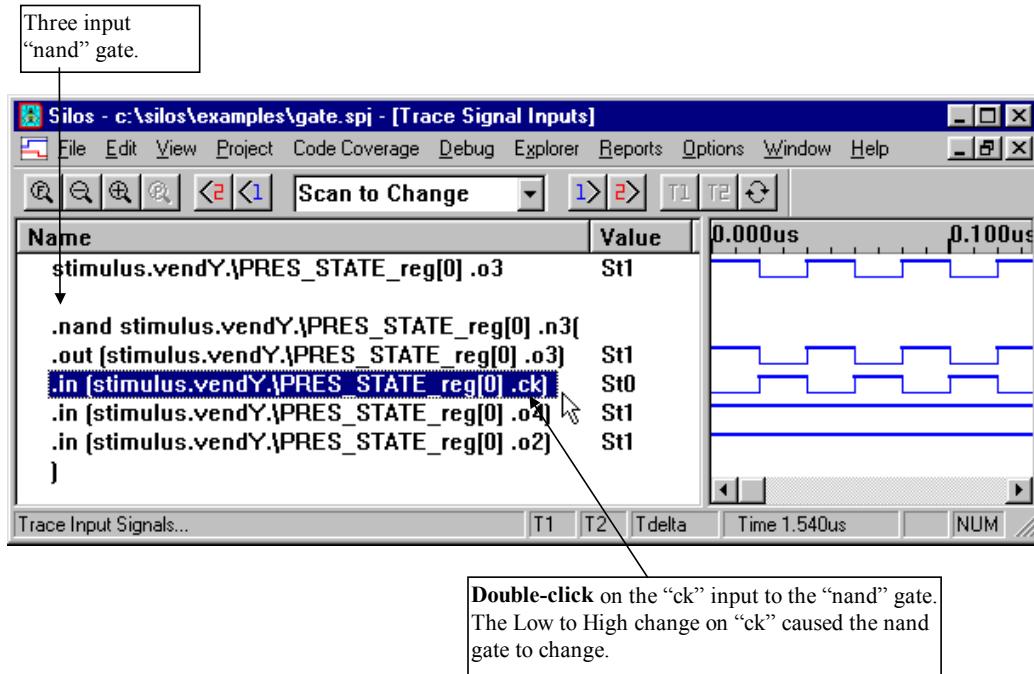
## Gate Level Debugging



- Signal "stimulus.vendY.\PRES\_STATE\_reg[0].qqbar" is driven by another three input nand gate. The "qq" signal and the "qqbar" signal form a latch of back to back "nand" gates, so further back tracing the "qq" input is pointless. Instead, **double click** on signal "stimulus.vendY.\PRES\_STATE\_reg[0].o3" to continue the traceback because this signal changes to Low at time=0.020us causing the output "qqbar" to change.

(continued on next page)

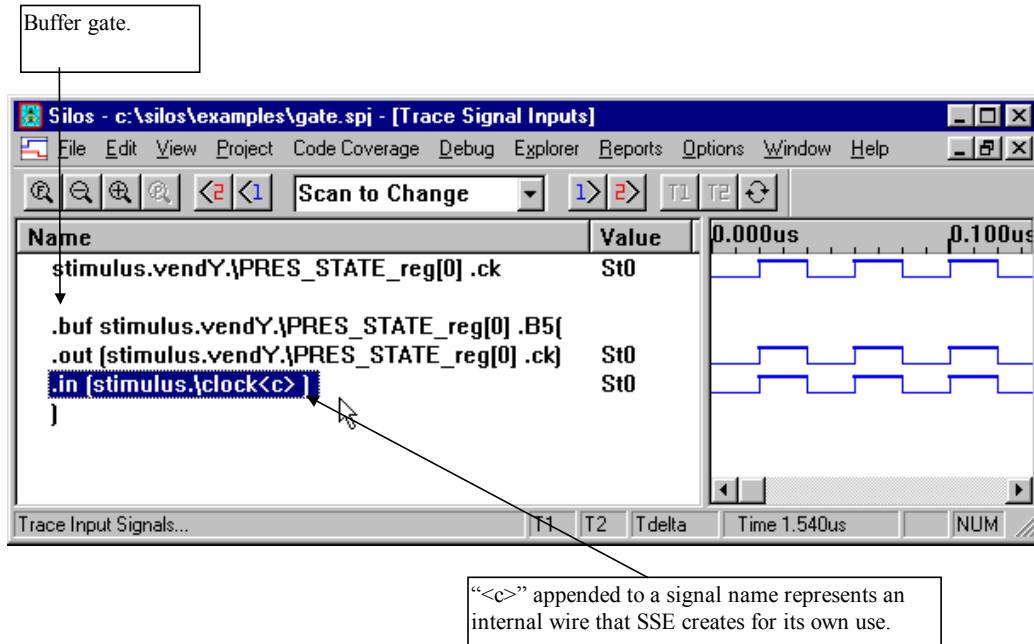
## Gate Level Debugging



- Signal "stimulus.vendY.\PRES\_STATE\_reg[0].o3" is driven by a three input nand gate. **Double click** on signal "stimulus.vendY.\PRES\_STATE\_reg[0].ck" because it goes High at time=0.020us causing the nand gate output to change.

(continued on next page)

## Gate Level Debugging



- Signal “stimulus.vendY.\PRES\_STATE\_reg[0].ck” is driven by a buffer whose input is “stimulus.\clock<c>”. The “<c>” appended to the signal name “clock” means it is a wire created for internal use by the Silos program. This internal signal is used to represent the signal “clock” across the port boundary from module “stimulus” into module “vend”.

Since we have traced back to the input stimulus signal “clock”, it appears the clock signal going High at time=0.020us propagated through the design, causing the “stimulus.newspaper” signal to go from an Unknown level to a Low level at time=0.021us. Had this been a design problem, the user would either correct the gate level design, or more likely, the RTL level design, and re-synthesize.

## Smaller Save Files

### 2.6 Creating Smaller Save Files for Logic Simulation

Silos is very efficient in writing the logic simulation results to the simulation history save file, “project\_name.sim”. The difference in simulation speed between saving everything and saving nothing is a maximum of 10%, so saving information to disk has a minimal impact on simulation speed. However, once the save file size exceeds one to two gigabytes, the user may see slower painting for the waveform display in the Silos Data Analyzer. For this reason, and if there is not enough available disk space, the user may wish to limit the size of the simulation history save file.

The save file for the simulation history is only used by the Silos GUI. The simulation history save file is not used by batch simulations, so by default it is turned off when running batch simulations (see the “Example for Batch Logic Simulation” on page 2-95).

There are two strategies the user can employ for limiting the size of the simulation history save file:

- Saving all of the simulation results for a specified time duration during simulation, such as 4,000 nanoseconds. The user would use this strategy to view the simulation results near the end of the simulation.
- Saving selected wires and selected module instances of the design’s hierarchy for the entire simulation. The user would choose this strategy to view the simulation results from time=0 to the end of the simulation.

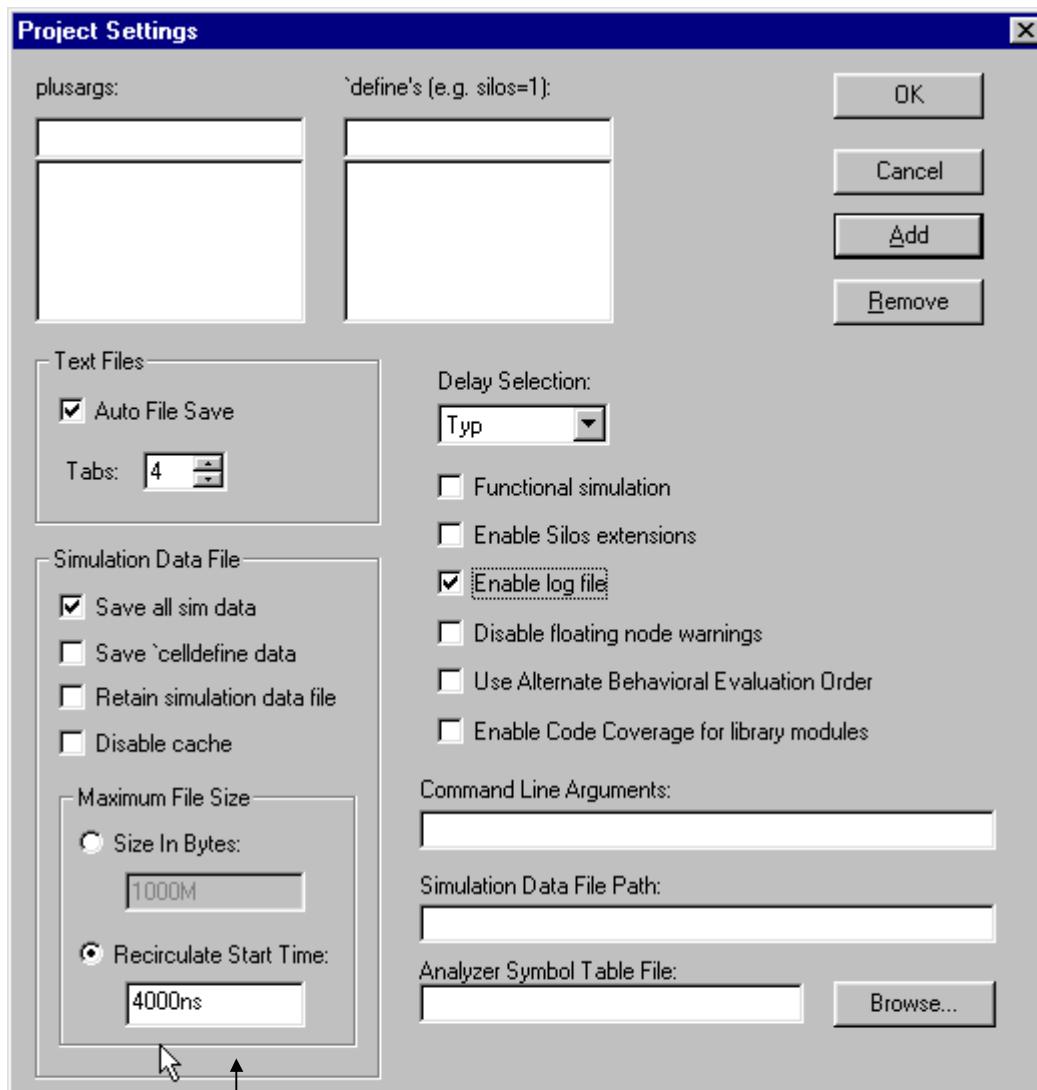
#### 2.6.1 Saving Across a Time Interval for a Constant Save File Size

A recirculating save file, similar in concept to a FIFO, can be used to keep the save file size constant while storing the simulation history for every variable across an interval of time. When the current simulation information is written to the save file, the earliest information in the save file is discarded, keeping the size of the save file constant.

To specify the time interval for the recirculating save file, select the “Project/Project Settings” menu selection to open the “Project Settings” dialog box. The “Recirculate Start Time” box at the bottom of the “Project Settings” dialog box can be used to specify the time interval.

(continued on next page)

## Recirculating Save File



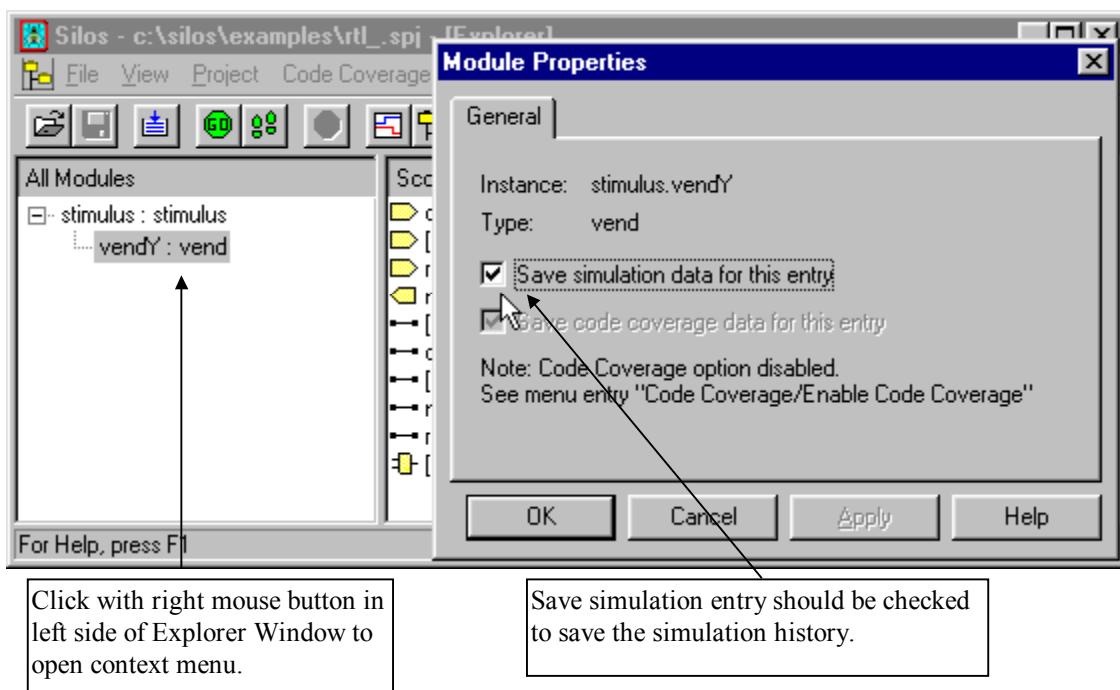
The time interval for saving the recirculating save files is set to 4000ns.

(continued on next page)

## Graphical Saving

### 2.6.2 Saving Selected Wires and Instances

Saving selected module instances and/or excluding selected module instances can be graphically specified by using the Silos Explorer. To graphically save or exclude module instances, open the context menu in the left side of the Silos Explorer by clicking with the right mouse button on the module instance name. Next select the “Properties” menu to open the “Module Properties” dialog box. When the “Save simulation data for this entry” in the “Module Properties” dialog box is enabled, Silos will save the simulation history for every variable in that module, and for all modules below it. When the “Save simulation data for this entry” in the “Module Properties” dialog box is disabled, Silos will not save the simulation history for any variable local to that module, nor for the modules instances below it. Port variables are still saved if the module instances above it are enabled.



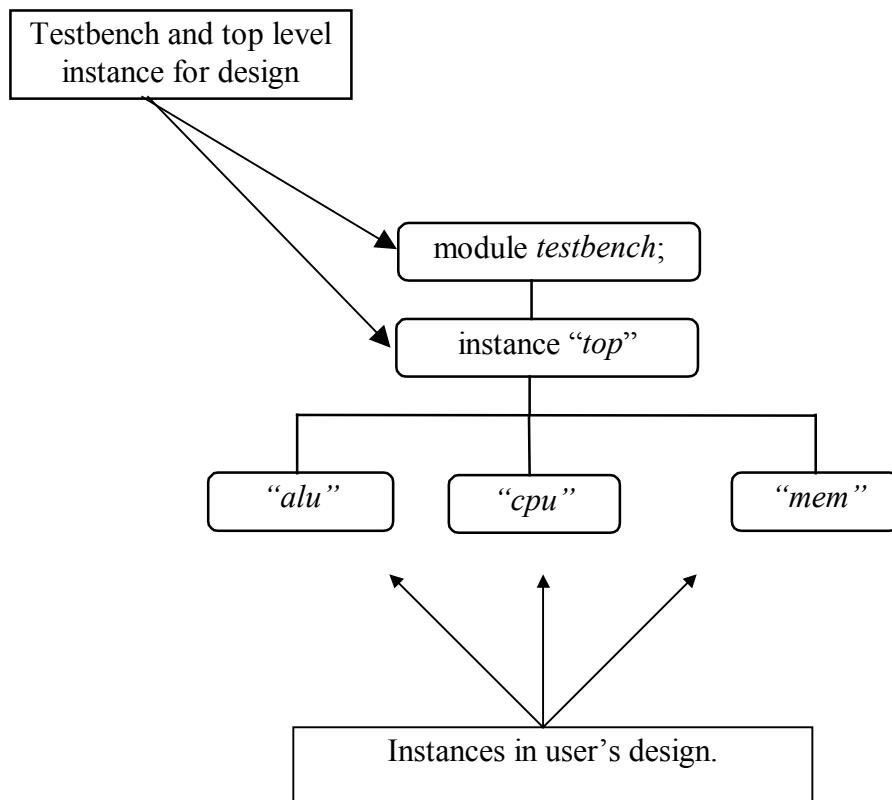
(continued on next page)

## Saving Wires

### (Saving Selected Wires and Instances)

Saving selected wires and selected module instances of the design's hierarchy can be specified also with the SILOS "keep" and "mkeep" commands (for more information, see "Keeping Simulation Node States" on page 5-27 and "Keeping Module Instance Simulation Variable Values" on page 5-29). The user can also exclude parts of the design's hierarchy by using the SILOS "mexclude" command (see "Exclude Saving Module Instance Variable Values" on page 5-28). The user usually puts the "keep", "mkeep", and "mexclude" commands at the very top of the file that contains the user's testbench, or in a separate file that is read in to Silos. An example for using these commands follows.

### Example for Reducing Save File Size: Design Hierarchy



(continued on next page)

## Saving Wires

To save only the variables for the testbench for the example, put the commands that follow at the top of the file containing the testbench (or in a separate file). Notice that the “mkeep” and “mexclude” commands use a left parentheses “(“ as the SILOS style of hierarchical delimiter.

```
'ifdef silos
!control .sav=1
!mkeep (testbench
!mexclude (testbench(top
`endif
module testbench;
...
endmodule
```

To save the variables in the testbench, instance “top”, and in instance “cpu” (and below), but not for instances “alu” and “mem”, use the commands that follow:

```
'ifdef silos
!control .sav=1
!mkeep (testbench
!mexclude (testbench(top(alu (testbench(top(mem
`endif
module testbench;
...
endmodule
```

## Batch Runs

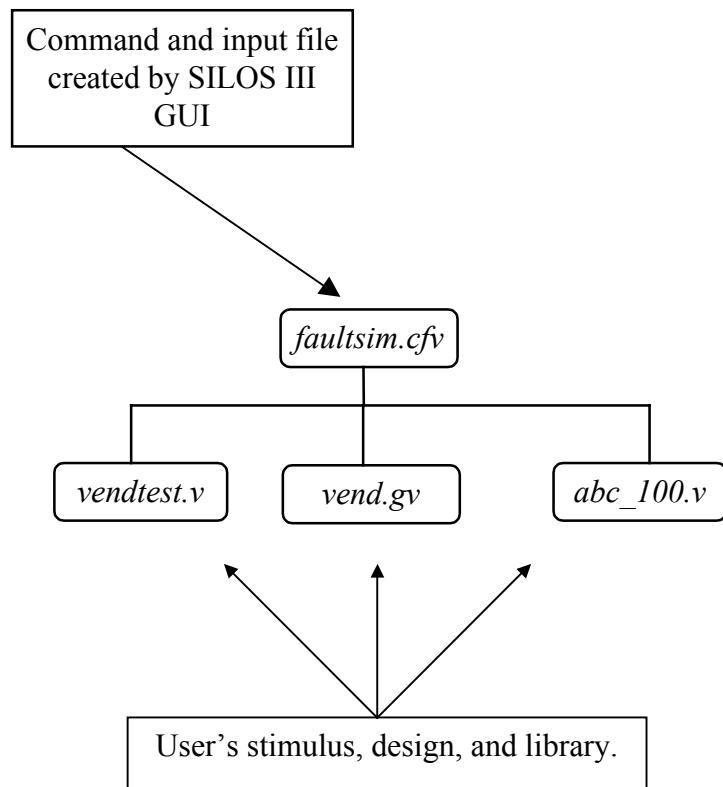
## 2.7 Example for Batch Logic Simulation

The below example sets up logic simulation for a batch run (**bold** is required syntax, plain is optional syntax, *italics* is user specified). When you simulate with the Silos GUI, it automatically creates a command file “`project_name.cfv`”, that can be used to run your design in a batch mode. The command file to run this example is automatically created when you run the example “Gate level Debugging” on page 2-81 using the Silos GUI. In the “`gate.cfv`” file, make sure you set the flags to not save the simulation history (it has little use in batch simulation), as shown in the description for “`gate.cfv`” in this example. The command line option to run this example is:

**silos -f gate.cfg** (for Windows NT)

**runsilos -f *gate.cfg*** (for Unix)

## Diagram of the Files for the Batch Logic Simulation Example



(continued on next page)

## Batch Runs

### (Example for Batch Logic Simulation)

#### Command File “gate.cfv” (generated by Silos)

```
// Simucad Generated Commands Start Here
// Simucad define
+define+sse                                // Defines "sse" as true when running the GUI
// Project Settings
-v "abc_100.v"                             // Specify the library file
+typdelays                                    // Sets delay parsing for "typical" delays
-!"file .sav=\\gate\\"
                                         // Renames the save file to the project name
-!"control .sav=0"                           // Set this to "0" so nothing is saved for batch simulation
-!"control .savcell=0"                        // Set this to "0" so nothing is saved for batch simulation
-!"control .disk=1000M"                      // Limits the save file size to 1 gig.

// Input Files
vndtest.v                                     // User file for testbench
vnd.gv                                         // Description of gate level design
```

# Errors

## 2.8 Error reporting

### Benefit

If you have syntax errors in your design, the errors will be automatically reported to the Output Window (standard output for Silos). Whenever the file name and line number are reported with the syntax error, you can double click on the error in the Output Window, and the source file window will be automatically opened (if it is not already open) with the error highlighted. This can save time for locating errors in designs that have many files.

To report syntax errors for your design, you can also select the “Reports/Errors” menu selection and the syntax errors will be reported in separate report window.

### Procedure

For the Tutorial, project “rtl\_err.spj” has a syntax error. To view the syntax error:

- **Select** the Project/Open menu to open the “Open Project” dialog box.
- **Highlight** the “rtl\_err.spj” project file (RTL level design for the Newspaper Vending Machine FSM) and **click on** the Open button in the “Open Project” dialog box.
- **Click on** the “Go” button to attempt to simulate the design.

(continued on next page)

## Errors

Silos - c:\silos\examples\rtl\_err.spi

File Edit View Project Code Coverage Debug State Machine Explorer Reports Window Help

CC SS Command: [ ]

Output

```

Reading "venderr.v"
"C:\Silos\Examples\venderr.v" (8) : error 3.229 : expecting ";"
```

Reading "vendr.v" [

sim to 0

error 2.188 : errors are too severe to simulate

Ready: sim

For Help, press F1 T1 T2 Tdelta Time 0 NUM

Double click on the error message to automatically open the source file and see the error.

- **Double click** on the syntax error that is reported in the Output Window. This will open file “vend\_err.v” and highlight the syntax error. Notice that the real error occurred on the line above due to the missing semi-colon “;” for terminating the line. This is very common because Silos could not determine there had been an error until it got to the next line, where, instead of finding the module port list, Silos encountered a register statement.

Silos - c:\silos\examples\rtl\_err.spi - [venderr.v]

File Edit View Project Code Coverage Debug State Machine Explorer Reports Options Window Help

CC SS Command: [ ]

```

2 Adapted from Example 14-8 Stimulus for Newspaper Vending Machine
3 "Verilog HDL A Guide to Digital Design and Synthesis"
4 By Samir Palnitkar, Prentice Hall, ISBN 0-13-451675-3
5 ****
6 `timescale 1ns / 1ns
7 module stimulus // missing ";" from module header
8   req#(clock) [ ]
9   req#(1:0) coin;
```

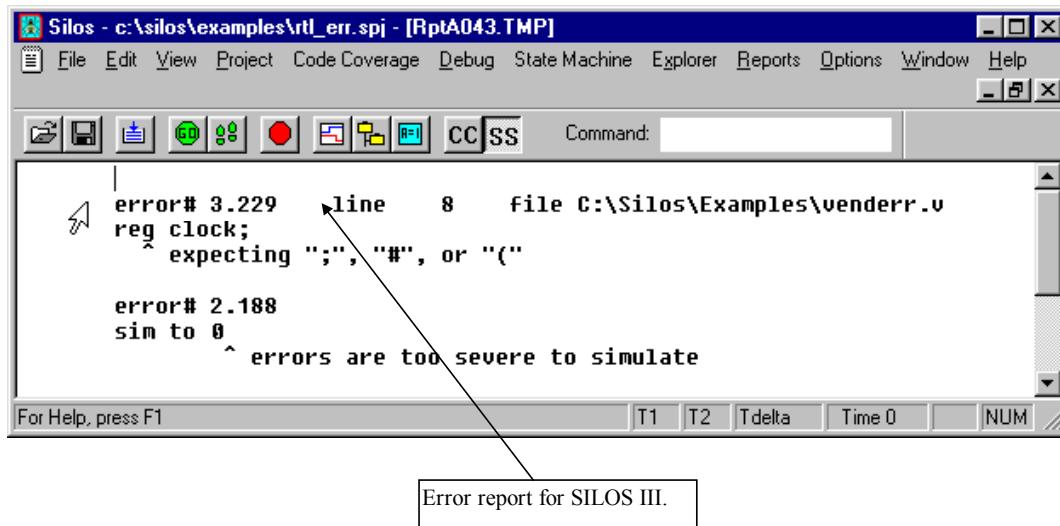
For Help, press F1 T1 T2 Tdelta Time 0 NUM

SILOS III highlights the next line because the error could not be detected until that line is interpreted.

Error is caused by semi-colon ";" missing from the module header.

(continued on next page)

# Errors



The screenshot shows the Silos software interface with the title bar "Silos - c:\silos\examples\rtl\_err.spi - [RptA043.TMP]". The menu bar includes File, Edit, View, Project, Code Coverage, Debug, State Machine, Explorer, Reports, Options, Window, and Help. The toolbar contains icons for file operations and simulation modes (CC, SS). A command line input field is present. The main window displays two error messages:

```
error# 3.229    line     8    file C:\Silos\Examples\venderr.v
reg clock;
^ expecting ";" , "#" , or "("

error# 2.188
sim to 0
^ errors are too severe to simulate
```

A callout arrow points from the text "Error report for SILOS III." to the bottom right of the main window.

To obtain a report for the error, select the “Reports/Errors” menu.

This concludes the logic simulation example. To run the other examples for the Tutorial, see:

- "Analog Behavioral Modeling (AHDL)" on page 2-100

# Analog Waveforms

## 2.9 Analog Behavioral Modeling (AHDL)

This example demonstrates analog behavioral modeling using Silos (for more information, see “Analog Extensions” on page 4-20).

Skills presented in this section are:

- Setting up a project for analog simulation.
- Using analog behavioral modeling in a gate level simulation.
- Viewing analog and digital waveforms in the Data Analyzer Window.

The file used for this example is listed below:

- `analog.v`: Shows a simple example of an A/D converter modeled with analog behavioral modeling and gate level logic.

### 2.9.1 Specifying the Analog Behavioral Modeling Project

For this example, file “`analog.v`” shows an A/D converter with comments. For additional information, see “Analog Extensions” on page 4-20. The essential ideas for analog behavioral modeling are:

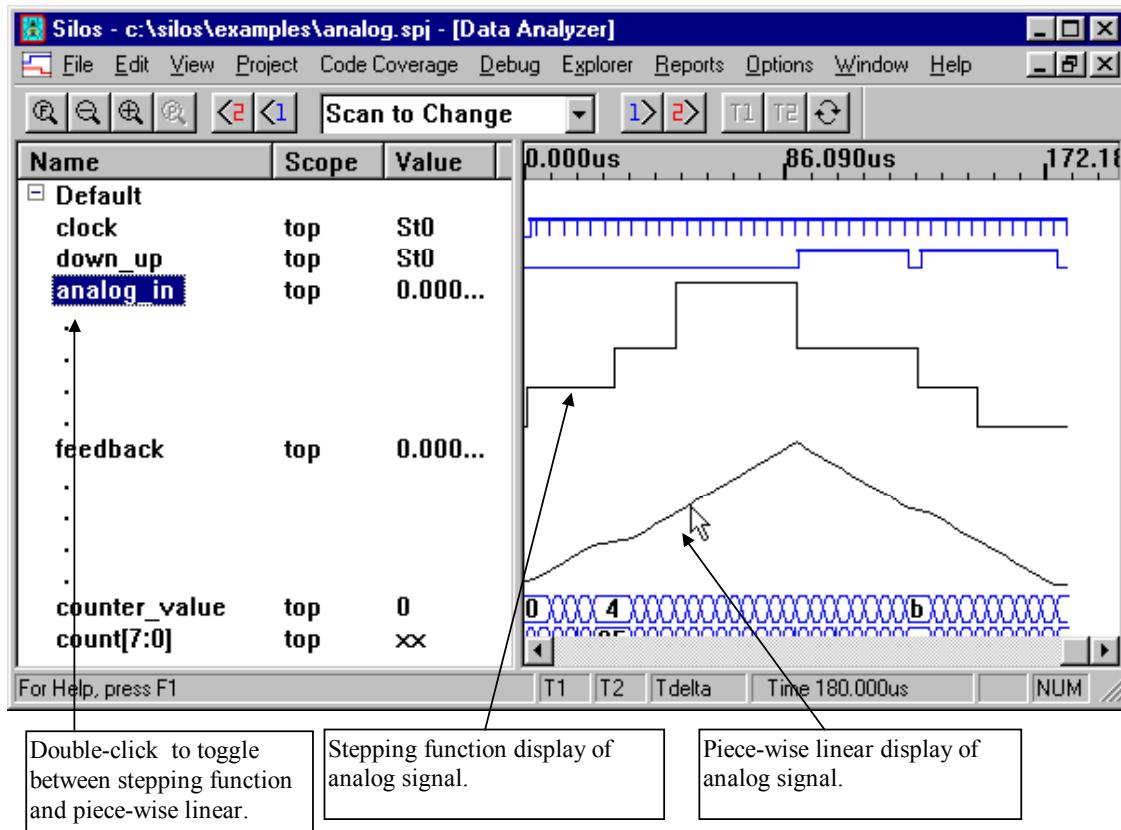
- Silos has the ability to pass real variables and integer variables in module ports. There is no need to convert reals or integers to bit vectors. This is an extension to the IEEE standard for Verilog HDL. If you need the real or integer variables to behave as wires you can use the “wire real” or “wire integer” declaration. (For more information, see “Real and Integer Data Types” on page 4-20)
- Silos supports analog extensions that allow you to put most of the standard math functions, such as “sine”, “cosine”, “log”, “power”, directly in your Verilog HDL code. This is an extension to the IEEE standard for Verilog HDL (for more information, see “Utility Transcendental Functions” on page 4-21).

The project for analog behavioral modeling is already set up. To open the project, **select** the “Project/Open” menu selection. Then change to the “examples” subdirectory of the installation directory, **select** project “`analog.spj`” and then **click-on** the “OK” button to close the dialog box.

## Analog Waveforms

### 2.9.2 Running the Analog Behavioral Modeling Simulation

**Click-on** the “Go” button on the Toolbar to load the input file and run logic simulation. The logic simulation will run until it encounters the \$finish system task in file “analog.v”.



To display the logic simulation results, **click-on** the “Open Analyzer” button on the Toolbar to open the Analyzer Window. You should see both analog and digital waveforms displayed in the Waveform Display Window.

(continued on next page)

## Analog Waveforms

You can double-click on the analog signal names “top:feedback” and “top:analog\_in” to toggle between a piece-wise linear or analog display (see “Digital and Analog Signal Display” on page 3-50). The integer “top:counter\_value” can also be displayed as an analog waveform by selecting the “Options/Analog Integer Display” menu selection. You can use the timing markers to display the analog values.

This concludes the analog behavioral modeling example. To run the other examples for the Tutorial, **see** :

- “RTL (Behavioral) Debugging” on page 2-5

**Exit**

## **2.10 Exiting Silos**

Exit Silos by selecting the “File/Exit” menu selection.

This concludes the Tutorial for Silos.

## 3. Menus

### 3.1 Menus Overview

#### 3.1.1 Pull-Down Menu Bar

Silos provides the following top-level pull-down menus:

- File menu;
- Edit menu;
- View menu;
- Project menu;
- Code Coverage menu;
- State Machine menu
- Explorer menu;
- Reports menu;
- Debug menu;
- Options menu;
- Window menu;
- Help menu.

The top-level pull-down menus for Silos change depending on which window has the focus (the window that is in focus has its title bar highlighted). For example, when the Data Analyzer Window has the focus the available top-level pull-down menus are different from the Silos Window.

Many of the menu selections can be accessed by clicking-on buttons on the Toolbar. To see a label for each button on the Toolbar, place the mouse cursor over the button for a few seconds and an explanatory text message will appear.

#### 3.1.2 Context Menus

Many of the windows in Silos have a context menus that can be accessed by putting the mouse cursor in the window and then clicking on the right mouse button (for more information, see “Context Menus” on page 3-57). For example, if you put the mouse cursor in the left-hand side of the Explorer Window and click on the right mouse button, you will see the context menu for the Explorer.

(continued on next page)

## Menus

### 3.1.3 Dialog Box Conventions

The following conventions should be noted for the dialog boxes:

- Selecting the “OK” button will close the dialog box and any active options or specifications will be used.
- Selecting the “Cancel” button will close the dialog box and not affect any options or specifications.
- Selecting the “Close” button will close the dialog box, however, options selected for the dialog box are not canceled.
- Selecting the “STOP” button on the Toolbar or the Escape key (Esc) on the keyboard will stop the current process, such as inputting a file or running logic simulation. If Silos “hangs” and does not respond to the “STOP” button, see “Nonconvergence (“Hanging”) for Behavioral Designs” on page 3-33. For the command-line version (“silos.exe” on the PC and “silos” on Unix), pressing the “Ctrl” and “c” keys on the keyboard will stop the current process.

# File Menu

## 3.2 File Menu

The “File” menu provides the following menu selections:

- New menu selection;
- Open menu selection;
- Save menu selection;
- Save As menu selection;
- Export menu selection;
- Print menu selection;
- Print Setup selection;
- Exit menu selection.

### 3.2.1 New

The “File/New” menu selection opens a new source window for editing.

### 3.2.2 Open Menu

The “File/Open” menu selection opens a source window for an existing file so that you can view or edit the file. More than one source window can be open at the same time. Use the Window menu to switch among the multiple open documents.

To simulate a project, use the Project/New menu to create a new project (see “New Menu Selection” on page 3-12 for the Project menu).

Shortcuts:

Toolbar button

Keys:      CTRL+O

### 3.2.3 Save

The “File/Save” menu selection saves the contents of the source file window that has the focus. When you choose Save, the document remains open so you can continue working on it.

To save the simulation results for logic simulation, see the “Save Project State Menu Selection” on page 3-15 in the Project menu.

## File / Print

### 3.2.4 Save As

The “File/Save As” menu selection allows you to specify a file name and then save the contents of the source file window that has the focus.

When you choose Save As, the document remains open so you can continue working on it.

To save a project to a different project name, see the “Save As Menu Selection” on page 3-15 for the Project menu.

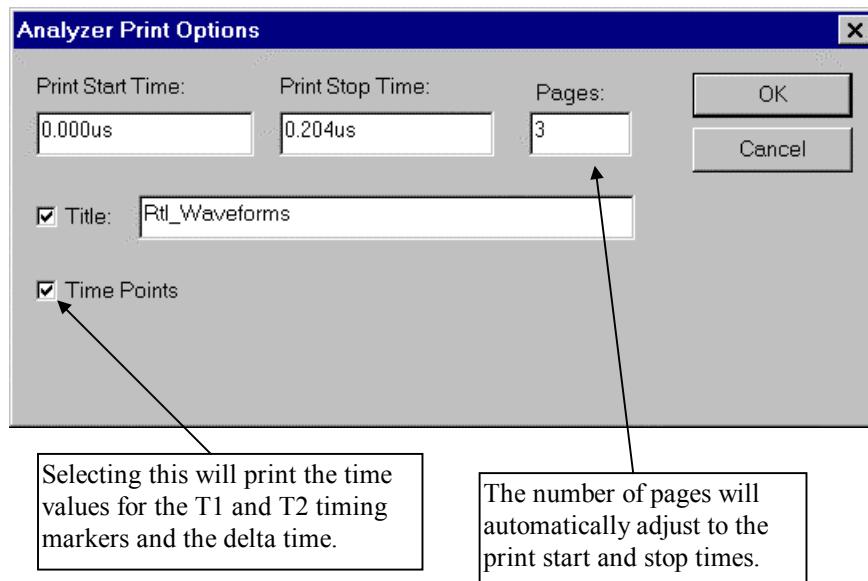
### 3.2.5 Export

The “File/Export” menu selection is used to export the results for the Code Coverage Line report and the Code Coverage Operator report to spreadsheet programs, such as Microsoft Excel. The “File/Export” menu selection opens the “Export Code Coverage Data” dialog box for writing comma delimited or tab delimited files, which can then be read into spreadsheet programs.

### 3.2.6 Print menu

The “File/Print” menu selection uses a Windows common dialog box to print the Output Window, the source windows and the report windows.

The Data Analyzer waveforms use the “Analyzer Print Options” dialog box for print.



The fonts for the Data Analyzer window can be modified by the “Options/Fonts” menu selection.

The “Analyzer Print Options” dialog box will print multiple pages of the waveform display. The number of pages to be printed is specified in the pages box. When printing multiple pages, the pages are automatically determined based on the Print Start and Print Stop times, and the number of pages specified.

## **File / Exit**

### **3.2.7 Print Preview**

The File/Print Preview menu selection displays a preview of the printout for the Output Window, the source windows, and the report windows.

### **3.2.8 Print Setup**

The “File/Print” setup menu selection uses a Windows common dialog box to setup the printer for the Output Window, the source windows, and the report windows.

### **3.2.9 Exit Menu**

The “File/Exit” menu selection exits Silos. When Silos is exited the simulation history is lost unless the “Project/Save Project State” menu is selected before exiting.

Shortcuts:

Mouse: Double-click the application's Control menu button.  
Keys: ALT+F4

# Edit Menu

## 3.3 Edit Menu

The “Edit” menu provides the following menu selections:

- Undo menu selection;
- Cut menu selection;
- Paste menu selection;
- Clear menu selection;
- Select All menu selection;
- Find menu selection;
- Find Next menu selection;
- Replace menu selection;
- Goto Line menu selection;
- Copy Image to Clipboard;
- Delete menu selection;
- Copy Image to Clipboard menu selection.

### 3.3.1 Undo menu selection

The “Edit/Undo” menu selection undoes your last editing or formatting action, including cut and paste actions. If an action cannot be undone, Undo appears dimmed on the Edit menu.

### 3.3.2 Cut menu selection

The “Edit/Cut” menu selection deletes text from a document and places it onto the Clipboard, replacing the previous Clipboard contents.

### 3.3.3 Copy menu selection

The “Edit/Copy” menu selection copies text from a document onto the Clipboard, leaving the original intact and replacing the previous Clipboard contents. When the Data Analyzer window has the focus, the “Edit/Copy” menu selection copies the full path for the selected signal name from the Data Analyzer window to the Clipboard.

## Edit Menu

### 3.3.4 Paste menu selection

The “Edit/Paste” menu selection pastes a copy of the Clipboard contents at the insertion point, or replaces selected text in a document.

### 3.3.5 Clear menu selection

The “Edit/Clear” menu selection deletes selected text from a document, but does not place the text onto the Clipboard.

Use Clear when you want to delete text from the current document but you have text on the Clipboard that you want to keep.

### 3.3.6 Select All menu selection

The “Edit>Select All” menu selection selects all the text in a document at once.

You can copy the selected text onto the Clipboard, delete it, or perform other editing actions.

### 3.3.7 Find menu selection

The “Edit/Find” menu selection searches for characters or words in a document.

You can match uppercase and lowercase letters and search forward or backward from the insertion point.

### 3.3.8 Find Next menu selection

The “Edit/Find Next” menu selection repeats the last search without opening the “Find” dialog box.

### 3.3.9 Replace menu selection

The “Edit/Replace” menu selection replaces one string with another.

### 3.3.10 Goto Line menu selection

The “Edit/Goto Line” menu selection goes to the source line number that was specified.

## Edit Menu

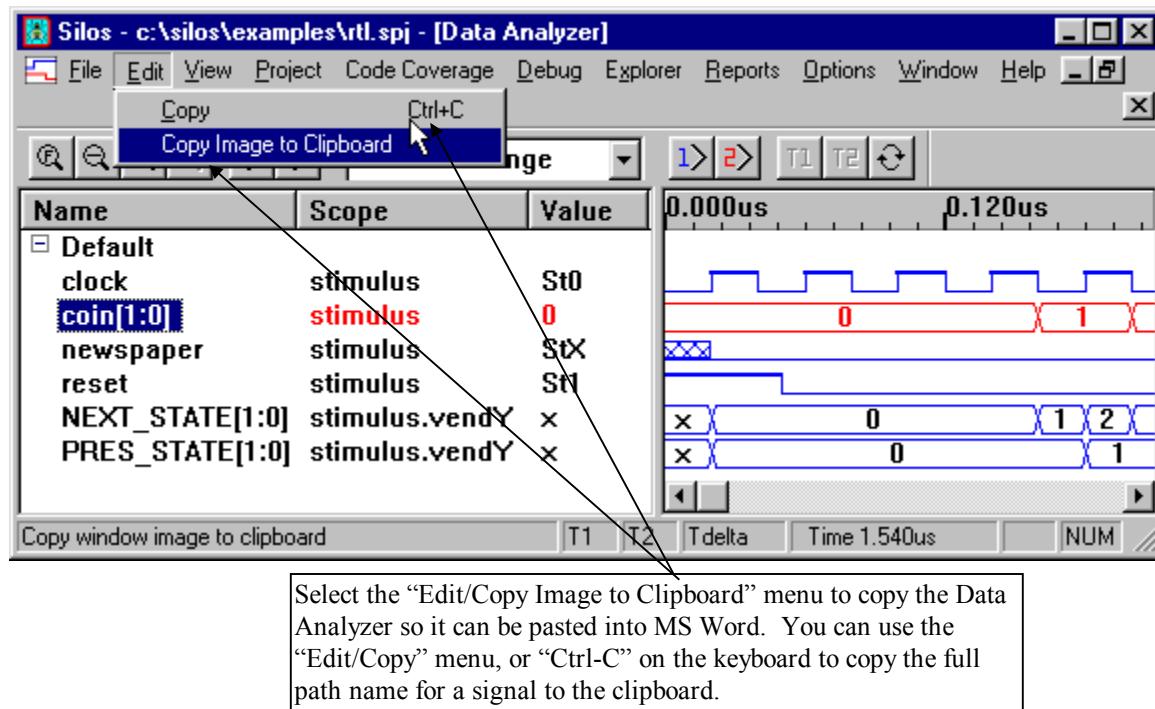
### 3.3.11 Delete menu selection

When the Finite State Machine (FSM) window has the focus, the “Edit/Delete” menu selection will delete the selected object or objects in the FSM window.

### 3.3.12 Copy Image to Clipboard menu selection

When the Finite State Machine (FSM) window has the focus, the “Edit/Copy Image to Clipboard” selection will copy the current image on the screen for the Finite State Machine (FSM) window to the Windows Clipboard.

When the Data Analyzer window has the focus, the “Edit/Copy Image to Clipboard” menu selection copies the Data Analyzer window (signal names, scope, values, and waveforms) to the Clipboard so it can be pasted into Microsoft Word.



## View Menu

### 3.4 View Menu

The “View” menu provides the following menu selections:

- Zoom menu selections;
- Toolbar menu selections;
- Status Bar menu selection.

#### 3.4.1 Zoom selections

The View menu has Zoom selections for the Data Analyzer Window. A check mark appears next to the menu item that is used. These zoom selections are also buttons on the Toolbar.

##### 3.4.1.1 Zoom-all menu selection

- Zoom-all - will display the entire simulation time range.

##### 3.4.1.2 Zoom-out menu selection

- Zoom-out - zoom out by a factor of two.

##### 3.4.1.3 Zoom-in menu selection

- Zoom-in - zoom in by a factor of two.

##### 3.4.1.4 Zoom-markers menu selection

- Zoom markers -will zoom-in-between the T1 and T2 timing markers if they are displayed.

## View Menu

### **3.4.2 Main Toolbar menu selections**

A check mark appears next to the Main Toolbar when it is displayed.

Many of the selections for pulldown menus can also be accessed by clicking-on buttons on the Main Toolbar. To obtain a text message of each button's function, place the mouse cursor over the button for a few seconds and an explanatory text message will appear.

The location of the Main Toolbar can be changed by using the mouse to grab an edge of either toolbar and dragging the toolbar to the desired location.

### **3.4.3 Analyzer Toolbar menu selections**

A check mark appears next to the Analyzer Toolbar when it is displayed.

Many of the selections for pulldown menus and context menus can also be accessed by clicking-on buttons on the Analyzer Toolbar. To obtain a text message of each button's function, place the mouse cursor over the button for a few seconds and an explanatory text message will appear.

The location of the Analyzer Toolbar can be changed by using the mouse to grab an edge of either toolbar and dragging the toolbar to the desired location.

### **3.4.4 FSM Toolbar menu selections**

A check mark appears next to the FSM Toolbar when it is displayed.

Many of the selections for pulldown menus can also be accessed by clicking-on the buttons on the FSM Toolbar. To obtain a text message of each button's function, place the mouse cursor over the button for a few seconds and an explanatory text message will appear.

The Instance box on the FSM Toolbar provides the name of each instance for the FSM. The drop down arrow to the right of the Instance box can be used to select an instance name. The "Open Instance" button can be used to display a FSM window for the instance. The state symbols in the FSM window are animated when the T1 timing marker is dragged in the Data Analyzer. The "FSM Scan" button can be used to scan to each state symbol that has been selected in an instance. As you click on the "FSM Scan" button, the T1 timing marker is moved to the time the state changed in the Data Analyzer. Editing changes that are saved for the FSM window will affect all instances of that FSM.

The location of the FSM Toolbar can be changed by using the mouse to grab an edge of the toolbar and dragging the toolbar to the desired location.

## View Menu

### 3.4.5 Status Bar

The View/Status Bar menu selection displays or hides the Status Bar at the bottom of Silos.

The left area of the Status Bar describes actions of menu items as you use the arrow keys to navigate through menus. This area similarly shows messages that describe the actions of toolbar buttons as you depress them, before releasing them. If you wish not to execute the toolbar button after viewing the description of the toolbar button, then release the mouse button while the pointer is off the toolbar button.

The right area of the Status Bar displays the time values for the T1, T2, the delta time, and the current time. The farthest right area of the status bar displays which of the following keys are latched down:

Indicator	Description
CAP	The Caps Lock key is latched down.
NUM	The Num Lock key is latched down.
SCRL	The Scroll Lock key is latched down.

# Project Menu

## 3.5 Project Menu

The “Project” menu provides the following menu selections:

- New menu selection;
- Open menu selection;
- Files menu selection;
- Save As menu selection;
- Close menu selection;
- Save Project State menu selection;
- Restore Project State menu selection;
- Load/Reload Input Files menu selection;
- Reload and Go menu selection;
- Project Settings menu selection;
- Filters menu selection;
- Project List Size.

### 3.5.1 New Menu Selection

The Project/New menu selection opens the “Create New Project” dialog box. The “Create New Project” dialog box enables you to specify the name and working directory for a new project. Projects provide a useful method for organizing your source files, as the files and libraries for the project may be scattered across many directories.

The new project name can be typed into the “File Name” box. Silos will automatically append the suffix “.spj” to the project name if you do not add a suffix to the project name. Click on “Save” to create the project name and exit the dialog box. Click-on “Cancel” to exit the dialog box without creating a project name.

After selecting the “Save” button, the “Project Files” dialog box is automatically opened. The “Project Files” dialog box enables you to specify all the source files, library files, and PLI library files associated with a project.

To create a new project that is similar to an existing project, you may want to use the “Project/Save As” menu selection (see “Save As Menu Selection” on page 3-15).

## Project Menu

### 3.5.2 Open Menu Selection

The Project/Open menu selection opens the “Open Project” dialog box to specify the name for opening an existing project. Click on the “Open” button to open the project. Click on the “Cancel” to exit the menu. Before opening a project Silos is automatically reset so that the results from any previous project are lost.

### 3.5.3 Files Menu Selection

The Project/Files menu selection opens the “Project Files” dialog box for specifying the input files and library files for a project. To select a project for the “Project Files” dialog box use the Project/New or Project/Open menu selections.

The “File Group” list box allows you to select Verilog HDL “Source Files”, “Library Files”, and “PLI Library Files”. To add source files, have the Source Files option selected in the File Group box. Next double-click on a file name in the list box, or highlight a file name in the list box and click on the Add button to add the source file to the “Files in Group” list box. Files can be deleted from the project by highlighting the file name in the “Files in Group” box and selecting the “Remove” button. The “Move Up” and “Move Down” buttons allow you to rearrange the file names in the “Files in Group” list box. Click on “Ok” to update the project and close the dialog box and “Cancel” to exit the dialog box without affecting the project.

To specify library files, click-on the drop-down arrow in File Group list box in the Project Files dialog box, and select “Library Files”. Double-click on library file names in the list box to add them to the “Files in Group” list box. You can also select more than one library file by clicking on a file name to highlight it, and then hold down the “Shift” key or the “Ctrl” key on the keyboard while clicking on additional file names. To specify a directory of library files whose file name extensions end in “.v”, enter the directory name followed by “{.v}” in the “File Name” box, and click-on the Add button. An example for specifying all the files ending with “.v” for directory “library” would be “library{.v}”. For more information on specifying library files, see the “-y” and “-v” “Command-line Options” on page 7-4 or Silos “Library Command” on page 6-2.

To specify PLI library files, click-on the drop-down arrow in File Group list box in the Project Files dialog box, and select “PLI Library Files”. Double-click on PLI library file names in the list box to add them to the “Files in Group” list box as PLI library file names.

To actually input the files for a project, use the “Project /Load/Reload Input Files” menu selection or the “Load/Reload Input Files” button on the Toolbar.

(continued on next page)

## Project Menu

### (Files)

Silos projects also allow relative paths to files so that the path names in a project will work at different sites. To specify this, you need to specify a `define compiler directive in the Project/Project Settings dialog box, such as "foo=c:\sse\examples". Then edit the project file (.spj file) and put the `define compiler directive before the path names, such as:

```
[Files]
0=`foo\vendtest.v
1=`foo\vend.v
```

Silos now preprocesses the information in the project file (.spj file), and substitutes the string "c:\sse\examples" for `foo. This feature will work for path names that are set in the Project Files dialog box, such as the Source Files, Library Files, Include Directories, and PLI Library Files. This also works with path names that are set from the Project Settings dialog box, such as Command Line Arguments, Simulation Data File Path, and the plusargs settings.

## Project Menu

### 3.5.4 Save As Menu Selection

The Project/Save As menu selection saves the project to the project name that you specify. It does not save the simulation history for the Data Analyzer. The Save As feature can be used to easily clone projects for testing purposes without having to modify the original project.

### 3.5.5 Close Menu Selection

The Project/Close menu selection closes a project. All “child” processes such as the Output Window and the Data Analyzer Window are also closed.

### 3.5.6 Save Project State Menu Selection

The “Project/Save Project State” menu selection saves the simulation results and the current state of the simulator to disk. This feature is very useful for preventing simulation data loss so that you do not have to re-simulate your design if you exit Silos or if Silos crashes. The Project/Save Project State menu can be selected at any time after simulation has halted.

When Silos is restarted, the Project/Restore Project State menu selection can be used to reload the simulation up to the last time point that was saved. The Data Analyzer can then be used to view the simulation results and the simulation can be continued. The Project/Restore Project State menu must be selected immediately after Silos is reopened or a project is opened. The Project/Restore Project State menu can not be selected after selecting the Project/Load/Reload Input Files menu or the “Go” button.

The simulation history is stored on disk in the file named “project\_name.sim”, and the simulation state is stored on disk in the file named “project\_name.cmm”. The “project\_name.cmm” file will be slightly larger than the size of your Silos simulation in RAM memory.

### 3.5.7 Restore Project State Menu Selection

When Silos is restarted, the Project/Restore Project State menu selection can be used to reload the simulation up to the last time point that was saved by the “Project/Save Project State” menu selection. The Data Analyzer can then be used to view the simulation results and the simulation can be continued. The Project/Restore Project State menu must be selected immediately after Silos is reopened or a project is opened. The Project/Restore Project State menu can not be selected after selecting the Project/Load/Reload Input Files menu or the “Go” button.

The simulation history is stored on disk in the file named “project\_name.sim”, and the simulation state is stored on disk in the file named “project\_name.cmm”. The “project\_name.cmm” file will be slightly larger than the size of your Silos simulation in RAM memory.

(continued on next page)

## Project Menu

### 3.5.8 Load/Reload Input Files Menu Selection

The Project/Load/Reload Input Files menu selection automatically resets Silos and inputs the files specified for the project that is open.

Logic simulation is then run to time=0 and you can begin debugging your project by setting breakpoints, single stepping, etc. The Data Analyzer can be opened to display the results during simulation. Choosing the “Go” button on the Toolbar will run logic simulation until a “\$stop” or “\$finish” is encountered in the design, or until you choose the “STOP” button on the Toolbar.

### 3.5.9 Reload and Go Menu Selection

The “Project/Reload and Go” menu selection automatically resets Hyperfault, inputs the files specified for the project, and then runs logic simulation until a \$stop or a \$finish system task is encountered, or until you choose the “STOP” button on the Toolbar.

## Project Menu

### 3.5.10 Project Settings Menu Selection

The Project/Project Settings menu selection allows you to set the command-line options and other options for Silos. For these settings to take effect, you must click on the Load/Reload Input Files button, or select the Project/Load/Reload Input Files menu selection.

Available options for the Project Settings dialog box are:

- Analyzer Symbol Table File: This specifies a file that substitutes text strings for state values for vectors displayed in the Data Analyzer. The text strings can be symbols for a state machine, making the state machine much easier to debug. For an example, see “Displaying Vectors using Symbolic Names” on page 2-22.
- Auto File Save: Enabling this feature will cause Silos to automatically save any source file you have modified whenever you click on the Load/Reload Input Files button on the Main toolbar, or select the Project/Load/Reload Input Files menu selection.
- Command Line Arguments: You can use Verilog style command line arguments for Silos. The command line arguments can be entered from the Command Line Arguments box in the Project Settings dialog box, or, from the command line if you are running a batch simulation. For a list of the available command line arguments, see “Command-line Options” on page 7-4. The command line arguments can make use of relative names (see “Files Menu Selection” on page 3-13).
- `define's: The user can enter `define statements that are project specific. These `define compiler directives will be used in addition to any `define compiler directives in the design. When entering the `define, use just the <text\_macro\_name> <MACRO\_TEXT> part of the `define syntax. For example, for the Verilog compiler directive:

```
'define wordsize 8
```

you would enter in the Project Settings dialog box:

```
wordsize 8
```

Silos allows relative paths to files so that the path names in a project will work at different sites. To specify this, you need to specify a `define compiler directive in the Project/Project Settings dialog box, such as "foo=c:\sse\examples". Then edit the project file (.spj file) and put the `define compiler directive before the path names, such as:

```
[Files]
0='foo\vendtest.v
```

Silos preprocesses the information in the project file (.spj file), and substitutes the string "c:\sse\examples" for `foo. This feature works for path names set in the Project Files dialog box, such as the Source Files, Library Files, Include Directories, and PLI Library Files. This also works with path names set from the Project Settings dialog box, such as Command Line Arguments, Simulation Data File Path, and the plusargs settings.

## Project Menu

### (Project Settings)

- Delay Selection: You can select the “min/typ/max” delay setting for all delays in the project.
- Disable cache: This feature disables the caching to RAM memory for the part of the simulation history save file that is used to draw the waveforms in the Data Analyzer. For example, if you do a Zoom Full in the Data Analyzer, the simulation information for the signals currently displayed is saved in RAM memory in a format that is faster to access. If you then zoom in, the information is taken from RAM memory instead of disk, and the Data Analyzer is updated much faster. However, if your computer does not have enough RAM memory, you may choose to disable this feature.
- Disable floating node warnings: This feature disables the warning message that informs the user that a wire is not driven by anything, i.e. a floating node.
- Enable Code Coverage for library modules: This feature causes Code Coverage to be enabled for module definitions in libraries.
- Enable log file: This feature causes standard output to be written to a log file when the Silos program is exited. The default log file name is the project name with the “.log” file extension. When this option is selected Silos will write to both standard output and the log file.
- Enable Silos extensions: This feature enables extensions to the Verilog HDL language, such as assigning to wires in procedural code and global variables (see “Silos Extensions to Verilog HDL” on page 4-24).
- Functional simulation: This feature reduces the memory used and increases simulation speed by eliminating the specify blocks from logic simulation. To prevent nonconvergence problems, the unit delay mode is used for all modules. When this command is used, the back annotation of SDF delays is disabled.
- Maximum File Size: You can select one of two options to control the maximum simulation save file size on disk, the “Size In Bytes” or the “Recirculate Start Time”.
- The “Size In Bytes” option sets the maximum size on disk for the simulation history save file. This prevents Silos from crashing when all available disk space is used. Silos will return to the “Ready:” prompt when the simulation history save file reaches the limit. The “reset savfile” command can be entered in the Command window in the Main Toolbar to reset the simulation history save file to a few bytes. The simulation can be continued and the simulation results that are after the “reset savfile” can be reviewed.
- The “Recirculate Start Time” option specifies the maximum time interval for saving the simulation history results. When the simulation time exceeds this interval, the new simulation results are saved and the earliest simulation results are discarded so that the save file interval and file size on disk stay relatively constant, very much like a FIFO. The interval can be specified in time units, such as “100ns” for 100 nanoseconds.

(continued on next page)

# Project Menu

## (Project Settings Menu Selection)

- plusargs: You can enter “+” command-line arguments that are project specific, such “+compare”, “+sdf”, .etc. For example, suppose you wanted to specify the SDF file only if you entered “+sdf” in the “plusargs” box for the Project Settings dialog box. Then your test bench may look like:

```
module test_bench;
initial
    if ( $test$plusargs( "sdf" )
        $sdf_annotate("test.sdf"); // only execute if "+sdf" is an argument
    endmodule
```

The plusargs arguments can make use of relative names (see “Files Menu Selection” on page 3-13).

- Retain simulation data file: This prevents the simulation history save file from being deleted from disk when Silos is exited. The default is to delete the simulation history file when Silos is exited. Checking this option is not recommended, as it has no use and may clutter up your disk with large save files.
- Save all sim data: This feature will save the simulation history for every variable. If “Save all sim data” is not enabled, then the only variables saved are those specified on each instance in the hierarchy by the Properties menu selection in the Explorer context menu.
- Simulation Data File Path: This specifies the directory where the simulation history file is stored. This enables you to use disk drives with more space or that are more convenient. The simulation data file path can make use of relative path names so that projects are independent of the file systems they are run on (see “Files Menu Selection” on page 3-13).
- Save `celldefine data: This feature determines if variables in `celldefine - `endcelldefine boundaries are saved. This feature is useful for excluding variables that are inside of library cells from the save file, thus reducing the size of the save file on disk. When using the Data Analyzer, if you see “No Saved Data” instead of a waveform, this may mean that the signal is inside of a `celldefine boundary. To correct this, enable the "Save `celldefine data" option and re-simulate.
- Tabs: This feature will set the spacing for tabs in the source file windows. This can be useful for customizing the tab spacing.
- Use Alternate Behavioral Evaluation Order: This instructs the simulator to evaluate selected behavioral code in a similar order of execution as used by other Verilog HDL simulators.

## Project Menu

### 3.5.11 Filters

The Project/Filters menu selection opens the “Recent Project List” dialog box for specifying file name filters for the “Project Files” dialog box (Project/Files menu selection). If you specify a file filter, then the default filters are hidden for the “Project Files” dialog box.

The name filtering for the “Project Files” dialog box uses the standard Windows style wildcard characters for file name expansion:

- The asterisk character “ \* ” is used to match any pattern, including null.
- The question mark character “ ? ” is used to match any single character.

### 3.5.12 Project List Size

- The Project/Project List Size menu selection opens the “File Filters” dialog box for specifying the number of recently used projects that are listed at the end of the Projects menu.

## Code Coverage Menu

### 3.6 Code Coverage Menu

The “Code Coverage” menu provides the following menu selections:

- Enable Code Coverage menu selection;
- Enable Operator Coverage menu selection;
- Select Files to Merge menu selection;
- Line Report menu selection;
- Operator Report menu selection;

The Code Coverage Plug-in feature reports behavioral lines that did not execute, and operators not fully exercised. The Code Coverage plug-in provides a fundamental functional analysis of the design, it is limited to basic Verilog language constructs such as behavioral operators, behavioral blocks, and executable lines of code in the design. The Code Coverage plug-in examines the testbench's ability to fully exercise these language constructs. For example, in the Verilog expression "(a | b)" which or's single bit's "a" and "b", the Code Coverage plug-in examines if the "|" operator is fully exercised during testbench application. Each operand individually at some time must produce a "1" result, and both operands at some time must be "0" to produce a "0" result.

#### Code Coverage vs. Fault Simulation:

Use the Code Coverage plug-in to improve your testbench and report if the language components of your design are fully exercised by the testbench. For example, the Code Coverage plug-in reports on whether a bit-or "|" operator and its associated operands are fully exercised by the testbench. Because the Verilog language supports hierachal constructs, many instances of each operator can be used. However, the Code Coverage plug-in reports only if “all uses combined” have fully exercised the operator. The Code Coverage plug-in does not report on each specific instance of an operator. Fault Simulation should be used to tell you if each operator is fully exercised, and if your testbench would detect chip failures due to specific chip defects.

#### 3.6.1 Enable Code Coverage Menu Selection

The “Code Coverage/Enable Code Coverage” menu selection enables saving of line coverage information to an ASCII file “project\_name.codecov”, where “project\_name” is the name of your current project. The “project\_name.codecov” file is then used by the Line report after simulation. The “CC” button on the Main toolbar is a shortcut to the “Code Coverage/Enable Code Coverage” menu selection. The Code Coverage plug-in must be selected before logic simulation begins, such as when you first open a project. If the “Code Coverage/Enable Code Coverage” menu selection or the “CC” button is selected at simulation time=0 or later, a message box will notify you that the project files will be automatically reloaded before code coverage is set.

## Code Coverage Menu

### 3.6.2 Enable Operator Coverage Menu Selection

The “Code Coverage/Enable Operator Coverage” menu selection enables saving of operator coverage information to an ASCII file “*project\_name.codecov*”, where “*project\_name*” is the name of your current project. The “*project\_name.codecov*” file is then used by the Operator report after simulation. The “Code Coverage/Enable Operator Coverage” menu selection is a separate option from Line coverage because saving operator information may slow down the logic simulation more than Line coverage. Operator coverage must be selected before logic simulation begins, such as when you first open a project. If the “Code Coverage/Enable Operator Coverage” menu selection is selected at simulation time=0 or later, a message box will notify you that the project files will be automatically reloaded before code coverage is set.

### 3.6.3 Select Files to Merge Menu Selection

The “Code Coverage>Select Files to Merge” menu selection can be used to merge the code coverage results from simulations using the same design. This option merges the ASCII files “*project\_name.codecov*”, where “*project\_name*” is the name of each project. If you do not change the project name for each simulation, then you need to rename the “*project\_name.codecov*” file before simulating each testbench. The “Code Coverage>Select Files to Merge” menu selection opens the “Code Coverage File Select” dialog box. This dialog box is additive, so you can select more than one code coverage results file before closing the dialog box, and you can use this dialog box more than once to add additional code coverage results files. For an example on using this feature, see “Merging Code Coverage Reports” on page 2-64.

## Code Coverage Menu

### 3.6.4 Line Report Menu Selection

The “Code Coverage/Line Report” menu selection opens the “Code Coverage Line Report” list box. This list box reports the execution of individual lines. For Example:

Module/Task/Function	Hits	Line Number	File Name
Alu	10	24	foo.v

Clicking on a heading will sort the displayed lines as follows:

Module/Task/Function	Alphabetically by name
Hits	Modules with the greatest number of zero hits.
Line Number	Numerically by line number then by file name.
File Name	Alphabetically by name then by hit.

If you double click on one of the entries in the list box, Silos will automatically open the source file, and highlight the line of behavioral code. Lines that are not executed have a purple dot to the left of the line. Lines that are executed have a green “E” to the left of the line. Holding the mouse cursor over a green “E” will show the number of times that the line was executed.

### Exporting Code Coverage Analysis

The code coverage Line report can be exported to a spread sheet program such as Microsoft Excel by using the “File/Export” menu selection to open the “Export Code Coverage Data” dialog box, which has options for comma or tab separated data.

## Code Coverage Menu

### 3.6.5 Exporting Code Coverage Analysis

The code coverage Line report and Operator report can be exported to a spread sheet program such as Microsoft Excel by using the “File/Export” menu selection to open the “Export Code Coverage Data” dialog box, which has options for comma or tab separated data. For specifying the “ccexport” command, see “Exporting Code Coverage Results” on page 5-12.

### 3.6.6 Restricting Code Coverage Analysis

There are several options which allow you to choose what code coverage analyzes. The default is to analyze all code except for code from libraries.

- 1) Identify the Device Under Test (DUT) to prevent Code Coverage plug-in from including your testbench in the analysis. You can add the following lines to your testbench to identify the Device Under Test:

```
'ifdef silos  
    initial $fs_dut("testbench.chip");  
'endif
```

- 2) To exclude specific lines from code coverage, enclose the lines that you do not want code coverage analyzed for with the directives (for an example, see “Merging Code Coverage Reports” on page 2-64):

```
'disable_codecoverage  
'enable_codecoverage
```

- 3) To enable code coverage of libraries, use the command line argument  
"+libcodecoverage".

The directives "'disable\_codecoverage'" and "'enable\_codecoverage'" in a library file have precedence over "+libcodecoverage".

(continued on next page)

## Code Coverage Menu

### (Restricting Code Coverage Analysis)

- 4) To exclude a module instance and every module instance below from contributing to coverage, use the command (for more information, see “Exclude Code Coverage for Module Instances” on page 5-13):

```
"!ccmexclude <verilog instance name> ..."
```

To include a module instance below one that is excluded, use the command (for more information, see “Keeping Code Coverage for Module Instances” on page 5-14):

```
"!ccmkeep <verilog instance name> ..."
```

Note the instance names on these commands must proceed from the top of the design downward.

To do the equivalent from the GUI explorer, select a module instance, right-mouse click and go to the properties tab dialog. Check or uncheck "Save code coverage data for this entry". Note this information is written into the ".cfv" file to permit batch use at a later time.

### 3.6.7 Operator Report Menu Selection

The “Code Coverage/Operator Report” menu selection reports exercising of each operator. Only operators which have been executed are reported. Use the line report to locate lines containing operators that have not been executed.

Module/Task/Function Name	Line Number	File Name	Fails	Operator	Failure Cause
alu	10	cpu33.v	1		never true(1)

Clicking on the heading will sort the contents displayed as follows:

Module/Task/Function	Alphabetically by name
Line Number	Numerically by line number then by file name
File Name	Alphabetically then by decreasing failure
Fails	Decreasing failure count
Operator	Operator then by decreasing failure count
Failure Cause	Alphabetically by cause

If you double click on one of the entries in the list box, Silos automatically opens the source file and highlights the corresponding line. Lines where an operator fails have a purple dot to the left of the line. Lines that have no operators that fail have a green “E” to the left of the line. Holding the mouse cursor over a purple dot shows one operator that failed on the line.

## Code Coverage Menu

### Operator Coverage Options:

When you select the “Code Coverage/Operator Report” menu selection, the “Select Operator Report Option/s” dialog box is opened. This dialog box contains six options that determine how the Operator report is displayed. You can select one or more of these options to obtain a combined report for the operators.

“DEFAULT Operator 1/0 Result” reports the result of the operator. For example, if the results for logical operators (“<” and “>”) were both true and false during the simulation, then the operator had no failures and a green “E” is placed to the left of the line with the operator. If the results of the operator were only true or only false during the simulation, then the operator had a failure and a purple dot is placed to the left of the line with the operator.

“Logic Operand Independence” reports when either operand fails to independently (of the other operand) cause the operator to be true or false.

“Logic Operand Bit Independence” reports when the operands are vectors, report when any bit in either operand fails to independently (of the other operand) cause the operator to be true or false.

“Bit Operand Independence” reports when either operand fails to independently (of the same bit in the other operand) toggle the resulting bit.

“Math Result Bits” reports any bit in the result never 1 or never 0.

“Reduction Operand Independence” reports when operand bits fail to independently toggle the result.

### Exporting Code Coverage Analysis

The code coverage Line report can be exported to a spread sheet program such as Microsoft Excel by using the “File/Export” menu selection to open the “Export Code Coverage Data” dialog box, which has options for comma or tab separated data.

## State Machine Menu

### 3.7 State Machine Menu

The “State Machine” menu provides the following menu selections:

- New menu selection;
- Open menu selection;
- Save menu selection;
- Save As menu selection;
- Select Mode menu selection;
- State Mode menu selection;
- Expression Mode menu selection;
- Note Mode menu selection;
- Transition menu selection.

Several items that are on the FSM Toolbar but not in the State Machine menu are:

- The Page button can be used to add one or more pages to the Finite State Machine (FSM) window. The pages can then be used to print different pages when you print the diagram for the state machine.
- The Delete button can be used to delete any drawing symbol that has been selected by the user.
- The Zoom In button can be used to zoom in on the FSM window. To zoom in, click on the Zoom In button, and then click on the FSM window where you want to center the zoom in.
- The Zoom Out button can be used to zoom out for the FSM window.

The Edit menu has a Copy Image to Clipboard selection that will copy the current image on the screen to the Windows Clipboard.

## State Machine Menu

### 3.7.1 New Menu Selection

The State Machine/New menu selection opens a blank master FSM (Finite State Machine) window. In the FSM window you can draw the diagram for a new state machine. When you save the state machine diagram, the Verilog HDL source code for the state machine is generated. You can then include the source code into your design using a `include compiler directive.

When you open an instance for the state machine diagram, it can be animated to reflect simulation changes from state to state, such as during single stepping. After simulation, the state machine diagram is animated as you move the T1 cursor in the Data Analyzer. When the state diagram is animated, the color is changed for the state that is active. If the color for a state is red, then the state is at an Unknown level.

### 3.7.2 Open Menu Selection

The State Machine/Open menu selection opens an existing FSM window.

### 3.7.3 Save Menu Selection

The “State Machine/Save” menu selection saves the FSM window that has the focus. When you choose Save, the FSM window remains open so you can continue working on it. Each time that you save the FSM window, the Verilog HDL source description for the FSM diagram is written to a “.v” file that has the state machine’s name.

To save the simulation results for logic simulation, see the “Save Project State Menu Selection” on page 3-15 in the Project menu.

### 3.7.4 Save As Menu Selection

The “State Machine/Save As” menu selection allows you to specify a state machine name and then save the FSM window that has the focus. Each time that you save the FSM window, the Verilog HDL source description to the FSM diagram is written to a “.v” file that has the state machine’s name.

When you choose Save As, the FSM window remains open so you can continue working on it.

### 3.7.5 Select Mode Menu Selection

The State Machine>Select Mode menu selection chooses the “select” drawing capability for the FSM window. The Select Mode lets you use the left mouse button to select a drawing symbol in the FSM window, and to select the box at the top of page one to enter the “clock” name for the state machine. To individually select more than one object, hold down the “Ctrl” key on the keyboard as you select objects with the left mouse button. To select portions of the window, you can hold down the left mouse button and drag the mouse cursor across the portion of the window that you want to select.

## State Machine Menu

### 3.7.6 State Mode Menu Selection

The State Machine/State Mode menu selection chooses the “State” drawing tool for the FSM window. The State Mode places an oval state symbol when you click with the left mouse button in the FSM window.

### 3.7.7 Expression Mode Menu Selection

The State Machine/Expression Mode menu selection activates the “Expression” drawing tool for the FSM window. The Expression tool creates a text box for entering a Verilog HDL expression, such as “A && B”. If an expression box is dragged over the top of a transition line it is automatically attached to the transition.

### 3.7.8 Note Mode Menu Selection

The State Machine/Note Mode menu selection lets you enter notes in the background portions for the FSM window. The notes are written as comments to the Verilog HDL source description when the state machine is saved.

### 3.7.9 Transition Mode Menu Selection

The State Machine/Transition Mode menu selection chooses the “Transition” drawing tool for the FSM window. A transition (an arrow) between two state symbols can be drawn by holding the mouse cursor over a state symbol, and then click and release with the left mouse button to select it. When a state symbol is selected, passing the mouse cursor over the state symbol causes a small circle to appear on the state symbol to show where the transition line will begin. To start drawing the transition, click and hold down the left mouse button at the edge of the selected state symbol. With the left mouse button held down, a transition will be drawn as you drag the mouse cursor to the next state symbol, where you can release the left mouse button. An arrow will appear at the end of the transition next to the state that it is connected to. When the transition is drawn, a text box for entering a Verilog HDL expression is automatically attached to the transition. An example expression would be “A && B”. Reset lines can be drawn by starting a transition away from a state, and drawing it to the state. Set lines can be drawn by starting a transition at a state, but not connecting the transition to another state.

If you want to redraw a transition, select the transition line by clicking and releasing with the left mouse button on the transition line. A small circle appears on the transition line whenever you hold the mouse cursor over it. To begin redrawing the transition, place the mouse cursor anywhere on the transition line and hold down the left mouse button and the small circle will appear. Next move the mouse cursor tangential to the transition line, and then continue in the direction that you want to draw the transition. Release the left mouse button when you have completed drawing the transition.

# Reports Menu

## 3.8 Reports Menu

The “Reports” menu provides the following menu selections:

- Activity menu selection;
- Errors menu selection;
- Iteration menu selection;
- Nonconvergence menu selections;
- Sizes menu selection.

### 3.8.1 Activity Menu Selection

The Reports/Activity menu selection can be used to pre-grade the test vectors for fault simulation by reporting nodes that have no activity (level transitions) during a logic-simulation. The logic simulation is much faster to run than fault simulation. An ACTIVITY report can be very useful for developing input test patterns to detect circuit faults for fault simulation. The number of level transitions at a node indicates an input test pattern's effectiveness. Faults at nodes which make no level transitions can not be detected.

The Activity menu selection can be used to generate the following report:

- An activity **table** that lists the node names of nodes that do not transition.
- An **HDL Code Coverage table** that lists the lines of each file that were not executed. This can be used to find problems with the HDL code, or stimulus that is not exercising all of the HDL code.
- An activity **summary** that lists totals for the number of nodes at each level of activity count.
- An activity **histogram** that shows known and potential level transitions versus time.

## Reports Menu

### **Activity Menu Selection (continued)**

For the activity table for the Activity report, you will see the following legend:

Legend for "TRANSITION COUNT" column:

value	- number of definite transitions
(value)	- number of possible transitions
"H"	- no definite transitions, node High at Min time specified
"L"	- no definite transitions, node Low at Min time specified

The legend is stating that a “value” is reported to the left of a node name. A “definite” transition is defined as a change from a Low to High level or High to Low level, even if it goes through an intermediate Unknown level. A “possible” transition is defined as a change from a High or Low level to the Unknown level or from the Unknown level back to a High or Low level. A “H” reported to the left of the node name, means the node never had a definite transition, and the node was High at the time specified as the minimum time for the time range for the report. The default minimum time is time=0. A “L” reported to the left of the node name, means the node never had a definite transition, and the node was Low at the time specified as the minimum time for the time range for the report.

For the Summary, the percentages are based on the nodes listed that are within the range specified for the Activity report. The default range is zero transitions for the maximum and minimum number of transitions. The default is zero because the purpose of the Activity report is to report nodes that did not toggle for fault simulation. If you want to set the maximum number of transitions to greater than zero so you can see how many times the nodes are toggling, see the “Activity Report For Nodes” on page 5-2.

### **3.8.2 Errors Menu Selection**

The Reports/Errors menu selection reports the errors that occurred during read-in, preprocessing or simulation. An error level of “1” indicates a warning. Error levels 2 through 5 will prevent simulation until the error has been corrected.

### **3.8.3 Fault Menu Selection**

The Reports/Fault menu selection reports the fault simulation results.

### **3.8.4 Iteration Menu Selection**

The Reports/Iteration menu selection is useful for finding order of evaluation problems and race conditions. This menu selection displays every signal that has more than one iteration at a timepoint from time=0 to the current timepoint. To view a graphical display for iterations at a timepoint, see “Display Iteration Data Menu Selection” on page 3-71.

# Nonconvergence

## 3.8.5 Nonconvergence Menu Selection

The Reports/Nonconvergence menu selection generates a report of any nonconverged nodes and their oscillating states for the time point that nonconvergence occurred. The Nonconvergence menu selection can **only** be used to debug nonconvergence in **gate** level designs. For behavioral level designs, the Nonconvergence menu selection will produce a report that states there is no data to report. To debug nonconvergence for **behavioral** designs see “Nonconvergence (“Hanging”) for Behavioral Designs” on page 3-34.

### 3.8.5.1 Nonconvergence For Gate Designs

The Reports/Nonconvergence menu selection reports the following information:

- names of the unresolved devices and nodes.
- the “type” of device and node as either a “.type” data keyword, “NODE” for a wired connection with at least one bi-directional device or “BUS” for a wired connection between two or more unidirectional enabled gates.
- the node state values for the nonconvergence time point. State values reported are preceded by a “...” to indicate possible previous states.

Nonconvergence only occurs when gate delays are zero. Zero delays can cause the circuit to oscillate at a single timepoint. As the circuit oscillates, the simulator must iterate over the circuit trying to resolve the circuit to a single set of values. When the iteration limit is exceeded, the circuit nonconverges. A simple example of a circuit that would cause nonconvergence is a ring of three inverters whose delays are set to zero. Zero-delays occur during logic simulation when either a zero delay, or no delay is specified for devices (the default delay for Verilog HDL gate devices is zero). Zero-delays also occur during logic initialization (LINIT command), which forces delays to be zero so that Silos can perform a steady-state, DC solution of the circuit at time=0.

Nonconvergence may be due either to circuit path length or problems with designs involving feedback. To eliminate oscillations caused by problems involving feedback, the circuit design must be corrected. When nonconvergence is due to path length, increasing the iteration limit should enable the circuit to converge. In general, each node in a serial path length requires one iteration to propagate a signal. The iteration limit during simulation is specified by the “.CONTROL .MXITR” command. The iteration limit at time=0 is specified by the “.CONTROL .MXDCI” command. Arbitrarily increasing the iteration limits is not recommended as it may dramatically increase the execution time necessary to reach nonconvergence.

(continued on next page)

## Nonconvergence

To debug a non-convergence due to a problem in the circuit design, you will need to use the NONCONV command to store the nonconvergence report (see the “Nonconvergence Summary” on page 5-30). Many times an important clue in solving a nonconvergence is to know which signal started oscillating first. The "probe" command can be used to find out which signal started oscillating (see “Probing Node States” on page 5-35). Use following steps to debug your circuit:

- In the Command window for the Main toolbar, enter the "disk" command to name an output file, such as:

```
disk file1
```

- Then, in the Command window for the Main toolbar, enter the "nstore nonconv" command to store the nonconvergence report in file1.
- Next edit file1 and copy and paste the net names from the nonconvergence report to another file (file2) that you have the probe command in, for example:

```
!probe iter time1 time_end net1,,net2,,net3
```

where time1 is the timepoint before the non-convergence and time\_end is the nonconvergence timepoint. If the net names use the SILOS style "(" to delimit hierarchy, then you will have to change the "(" to ".".

- Then, in the Command window for the Main toolbar, prompt input the file that has the !probe command:

```
input file2
```

From the probe report, determine which net is oscillating first.

- Next open the Data Analyzer and the Explorer Window. In the Explorer Window, select the signal that was first oscillated and drag and drop it to the Data Analyzer. Then highlight the signal in the Data Analyzer and use the "Trace Signal Inputs" menu to trace backwards from the net that started oscillating so that you can draw the circuit for that net and figure out the cause of the nonconvergence.

The Nonconvergence menu selection only reports the basic options for the nonconvergence report. For additional options, such as the INPUT option that reports the states for all inputs of devices which drive the oscillating nodes, and the ITER=val option that specifies the iteration number to the 1st of eight states for each node reported for nonconvergence, see the “Nonconvergence Summary” on page 5-30.

(continued on next page)

## Nonconvergence

### 3.8.5.2 Nonconvergence (“Hanging”) for Behavioral Designs

When nonconvergence occurs in behavioral designs, Silos may be able to stop the simulation and report an error stating that there has been nonconvergence. For nonconvergence during behavioral simulation, the “Reports/Nonconvergence Menu” selection report may produce a report that states there is no data to report. When this happens, you can click on the Step button on the Main Toolbar and immediately begin to single step in the nonconverged source code.

Infinite loops in the user’s behavioral code can cause Silos to “hang” and not respond. When the infinite loop occurs at time=0 Silos will “hang” and never get to the “Ready:” prompt. When the infinite loop occurs during simulation, Silos will hang and does not respond to the "STOP" button or the Esc key on the keyboard. Different techniques are used to debug “hangs” at time=0 and “hangs” during simulation.

When Silos hangs at time=0 and the Output Window never displays the "Ready:" prompt, this is usually due to an error in the behavioral code in an initial or an always block. The below example hangs at time=0 due to the incorrect code "i = +1" which should be "i = i+1" in the below "for" loop:

```
module hang_at_0;
    reg a;
    integer I;
    initial for (i = 0; i < 10; i = +1)
        a = ~a;
endmodule
```

To debug a design that hangs at time=0, use comments “/\* \*/” to comment out portions of the design until it no longer hangs at time=0. Then inspect the commented code and fix the problem in the behavioral code.

(continued on next page)

## Nonconvergence

When Silos hangs during simulation the program will not respond to the STOP button or the Esc key on the keyboard. User errors in behavioral code are usually what cause Silos to "hang". These errors are usually caused by a loop with no delay, such as in these code segments:

Code Segment 1:

```
for (i = a; b < c; i = i + 1) begin
    if (d & 9'h100) == 0) begin
        b = b + 1;
    end
end
```

Code Segment 2:

```
always @a
    b = ~b;
always @b
    a = ~a;
```

Code segment 1) hangs when the "if" test is not true, causing the "for" loop to infinitely loop at a time step. Code segment 2) hangs because each "always" block is triggering the other "always" block and neither "always" block has delay.

To debug a "hang" during simulation, run the simulation until it hangs, and then note the simulation time value on the status bar in the lower right-hand corner of Silos. Next kill Silos, restart Silos, and press the "Load/Reload Files" button. When Silos stops at time=0, enter a simulation time to one less than when the simulation hangs in the Command window for the Main toolbar. Such as, if the simulation hangs at time=11251ns, enter "sim 11250ns" so that Silos stops just before it hangs. Then press the "Step" button. Keep single-stepping until you find the problem code.

If you single-step past the time point where the simulation was hanging, then the time value for the status bar had not yet been updated when Silos hung. To find the true time value that Silos hangs at, keep entering short simulation times at the "Ready:" prompt until the simulation hangs, such as enter "sim 10ns" or "sim 1ns" until the simulation hangs. Then restart Silos, re-simulate to just before the simulation hangs, and single-step until you find the problem.

## Size

### 3.8.6 Size Menu Selection

The Reports/Size menu selection reports the memory usage for Silos. A simpler method of obtaining the memory usage is to select the Help/About Silos menu selection to open the About Silos dialog box, which has the memory usage for Silos. The memory usage will increase during circuit read-in and preprocessing until simulation starts at time=0. After simulation starts the memory usage remains constant.

## Explorer Menu

### 3.9 Explorer Menu

The “Explorer” menu provides the following menu selections:

- Open Explorer menu selection;
- Go to Module Source menu selection.

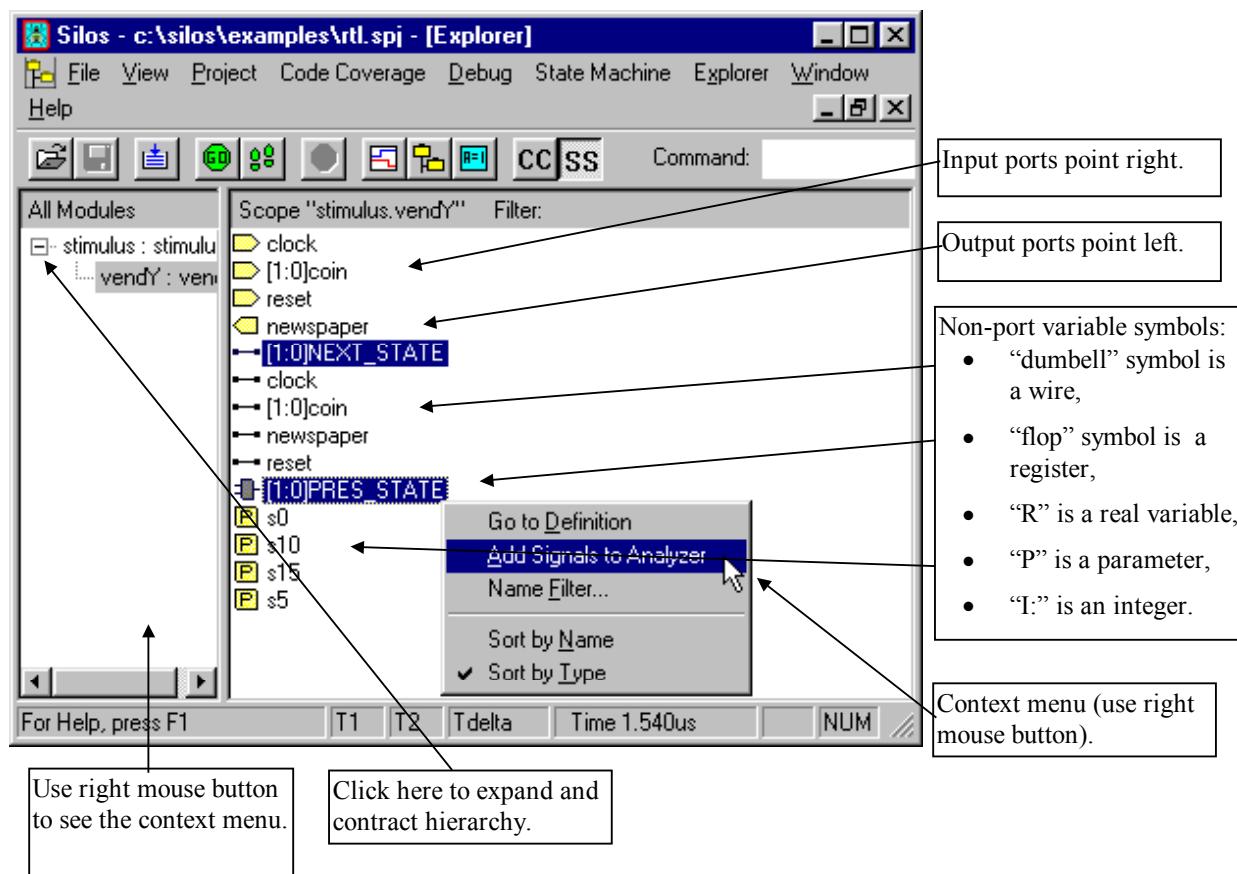
#### 3.9.1 Open Explorer Menu Selection

The Explorer/Open Explorer menu selection opens a hierarchical Explorer Window. The Explorer displays the name of every module instance and variable in the design in a tree structure similar to the directory structure for the Windows Explorer. The **shift** and **control** keys can be used to select variable names in a similar manner to the Windows Explorer. Names can then be dragged and dropped to the other windows, such as the Data Analyzer Window and the Watch Window. The Explorer Window also has context menus that can be accessed by using the right mouse button (see “Explorer Window” on page 3-58).

## Explorer Menu

### (Open Explorer Menu Selection)

The hierarchical Explorer Window is divided into two vertical windows with the hierarchy of module instances and gates listed in the left window and the variable names listed in the right window. To traverse the hierarchy of the design in the left window, click-on the plus sign to the left of the instance, or double-click on module instances. As each module instance or gate is selected in the left window, the names of the variables in that instance are displayed in the right window. Symbols to the left of the variables distinguish port variables (the symbol is a pad) from non-port variables (the symbol for a wire is a wire connecting two points, the symbol for a register is a “flop” symbol, “P” is used for parameters, “R” for real variables and “I” for integer variables.). Input ports have the pad symbol pointing to the right, output ports have the pad symbol pointing to the left, and inout ports have the pad symbol pointing in both directions.



### 3.9.2 Go to Module Source Menu Selection

The “Explorer/Go to Module Source” menu selection opens a source window that displays the source code for the hierarchical instance you selected in the left hand “Tree” side of the Explorer Window. This enables you to quickly find the source code for a module instance when you have a large design that spans many files scattered across many directories.

# Debug Menu

## 3.10 Debug Menu

The “Debug” menu provides the following menu selections:

- Enable Single Step/Breakpoints menu selection;
- Go menu selection;
- Break Simulation menu selection;
- Finish Current Timepoint menu selection;
- Step menu selection;
- Breakpoints menu selection.

### 3.10.1 Enable Single Step/Breakpoints Menu Selection

The “Debug/Enable Single Step/Breakpoints” menu selection turns on or off the single stepping capability, and the breakpoints capability. The advantage of turning of the debugging would be to increase simulation speed, perhaps by as much as 25%.

### 3.10.2 Go Menu Selection

The “Debug/Go” menu selection performs logic simulation. The “Go” button is also available on the Toolbar.

When “Go” is selected the files specified for the project are automatically input into Silos (unless they have already been input). Logic simulation is then run until a “\$stop” or “\$finish” is encountered in the design. During simulation, the “Go” button will change into a “Stop” button which can be clicked on to halt the simulation at any time.

The “Go” button can be useful during single stepping to skip across uninteresting source code to the next breakpoint, at which point the single stepping can be resumed.

#### 3.10.2.1 Simulation Suggestions

##### Libraries

To increase the speed of processing your design, use the library feature to specify large library files from vendors. Any file of Verilog source code can be specified as a library file. To specify library files, see the “Files Menu Selection” on page 3-13. Library files can also be specified from the command line, see “Command-line Options” on page 7-4.

(continued on next page)

# Simulation

## Simulation Save File Size

Silos is very efficient in saving the simulation results to disk. In general, there is only a 10% to 15% difference in speed between the saving every variable in the hierarchy and saving nothing. However, if the save file on disk becomes large enough (over a few hundred megabytes) then the time spent writing to disk may be significant.

To reduce simulation save file size on disk, you can select which parts of the hierarchy that you want to save the simulation results for. To specify which instances to save information for during simulation, select the “Properties” menu selection in the context menu for the left-hand side of the Explorer Window.

## Restarting Simulation

Silos has the ability to restart the simulation from any timepoint by using the save and restore feature (see “Save Project State Menu Selection” on page 3-15 and “Restore Project State Menu Selection” on page 3-15). Using this feature eliminates the time required to re-input and re-simulate the design. A debugging strategy may be to simulate to a timepoint, and then use the Project/Save Project State menu to save the state of the simulation. You can then set variables to a value and continue simulation. To restart the simulation, exit Silos, then restart Silos, and select the Project/Restore Project State menu to return the simulation to the timepoint the simulation was saved at.

To restart the simulation from time=0, you do not need to exit Silos. Instead, click on the “Load/Reload Input Files” button on the Main Toolbar. This will re-input the design and simulate to time=0.

## Loss of Simulation Data

To prevent the loss of the simulation results due to unexpected interruptions, you can save the state of Silos after logic simulation. When simulation has completed, save the simulation results by selecting the “Project/Save Project State” menu. Then open the Data Analyzer, review your simulation results, and debug your design. If you need to re-enter Silos, you can select the “Project/Restore Project State” menu, open the Data Analyzer and view the waveforms without re-simulating.

### 3.10.3 Break Simulation Menu Selection

The “Debug/Break Simulation” menu selection will stop the Silos program from simulating logic simulation. The Escape key (“Esc”) on the keyboard or the “STOP” button on the Toolbar performs the same function.

## Debug Menu

### 3.10.4 Finish Current Timepoint Menu Selection

The “Debug/Finish Current Timepoint” menu selection is used to continue the simulation until the end of the current timepoint. This is useful to complete the time step during single-stepping so that the Data Analyzer waveforms are updated.

### 3.10.5 Restart Simulation Menu Selection

The “Debug/Rerun Simulation” menu selection restarts the Silos program from time zero. This is useful if you have reviewed the simulation results and want to rerun the simulation to set breakpoints or to force signals to a value.

### 3.10.6 Step Menu Selection

The “Debug/Step” menu selection single steps through the HDL source code for the project. As Silos single steps it places a “yellow arrow” to the left of the line.

Single stepping can be very useful when combined with breakpoints and the Watch Window for debugging behavioral code. As you step through the HDL source code you can highlight variables and expressions and drag and drop them into the Data Analyzer Window and the Watch Window. The Toolbar also has a “Step” button.

(continued on next page)

## Debug Menu

### 3.10.7 Breakpoints Menu Selection

The “Debug/Breakpoints” menu selection opens the “Breakpoints” dialog box for setting breakpoints. A typical method to debug a design using breakpoints would be to set a breakpoint in a module instance, then click on the “Go” button on the toolbar and simulate until the simulation stops in the module with the breakpoint. Next single step through the module to review how the source code is executing and watch variables change value in the Watch Window and the Data Analyzer Window. The “Go” button could then be used again to simulate until the simulation stops in the module and single-stepping is resumed.

There are four types of breakpoints:

Break at Simulation Time:

The "Break at Simulation Time" selection allows you to specify a simulation time and stops the logic simulation before the specified time is simulated. To specify a stop time, select “Break at Simulation Time” in the “Type” box and enter the stop time in the “Timepoint” box. Then choose the “Add” button.

Break at Location:

The “Break at Location” selection stops logic simulation before the selected source line is simulated. To select a breakpoint location, you can use the “Toggle Breakpoint” button on the Toolbar or you can use the Breakpoints dialog box. To use the “Toggle Breakpoint” button, first open a source file window by single-stepping with Silos, or use the “File/Open” menu selection or “Open” button on the Toolbar to open the source file window. Next, put the mouse cursor on an HDL source line you want to stop at. Then click-on the “Toggle Breakpoint” button on the Toolbar to set the breakpoint. A red stop sign symbol will be placed to the left of the line next to the line number. You can also see that the breakpoint has been set in the “Breakpoints” dialog box.

Break in Module Instance:

The “Break in Module Instance” selection allows you to select a module instance and then stop logic simulation each time a source line in the selected module instance is to be simulated. To select a breakpoint location, use the Explorer Window to highlight the module instance you want to select. In the “Breakpoints” dialog box select “Break in Module Instance” for the “Type” box. The name of the module instance that you selected from the “Explorer” Window will then appear in the “Scope” box. Next press the “Add” button to add the name of the module instance to the list of breakpoints.

(continued on next page)

## Debug Menu

### (Breakpoints Menu Selection )

Break in Module (Any Instance):

The “Break in Module (Any Instance)” selection allows you to select a module instance and then stop logic simulation each time a source line in any instance of the module is to be simulated. To select a breakpoint location, use the Explorer Window to highlight the module instance you want to select. In the “Breakpoints” dialog box select “Break in Module (Any Instance)” for the “Type” box. The name of the module instance that you selected from the “Explorer” Window will then appear in the “Scope” box. Next press the “Add” button to add the name of the module instance to the list of breakpoints.

The “Add” button adds the specified breakpoint to the “Breakpoints” list. Active breakpoints are preceded by a plus (“+”) sign. Inactive breakpoints (“Disable” button) are preceded by a minus (“-”) sign. Individual breakpoints can be deleted with the “Delete” button. All breakpoints can be deleted by the “Clear All” button. The “OK” button closes the “Breakpoints” dialog box and saves the changes. The “Cancel” button closes the “Breakpoints” dialog box without saving the changes.

# Options Menu

## 3.11 Options Menu

The “Options” menu provides the following menu selection:

- Fonts menu selection;
- Tabs menu selection;
- Snap to Edges;
- Title Tips menu selection;
- Analog Integer Display;
- Strength Color Coding;
- Trace Color Coding;
- White Background;
- Black Background;
- Full Path Title;
- Data Tips;
- Syntax Color Coding.

### 3.11.1 Fonts menu selection

The “Options/Fonts” menu selection opens the “Fonts” dialog box for setting the fonts for the Data Analyzer Window and the source windows..

### 3.11.2 Tabs menu selection

The “Options/Tabs” menu selection opens the “Tabs” dialog box for setting the number of spaces in the Tab Interval for the source windows..

### 3.11.3 Snap to Edges

When setting the T1 and T2 timing markers for the Data Analyzer, they will **snap to the nearest edge** if the “Options/Snap to Edges” menu selection is active. When setting a timing marker, you can hold down the **shift key** to temporarily toggle the “Snap to Edges” selection to its opposite effect. Such as, if the “Snap to Edges” is not selected, you can have the timing marker snap to the nearest edge when you set it by holding down the shift key as you click-on the left or right mouse button with the mouse indicator (arrow) in the Waveform Display Window.

(continued on next page)

## Options Menu

### (Options Menu)

#### 3.11.4 Title Tips menu selection

The “Options/Title Tips” menu selection enables the title tips for the signal names in the Data Analyzer. The title tips show the full hierarchical path name for a signal.

#### 3.11.5 Analog Integer Display

Integer variables can be displayed as a vector type of waveform or as an analog waveform. The “Options/Analog Integer Display” menu selection sets the method of displaying integers.

#### 3.11.6 Strength Color Coding

The “Options/Strength Color Coding” menu selection will use different colors to denote waveform strength in the Data Analyzer:

- Supply strength - black
- Strong strength - blue
- Pull strength - green
- High-Z, Unset, and Uncertain strength - red
- mixed strength for inserted groups and vectors - purple

#### 3.11.7 Trace Color Coding

The “Options/Trace Color Coding” menu selection will uses a single color to denote waveforms in the Data Analyzer:

- Black background - white signal traces
- White background - black signal traces

#### 3.11.8 White Background

- The “Options/White Background” menu selection will set the background color for the Data Analyzer to white.

(continued on next page)

## Options Menu

### (Options Menu)

#### 3.11.9 Black Background

- The “Options/Black Background” menu selection will set the background color for the Data Analyzer to Black.

#### 3.11.10 Full Path Title

The full directory path for a source file can be turned “on” or turned “off” by using the “Options/Full Path Title” menu selection.

#### 3.11.11 Data Tips

The Data Tips feature for displaying the value, scope, radix, and simulation time point for a variable or expression in the source window can be toggled “on” or “off” with the Options/Data Tips” menu selection. Any variable or expression in a source window can be viewed by opening the source window and holding the mouse cursor over a variable or by highlighting an expression and holding the mouse cursor over the expression. This feature enables the designer to trace the cause of problems directly in a Verilog HDL source code window. The “Go to Source code” menu selection can be used to quickly display the module definition for any instance in the hierarchy. This saves time opening the source window for displaying the value for variables and expressions when there are numerous files in the design.

#### 3.11.12 Syntax Color Coding

The “Options/Syntax Color Coding” menu selection will set Verilog HDL keywords and constructs to different colors in the source windows for Silos, thus increasing the readability of the Verilog HDL text.

# Window Menu

## 3.12 Window Menu

The “Window” menu provides the following menu selections:

- Cascade menu selection;
- Tile menu selection;
- Arrange Icons;
- Open Explorer menu selection;
- Open Watch menu selection;
- Open New Data Analyzer menu selection.

### 3.12.1 Cascade Menu Selection

The Window/Cascade menu selection arranges multiple opened windows in an overlapped fashion.

### 3.12.2 Tile Menu Selection

The Window/Tile menu selection arranges multiple opened windows in a non-overlapped fashion.

### 3.12.3 Arrange Icons Menu Selection

The Window/Arrange Icons menu selection arranges icons for the minimized windows.

## Window Menu

### 3.12.4 Open Explorer Menu Selection

The Window/Open Explorer menu selection opens the hierarchical Explorer Window that displays the name of every module instance and variable in the design in a tree structure similar to the directory structure for the Windows Explorer (for more information, see “Open Explorer Menu Selection” on page 3-37).

### 3.12.5 Open Watch Menu Selection

The Window/Open Watch menu selection opens the Watch Window that can be used to display the state value for specified variables and expressions as you single-step through the design. Variables or expressions can be dragged and dropped into the Watch Window from any source file window, from the Explorer Window, and from the Data Analyzer window. The Watch Window also has a context menu for setting and forcing variables to a value. The context menu can be accessed by using the right mouse button. For more information on the Watch Window, see “Setting and Forcing Values” on page 2-51.

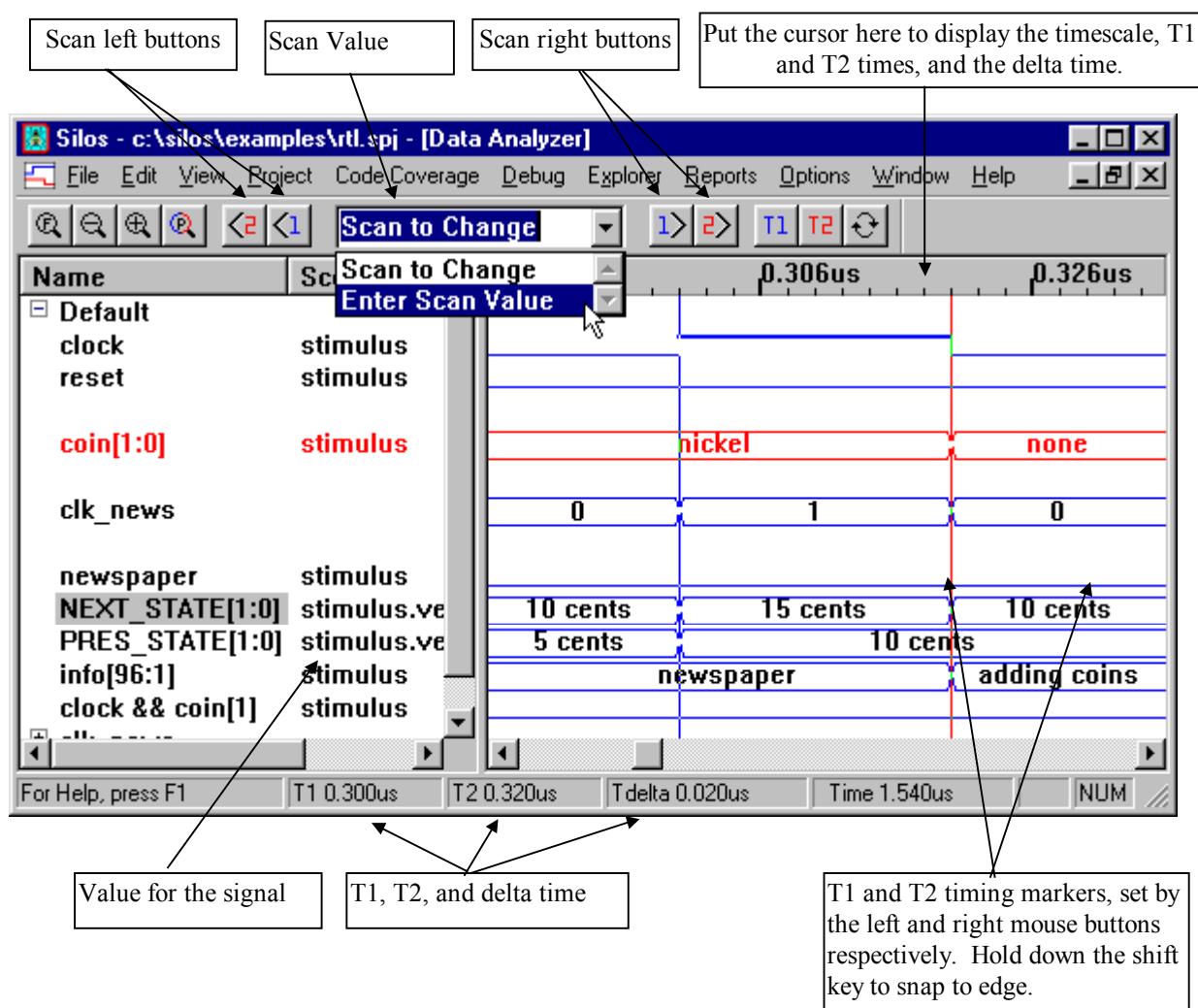
(continued on next page)

## Data Analyzer

### 3.12.6 Open New Data Analyzer Menu Selection

The Window/Open New Data Analyzer menu selection opens the Data Analyzer Window that can be used to display the waveforms for specified variables and expressions as you single-step through the design. Variables or expressions can be dragged and dropped into the Signal Window for the Data Analyzer from any source file window and from the Explorer Window.

The Data Analyzer Window displays the logic simulation results as waveforms. The list box to the left of the waveforms shows the signal's "Name", its "Scope", and the "Value" of the signal at either the left axis of the waveform window or the T1 timing marker. To copy the waveform display to Microsoft Word, select the "Edit/Copy Image to Clipboard" menu selection when the Data Analyzer Window has the focus, and then paste it into Word.



# Waveforms

## 3.12.6.1 Digital and Analog Signal Display

### Digital Signals

For digital signals, simulation results are displayed with waveforms denoting signal levels and colors denoting signal strength (on color monitors).

- Supply strength - black
- Strong strength - blue
- Pull strength - green
- High-Z, Unset, and Uncertain strength - red
- mixed strength for inserted groups and vectors - purple

Either the High level or Low level for the signal trace can be displayed as a thicker horizontal line. The default setting is for the High level to be a thicker line.

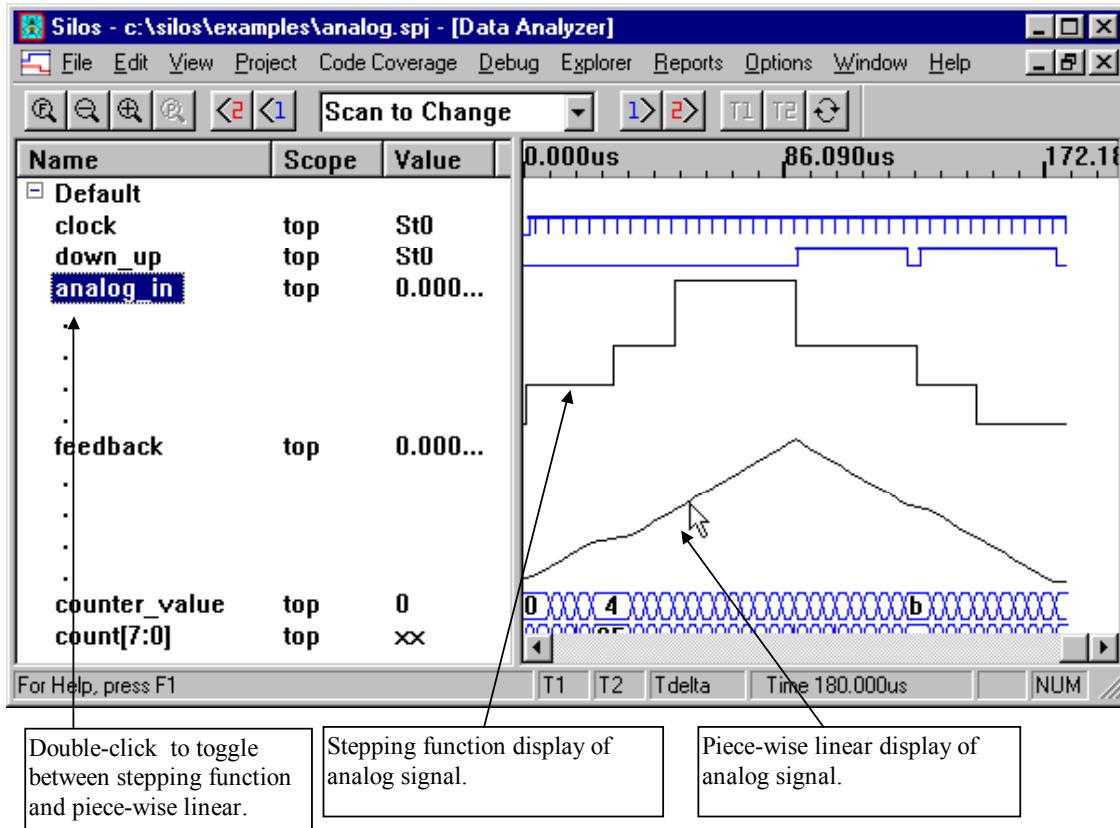
For group names that are inserted into other groups, vector names, and the name for real or integer variables, the value is displayed in the center of the waveform (resolution permitting). Double-clicking on an inserted group or vector name will show (or hide) the individual bits. Color is used to denote the strength for vector signals. Purple is used to represent a vector whose bits are at different strengths. If the timing markers have been set, then the vector value is provided at the top of the display along with the vector strength. If the bits for the vector are at different strengths, then “Mixed Strength” is displayed at the top of the display.

When the timing markers have been set, their value is shown in the status bar at the bottom of the Data Analyzer. If the bits are at different strengths then “Mixed Strength” is displayed for the timing marker value.

(continued on next page)

# Analog Signals

## Analog Signals



Analog signals for real variables are displayed as piece-wise linear or step waveforms. Double-clicking on the signal name will toggle between the two methods of displaying analog signals for real variables.

Integer variables can be displayed as a vector type of waveform or as an analog waveform. Select the "Options/Analog Integer Display" menu selection to set the method of displaying integers. Displaying integer variables as analog waveforms can be very useful for designing digital filters, etc.

## Data Analyzer

### 3.12.6.2 Notes on using the Data Analyzer Window

#### Viewing More Waveforms

To create more space to display waveforms:

- Decrease the size of the “Name”, “Scope”, or “Value” buttons by grabbing and moving the vertical edge for the buttons.
- Increase the size of the Waveform Display by grabbing the vertical line that separates the “Name” list from the Waveform Display and moving the vertical line to the left or right.
- Use the “View” menu to hide the Status Bar or Tool bars.
- Grab the Tool bars and move them to any part of your monitor display, even outside of the Silos program.

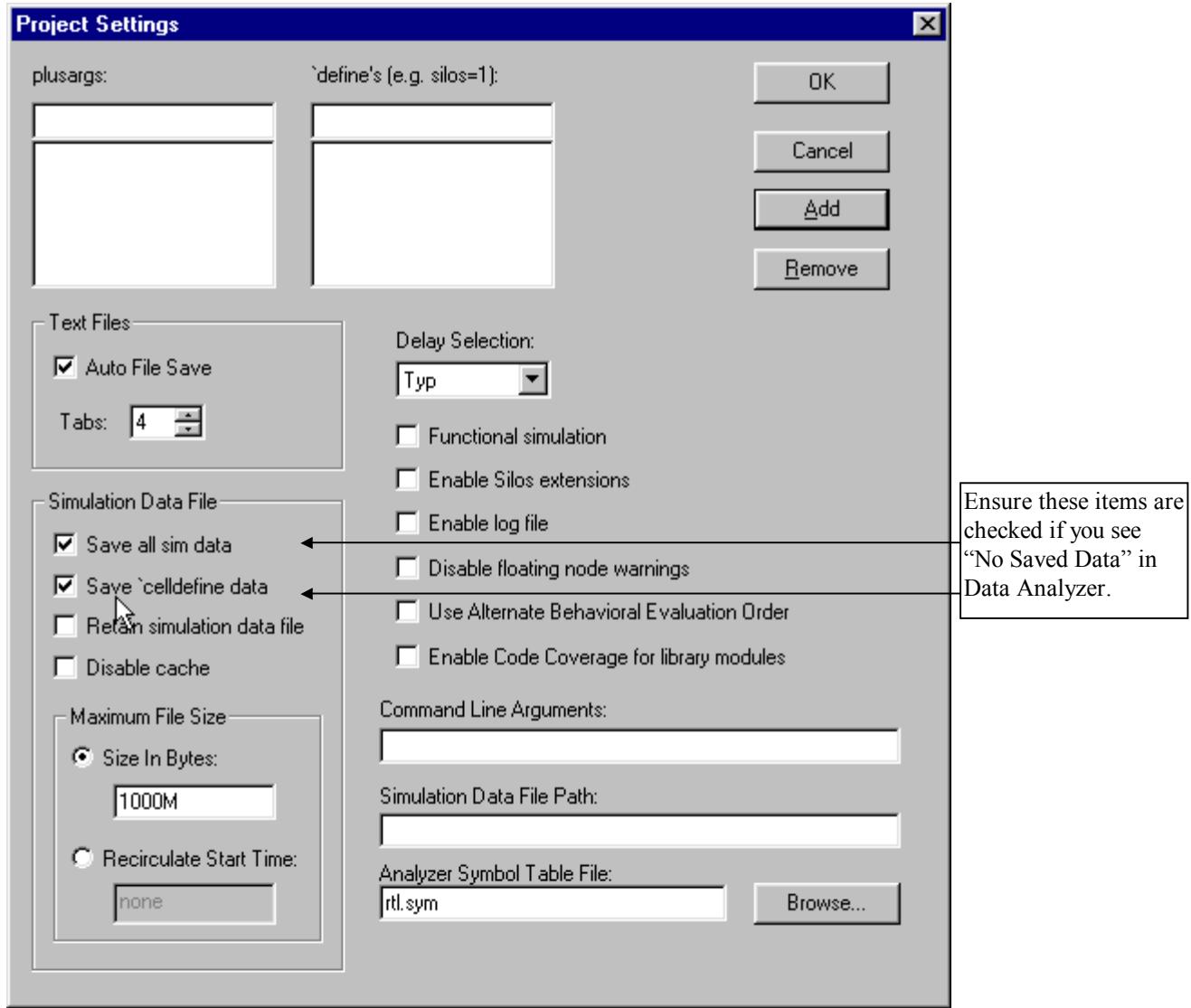
#### Multiple Data Analyzers

To open one or more Data Analyzer windows, you can use the “Open Analyzer” button on the Toolbar. Each time the Analyzer is started, it can be used to simultaneously display another copy of the simulation results.

(continued on next page)

## Data Analyzer

**“No Saved Data”**



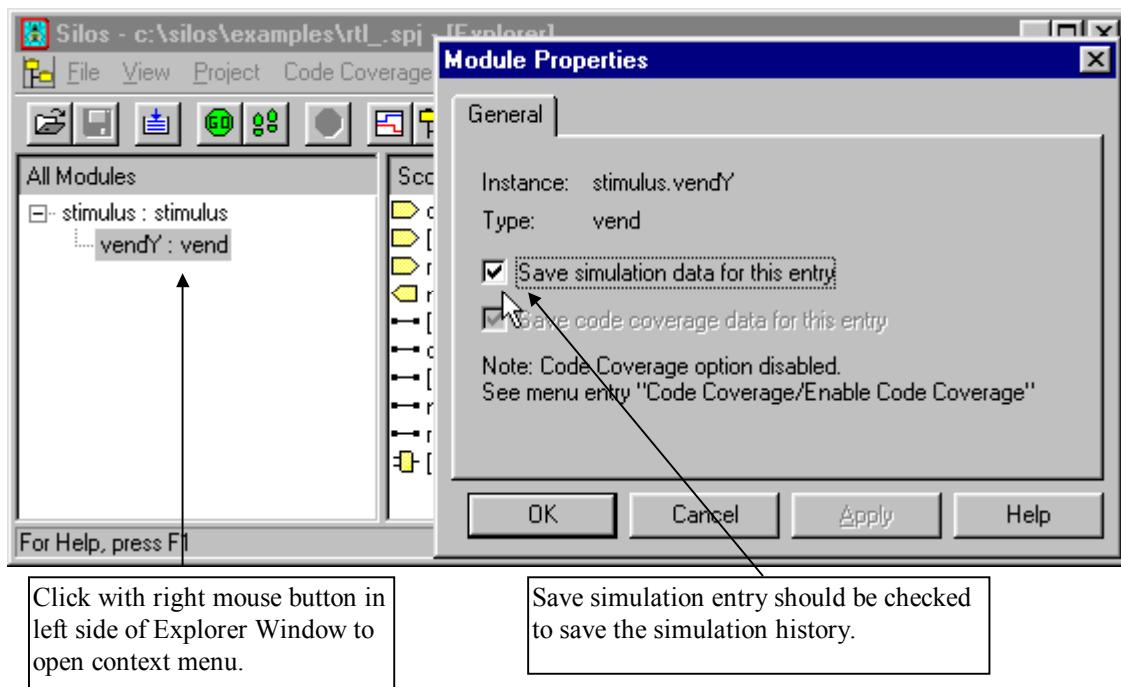
If the Data Analyzer reports “No Saved Data” for a waveform, check the following:

- The “Save `cellddefine data`” option in the “Project Settings” dialog box may need to be enabled (Project/Project Settings menu). This will save the local variables for modules in library cells that are bounded by `cellddefine` and `endcellddefine`.
- The “Save all sim data” option in the “Project Settings” dialog box may need to be enabled (Project/Project Settings menu). This will save the simulation history for every variable.

(continued on next page)

## Data Analyzer

- The “Save simulation data for this entry” in the “Module Properties” dialog box may need to be enabled (Explorer Window/Tree/Properties menu). See “Keeping Module Instance Simulation Variable Values” on page 5-29 for a simpler method of saving instances in the hierarchy.



## Help Menu

### 3.13 Help Menu

#### 3.13.1 Contents menu selection

The Help/Contents menu selection opens the contents listing for the “Silos User’s Manual” on-line help file.

#### 3.13.2 Using Help menu selection

The Help/Using Help menu selection opens a Microsoft help file for an explanation of how to effectively use the Index and on-line Help.

#### 3.13.3 Quick Start Guide menu selection

The Help/Quick Start Guide menu selection has instructions for opening a project, creating a new project, and simulating.

#### 3.13.4 User’ Manual menu selection

The Help/User’s Manual menu selection provides the complete Silos User’s Manual.

#### 3.13.5 Verilog LRM menu selection

The Help/Verilog LRM menu selection provides the complete OVI Verilog Language Reference Manual version 1.0.

#### 3.13.6 SDF Manual menu selection

The Help/SDF Manual menu selection provides the complete OVI Standard Delay Format (SDF) Manual version 2.0.

#### 3.13.7 About Silos

The Help/About Silos menu selection opens the “About Silos” dialog box. This dialog box contains the Silos version number, copyright notice, and the “Sim Engine Mem” value for the Silos logic simulation engine memory usage. The first number listed for the “Sim Engine Mem” is the memory allocated for the Silos logic simulation engine. The second number listed for the “Sim Engine Mem” is the total amount of memory that Silos thinks it can possibly allocate for your computer. The memory usage for Silos is constant for logic simulation. To determine the memory usage for logic simulation, select the “Load/Reload Input Files” button on the Main Toolbar and Silos will simulate to time=0. You can then use the “About Silos” dialog box to determine the RAM memory usage for your design during logic simulation.

# Registration

## 3.13.8 User Registration

The Help/User Registration menu opens the “User Registration” dialog box to update the Silos registration.

The following is an explanation for the registration fields:

- “Name”: This field is where you can personalize this copy with your name (optional).
- “Company”: This field is for your company name (optional).
- “Serial #”: This is the serial number of your security block on the parallel port. Silos automatically interrogates the parallel ports on your computer for the serial number.
- “NetId: This is your Ethernet card address.

When you have completed the registration information, click on the “OK” button.

## Context Menus

### 3.14 Context Menus

Silos has context menus for the Watch Window, Explorer Window, Data Analyzer Window, Output Window, and the source windows,. The context menus for the Output Window and the source windows allow you to Cut, Copy, and Paste.

The context menus for windows are:

- The right-hand side of the Explorer Window has a context menu for the “Go to Definition”, “Add Signals to Analyzer”, “Name Filter”, “Sort by Name”, and “Sort by Type” menu selections. To invoke this context menu, use the right mouse button to click on any part of the right-hand side (the side with the signal names) of the Explorer Window. The context menu will remain open while the left mouse button is used to select a menu item.
- The left-hand side of the Explorer Window has a context menu for the “Copy Scope”, “Go To Source code”, “Go To Scope”, “Instance Name Filter” and “Properties” menu selections. To invoke this context menu, use the right mouse button to click on any part of the left-hand side (the side with the hierarchical tree) of the Explorer Window. The context menu will remain open while the left mouse button is used to select a menu item.
- The Watch Window has a context menu for the “Add Signal/Expression” “Set Value”, “Free Forced Wire”, and “Clear All” menu selections. To invoke this context menu use the right mouse button to click on any part of the Watch Window. The context menu will remain open while the left mouse button is used to select a menu item.
- The Data Analyzer has a context menu for the “Goto Timepoint”, “Pan to T1”, “Pan to T2”, “Pan to Last View”, “Timescale”, “Snap to Edges”, “Add Bookmark” and “Delete Bookmark” menu selections. To invoke this context menu use the right mouse button to click on any part of the time point display area (the gray area just above the Waveform Display Window). The context menu will remain open while the mouse is used to select a menu item.
- The Data Analyzer has a context menu for the “Trace Signal Inputs”, “Display Iteration Data”, “New Group”, “Delete Group”, “Insert Group”, “Show Groups”, “Set Radix”, “Add “One” Bit”, “Add “Zero” Bit”, “Reverse Bit Order”, “Add Signal”, “Add Blank Line”, “Set Strength Color Coding”, “Set Trace Color”, “Delete Item/s”, “Clear Signal List”, and “Reload Groups”. To invoke this context menu, use the right mouse button to click on any part of the Signal List Box. The context menu will remain open while the left mouse button is used to select a menu item.

# Explorer Name Filter

## 3.14.1 Explorer Window

### 3.14.1.1 Go To Definition Menu Selection

The “Explorer/Signal/Go to Definition” menu selection is located in the context menu in the right hand "Signal" side of the Explorer Window. The “Go to Definition” menu selection will automatically open the source file, and highlight the definition line for the variable that is selected in the right side of the Explorer Window.

To invoke this menu selection, use the right mouse button to click on a variable in the right hand side of the Explorer Window to open the context menu. The context menu will remain open while the mouse is used to select the “Go to Definition” menu item.

### 3.14.1.2 Add Signals to Analyzer Menu Selection

The “Explorer/Signal/Add Signals to Analyzer” menu selection is located in the context menu in the right hand "Signal" side of the Explorer Window. The Add Signals to Analyzer menu selection is useful for adding signals to the Data Analyzer when it is difficult to drag and drop the signals due the screen size.

To invoke this menu selection, use the right mouse button to click on any part of the right hand side of the Explorer Window to open the context menu. The context menu will remain open while the mouse is used to select the Add Signals to Analyzer menu item.

### 3.14.1.3 Name Filter Menu Selection

The “Explorer/Signal/Name Filter” menu selection is located in the context menu in the right hand "Signal" side of the Explorer Window. The Explorer Window supports the ability to filter the names in the right-hand window of the Explorer.

To invoke this menu selection, use the right mouse button to click on any part of the right hand side of the Explorer Window to open the context menu. The context menu will remain open while the mouse is used to select the Name Filter menu item.

The name filtering in the Explorer Window uses regular expressions. Explanations for regular expressions can be found in many programming books. A short description of regular expressions is provided in this help file.

(continued on next page)

## Explorer Name Filter

### (Name Filter)

A regular expression is a notation for specifying and matching strings. Regular expressions have two basic kinds of characters;

- special characters      These are characters that have special meaning for matching strings.

The special characters for regular expressions are:

\ ^ \\$ . [ ] \* + ?

where:

- \      this escapes or quotes other characters, such as \\$ matches the “\$”. A useful quoted string is \| which means “or”. Another useful quoted string is \(` and \)` which allow the parenthesis to be used to group regular expressions. For example, ac\* means the character “a” and zero or more characters of “c”. However, \(`ac`)\* means zero or more occurrences of the character string “ac”.
- ^      this matches the preceding character at the beginning of a string. When ^ is the first character in a character class it means the compliment of the character class.
- \$      this matches the preceding character at the end of a string.
- .      this matches any single character.
- [ ]      characters enclosed in brackets are a character class.
- \*
- +      this matches one or more occurrences of the character that precedes the +.
- ?
- ordinary characters      These are all the other available characters. These characters match themselves, such as “a” would match the letter “a”.

The character class [ ] has special rules. Inside a character class, all characters have their literal meaning, except for the quoting character \, ^ at the beginning, and - between two characters. Each of the characters in a character class are treated as an “or” search. For example, [ab] will find any single character name “a” or “b”. The character class [a-z] will match any single character lower case name. The character class [^a-zA-Z] will match any single character name that is not an alpha.

(continued on next page)

## Explorer Name Filter

### (Name Filter)

Some examples of using regular expressions are listed below:

<b>a</b>	matches only the name lowercase “a”
<b>a.*</b>	matches any name that begins with “a”.
<b>.*a</b>	matches any name that ends with “a” and the name “a”.
<b>.*a.*</b>	matches any name that has an “a” anywhere in the name.
<b>[abc]</b>	matches only the names “a” or “b” or “c”.
<b>[a-z]</b>	matches any name that is a single lower case character, such as “b”.
<b>[a-z]*</b>	matches any name that is only lower case characters, for example “data”.
<b>[a-zA-Z]*</b>	matches any name that is upper and/or lower case characters, for example “data”, “DATA” and “Data”.
<b>[a-zA-Z0-9]*</b>	matches any name that has upper and/or lower case characters and/or digits, for example “data”, “DATA”, “Data”, “123”, “data1”.
<b>[a-zA-Z0-9_]*</b>	matches any name that has upper and/or lower case characters, and/or digits, and/or “_”, for example “data”, “DATA”, “Data”, “123”, “data1”, and “DATA_bus1”.

Some programming books (such as AWK and Perl) show regular expressions enclosed with slashes “/ /”. The Silos style of searching for regular expressions is a character search and forward slashes have no special meaning. For example, using the regular expression /a/ to try to find any name that contains an “a” will find only the name “/a/”.

Regular expressions **are not** like the Unix style wildcards (where “?” matches any single character, “\*” matches any pattern, and [list] matches any character in list including ranges). For example, using the regular expression \*a\* to find any name that contains an “a” will not find any names.

If you want use regular expressions to find every name that has an “a”, you can enter:

**.\*a.\***

For the above expression, “.\*a.\*” means search for zero or more occurrences (the first “\*”) of any character (the first “.”), followed by a single character “a”, followed by zero or more occurrences (the second “\*”) of any character (the second “.”). So, with the search “.\*a.\*” you could find names such as “a”, “data”, “read”, etc.

## Explorer Context Menus

### 3.14.1.4 Sort by Name or Type Menu Selections

The “Explorer/Signal/Sort by Name” and the “Explorer/Signal/Sort by Type” menu selections are located in the context menu in the right hand "Signal" side of the Explorer Window. The Explorer Window supports the ability to sort the names in the right-hand window by the type of variable (port, wire, register, etc.), or by name.

### 3.14.1.5 Copy Scope Menu Selection

The “Explorer/Tree/Copy Scope” menu selection is located in the context menu in the left hand "Tree" side of the Explorer Window. When you select a hierarchical instance in the Explorer Window, the “Copy Scope” menu selection (or simultaneously holding down the “Ctrl” and the “C” keys on the keyboard) will copy the hierarchical instance name to the Windows Clipboard. This enables you paste the hierarchical name (simultaneously holding down the “Ctrl” and the “V” keys on the keyboard).

To invoke this menu selection, use the right mouse button to click on any part of the left hand side of the Explorer Window to open the context menu. The context menu will remain open while the mouse is used to select the “Go To Source code” menu item.

### 3.14.1.6 Go to Source code Menu Selection

The “Explorer/Tree/Go to Source code” menu selection is located in the context menu in the left hand "Tree" side of the Explorer Window. When you select a hierarchical instance in the Explorer Window, the “Go to Source code” menu selection opens a source window that displays the source code for the hierarchical instance. This enables you to quickly find the source code for any instance in your design.

To invoke this menu selection, use the right mouse button to click on any part of the left hand side of the Explorer Window to open the context menu. The context menu will remain open while the mouse is used to select the “Go to Source code” menu item.

### 3.14.1.7 Go to Scope Menu Selection

The “Explorer/Tree/Go to Scope” menu selection is located in the context menu in the left hand "Tree" side of the Explorer Window. This feature enables you to quickly find an instance in the Explorer Window. The “Go to Scope” menu selection opens the “Enter Scope” dialog, where you specify the scope for the instance that you want to find. Then click on the “Ok” button and the Explorer Window will highlight the instance you selected.

To invoke this menu selection, use the right mouse button to click on any part of the left hand side of the Explorer Window to open the context menu. The context menu will remain open while the mouse is used to select the Go To Scope menu item.

## Explorer Context Menus

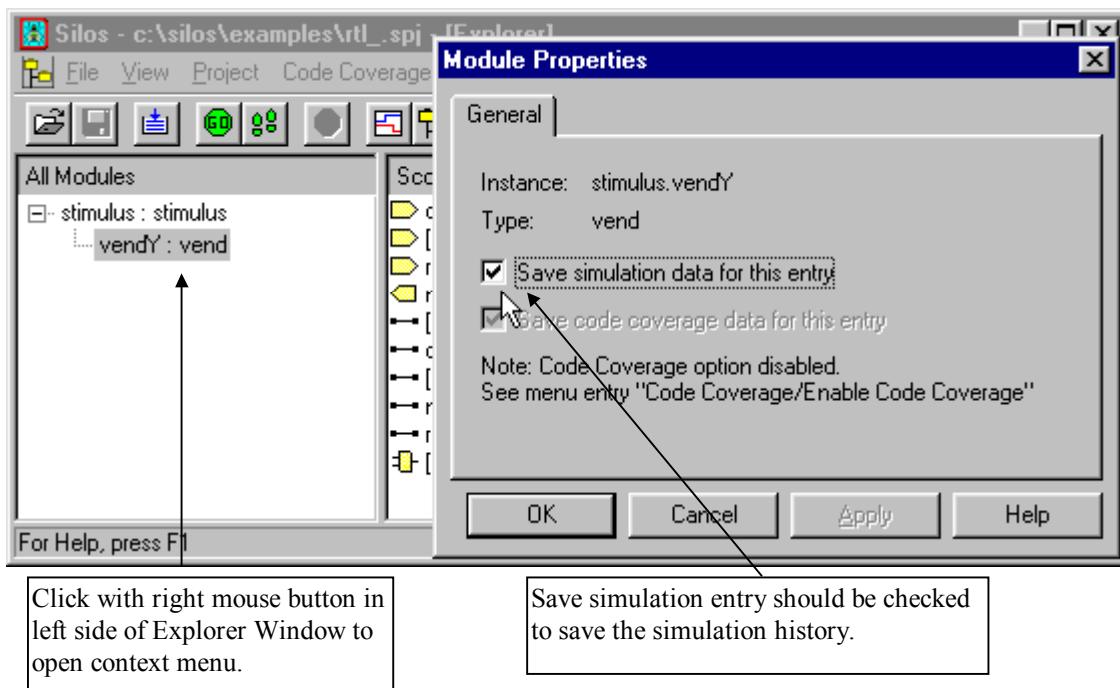
### 3.14.1.8 Instance Name Filter Menu Selection

The “Explorer/Tree/Instance Name Filter” menu selection is located in the context menu in the left hand “Tree” side of the Explorer Window. This feature allows you to use a regular expression to filter the instance names to assist you with quickly finding an instance in the Explorer Window. The “Instance Name Filter” menu selection opens the “Name Filter” dialog, where you specify the filter for the instance that you want to find. Then click on the “Ok” button and the Explorer Window will show only the instances you selected. For more information on regular expressions, see “Name Filter Menu Selection” on page 3-58.

## Explorer Context Menus

### 3.14.1.9 Properties

The “Explorer/Tree/Properties” menu selection is located in the context menu in the left hand “Tree” side of the Explorer Window. The “Properties” menu selection opens the “Module Properties” dialog box. The Module Properties dialog box enables you to specify which module instances you want to save the simulation data for during simulation. See “Keeping Module Instance Simulation Variable Values” on page 5-29 for an additional method of saving instances in the hierarchy.



If the “Save simulation data for this entry” is selected, then Silos saves the simulation data for the following items during simulation:

- all local variables for the module instance;
- all port variables for the module instance (even if the ports connect to module instances that are not saved below it in the hierarchy);
- any instance below it in the hierarchy of the design, unless the instance below it is specifically not saved.

## Explorer Context Menus

### (Properties)

If the “Save simulation data for this entry” is not selected, then Silos does not save the simulation data for following items during simulation:

- all local variables for the module instance not selected;
- the simulation data for any instance below it in the hierarchy of the design.

Before the “Save simulation data for this entry” specifications can take effect, the Project must be reloaded by using the Project/Open menu selection or by selecting a project name from the Most Recently Used list of projects at the end of the Project menu.

The “Module Properties” dialog box also has a check box for “Save code coverage data for this entry”. When this is checked, code coverage information is saved for this instance, and the Silos Explorer has a “CC” to the left of the instance name. If this box unchecked, code coverage information is not saved for this instance, and the Silos Explorer has a “CC” covered by a red stop symbol to the left of the instance name. The “Save code coverage data for this entry” will be grayed out if the module instance is outside of the device under test (“\$fs\_dut” system task), or if code coverage is not enabled.

To invoke this menu selection, use the right mouse button to click on any part of the left hand side of the Explorer Window to open the context menu. The context menu will remain open while the mouse is used to select the Properties menu item.

# Watch Context Menus

## 3.14.2 Watch Window

### 3.14.2.1 Add Signal/Expression Menu Selection

The “Watch/Add Signal/Expression” menu selection opens the “Specify Signal/Expression” dialog box. To invoke this menu selection, use the right mouse button to click on any part of the Watch Window to open the context menu. The context menu will remain open while the mouse is used to select the Add Signal/Expression menu item.

The “Specify Signal/Expression” dialog box contains an edit box “Scope” to specify the scope for the signal. The “Specify Signal/Expression” dialog box also contains an edit box “Signal/Expression” to enter a signal or an expression. Any valid Verilog HDL expression can be entered. If the expression or scope is not valid then the expression will list an “Error”.

The OK button closes the dialog box and displays the specified expression in the Watch Window. The Cancel button closes the dialog box and does not display the expression.

### 3.14.2.2 Set Value For Watch Window

The “Watch/Set Value” context menu selection can be used to force or set a vector in the Watch Window. To invoke this menu selection, use the right mouse button to click on any part of the Watch Window to open the context menu. The context menu will remain open while the mouse is used to select the Set Value menu item.

### 3.14.2.3 Free Forced Wire For Watch Window

The “Watch/Free Forced Wire” context menu selection frees a wire that has been forced in the Watch Window. To invoke this menu selection, use the right mouse button to click on any part of the Watch Window to open the context menu. The context menu will remain open while the mouse is used to select the Free Forced Wire menu item.

(continued on next page)

## Watch Context Menus

### 3.14.2.4 Clear All For Watch Window

The “Watch/Clear All” context menu selection clears all of the expressions from the Watch Window. To invoke this menu selection, use the right mouse button to click on any part of the Watch Window to open the context menu. The context menu will remain open while the mouse is used to select the Clear All menu item.

# Analyzer Context Menus

## 3.14.3 Data Analyzer Context menus

### 3.14.3.1 Data Analyzer Timeline Area

#### 3.14.3.1.1 Goto Timepoint menu selection

The “Analyzer/Timepoint/Goto Timepoint” menu selection opens the “Goto Timepoint” dialog box for specifying the time point for the left axis or the center of the Waveform Display Window. The “Goto Timepoint” dialog box enables you to precisely position the Waveform Display Window for debugging and printing. The time value for the “Time Point” box can be specified in any standard time unit, such as “ns” for nano seconds, “ps” for pico seconds, etc., i.e.:

12000.3ns

To invoke the context menu use the right mouse button to click on any part of the timeline display area (the gray area just above the Waveform Display Window). The context menu will remain open while the mouse is used to select the Goto Timepoint menu item.

#### 3.14.3.1.2 Pan to T1, Pan to T2, and Pan to Last View menu selection

The “Analyzer/Timepoint/Pan to T1” menu selection and the “Pan to T2” menu selection will center the Waveform Display Window around the T1 or T2 timing marker. The “Pan to Last View” will return the Waveform Display Window to the preceding view. These menus are useful during debugging for jumping between views of the simulation results.

To invoke the context menu use the right mouse button to click on any part of the timeline display area (the gray area just above the Waveform Display Window). The context menu will remain open while the mouse is used to select the Pan to T1, Pan to T2, or Pan to Last View menu item.

## Analyzer Context Menus

### 3.14.3.1.3 Timescale menu selection

The “Analyzer/Timepoint>Select Timescale” menu opens the “Time Scale” dialog box for setting the number of time units per division of display. Setting the timescale is useful for debugging the design.

The current time scale, T1 time value, T2 time value, and delta time value can be displayed by holding the mouse cursor over the timeline for a few seconds. To modify the time scale, open the “Time Scale” dialog box and enter a value in the “Time/Div” box. You can use any standard time unit, such as “ns” for nano seconds, “ps” for pico seconds, etc. i.e.:

12000.3ns

The “OK” button closes the dialog box and causes the Data Analyzer to use the selected time scale. The “Cancel” button closes the dialog box and does not affect the Data Analyzer.

To invoke the context menu use the right mouse button to click on any part of the time point display area (the gray area just above the Waveform Display Window). The context menu will remain open while the mouse is used to select the Select Timescale menu item.

### 3.14.3.1.4 Snap to Edges Menu Selection

When setting the T1 and T2 timing markers for the Data Analyzer, they will **snap to the nearest edge** if the “Analyzer/Timepoint/Snap to Edges” menu selection is active.

When setting a timing marker, you can hold down the **shift key** to temporarily toggle the “Snap to Edges” selection to its opposite effect. Such as, if the “Snap to Edges” is not selected, you can have the timing marker snap to the nearest edge when you set it by holding down the shift key as you click-on the left or right mouse button with the mouse indicator (arrow) in the Waveform Display Window.

To invoke the context menu use the right mouse button to click on any part of the time point display area (the gray area just above the Waveform Display Window). The context menu will remain open while the mouse is used to select the Snap to Edges menu item.

## Analyzer Context Menus

### 3.14.3.1.5 Add Bookmark menu selection

The “Analyzer/Timepoint/Add Bookmark” menu selection enables you to place a virtual marker for the time and the timescale resolution of the center of the Waveform Display Window. When debugging your design, the bookmakers you set enable you to jump back and forth between waveform views with the same or different timescale.

After opening the “Add Bookmark” dialog box you will see the default bookmarks, i.e. “Bookmark1”, “Bookmark2”, etc. You can specify any string of characters for the bookmark name and then click-on OK to set the bookmark. The bookmarks you have set are listed at the bottom of the context menu. To go to a bookmarker select it with the left mouse button.

To invoke the context menu use the right mouse button to click on any part of the time point display area (the gray area just above the Waveform Display Window). The context menu will remain open while the mouse is used to select the Add Bookmark menu item.

### 3.14.3.1.6 Delete Bookmark menu selection

The “Analyzer/Timepoint/Delete Bookmark” menu selection enables you to delete a bookmarker.

After opening the “Delete Bookmark” dialog box you will see the bookmakers, i.e. “Bookmark1”, “Bookmark2”, listed in the “Bookmarks” list box. You can delete a bookmark by selecting it with the mouse can clicking-on the “Delete” button. Click on the OK button to close the dialog box.

To invoke the context menu use the right mouse button to click on any part of the time point display area (the gray area just above the Waveform Display Window). The context menu will remain open while the mouse is used to select the Delete Bookmark menu item.

## Trace Signal Inputs

### 3.14.3.2 Data Analyzer Signal List Box

#### 3.14.3.2.1 Trace Signal Inputs Menu Selection

The “Analyzer/Name box/Trace Signal Inputs” menu selection opens a Trace Signal Inputs Window. The Trace Signal Inputs Window allows you to interactively trace an incorrect value at a net to its cause by displaying the waveforms of all the devices that are driving the net.

To invoke the context menu, use the right mouse button to click on any part of the Name list box. The context menu will remain open while the mouse is used to select the Trace Signal Inputs menu item.

When you wish to trace a net, highlight a signal name in the Name list box of the Data Analyzer. With the mouse cursor still in the Name list box, click on the right mouse button to open the context menu. Next, select the “Trace Signal Inputs” menu selection in the context menu. A Trace Signal Inputs Window will then be opened and it will display the signals that are driving the net you selected.

To continue the fan-in tracing, double-click on any input and the waveform for that node will be displayed along with the waveforms for the devices driving it. To “undo” your signal tracing, double-click on the node name that is being traced.

When the waveforms for the Trace Signal Inputs Window are displayed, the name of the node being traced is listed first in the Name box. Blank lines delineate each device listed in the Name box. The format for each device is similar to the format for a module instance when passing ports by name in Verilog HDL. For example, the name:

```
not top.bit4.dff2.n6 (
    .out (top.bit4.dff2.\q_<specify> )
    .in (top.bit4.dff2.\qbar_<specify> ))
```

the device is a “**not**” gate and the instance name is “**top.bit4.dff2.n6**”. The output port name “**.out**” has the instance name “**top.bit4.dff2.\q\_<specify>**“. The input port name “**.in**” has the instance name “**top.bit4.dff2.\qbar\_<specify>**“.

You can continue to double-click on device inputs and trace your way back through the topology. To “undo” your signal tracing, double-click on the signal name that was traced. If you see “No Saved Data” instead of a waveform, this may mean that the signal is inside of a ‘celldesign’ boundary. To save data within ‘celldesign’ boundaries, see the “Save ‘celldesign’ data” option in the “Project Settings Menu Selection” on page 3-15.

## Display Iteration

### 3.14.3.2.2 Display Iteration Data Menu Selection

The “Analyzer/Name box/Display Iteration Data” menu selection is useful for finding order of evaluation problems and race conditions.

To invoke the context menu, use the right mouse button to click on any part of the Name list box. The context menu will remain open while the mouse is used to select the Display Iteration Data menu item.

When the “Display Iteration Data” menu selection is selected it opens an Iteration Data Window that displays all of the iterations at a single timepoint for each signal in the Name list box. The timepoint to display the iterations is specified by the T1 timing marker.

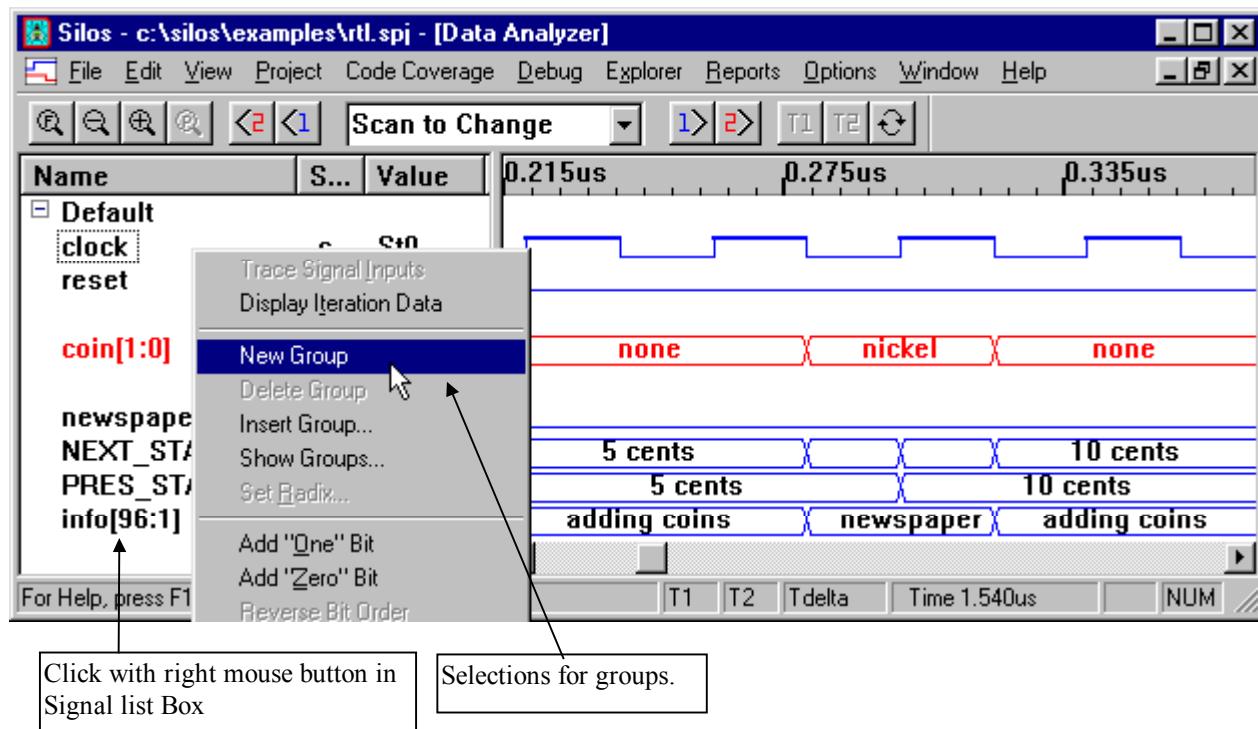
To display a text report for iterations at a timepoint, see “Iteration Menu Selection” on page 3-31.

## Groups

### 3.14.3.2.3 Groups of Signals

Using groups provides a convenient method for organizing your signals in the Data Analyzer Window. Not only does this help organize the display of your signals, it also prevents you from losing your list of signals if the default group gets inadvertently changed or lost. Display groups are also useful for assisting engineers who are unfamiliar with the design, and for record keeping if the design is reused.

The context menu for the Name list box provides the following selections for groups:



(continued on next page)

# Groups

## (Groups of Signals)

- New Group: This selection can be used to add a new group to the Name list box.
- Delete Group: This selection can be used to delete a group.
- Insert Group: This selection opens the “Add Group” dialog box. This dialog box will insert a group within a group. The inserted group is displayed as a bus, which can be expanded and hidden by double-clicking on it.
- Show Groups: This selection opens the “Select Signal Groups” dialog box. This dialog box can be used to select which groups are displayed in the Data Analyzer.

To invoke the context menu, use the right mouse button to click on any part of the Name list box. The context menu will remain open while the mouse is used to select the group menu item of interest.

When the Data Analyzer Window is opened, the “Default” group is displayed. To save any signals that are added to the Default group, click on the minus sign “-” just to the left of the Default group in the Name list box. Silos will ask you if you want to save the changes to the Default group.

### 3.14.3.2.4 Set Radix Menu Selection

The “Analyzer/Name box/Set Radix” menu selection is located in the context menu for the Name list box. Setting the radix for a vector can assist with debugging the design. The radix can be set to binary, octal or hexadecimal to conveniently display the vector. Symbolic names can be used to represent the values for a state machine. ASCII vectors can be displayed to create a timeline of events for the Data Analyzer display.

To invoke the context menu, use the right mouse button to click on any part of the Name list box. The context menu will remain open while the mouse is used to select the Set Radix menu item.

To set the radix for a vector:

- Click on the vector with the right mouse button to open the context menu.
- Next, select the “Set Radix” menu selection in the context menu.
- The “Set Radix” dialog box will then be opened and you can select the Radix.
- If you select the “Symbol Table” radix, then select the correct symbol table in the “Symbol Table” box.

The OK button closes the dialog box and sets the vector to the selected radix. The Cancel button closes the dialog box and does not affect the vector’s radix.

## Bit Menus

### **3.14.3.2.5 “Bit” Menu Selections**

The context menu for the Name list box for the Data Analyzer has the following selections:

- Add "One" Bit - adds a single bit signal at a high level just ahead of the selected signal name.
- Add "Zero" Bit - adds a single bit signal at a low level just ahead of the selected signal name.
- Reverse Bit Order - reverses the signal order for the highlighted signals in the Name list box. This is useful for reversing the bit order for a group.

To invoke the context menu, use the right mouse button to click on any part of the Name list box. The context menu will remain open while the mouse is used to select the Set Radix menu item.

### **3.14.3.2.6 Add Signal Menu Selection**

The “Data Analyzer/Name box/Add Signal” menu selection opens the “Specify Signal/Expression” dialog box.

Adding a signal can be very useful for performing conditional searches. The added waveform can be any expression whether the expression exists in your HDL source code or not. For a conditional search, you can then use the scan to change feature and the States List Box to review the signal values for the conditional search.

The Add Signal feature can also be useful for adding expressions that exist in your source code, such as for an “if” test, and then viewing then the expression is true (high).

The “Specify Signal/Expression” dialog box contains an edit box “Scope” to specify the scope for the signal. The “Specify Signal/Expression” dialog box also contains a “Signal or (Expression)” edit box to enter a signal or an expression. Any valid Verilog HDL expression can be entered, however, the expression must be enclosed by parentheses. You can copy and paste an expression that is in your source code window by highlighting the expression and using the “Ctrl c” keys on the keyboard to copy and the “Ctrl v” keys to paste the expression into the Signal or (Expression) list box. If the expression or scope is not valid then the waveform will be blank.

The OK button closes the dialog box and displays the specified expression in the Data Analyzer. The Cancel button closes the dialog box and does not display the expression.

If you single click on the expression that you added to the Data Analyzer, pause, then click on the expression again, you can modify the expression.

### **3.14.3.2.7 Add Blank Line Menu Selection**

The “Data Analyzer/Name box/Add Blank Line” menu selection inserts a blank line in the Data Analyzer just above the signal name that is highlighted.

## Clear Signal

### 3.14.3.2.8 Set Strength Color Coding Menu Selection

The “Data Analyzer/Name box/Set Strength Color Coding” menu selection uses color to denote waveform strength.

### 3.14.3.2.9 Set Trace Color Menu Selection

The “Data Analyzer/Name box/Set Trace Color” menu selection opens the “Select Trace Color” dialog box, which can be used to select a color for a waveform.

### 3.14.3.2.10 Delete Item/s Menu Selection

The “Data Analyzer/Name box/Delete Items” menu selection deletes all of the signal names that are selected by the user in the Name list box.

### 3.14.3.2.11 Clear Signal List Menu Selection

The “Data Analyzer/Name box/Clear Signal List” menu selection deletes all of the signal names in the Name list box.

### 3.14.3.2.12 Reload Groups Menu Selection

The "Data Analyzer/Name box/Reload Groups" menu selection will cause the Data Analyzer to clear the list box and reload the group information from the project file. This is useful when a user written program is used to modify the groups while the Silos is running.

## Source Window Context Menus

### 3.14.4 Source Window Context menus

#### 3.14.4.1 Undo menu selection

The “Source Window/Undo” menu selection undoes your last editing or formatting action, including cut and paste actions. If an action cannot be undone, Undo appears dimmed on the Edit menu.

#### 3.14.4.2 Cut menu selection

The “Source Window/Cut” menu selection deletes text from a document and places it onto the Clipboard, replacing the previous Clipboard contents.

#### 3.14.4.3 Copy menu selection

The “Source Window/Copy” menu selection copies text from a document onto the Clipboard, leaving the original intact and replacing the previous Clipboard contents.

#### 3.14.4.4 Paste menu selection

The “Source Window/Paste” menu selection pastes a copy of the Clipboard contents at the insertion point, or replaces selected text in a document.

(continued on next page)

## Source Window Context Menus

### (Source Window Context menus)

#### 3.14.4.5 Add/Remove Breakpoint menu selection

The “Source Window/Add or Remove Breakpoint” menu selection places or removes a simulation breakpoint at the location of the cursor in the source window.

#### 3.14.4.6 Data Tips menu selection

The “Source Window/Data Tips” menu selection toggles the Data Tips capability “on” or “off”. This feature enables the designer to trace the cause of problems directly in a Verilog HDL source code window. The Data Tips capability displays the value, scope, radix, and simulation time point for a variable or expression in the source window. Any variable or expression in a source window can be viewed by opening the source window and holding the mouse cursor over a variable or by highlighting an expression and holding the mouse cursor over the expression. If Silos does not know the scope for the variable or expression, the Data Tips message will request the user open the context menu in the source window by clicking on the variable or expression with the right mouse button, and select the “Go to Instance” menu to set the scope.

#### 3.14.4.7 Data Tip Radix menu selection

The “Source Window/Data Tip Radix” menu selection sets the radix for the Data Tips capability. The allowed radices are binary, octal, hexadecimal, decimal and string. This feature enables the designer to trace the cause of problems directly in a Verilog HDL source code window.

#### 3.14.4.8 Go to Instance menu selection

When viewing source code, you can select a line of source code and use the “Source Window/Go to Instance” selection in the context menu of the source window to open the Explorer at the instance for that line. This provides you with a hierarchical representation for the selected line of source code.

## Output Window Context Menus

### 3.14.5 Output Window Context menus

#### 3.14.5.1 Copy menu selection

The “Output Window/Copy” menu selection copies text from a document onto the Clipboard, leaving the original intact and replacing the previous Clipboard contents.

#### 3.14.5.2 Select All menu selection

The “Output Window>Select All” menu selection selects all the text in a document at once. You can copy the selected text onto the Clipboard, delete it, or perform other editing actions.

## 4. Verilog HDL Extensions

### 4.1 Silos PLI Interface

Silos dynamically interfaces the user written or vendor supplied Programming Language Interface (PLI) routines with the Silos executable at runtime. Interfacing the PLI routines at run time has the following advantages:

- The user does not have to create a new Silos executable or have a different executable for each set of vendor supplied PLI routines.
- Dynamically linking the PLI routines at runtime is faster than having to recompile and create a new executable each time.
- Upgrading to new versions of Silos is simple because there is no need to recompile and create a new executable.

#### 4.1.1 Silos PLI Interface on the PC

The PLI can be used with Silos on Windows NT version 4.0 and greater, and on Windows 2000 and Windows 98. Contact Simucad for an updated list of supported platforms. You may also need a "C" compiler, such as the MS Visual C++ compiler, to compile any user written PLI routines and to create a ".dll" file to link with Silos. To use PLI on the PC platform:

- Create one or more ".dll" files that contain the object code for the PLI. The object code must have been compiled for the type of platform you are using. For example, object code from the Sun will not work on the PC.
- The Silos "pliload" command is used to specify the ".dll" files for Silos at runtime. The "pliload" command is cumulative so that one or more "pliload" commands is allowed before starting simulation. The "pliload" command can be entered in the Command window for the Main toolbar, from the Silos command line, or from a file.

**pliload mypli.dll**

example for entering the pliload command in the Command window for the Main toolbar.

silos.exe myfile.v -“!pliload mypli.dll”

example for entering the pliload command at the command line.

```
module foo;
...
endmodule
`ifdef silos
!pliload mypli.dll
`endif
```

entering the pliload command from a file.

## (Silos PLI Interface on the PC)

- For more information, see the “README.TXT” file in the PLI subdirectory for the Silos installation.
- For an example of using PLI with Silos, see file “pli01.spj” in the PLI subdirectory for the Silos installation, or contact Simucad.

### 4.1.2 Silos PLI Interface on the Workstation

The PLI can be used with Silos on the Sun and HP workstations supported by Silos. Contact Simucad for an updated list of supported platforms. To use PLI on the workstation:

- Create one or more “.so” files that contain the object code for the PLI. For example, to create a “.so” file on Solaris, enter the following load command.

```
ld -o mylib.so -dy -G *.o
```

- The Silos “pliload” command is used to specify the “.so” files for Silos at runtime. The “pliload” command is cumulative so that one or more “pliload” commands is allowed before starting simulation. The “pliload” command can be entered at the in the Command window for the Main toolbar, from the Silos command line, or from a file.

**pliload mypli.so**

example of entering the pliload command at the in the Command window for the Main toolbar.

silos.exe myfile.v -“!pliload mypli.so”

example of entering the pliload command at the command line.

```
module foo;
...
endmodule
`ifdef silos
!pliload mypli.so
`endif
```

entering the pliload command from a file.

- For more information, see the “README.TXT” file in the PLI subdirectory for the Silos installation.
- For an example of using PLI with Silos, see file “pli01.spj” PLI subdirectory for the Silos installation, or contact Simucad.

(continued on next page)

**PLI**

#### **4.1.3 List of Implemented PLI Routines**

Silos uses the IEEE 1364 “Standard Hardware Description Language Based on the Verilog Hardware Description Language” manual as the specification for the PLI. Many of the "tf\_" PLI routines for linking user "C" programs to Silos have been implemented. The user "C" programs could be used for modeling a circuit or for creating test vectors. Selected "acc\_" PLI routines have also been implemented. Contact Simucad for the list of implemented PLI routines.

## 4.2 Standard Delay Format

Silos supports the Standard Delay File (SDF) format. SDF is a text file that contains the instance names and delay values necessary to back-annotate delays into a Verilog HDL description. SDF is usually generated by another tool such as a place and route tool.

The `$sdf_annotate` system task is used to specify the SDF file (do not input the SDF file). The format specification for the `$sdf_annotate` system task is:

```
$sdf_annotate("file_name", module_instance);
```

where:

`file_name` represents any valid file path and file name specification.

`module_instance` represents the name of the module instance. The hierarchy of this instance is used for back annotation. The names in the SDF file are relative paths to the `module_instance` or full paths with respect to the entire Verilog HDL description. For example, if you use the `module_instance` name “`top.dff1`” then the instance names in the SDF file are relative to “`top.dff1`”. If you omit `module_instance`, Silos uses the module containing the call to the `$sdf_annotate` system task as the `module_instance` for annotation.

When you run the simulation, you will see the following lines in the Output window for Silos that shows the SDF file is read in:

```
Beginning '$sdf_annotate("file")' into the hierarchy at 'testbench.design'  
Done $sdf_annotate.
```

Where “`file`” is the name of your SDF file, and “`testbench.design`” is the hierarchical name of your design. When errors or warnings occur during readin, there will be an additional message.

**Note:** If you have selected the “Functional simulation” check box in the “Project Settings” dialog box for Silos, then the back annotation of SDF delays is disabled and the SDF file is not readin to Silos.

For examples on using SDF, see the example [on the next page](#), and see project `fltsim.spj` (fault simulation example) in the examples subdirectory for the installation.

[\(continued on next page\)](#)

**SDF****(Standard Delay Format)**

For the below example and diagram, if the SDF file contained the following instance name:

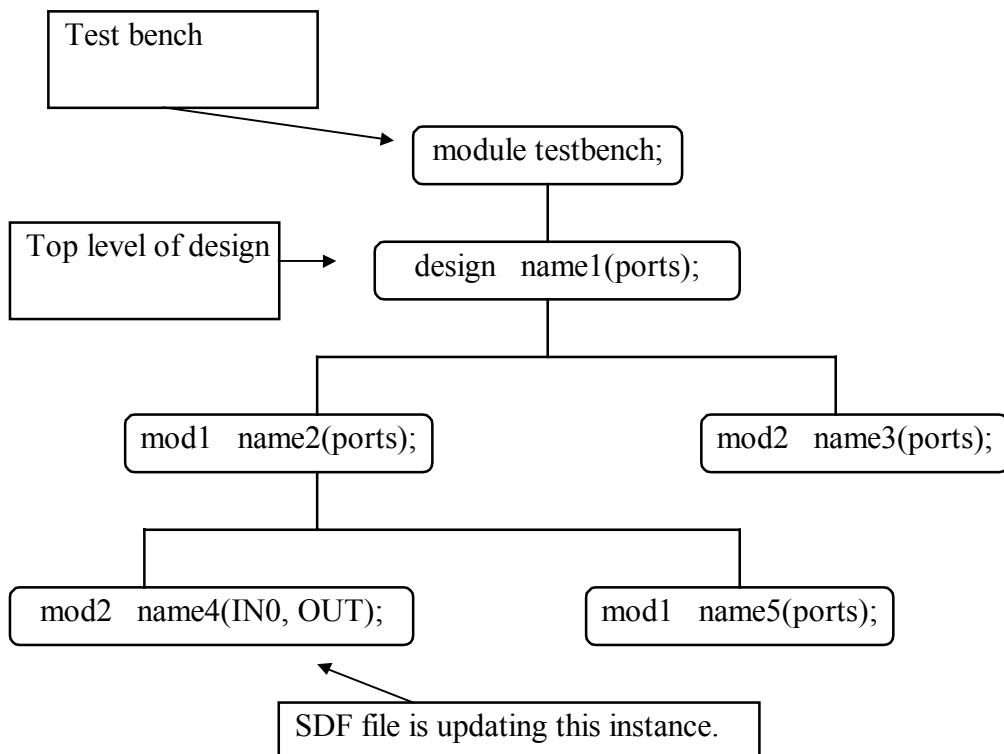
```
(INSTANCE name2.name4)
(DELAY
(ABSOLUTE
(IOPATH IN0 OUT (2420:2420:2420) (2420:2420:2420)))
```

then the \$sdf\_annotate system task to specify the SDF file and its relative position in the design's hierarchy would be:

```
module testbench;
initial $sdf_annotate("filename.sdf", testbench.name1);
...
endmodule
```

When Silos reads file "filename.sdf" to update the design, the path "testbench.name1" is concatenated with path "name2.name4" to form path "testbench.name1.name2.name4". Signals "IN0" and "OUT" are then updated as specified in the above example.

### Diagram of the Design Hierarchy for the SDF Example



# Stimulustable

## 4.3 Expected Values and Stimulustable

The IEEE Verilog specification does not describe any syntax for tabular representation of input data, nor expected value information. The stimulustable statement is a Silos enhancement which provides tabular format for input data. The stimulustable statement also can combine expected value information with the tabular format for input data.

### 4.3.1 BNF

```

stimulustable <id> ;
  table <# delay-expression>? <probe> <,<probe>>* ;
    <delay-constant>? <data>+ ;
    <delay-constant>? <data>+ ;
    .....
    endtable
endstimulustable

probe ::= <data-format>? <variable> <(variable)?@ <strobe>>?
  if <data-format> is omitted then %h is assumed.

data-format ::= %h
  ||= %o
  ||= %b

```

For replacing an existing stimulus table after prep:

```

stimulustable <id> ;
  table <# delay-expression>? ;
    <delay-constant>? <data>+ ;
    <delay-constant>? <data>+ ;
    .....
    endtable
endstimulustable

```

## Stimulustable

### 4.3.2 Stimulustable

“stimulustable” is a behavioral statement and can be located anywhere any statement can be placed, e.g.

```
for (i=1; i<=8; i = i+1) // repeat 8 times the input pattern  
  stimulustable  
    ...  
  endstimulustable
```

There is no limit to the number of stimulustable statements. They are not required to be located in top-level modules. The syntax for the stimulustable keywords must be lower case. Such as, the keyword "table" must be lower case. Variable names are upper/lower case sensitive.

Any number of input or expected value columns may appear in a stimulustable. Each input column is identified by a variable which is driven by the data in the column. Each expected value column is identified by an @ sign. The variable on the left side of an @ sign is verified against the data in the column. The variable on the right side of the @ sign is used as a strobe. Variables used in the table can be a wire, register, memory element, integer or real variable of any width and they can have any valid Verilog name.

(continued on next page)

## Stimulustable

### Stimulustable

In the example below for stimulustable **s1**, register variable **in1** and memory element **in2** are driven by the data in the table. Wire **out1** is verified against the data in the table whenever the variable **strobe1** is high. To prevent possible race conditions, the rising or the falling edge of the strobe signal “**strobe1**” should not coincide with a change in the expected output for “**out1**” in the stimulustable.

```

!control .ext=stim
`timescale 1ns/100ps
module test;
reg [7:0] in1, in2[1:0];
wire[7:0] out1 = ~in1 | in2[0];
reg strobe1;
initial strobe1 = 0;
always @out1 begin #0.1 strobe1 = 1; #0.1 strobe1 = 0; end
initial
begin
#5;
stimulustable s1;
table #1.2    in1,    in2[0],        out1@strobe1;
  00 00 ff;
  0e 0a f6;
  ff ff 00;
endtable
endstimulustable
end
endmodule

```

**Radix****4.3.3 Radix**

Data in the table can be in hexadecimal (%h is the default), octal (%o), or binary (%b) for register data-types, and integer or floating point for integer and real data types. There must be one or more blank spaces between the radix symbol and the variable name it refers to, such as %h in2. Each row of values in the table is terminated by a semicolon “;”. Blank spaces and tabs (not carriage returns) can be used to delineate the values between different variables, however, white space is not allowed between the values for a vector variable. An example for specifying the radix is shown below:

```
table #1.2  %b in1,  in2,  out@strobe;
  000000000000 ff;
  000011100a f6;
  11111111ff 00;
```

For single bit wires, SILOS state symbols may be used to enter states other than 1, 0, x, z. See “Symbol Modification For Output” on page 5-48.

**4.3.4 Delay Time**

The delay time for a constant increment of time (delta time) between application of subsequent table lines can be specified as a single expression:

```
table #delta .... ;
```

When the delta delay is specified on the table header, then the first table line is applied immediately upon execution of the stimulustable statement.

The delay time can also be specified on each table line. When no # sign is specified on the table header, then the delay values are added to the simulation time as the stimulustable is read. The delay is applied prior to the application of the table line. The time units for the delay value can be specified by preceding the module containing the stimulustable statement with a `timescale statement. Below is an example:

```
`timescale 1ns / 1ns
```

```
#10                                // time=10
table in1, in2, out@strobe;
1.2    00  00  ff;                // time=11.2
1.6    0e  0a  f6;                // time=12.8
2.1    ff  ff  00;                // time=14.9
```

(continued on next page)

## Delay Time

### (Delay Time)

If two ## signs are specified on the table header, then the delay values are relative to the time the stimulus table is started (very much like delay values in a fork/join statement). For example:

```
#10                                // time=10
table ##  in1, in2, out@strobe;
1.2  00  00  ff;                  // time=11.2
1.6  0e  0a  f6;                  // time=11.6
2.1  ff  ff  00;                  // time=12.1
```

To have each delay value represent absolute time, start the stimulustable at time=0. For example:

```
initial begin
  stimulustable s1;
table ##  in1, in2, out@strobe;
1.2  00  00  ff;                  // time=1.2
1.6  0e  0a  f6;                  // time=1.6
2.1  ff  ff  00;                  // time=2.1
```

Note that mixing of both delay styles in the same stimulus is not allowed.

# Memory

## 4.3.5 Memory Utilization

Data specified in tables is not stored in RAM, so as to reduce memory used when there is a large pattern.

## 4.3.6 Strobe

Expected value information is conditioned by a strobe. When the strobe is high, the variable must agree with the data in the column as follows:

1 <==> 1  
0 <==> 0  
x <==> don't care  
z <==> High impedance strength (0,1, or x)

The expected value check is engaged when the stimulustable statement begins execution, and persists through one strobe cycle following the conclusion of the stimulustable statement. During engagement of the expected value check, a high (positive) strobe is required to check that the variable agrees with the expected value data. To prevent possible race conditions, the rising or the falling edge of the strobe signal should not coincide with a change in the expected output signal in the stimulustable.

(continued on next page)

## Memory

For example, in stimulustable **s1** (shown below) variable **strobe1** strobes **out1** every 0.2 nano seconds. This is faster than the input values change (every 1.2 nano seconds) for variables **in1** and **in2**. When the second entry in the table is executed the calculated value for **out1** (f6) does not equal its expected value. The violation is recorded at the next high pulse for variable **strobe1** and then is not recorded again until after the next entry in the table occurs.

```

!control .ext=stim
`timescale 1ns/100ps
module test;
    reg [7:0] in1, in2[1:0];
    wire[7:0] out1 = ~in1 | in2[0];
    reg strobe1;
    initial strobe1 = 0;
    always @out1 begin #0.1 strobe1 = 1; #0.1 strobe1 = 0; end
    initial
        begin
            #5;
            stimulustable s1;
            table #1.2    in1,    in2[0],          out1@strobe1;
                00 00 ff;
                0e 0a f6;
                ff ff 00;
            endtable
        endstimulustable
    end
endmodule

```

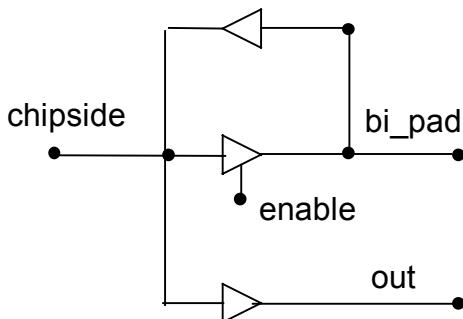
Using the disable statement to disable the block containing the stimulustable statement immediately terminates expected value checking.

Each expected value column has one strobe, however multiple columns may each have different strobes.

## I/O Pad

### 4.3.7 I/O Pad

The stimulustable can be used to model a bi-directional I/O pad. In the example below for stimulustable **s1**, variable **enable** controls the bi-directional I/O pin **bi\_pad**. When enable is high (1), pin **bi\_pad** acts as an output pin and expected value checking is performed every time strobe1 goes high. The stimulustable also ignores any values in the table for pin “**bi\_pad**” when “**enable**” is high. When enable is low (0), pin **bi\_pad** acts as an input pin and the stimulustable applies the values in the table for pin **bi\_pad** as input stimulus. Expected value checking is ignored for pin **bi\_pad** when enable is low.



```

!control .ext=stim
`timescale 1ns/100ps
module test;
    wire chipside, bi_pad, enable, out;
    buf(chipside, bi_pad);           // buf(out, in);
    bufif1(bi_pad, chipside, enable); // bufif1(out, in, enable);
    buf(out, chipside);
    reg strobe1;
    initial strobe1 = 0;
    always @bi_pad begin #8 strobe1 = 1; #1 strobe1 = 0; #1; end
    initial
        begin
            #5;
            stimulustable s1;
            table #10 chipside, enable, out@strobe1, bi_pad(enable)@strobe1;
                1 1 1 1; // output cycle
                0 1 0 0; // output cycle
                1 0 1 1; // input cycle
                0 0 0 0; // input cycle
            endtable
        endstimulustable
    end
endmodule

```

## Expected Value Error

### 4.3.8 Expected Value Error

Expected value errors trigger a global register named `ExpectedValueError`. This register is simultaneously set with specific information about the expected value violated.

The `ExpectedValueError` variable can be accessed either:

- from the data file to cause immediate interaction with the simulation, e.g.  
`always @ExpectedValueError $stop;`
- from the `$monitor` system task or the Silos probe command the variable prints a terse string indicating the stimulus name and column name violated, e.g.  
`$monitor($time,,ExpectedValueError);`

To obtain all violations at a single time-point:

```
probe iter ExpectedValueError
```

To obtain time points for which there is at least one violation:

```
probe ExpectedValueError
```

Note other variables may simultaneously be probed, e.g.

```
probe out,,ExpectedValueError
```

When the `ExpectedValueError` signal is displayed in the Data Analyzer, value for the `ExpectedValueError` signal is “none” when there is no violation, and “x” when there is a violation. The “Scan T1 Right” and the “Scan T1 Left” buttons on the Analyzer Toolbar can be used to scan to expected value violations. If the `ExpectedValueError` signal shows a series of contiguous “none” values, then the rising and falling edge of the strobe signal may be coincident with the changes for the expected value signal in the stimulustable. For example, the rising and falling edge of the strobe signal “strobe1” does not coincide with expected output signal “out1” in the next example.

(continued on next page)

## Expected Value Error

The below example shows how to use the `ExpectedValueError` variable with `$monitor`:

```

!con .ext=stim
//title example with $monitor
`timescale 1ns/100ps
module test;
  reg [7:0] in1, in2[1:0];
  wire[7:0] out1 = ~in1 | in2[0];
  reg strobe1;
  initial strobe1 = 0;
  always @out1 begin #0.1 strobe1 = 1; #0.1 strobe1 = 0; end
  initial
    begin
      $timeformat(-9,3,"ns",-15);
      $monitor("%t",$realtime,, ExpectedValueError);
      #5;
      stimulustable s1;
        table #1.2 in1,      in2[0], out1@strobe1;
          00 00 ff;
          0e 0a f6;
          ff ff 00;
        endtable
      endstimulustable
      #10 $finish;
    end
  endmodule

```

The output from the `$monitor` is shown below:

```

6.300ns :s1 out1 fb != f6:
6.400ns
7.500ns :s1 out1 ff != 00:
7.600ns

```

For the first line:

```
6.300ns :s1 out1 fb != f6:
```

The “6.300ns” is the time the difference occurred, the “s1” is the instance name for the `stimulustable`, the “fb” is the simulation value for variable “`out1`”, and the “f6” is the expected value for “`out1`”.

## Incremental Update

### 4.3.9 Expected Value Error Storage

The expected values are stored in variables that use the root name of the variable whose value is checked with an <expected><number> appended to the name. These variables can be accessed with the probe or print commands, or viewed in the Data Analyzer.

### 4.3.10 Incremental Update

“stimulustable” data can be incrementally replaced without having to re-input all the files in the design. This allows quick iteration of different stimulus/expected-value patterns. The incremental stimulustables can be specified at any time after preprocessing (the “!prep” command). When specifying the incremental stimulustables, each incremental stimulustable must be specified outside of any module, and the variable names can not be put on the table line. The name of the stimulustable is used to determine which stimulustable is updated.

The file below, “test.v”, shows the top level module with a stimulustable that is simulated until the “\$finish” is encountered:

File “test.v”:

```

!con .ext=stim
`timescale 1ns/100ps
module main;
  reg [8:0] r9;
  reg      r1;
  initial
    begin
      stimulustable s1;
      table #10 %b r9,      r1;
          000000000 1;
          000010000 0;
          111111111 x;
          100000001 0;
    endtable
    endstimulustable
    #10 $finish;
  end
endmodule

!sim
`include "test1.v"

```

(continued on next page)

## Incremental Update

The next file, “test1.v” shows the new stimulustable values. To simulate the new values, use ‘include to input file “test1.v”. Notice that the delta delay value was changed from “#10” for the first version of stimulustable “s1” to “#20” for the second version of stimulustable “s1”. The command “!sim 0 2100m” will restart the simulation at time=0 and simulate until the “\$finish” in file test.v:

File “test1.v”:

```
stimulustable s1;
  table #20;
    001100000 0;
    011010000 x;
    111111111 x;
    100010001 1;
  endtable
endstimulustable

!sim 0 2100m
```

## Behavioral to Tabular

### 4.3.11 Changing Behavioral Stimulus to a “stimulustable” Format

Using a stimulustable statement instead of behavioral stimulus has the following advantage:

- The stimulustable is input in chunks so it requires less memory.

To change behavioral stimulus to a stimulustable format, you can simulate the behavioral stimulus with Silos and then store the results from the “probe” command as a file of tabular ones and zeros. The file can then be edited to remove the title for the “probe” command report. Each line of tabular values in the file must end with a semicolon “;”. To add a semicolon at the end of each line, put a “;” at the end of the probe state, i.e.:

```
!store probe in1,,in2,,bi1_,,bi2_,";"
```

For example, for file “stimulus.v”:

```
'timescale 1ns / 100ps
module foo;
    reg in1, in2;
    reg bi1_, bi2_;
    wire bi1 = bi1_; // bi-directional inputs
    wire bi2 = bi2_; // bi-directional inputs
    initial
        begin
            in1=0; in2=0;
            bi1_=0; bi2_=0;
            #10 in1=1; bi1_=1;
            #10 bi2_=1'bz;
            #10 $finish;
        end
    endmodule
```

The following commands would be used from a file to simulate the stimulus:

```
'include "stimulus.v"
!control .savsim=2
!sim
!disk stim.v
!scope foo
!store probe in1,,in2,,bi1_,,bi2_,";"
```

(continued on next page)

## Behavioral to Tabular

Next, edit file “stim.v” and remove the report header for the probe report and any messages from the probe report.

Then, include file “stim.v” into a stimulustable statement:

```

!control .ext=stim
`timescale 1ns / 100ps
module foo;
    reg in1, in2;
    reg bi1_, bi2_;
    wire bi1 = bi1_; // bi-directional inputs
    wire bi2 = bi2_; // bi-directional inputs
    initial
        begin
            `timescale 100ps / 100ps // timescale for stimulustable
            stimulustable s1;
                table ## in1, in2, bi1_, bi2_;
                    `include "stim.v"
                endtable
            endstimulustable
            `timescale 1ns / 100ps // respecify the circuit's timescale
            #10 $finish;
        end
endmodule

```

When converting behavioral stimulus to tabular stimulus, you need to ensure that timescale for the tabular stimulus is correct. The units for the time values from the “probe” command are equal to the smallest resolution for the simulation. This may require a `timescale compiler directive before the stimulustable statement so that the delay values are scaled correctly. In the above example, the resolution of the “timescale 1ns/100ps” compiler directive for the circuit is “100ps”, so a “timescale 100ps/100ps” compiler directive must be used before the stimulustable statement.

Notice that the stimulustable for the above example also uses the “##” delay notation, so that the time values are relative to the time that the stimulustable statement is started.

Notice also that you may need to be careful when applying the stimulus for “inout” (bi-directional) pins in the circuit. The “inout” pins “bi1” and “bi2” are defined as the left hand side of continuous assignments. For this circuit, the stimulustable values should be applied to the registers “bi1\_” and “bi2\_”. Otherwise, registers “bi1\_” and “bi2\_” will remain at an Unknown level and will continue to drive wires “bi1” and “bi2” to an Unknown level due to the continuous assignments.

# Analog

## 4.4 Analog Extensions

Simucad has added extensions to the Verilog Hardware Description Language (HDL) that allow Silos to model analog circuits at the behavioral level.

### 4.4.1 Real and Integer Data Types

Silos supports real and integer data types as defined by the IEEE P1364 Standard Verilog HDL Language Reference Manual. To facilitate analog behavioral modeling, Silos also supports the following unique extension to the Verilog language:

- Real (floating point) and integer variables can be passed between module ports.

The advantages of directly passing real and integer variables between modules are:

- ease of programming style;
- no loss of information (as occurs with other Verilog simulators).

Passing numerical values between behavioral modules is particularly useful when modeling analog behavior for circuits such as analog to digital converters, phase lock loops, charge pumps, etc. For an example of an analog to digital converter, see file “analog.v” in the “examples” subdirectory of the installation directory.

#### **4.4.2 Utility Transcendental Functions**

To simplify the implementation of analog models, Silos supports a full range of transcendental math functions. The following functions accept a single floating-point argument x and return a floating-point value (except for pow, which has two floating point arguments x and y):

<u>Function Name</u>	<u>Description</u>
sin(x)	sine
cos(x)	cosine
tan(x)	tangent
asin(x)	inverse sine
acos(x)	inverse cosine
atan(x)	inverse tangent
sinh(x)	hyperbolic sine
cosh(x)	hyperbolic cosine
tanh(x)	hyperbolic tangent
sqrt(x)	square root
exp(x)	exponential
log10(x)	common logarithm
log(x)	natural logarithm
pow(x,y)	power xy

**Analog****4.4.3 Examples for Transcendental Math Functions**

The transcendental functions are used in the same way as any other Verilog function. The module below illustrates a simple use of displaying values for the math functions:

```
module math03;
initial
begin
real pi;
pi = 3.14159;
$display ("sin(0.0) = 0:", sin(0.0));
$display ("sin(0.5 * pi - 0.01) = 0.99995:", sin(0.5 * pi - 0.01));
$display ("cos(0.00234) = 0.999997:", cos(0.00234));
end
endmodule
```

The next example shows how to generate a sine wave using the “sin” function:

```
//title example for generating a sine wave
// The example below generates a sine wave "y" based on the value of "x".
module sine_wave;
real x, y;
initial
begin
x = 0;
#1000 $finish;
end
always
begin
#1 x = x + 0.1;
y = sin(x); // Built-in Silos "sin" function
end
endmodule
```

## Silos Extensions

### 4.5 “silos” and “sse” keywords

Silos has a reserved keyword “silos” that is always true. The “silos” keyword allows the user to enclose Silos specific code or commands within a `ifdef / `else / `endif compiler directive so that it will be available for Silos but not other Verilog simulators, e.g.:

```
`ifdef silos
    initial $stopsave();
    initial #1000000 $resetstartsave();
`endif
```

When running Silos, the reserved keyword “sse” is always true so that the user can enclose code or commands that is specific to the GUI within a `ifdef / `else / `endif compiler directive.

### 4.6 Extensions to Turn-off, Reset, and Turn-on Saving

When running a simulation that creates a large save file, the \$stopsave system task can be used to turn off saving to the save file. This can be used to keep the save file size fixed during the portion of the simulation that is of no interest for the designer.

The \$resetstartsave system task can be used to reset the save file and then start saving the simulation history. After the simulation is complete, the simulation history that has been saved after resetting the save file will be available for display with the Data Analyzer.

The below example stops saving at time=0, and starts saving at time=1000000:

```
`ifdef silos
    initial $stopsave();
    initial #1000000 $resetstartsave();
`endif
```

# Silos Extensions

## 4.7 Silos Extensions to Verilog HDL

Silos has a switch to issue syntax errors for extensions to the IEEE P1364 Standard Verilog HDL Language Reference Manual. The default setting for the switch is to check for IEEE compliance. To allow all extensions, enter “**!control .ext=all**” before inputting your model. The parameters to allow individual extensions are reported in the syntax error for each extension. **On the pages that follow** is a sample list of extensions that will be flagged as syntax errors:

### 4.7.1 Global Variables:

example:

```
wire xx;
module ... endmodule
```

Silos command to allow this extension:

**!control .ext=gvar**

(For more information, see section A.1 of the Verilog HDL Reference on-line help file)

### 4.7.2 Global tasks and functions:

example:

```
function
...
endfunction
module
...
endmodule
```

Silos command to allow this extension:

**!control .ext=gft**

(For more information, see section A.1 of the Verilog HDL Reference on-line help file)

## Silos Extensions

### (Silos Extensions to Verilog HDL)

#### 4.7.3 Functions with multiple outputs:

example:

```
function xx(in, out2);
    input in;
    output out2;
```

Silos command to allow this extension:

**!control .ext=fmout**

(For more information, see section 9.3.1 of the Verilog HDL Reference on-line help file)

#### 4.7.4 Functions without any inputs:

example:

```
x = funct();
```

Silos command to allow this extension:

**!control .ext=fzero**

(For more information, see section 9.3.4 of the Verilog HDL Reference on-line help file)

#### 4.7.5 Tasks and functions with ports declared like a module:

example:

```
task foo(in1,in2);
```

Silos command to allow this extension:

**!control .ext=formals**

(For more information, see sections 9.2.1 and 9.3.1 of the Verilog HDL Reference on-line help file)

## Silos Extensions

### 4.7.6 Procedural assignment to wires:

example:

```
module foo;
  wire w;
  initial w = 1;
```

Silos command to allow this extension:

**!control .ext=paw**

(For more information, see section 8.2 of the Verilog HDL Reference on-line help file)

### 4.7.7 Continuous assignments to register and memory variables:

example:

```
reg r;
assign r = in;
```

Silos command to allow this extension:

**!control .ext=aar**

(For more information, see sections 5.1 and 11.1 of the Verilog HDL Reference on-line help file)

### 4.7.8 Continuous assignments using intra-assignment/non-blocking delays:

example:

```
module foo;
  wire o, o1;
  assign o = #4 i;
  assign o1 <= #4 i;
```

Silos command to allow this extension:

**!control .ext=assign**

(For more information, see section 5.1 of the Verilog HDL Reference on-line help file)

## Silos Extensions

### (Silos Extensions to Verilog HDL)

#### 4.7.9 Default state value for UDP:

The "default" keyword for the UDP specifies the state value for the UDP's output when UDP input levels and transitions do not match any of the entries in the UDP table. When the "default" keyword is not used, the UDP default output state is "x".

example:

```
primitive udp1 (out, in);
    output out;
    input in;
    table
        // in  out
        0 : 1;
    default: 0;
    endtable
endprimitive
```

Silos command to allow this extension:

**!control .ext=udpdefault**

(For more information, see section 7.1 of the Verilog HDL Reference on-line help file)

#### 4.7.10 UDP additional states for High-Z on inputs or output:

example:

for row states other than "0,x,1", such as:  
 Z : 1;  
 <?HV> : 1;

Silos command to allow this extension:

**!control .ext=udpstate**

(For more information, see section 7.1 of the Verilog HDL Reference on-line help file)

## Silos Extensions

### 4.7.11 UDP edge for High-Z:

example:

for edges to High-Z, such as  
`(0Z) : 1;`

Silos command to allow this extension:

**`!control .ext=udpstate`**

(For more information, see section 7.1 of the Verilog HDL Reference on-line help file)

### 4.7.12 UDP Multiple Edges in a Row:

example:

`(01) (01): 1;`

Silos command to allow this extension:

**`!control .ext=udpstate`**

(For more information, see section 7.5 of the Verilog HDL Reference on-line help file)

### 4.7.13 Non-Constant Specify Block Delays:

example:

for non-constant specify block delay, such as  
`(in => out) = delay_var;`

Silos command to allow this extension:

**`!control .ext=ncsd`**

(For more information, see section 13.1 of the Verilog HDL Reference on-line help file)

### 4.7.14 Parameter for Specify Block Delays:

example:

for parameter used for specify block delay, such as  
`parameter dly=8  
 (in => out) = dly;`

Silos command to allow this extension:

**`!control .ext=psd`**

(For more information, see section 13.1 of the Verilog HDL Reference on-line help file)

## Silos Extensions

### (Silos Extensions to Verilog HDL)

#### 4.7.15 Stimulustable Extension:

example:

for using the stimulustable statement, such as  
stimulustable ... endstimulustable statement

Silos command to allow this extension:

**!control .ext=stim**

#### 4.7.16 “input/output/inout” declarations after the variable's declaration:

example:

```
module foo (in);
  wire in;
  input in;
```

Silos command to allow this extension:

**!control .ext=inout**

(For more information, see section 12.1 of the Verilog HDL Reference on-line help file)

#### 4.7.17 Using registers as module inputs:

example:

```
module xx(in);
  input in;
  reg in;"
```

Silos command to allow this extension:

**!control .ext=rsink**

(For more information, see section 12.4.6 of the Verilog HDL Reference on-line help file)

## Silos Extensions

### (Silos Extensions to Verilog HDL)

#### 4.7.18 Duplicate variable definitions:

example:

```
module foo;
  wire a;
  wire a;
```

Silos command to allow this extension:

**!control .ext=dvd**

(For more information, see section 3.1 of the Verilog HDL Reference on-line help file)

#### 4.7.19 Parameter used for sizing numbers:

example:

```
module foo;
  reg[7:0] xx;
  parameter size=8;
  initial
    xx = size'b010;
```

Silos command to allow this extension:

**!control .ext=psize**

(For more information, see section 2.3 of the Verilog HDL Reference on-line help file)

#### 4.7.20 Null statements:

example:

```
module foo;
  initial
  begin ;
```

Silos command to allow this extension:

**!control .ext=nstmtt**

(For more information, see sections 8.7.1 of the Verilog HDL Reference on-line help file)

## Silos Extensions

### (Silos Extensions to Verilog HDL)

#### 4.7.21 Timing checks without edge specifications for selected variables:

example:

```
$recovery( CLR, ...
```

Silos command to allow this extension:

```
!control .ext=neref
```

(For more information, see section B.9.6 of the Verilog HDL Reference on-line help file)

#### 4.7.22 More precision in “\$timeformat” than “`timescale”:

Silos command to allow this extension:

```
!control .ext=tfmt
```

(For more information, see section B.5.2 of the Verilog HDL Reference on-line help file)

#### 4.7.23 Missing port connections for VCS compatibility:

Missing port connections are set to ground for VCS compatibility.

Silos command to allow this extension:

```
!control .skip=.gnd
```

Note: Wires which are otherwise floating still remain HiZ, regardless of "!control .skip".

#### 4.7.24 VCS compatibility extension:

VCS compatibility extension for comma at the end of the port list, i.e.: module (xx(a,):

Silos command to allow this extension:

```
!control .ext=portcomma
```

## 5. Silos Commands

### 5.1 Commands Overview

The Commands section contains a short overview on command syntax, inputting commands from the in the Command window for the Main toolbar and inputting commands from a data file.

#### 5.1.1 Command Syntax

Usually, only the first two characters are required when specifying a command. A few commands (e.g. FAN, PRE, PRO) require three letters to prevent ambiguity.

#### 5.1.2 Inputting SILOS Commands

Most commands are a part of the menu structure. However, a Command window has been provided in the Main Toolbar for Silos to enter any command. On Unix, a command line executable, “silos”, is also provided that run Silos from the “Ready” prompt.

#### 5.1.3 Stopping Processes

To discontinue or stop an interactive process when running Silos, such as during a large report that is output to the terminal, enter:

- “Ctrl-C”: simultaneously hold down the “Ctrl” key and the “C” key on the keyboard for the Unix command-line version of HyperFault.
- “Esc” key on the keyboard for all Windows versions.
- “STOP” button on Toolbar for all Windows versions.

**ACTIVITY****5.2 Activity Report For Nodes**

The ACTIVITY command pre-grades the test vectors for fault simulation by reporting nodes that have no activity (level transitions) during a logic-simulation for the test vectors. The logic simulation is much faster to run than fault simulation.

The ACTIVITY command can also be used as an HDL code coverage report. This section of the Activity report lists the number of times that each line of HDL code was executed as specified by the MXTRAN and MNTRAN keywords.

To obtain a node activity report, enter:

**TYPE**

**STORE**      **ACTIVITY** [t1 TO t2] [ / keywrd=val, keywrd..]

**TYPE**            optionally directs the activity report to standard output or to a disk file.

**STORE**

...

**ACTIVITY**       generates a node activity report.

**t1 TO t2**       represent the minimum and maximum time point values for reporting node activity. This time point range must be within the logic simulation time point range. If the time points are not specified, the logic simulation times are used. (The TO keyword is optional).

**keywrd**          represents an optional keyword used to define a condition or specify a value. The first keyword must be preceded by a slash. Allowed keywords are:

**BLOCK**            reports the activity only for nodes that are included within fault blocking.

*(continued on next page)*

**ACTIVITY****(Activity Report For Nodes)**

(keyword)

|                   |  |
|-------------------|--|
| <b>MNTRAN=val</b> | specifies the lower limit for reporting node activity. Only nodes which have known level transitions greater than or equal to this minimum limit will be reported. (Default: MNTRAN=0) |
| <b>MXTRAN=val</b> | specifies the upper limit for reporting node activity. Only nodes which have known level transitions less than or equal to this maximum limit will be printed. (Default: MXTRAN=0)     |
| <b>val</b>        | represents the user-specified numerical value for MNTRAN or MXTRAN.  |
| <b>NOHIST</b>     | suppresses output of the activity versus time histogram.   |
| <b>NOSUM</b>      | suppresses output of the activity summary.   |
| <b>NOTAB</b>      | suppresses output of the node activity table.  |

**Application Notes:**

- 1) An ACTIVITY report can be very useful for developing input test patterns to detect circuit faults for fault simulation. The number of level transitions at a node indicates an input test pattern's effectiveness. Faults at nodes which make no level transitions cannot be detected.

*(continued on next page)*

# ACTIVITY

## (Activity Reports For Nodes)

### (Application Notes:)

- 2) The “t/s ACTIVITY” command can be used to generate the following reports:
- An activity **table** that lists the node names and their number of transitions (changes in level). The only nodes listed are those whose transition count falls between the minimum and maximum number of reported transitions specified by the ACTIVITY report “MNTRAN” and “MXTRAN” values. Registers are not listed, unless they are used to create an wire that is internal to the Silos program, such as a register that is an input to a gate will create an internal wire for Silos that actually drives the gate.
- The activity table lists the transition count and the node-name for each node. The "Legend" for the "Transition Count" column shows that a value is listed when the node has known (definite) transitions from a high to low or low to high. The value is enclosed by parenthesis if the node had possible transitions. A "H" is listed if the node toggled between an unknown level and a high level. A "L" is listed if the node toggled between an unknown level and a low level. A "U" is listed if the node is always at an unknown level.
- An activity **summary** that lists totals for the number of nodes at each level of activity count.
  - An activity **histogram** that shows known and potential level transitions versus time.
- 3) A “known” (definite) transition is defined as a change from a Low to High level or High to Low level, even if it goes through an intermediate Unknown level. A “possible” transition is defined as a change from a High or Low level to the Unknown level or High-Z; or, from the Unknown level or High-Z back to a High or Low level.
- 4) In the activity histogram, the number of known transitions is displayed first for each bar of the histogram as a column of stars ("\*"), and then the number of possible transitions is shown above the known transitions on each bar as a column of carats (^"). For example, if the height of the column of stars for the known transitions on a bar started at zero ended at 100, then there were 100 known transitions for that bar. If the height of the column of carats for the number of possible transitions started at 100 and ended at 159, then there were 59 possible transitions for that bar.

*(continued on next page)*

## ACTIVITY

### (Activity Reports For Nodes)

#### Examples:

- 1) To store the Activity report to a file for nets that do not toggle:

```
disk foo.out // specify file "foo.out" for the activity report  
store activity // write the activity report to file "foo.out"
```

- 2) To store the Activity report to a file to report all of the nets that toggle:

```
disk test_toggle.out // specify file "test_toggle.out" for the activity report  
store activity / mxtran=99 // write the activity report to file "test_toggle.out"
```

*(continued on next page)*

# ACTIVITY

## (Activity Reports For Nodes)

### (Examples)

- 3) The following simple circuit will be used as an example for the activity report:

```

module foo;
  reg in, in_U, in1_0;
  wire out_L, out_H, out_U, out1_0;
  not (out_L,in);  buf (out_H,in);  buf (out_U,in_U);  buf (out1_0,in1_0);
  initial
    begin
      $display("\t\time \tin \tin_U \tin1_0 \tout_L \tout_H \tout_U \tout1_0");
      in = 1'bx; in_U = 1'b1;      in1_0 = 1'b1;
      #10 in = 1'b1;  in_U = 1'bx;  in1_0 = 1'bx;
      #10 in = 1'bx;  in_U = 1'b1;  in1_0 = 1'b0;
      #10 in1_0 = 1'bx;
      #10 in1_0 = 1'b1;
      #10 $finish;
    end
    always @(in or in_U or in1_0 or out_L or out_H or out_U or out1_0 )
      $display($time,"t",in , "t",in_U , "\t",in1_0 , "t",out_L ,
              "\t",out_H , "\t",out_U , "\t",out1_0);
  endmodule
`ifdef silos
  !mkeep (foo          // save only the part of the design that is faulted
  !control .sav=1     // save only what is specified to be saved ("!mkeep")
  !sim
  !activity           // report all signals that do not toggle
  !activity / mxtran=99 // report all signals
  !error
  !exit
`endif

```

(continued on next page)

**ACTIVITY****(Activity Reports For Nodes)****(Examples:)**

- 3) Below is the simulation output for the example:

**Simulation Ouput for example on Activity report**

```

!sim

Highest level modules (that have been auto-instantiated):
    (foo foo
6 total devices.
Linking ...

8 nets total: 8 saved and 0 monitored.
67 registers total: 67 saved.
Done.
time   in    in_U   in1_0   out_L   out_H   out_U   out1_0
      x      1       1       x       x       1       1
      1      x       x       x       x       1       1
      1      x       x       0       1       x       x
      20     x      1       0       0       1       x       x
      20     x      1       0       x       x       1       0
      30     x      1       x       x       x       1       0
      30     x      1       x       x       x       1       x
      40     x      1       1       x       x       1       x
      40     x      1       1       x       x       1       1

26 State changes on observable nets in 0.10 seconds.
260 Events/second.

Simulation stopped at the end of time 50.

```

(continued on next page)

**ACTIVITY****(Activity Reports For Nodes)****(Examples:)**

- 3) The below report is for the “!activity” command in the example:

| <u>Activity Table for nets that do not toggle</u> |    |                  |  |
|---|----|------------------|--|
| Min Transitions (MNTRAN)                          | 0  |                  | Nets were reported that did not toggle.  |
| Max Transitions (MXTRAN)                          | 0  |                  |  |
| (   | 2) | H (foo(\in<wc>   | Net “in<wc>” is a net created by HyperFault to model input “in” (a register). The “H” means that the first transition was from unknown to high. The “(2)” means net “in<wc>” had 2 possible transitions from unknown to high to unknown. |
| (   | 2) | U (foo(\in_U<wc> | The “U” means that the first transition was from high to unknown, or low to unknown. The “(2)” means net “in_U>” had 2 possible transitions.   |
| (   | 2) | H (foo(out_H     |  |
| (   | 2) | L (foo(out_L     | The “L” means that the first transition was from unknown to low. The “(2)” means net “out_L” had 2 possible transitions.   |
| (   | 2) | U (foo(out_U     |  |

(continued on next page)

# ACTIVITY

## (Activity Reports For Nodes)

### (Examples:)

- 3) The below report is for the “!activity/mxtran=99” command in the example:

| <u>Activity Table for nets that do toggle</u> |                  |  |
|---|------------------|--|
| TRANSITION<br>COUNT                           | NODE-NAME        |  |
| 2   | (foo(\in1_0<wc>  | ← Nets were reported that did toggle (mxtran=99).  |
| ( 2)  | H (foo(\in<wc>   | Net “in1_0<wc>” is a net created by HyperFault to model input “in1_0” (a register). The “2” means the net had two transitions between high and low levels. |
| ( 2)  | U (foo(\in_U<wc> |  |
|   | 2 (foo(out1_0    |  |
| ( 2)  | H (foo(out_H     |  |
| ( 2)  | L (foo(out_L     |  |
| ( 2)  | U (foo(out_U     |  |

**BUSCON****5.3 Bus Contention Report**

The BUSCON command reports the logic simulation time points at which more than one “tri”, “triand”, “trior”, “trireg”, “tri0”, or “tri1” net types, or an enabled unidirectional device (“bufif1”, “bufif0”, “notif1”, “notif0”, “nmos”, “pmos”, “rnmos”, and “rmos” devices) are simultaneously driving a node (a bus contention).

To obtain a bus contention report, enter:

```
TYPE
STORE    BUSCON [ t1 TO t2 ]
WTYPE
NSTORE
```

|               |  |
|---------------|--|
| <b>TYPE</b>   | (optional) directs the bus contention report to standard output, or to a disk file.  |
| <b>STORE</b>  |  |
| ...           |  |
| <b>BUSCON</b> | generates a summary table of any contentions that have occurred between two time points.   |
| t1 TO t2      | represent the minimum and maximum time point values over which contention is to be checked. These time values must be within the logic simulation time point range. If not specified, the simulation time points will be used. The keyword “TO” is optional. |

**Application Notes:**

- 1) Contentions are reported only for nodes where two or more enabled unidirectional devices, or “tri”, “triand”, “trior”, “trireg”, “tri0”, and “tri1” net types, form a wired connection (often used to form a bus). Bi-directional transistors, non-enabled gates and gates without enable lines are ignored by the BUSCON report.
- 2) For each contention, the BUSCON command reports the starting and ending time points, the starting and ending node states, and the names of the enabled unidirectional devices or “tri”, “triand”, “trior”, “trireg”, “tri0”, and “tri1” net types connected to the node.

(continued on next page)

## BUSCON

### (Bus Contention Report)

#### Examples:

```
TYPE BUSCON 2K 100K  
nst busco
```

**ccexport****5.4 Exporting Code Coverage Results**

The ccexport command can be used to export code coverage results to other programs, such as Microsoft Excel.

The format for the ccexport command is:

```
ccexport /option1 /option2 filename
```

|                 |  |
|-----------------|--|
| <b>ccexport</b> | specifies exporting of code coverage results for use with another program.   |
| <b>/option1</b> | option1 can be either “/line”, which specifies to export a line report, or, “/oper”, which specifies to export an operator report. The default is to specify a line report. Both options can not be specified for the same ccexport command. |
| <b>/option2</b> | option2 can be either “/comma”, which specifies a comma delimited report, or, “/tab”, which specifies a tab delimited report. The default is to specify a comma delimited report.  |
| <b>filename</b> | represents the file name the exported results are written to. The default is to write the exported results to standard output.   |

**Example:**

To specify the ccexport command from a file, enter:

```
!sim
!ccexport /line /tab test.out
```

To specify the ccexport command from the command line, enter:

```
-!sim -“!ccexport /line /tab test.out”
```

**ccmexclude****5.5 Exclude Code Coverage for Module Instances**

The ccmexclude command disables code coverage for the specified module instance and every module instance below it.

The format for the ccmexclude command is:

```
ccmexclude mname ... mname
```

**ccmexclude** specifies module instances that will be disabled for code coverage and any module instance that is hierarchically below the disabled module instance.

**mname** represents the name of a module instance in Verilog HDL syntax. The instance names on these commands must proceed from the top of the design downward.

**Application Notes:**

- 1) The ccmkeep command can be used to specify module instances for which code coverage results are saved.
- 2) The effects to the ccmkeep and ccmexclude commands are cumulative.
- 3) The equivalent of the ccmexclude command can be specified from the GUI using the Silos Explorer. To do this, select a module instance, right-mouse click and select to the Properties menu selection. In the "Module Properties" dialog box, check or uncheck "Save code coverage data for this entry". Note this information is written into the ".cfv" file to permit batch use at a later time.

**Example:**

To specify the ccmexclude command from a file, enter:

```
!ccmexclude m1.alu m1.cpu  
!sim
```

To specify the ccmexclude command from the command line, enter:

```
-“!ccmexclude m1.alu m1.cpu “ -!sim
```

**ccmkeep****5.6 Keeping Code Coverage for Module Instances**

The ccmkeep command enables code coverage for the specified module instance and every module instance below it.

The format for the ccmkeep command is:

```
ccmkeep mname mname ... mname
```

**ccmkeep** specifies module instances that will be enabled for code coverage and any module instance that is hierarchically below the enabled module instance.

**mname** represents the name of a module instance in Verilog HDL syntax. The instance names on these commands must proceed from the top of the design downward.

**Application Notes:**

- 1) The ccmexclude command can be used to specify module instances for which code coverage results are disabled.
- 2) The effects to the ccmkeep and ccmexclude commands are cumulative.
- 3) The equivalent of the ccmkeep command can be specified from the GUI using the Silos Explorer. To do this, select a module instance, right-mouse click and select to the Properties menu selection. In the "Module Properties" dialog box, check or uncheck "Save code coverage data for this entry". Note this information is written into the ".cfv" file to permit batch use at a later time.

**Example:**

To specify the ccmkeep command from a file, enter:

```
!ccmkeep m1.alu m1.cpu  
!sim
```

To specify the ccmkeep command from the command line, enter:

```
-“!ccmkeep m1.alu m1.cpu “ -!sim
```

**CHGLIB****5.7 Encrypting Library Files**

The CHGLIB command can be used to encrypt and secure library files.

**CHGLIB [ / ENCRYPT] [ / SECURE=feature] [ / NODEMOLIMIT] output\_file infile1 ...**

**CHGLIB** encrypts library files.

**ENCRYPT** the resulting library file is unreadable except by the Silos program. Readable comments can be added by editing the encrypted library file before the first “#” character. Use of this option is controlled by a security license feature issued by Simucad.

**SECURE=feature** the resulting library file is unreadable except by the Silos program. When Silos attempts to access the resulting library file, the user must have the “feature” listed in the license file “silos.lic” for Silos. The “SECURE” option does not require the “ENCRYPT” option to encrypt the file. The “SECURE” option can also be used with the “NODEMOLIMIT” option.

**NODEMOLIMIT** the resulting library file is unreadable except by the Silos program. This is a special option that allows the demo version of Silos to read a library file of more than 200 gates. Use of this option is controlled by a security license feature issued by Simucad.

**output\_file** name of the output library file to be created by the CHGLIB command.

**infile1 ...** name(s) of one or more input files to be encrypted.

**Examples:**

```
chglib cmos12.lib cmos12.dat cmos13.dat cmos14.dat
chglib /encrypt chip.library chip.dat
```

# CONTROL

## 5.8 Control Parameters For Logic Simulation

The “CONTROL” command enables you to modify the parameters that control logic simulation.

The general format of the CONTROL command is:

|                |                      |                      |                     |
|----------------|----------------------|----------------------|---------------------|
| <b>CONTROL</b> | <b>.COMMENT=c</b>    | <b>.CUSTREPORT</b>   | <b>.DISK=val</b>    |
| +              | <b>.DISABLECACHE</b> | <b>.DMIN</b>         | <b>.DMAX</b>        |
| +              | <b>.EUNK=val</b>     | <b>.MXITR=val</b>    | <b>.MXDCI=val</b>   |
| +              | <b>.NONCON</b>       | <b>.RECIRCULATE</b>  | <b>.SAVCELL=val</b> |
| +              | <b>.SAVSIM=val</b>   | <b>.SKIP=val</b>     | <b>.SYNONYM=val</b> |
| +              | <b>.TPS=qual</b>     | <b>.XL_ORDER=val</b> |                     |

- val           represents the numerical value assigned to the control parameter.
- c           represents a single character.
- string       represents the prompt string.
- CONTROL**   indicates that the default simulation control parameters are to be modified.
- .COMMENT**   specifies the comment character. (Default: **.COMMENT=\$**)
- .CUSTREPORT** specifies that the save-file will be used in a Custom Report. (Default: not specified)
- .DISABLECACHE** turns off the caching mechanism for the Data Analyzer. Turning off the caching may reduce the RAM memory used by Silos, however, it may make the Data Analyzer slower. The cache is used to “remember” in RAM memory the simulation data that you have viewed with the Data Analyzer. For example, if you do a zoom full, then every change for the displayed signals is stored in memory. If you then zoom in or zoom out for these signals, the redraw time is much faster. If you add additional signals to the Data Analyzer, then the simulation data for the new signals has to be read from the simulation history save file on disk.
- .DISK**       assigns the approximate limit of disk storage in bytes that the simulation save-file can use. When the disk storage limit is exceeded, the simulation will terminate (see note 1). (Default: **.DISK=100M**)

(continued on next page)

**CONTROL****(Control Parameters For Logic Simulation)**

- .DMAX** specifies that the maximum delay value will be used when parsing the netlist (see note 2).
- .DMIN** specifies that the minimum delay value will be used when parsing the netlist (see note 2).
- .EUNK** defines the conductance for bi-directional transistors and unidirectional transfer gates when there is an Unknown level on the enable: “on” when .EUNK=1, “off” when .EUNK=0 or “Unknown” when .EUNK=\*. (See note 3) (Default: .EUNK=\*)
- NONCON** when nonconvergence is detected, the default is for Silos to issue a warning message, pick a possible solution if this is possible, and continue simulation. If the “CONTROL .NONCON” command is entered before logic initialization begins, then if nonconvergence is detected, Silos will issue an error message and stop the logic simulation.
- .MXDCI** assigns the maximum allowed iterations for each pass during LINIT. (See note 4) (Maximum: .MXDCI=9999) (Default: .MXDCI=100)
- .MXITR** assigns the iteration limit to reach convergence for each logic simulation time point. (See note 4) (Maximum: .MXITR=999) (Default: .MXITR=300)
- .RECIRCULATE=val** this option specifies the maximum time interval for saving the simulation history results. When the simulation time exceeds this interval, the new simulation results are saved and the earliest simulation results are discarded so that the save file interval and file size on disk stay relatively constant, very much like a FIFO. The interval can be specified in time units, such as “100ns” for 100 nanoseconds, i.e: !control .recirculate=100ns.

*(continued on next page)*

**CONTROL****(Control Parameters For Logic Simulation)**

- .SAVCELL** Causes Silos to not save (“.SAVCELL=0”) or to save (“.SAVCELL=1”) the simulation history for variables listed between the ‘celldesign’ and ‘endcelldesign’ compiler directives. Caution: Saving all variables between ‘celldesign’ and ‘endcelldesign’ compiler directives may slow down simulation and create larger save files on disk. (Default: .SAVCELL=0)
- .SAVSIM** sets the logic simulation save option to determine which simulation node state changes are saved on the “SAVE” disk file. The .savsim option must be specified before simulation begins. (Default: .SAVSIM=0)
- .SAVSIM=0** specifies that no simulation node values are to be saved. This has limited use, as no data is available.
  - .SAVSIM=1** specifies that node simulation values (logic-type, integer-type and double-type) are to be saved only for nodes named in the TABLE, PLOT, GNAME, TESTER, KEEP, MKEEP, HEX and OCT commands. Output results can be obtained only for the saved nodes. This option decreases simulation disk file size and reduces execution time.
  - .SAVSIM=2** specifies that all logic-type simulation node states are to be saved for all nodes in the circuit. This option prevents saving integer and floating-point values.
  - .SAVSIM=3** specifies that all (logic-type, integer-type and double-type) simulation node values are to be saved. Output values can be retrieved for any network node.
- .SYNONYM** Causes Silos to retain the hierarchical node names (synonyms) in addition to the “real” node name for the upper-most level that the node is connected to in the hierarchy. When all synonyms are saved, Silos will recognize the hierarchical name as well as the “real” name to each node in the design. Caution: Saving all synonyms may slow down input parsing and memory usage may go up. (Default: .SYNONYM=1)
- .SYNONYM=0** don’t save synonyms.
  - .SYNONYM=1** save all synonyms.

*(continued on next page)*

# CONTROL

## (Control Parameters For Logic Simulation)

- .SKIP** Causes Silos to set a skipped port to a level. The default level is High-Z unknown. An allowed level is ground for compatibility with VCS (!control .skip=.gnd). . (Default: **.SKIP=.gnd**)
- .TPS** specifies the default command for redirecting report outputs to standard output or disk file. Allowed qualifiers are: TYPE, WTYPE, STORE, NSTORE.
- .XL\_ORDER=val** specifies a switch so that the order of evaluation for always blocks is the same order as for Verilog-XL, where “val” is “1” for “xl\_order” being “on” (the same as Verilog-XL) and “0” for xl\_order being “off” (default). This switch may be useful for obtaining the same simulation results as Verilog-XL. This option must be parsed before any modules are parsed. An example would be the order of evaluation for:

```
always @posedge clock ....
always @posedge clock ....
```

### Application Notes:

- 1) When the disk storage limit (set by “.CONTROL .DISK”) is exceeded during logic, the simulation will stop. A message will be displayed showing the last simulation time point. To continue the simulation, you can increase the disk limit and re-enter the “SIMULATE” command. Another method would be to report the simulation results, enter “RESET SAVFILE” to clear the disk file “save.sim” (saves the simulation history) and then continue from the last simulation time point. “RESET ERRORS” must be entered before continuing the simulation.
- 2) .DMAX and .DMIN will not both affect the same simulation. The one that is specified last will remain in effect for all netlist parsing and subsequent SIMULATE commands. The .DMAX or .DMIN scaling factor should be specified before inputting the netlist so that the netlist is parsed correctly.
- 3) If “.CONTROL .EUNK=\*\*” has been defined and there is an Unknown level on the gate’s enable, MOS transistors will have an uncertain conductance and interval logic will be used to resolve their source and drain (see Interval Logic: Resolving Uncertain Strength at a Node in the Logic Simulation chapter). Transfer gates also have an uncertain conductance and their output will be resolved using interval logic. For tri-state gates, the output level will be set to Unknown. The output strength will be defined by the gate definition for a tri-state gate.

(continued on next page)

# CONTROL

## (Application Notes:)

- 4) When the iteration limit is exceeded for “.CONTROL .MXDCI” or “.CONTROL .MXITR”, a nonconvergence error stops execution. Nonconvergence may be due either to circuit path length or problems with designs involving feedback. To eliminate oscillations caused by problems involving feedback, the circuit design must be corrected. When nonconvergence is due to path length, increasing the iteration limit should enable the circuit to converge. In general, each node in the serial path length requires one iteration to propagate a signal. Arbitrarily increasing the iteration limits is not recommended as it may dramatically increase the execution time necessary to identify oscillating nodes.
- 5) The “NONCONV” command can be used to identify which parts of the circuit have caused a logic initialization or logic simulation to stop executing.

## Examples:

```
!con .mxd=200 .mxp=200  
CON .DISK=2M .MXOSC=30 .EUNK=*
```

# DELAY

## 5.9 Default Device Delay Times

Normally, if a device has no delay specification, the delay times default to zero. The DELAY command allows you to globally redefine the default values. When this command is used, the back annotation of SDF delays is disabled.

Default delay times are specified as follows:

**DELAY .DEFAULT = d1, d2**

**DELAY** indicates a default delay time specification.

**.DEFAULT** indicates that the default times for all unspecified delays are to be assigned. Normally, the default delays are  $d_1=d_2=0$ .

**d1** represents the nominal rise delay time where:

- “d1” must be an integer between 0 and 10000.

**d2** represents the nominal fall delay where:

- “d2” must be an integer between 0 and 10000.

### Examples:

**!DEL .DEF = 16,5**

**!del .default=0, 0**

**DISK****5.10 Disk File Name Reassignment**

The DISK command enables you to change the default file name for the “STORE”/“NSTORE” commands.

To change the “STORE/NSTORE” disk file name, enter:

**DISK filename**

**DISK** changes the “STORE/NSTORE” disk file name. If no file name is specified, the program will tell you the name of the present default disk file name.

**filename** represents the name of the disk file to which STOREd output will be written.  
(Default: filename=“store.out”)

**Application Notes:**

- 1) Whenever a DISK command is specified, any STOREd data will be written to that disk file until another file name is specified.
- 2) Each time a STORE or NSTORE command is specified, any existing data on the “DISK” file in effect may be overwritten (default), appended or a new cycle will be created.
- 3) The file name can be unlimited in length, but must conform to the file name syntax of your operating system. For the UNIX operating system, the file name is case sensitive.
- 4) The “FILE .STO=“ command can also be used to change the default file name.

**Examples:**

DISK sim.results  
DI PATTERN.INP

# ERRORS

## 5.11 Error Summary

When the program indicates that errors occurred during read-in, preprocessing or simulation, you should enter the “t/s ERRORS” command to determine their error level and type. For input errors, the line number and the input file name will also be reported.

To check the errors, enter:

```
TYPE
STORE    ERRORS [ / LEVEL=value ]
WTYPE
NSTORE
```

**TYPE** (optional) directs the error messages to standard output  
**STORE** or to a disk file.  
**...**

**ERRORS** reports any error and warning messages.

**LEVEL** (optional) indicates that only those errors with a severity level equal to “value” are to be output. If not specified, all errors will be output.

**value** represents a value from 1 to 5.

### Examples:

```
STOR ERROR / LEVEL=2
ty er
```

**EXCLUDE****5.12 Exclude Saving Simulation Node States**

The EXCLUDE command specifies nets whose state values will *not* be saved during simulation. To exclude registers, see “Exclude Saving Module Instance Variable Values” on page 5-28.

The format for the EXCLUDE command is:

```
EXCLUDE name name ... name
```

**EXCLUDE** specifies nets whose state values will not be saved during simulation.

name represents the name of a net whose state changes will not be saved during simulation.

**Application Notes:**

- 1) The KEEP command can be used to specify nets whose simulation states *are* to be saved. The MKEEP and MEXCLUDE commands will keep and exclude all variables (including registers and memory variables) within a module or macro instance.
- 2) The effects to the KEEP and EXCLUDE commands are cumulative. When an identical net name is specified in more than one KEEP or EXCLUDE command, the last KEEP or EXCLUDE command will determine if the simulation states for that net are saved.
- 3) The KEEP, EXCLUDE, MKEEP and MEXCLUDE commands can be used with the “CONTROL .SAVSIM=1” command option to save simulation state values.

**Examples:**

```
CONTROL .SAVSIM=1
```

```
EXCLUDE (REG15(QBAR A15
.exclude (m1(bit0 (m1(bit1
+ (m1(bit4 (m1(bit5 (m1(bit6 (m1(bit7 (driver
+ (obuf(pin34
```

**EXIT**

### **5.13 Exiting The Program**

The “EXIT” command is used for normal exit of Silos.

To exit the program, enter:

**EXIT**

**EXIT** commands the Silos program to stop execution and exit normally.

#### **Example:**

**EXI**

**FILE****5.14 File Name Specification**

The FILE command enables you to redefine the default file names used for the SAVE, STORE and BATCHFILE commands.

The format for the FILE command is:

```
FILE [.SAV=filename] [.STO=filename] [.BAT=filename]
+ [.MODE=.APPEND] [.MODE=.OVERWRITE]
```

|              |   |
|--------------|---|
| <b>FILE</b>  | redefines the file name defaults.   |
| <b>.SAV</b>  | changes the file name prefix “save” to a user specified name for all of the save files, including the save.dictionary file.   |
| <b>.STO</b>  | changes the file name for subsequent STORE and NSTORE commands.   |
| <b>.BAT</b>  | changes the default BATCHFILE command file name.  |
| <b>.MODE</b> | specifies whether a report STOREd will either append to the existing <b>.STO</b> file (or DISK command file) or overwrite the <b>.STO</b> file. (Default: <b>.MODE=.OVERWRITE</b> ) |
| filename     | represents the redefined name of the file. A file name can be unlimited in length. However, the name must conform to the syntax of your operating system.                           |

**Application Notes:**

- 1) The “FILE .SAV” command must be entered before using a FSIM command. Do not specify a file name extension; the program will automatically provide the correct extensions. (Default: filename=SAVE)
- 2) The “FILE .STO” command is equivalent to the DISK command. (Default: filename=STORE.OUT or STORE OUTPUT)
- 3) A file name specified on a subsequent BATCHFILE command will override the “FILE .BAT” command.

**Example:**

```
file .sav=run5
```

**KEEP****5.15 Keeping Simulation Node States**

The KEEP command specifies wires whose state values will be saved during simulation. To save state values to registers, see “Keeping Module Instance Simulation Variable Values” on page 5-29.

The format for the KEEP command is:

```
KEEP name name ... name
```

**KEEP**

specifies wires whose state values will be saved during simulation.

name

represents the name of a wire whose state changes will be kept during simulation.

**Application Notes:**

- 1) The EXCLUDE command can be used to specify wires to be excluded from the saved simulation results.
- 2) The MKEEP and MEXCLUDE commands will keep and exclude all variables (including registers and memory variables) within a module.
- 3) The effects to the KEEP and EXCLUDE commands are cumulative.
- 4) The KEEP, EXCLUDE, MKEEP and MEXCLUDE commands can be used with the “CONTROL .SAVSIM=1” option.

**Examples:**

```
CONTROL .SAVSIM=1
KEEP (MAC15(REG08 (MAC1(MEM(ADDR01
.keep (m1(bit0 (m1(bit1 (m1(bit2 (m1(bit3
+ (m1(bit4 (m1(bit5 (m1(bit6 (m1(bit7
+ (obuf(pin34
```

**MEXCLUDE****5.16 Exclude Saving Module Instance Variable Values**

The MEXCLUDE command excludes the internal variable values from being saved during logic simulation for module instances and any variable that is hierarchically below the excluded module instance.

The format for the MEXCLUDE command is:

**MEXCLUDE mname mname ... mname**

**MEXCLUDE** specifies module instances that will not have their internal variables and any variable that is hierarchically below the excluded module instance saved during logic simulation.

mname represents the name of a module instance.

**Application Notes:**

- 1) The MKEEP command can be used to specify module instances for which the simulation state values to all variables are saved.
- 2) The effects to the MKEEP and MEXCLUDE commands are cumulative.
- 3) The KEEP, EXCLUDE, MKEEP and MEXCLUDE commands can be used with the “CONTROL .SAVSIM=1” option.

**Examples:**

```
CONTROL .SAVSIM=1
MKEEP (mac1(a
MEXCLUDE (mac1(a(c
```

```
!mexclude (m1(bit0 (m1(bit1 (m1(bit2 (m1(bit3
+ (m1(bit4 (m1(bit5 (m1(bit6 (m1(bit7 (driver
+ (iobuf(pin34
```

**MKEEP****5.17 Keeping Module Instance Simulation Variable Values**

The MKEEP command saves the logic simulation state values for all variables in the specified module instances, and the state values for all variables hierarchically below each specified module instance.

The format for the MKEEP command is:

```
MKEEP mname mname ... mname
```

**MKEEP** specifies module instances whose logic simulation state values will be saved for all variables in the specified module instance, and for all variables hierarchically below each specified module instance.

**mname** represents the name of a module instance.

**Application Notes:**

- 1) The MEXCLUDE command can be used to specify module instances whose internal variables will not be saved during logic simulation.
- 2) The effects to the MKEEP and MEXCLUDE commands are cumulative.
- 3) The KEEP, EXCLUDE, MKEEP and MEXCLUDE commands can be used with the “CONTROL .SAVSIM=1” option.

**Examples:**

```
CONTROL .SAVSIM=1
MKEEP (mac1(a(b
```

```
.mkeep (m1(bit0 (m1(bit1 (m1(bit2 (m1(bit3
+ (m1(bit4 (m1(bit5 (m1(bit6 (m1(bit7 (driver
+ (obuf(pin34
```

**NOCONV****5.18 Nonconvergence Summary**

The “t/s NOCONV” command generates a report of any nonconverged nodes and their oscillating states for the time point that nonconvergence occurred.

To obtain a nonconvergence summary for nodes and devices, enter:

|               |                                    |
|---------------|------------------------------------|
| <b>TYPE</b>   |                                    |
| <b>STORE</b>  | <b>NOCONV [ / INPUT ITER=val ]</b> |
| <b>WTYPE</b>  |                                    |
| <b>NSTORE</b> |                                    |

**TYPE** (optional) directs the nonconvergence table to standard output  
**STORE** or to a disk file.

...

**NOCONV** reports the oscillating states for nonconverged nodes.

**INPUT** (optional) reports the states for all inputs of devices which drive the oscillating nodes.

**ITER=val** (optional) specifies the iteration number to the 1st of eight states for each node reported for nonconvergence. The default iteration for ‘val’ is computed such that the last of the eight states reported corresponds to the last iteration simulated before no convergence halted the simulation.

*(continued on next page)*

**NOCONV****(Nonconvergence Summary)****Application Notes:**

- 1) The “t/s NOCONV” command can be used to identify which parts of the circuit have caused a logic initialization or logic simulation to nonconverge.
- 2) The “t/s NOCONV” command reports the following information:
  - names of the unresolved devices and nodes.
  - the “type” of device and node as either a “.type” data keyword, “NODE” for a wired connection with at least one bi-directional device or “BUS” for a wired connection between two or more unidirectional enabled gates.
  - the node state values for the nonconvergence time point. State values reported are preceded by a “...” to indicate possible previous states.
- 3) Nonconvergence only occurs when gate delays are zero. Zero-delays occur during logic initialization , which forces delays to be zero; or, during zero-delay logic simulation ; or, when either zero delay or no delay is specified for devices (the default delay for Verilog HDL devices is zero).
- 4) When the iteration limit is exceeded while resolving node states at a time point, a nonconvergence error stops execution. Nonconvergence may be due either to the circuit path length or problems with designs involving feedback. The circuit design must be corrected to eliminate oscillations caused by problems involving feedback. When nonconvergence is due to path length, increasing the iteration limit should enable the circuit to converge. In general, each node in the serial path length requires one iteration to propagate a signal.
- 5) The maximum iterations per pass for logic initialization can be redefined by the “CONTROL .MXDCI” command. The maximum iterations at a time point for logic simulation can be redefined by the “CONTROL .MXITR” command. Arbitrarily increasing these parameters is not recommended as it may dramatically increase the execution time necessary to identify oscillating nodes.

*(continued on next page)*

## NOCONV

### (Nonconvergence Summary)

#### (Application Notes:)

- 6) The maximum number of passes for logic initialization is defined by the “CONTROL .MXPAS” command. When this limit is exceeded, the error message specifies the required number of passes to complete logic initialization. To reduce the number of passes and execution time, use “.INIT” to preset the state for critical nodes. Although arbitrarily setting “CONTROL .MXPAS” to a large value will very likely converge a circuit that is theoretically solvable, this is not recommended as the problem is usually due to incorrect circuit design.

#### Examples:

```
ty noc
```

```
nocon /iter=43
```

**NSTORE****5.19 Narrow Storing Outputs**

When a command that generates a report is preceded by the NSTORE command, the report output will be directed to a disk file.

To specify the NSTORE command, enter:

**NSTORE    command ...**

**NSTORE**      directs the output to a 79 column disk file. As a default, this file is named “store.out”.

**command**      represents a command structure which defines the type of data to be output. These commands are described within this section of the manual.

**Application Notes:**

- 1) If a command that generates a report is not preceded by the NSTORE command, then the default output device is specified by the CONTROL command.
- 2) To respecify the page width for the NSTORE command, use the “FORMAT .NSTORE” command.
- 3) The default file name for the NSTORE command may be redefined using the DISK command or the “FILE .STO=“ command.

**Examples:**

nstore output on change

NSTO NETWORK /FDD

# PREPROC

## 5.20 Preprocessing Data

Normally, data preprocessing is automatically performed prior to initialization or simulation (i.e., when the SIMULATE command are entered). However, you may wish to use the PREPROC command to check for syntax errors without simulating.

To preprocess data, enter:

**PREPROC**

### Application Notes:

- 1) During data preprocessing, the program resolves and checks all gate input connections, calculates fan-out connections and creates implicit nodes. Generally, the data is reformatted for more efficient simulation.
- 2) Once preprocessing has been performed, additional topological data cannot be entered.
- 3) Note that at least the first three letters of this command must be specified (i.e. **PRE**).
- 4) If serious errors occur during a phase of the preprocessing, you should correct the errors before proceeding. If the errors were corrected interactively, enter “RESET ERRORS” and then reissue the PREPROC command to continue preprocessing.

**probe****5.21 Probing Node States**

The probe command reports the value of variables and **expressions** in tabular format.

The format for the probe command is:

|              |                      |                       |                              |
|--------------|----------------------|-----------------------|------------------------------|
| <b>STORE</b> | <b>probe</b>         | <b>t1 t2 "format"</b> | (expression), expression ... |
| <b>STORE</b> | <b>probe ITER</b>    | <b>t1 t2 "format"</b> | (expression), expression ... |
| <b>STORE</b> | <b>probe STEP dt</b> | <b>t1 t2 "format"</b> | (expression), expression ... |

|                   |  |
|-------------------|--|
| <b>STORE</b>      | (optional) directs the probe output to a disk file. Use the DISK command to specify the file name for the stored output.   |
| <b>probe</b>      | reports the value of variables and expressions in tabular format. (Default: report the “on change” values)   |
| <b>STEP dt</b>    | (optional) causes the values to be reported between time “t1” to time “t2” at intervals of “dt”.   |
| <b>ITER</b>       | (optional) causes values of variables and expressions to be reported for each iteration at a time point.   |
| <b>t1 t2</b>      | (optional) represents an optional time point range over which the values will be reported. When “t1” and “t2” are not specified the probe command will use the simulation time values or the time values specified on the last probe command.  |
| <b>format</b>     | (optional) specifies the format for reporting the expressions. Any of the format specifications for \$display and \$monitor are allowed (Default radix: %h)  |
| <b>expression</b> | specifies any legal Verilog HDL expression. The parentheses around the first specified expression are not required when it is just a single variable. Full hierarchical path names can be used, otherwise the module instance selected by the “SCOPE” command will be used. A “,” can be used instead of a name to insert a blank column in the report. When an “@” sign is used in front of any expression, then values are reported when those expressions change. |

(continued on next page)

# probe

## (Probing Node States)

### Examples:

A typical way the probe command can be used is to declare the scope for a module instance, and then list variables in the module that you want to report on. Using two commas between variables would leave a blank column between variables:

```
scope main           // declare module instance "main".
pro a,,b,,c         // blank column between variables.
```

The probe command can be used to report the value for any expression, such as, you could use the following probe command to report the value for the assignment “out= (a+b) | (c+d)” for each change of variable “clock”:

```
probe @clock,, "out=", (a+b) | (c+d)
```

The STORE command can be used to store the probe report to a file:

```
store probe a,b      // stores the probe report to a file.
```

Some additional examples for the probe command are listed below:

```
probe main.i1.a       // report variable "a" inside instance "main.i1"
probe "output result = %o", out   // use string and octal radix formats.
probe 0 100 %o{a,b,c}    // report concatenated variables as octal
store probe %b a[0:2], %h a[3:6] // vary the radix for reporting values.
```

**QUIT**

## **5.22 Quitting Execution**

The QUIT command enables you to terminate an unwanted session without that session affecting any active SAVE files.

To quit program execution, enter:

**QUIT**

### **Application Notes:**

- 1) The QUIT command aborts execution of the program and all program results since the last SAVE command are lost.

**RESET****5.23 Resetting Selected Data**

The RESET command can be used to reset (i.e. delete) selected data information for the command line version of Silos. For the graphical interface version, the Silos, use the Load/Reload Files button.

The form of the command is:

|              |                |
|--------------|----------------|
| <b>RESET</b> | <b>ALL</b>     |
|              | <b>ERRORS</b>  |
|              | <b>OUTPUTS</b> |
|              | <b>PATTERN</b> |
|              | <b>SAVFILE</b> |

**RESET** resets selected program counters and/or flags as specified by the below options.

**ALL** resets everything (as if you just began execution). The program will not issue a warning.

**ERRORS** deletes data error flags and messages up through level 4. This can be used to continue a simulation after errors have been corrected, and to clear unnecessary warning and error messages.

**SAVFILE** resets the logic simulation save file data to eliminate disk storage.

*(continued on next page)*

## RESET

### (Resetting Selected Data)

#### Application Notes:

- 1) For RESET OUTPUTS, new output commands can be entered from the menu selections or input from a file.
- 2) Before using “RESET SAVFILE”, reports should be generated and/or the SAVE files should be copied to tape. After “RESET SAVFILE”, logic simulation can be continued from the last simulation time point but output reports are not available for simulation results prior to the last simulation time point.

#### Examples:

RES ERR

res savfile

# SCOPE

## 5.24 Scope For Printing Module Variables

The SCOPE command declares the module instance used when the PRINT or probe commands reports the values for variables and expressions in a module.

The format for the SCOPE command is:

```
SCOPE instance_name
```

**SCOPE**        declares the module instance used by the PRINT or probe command.

**instance\_name**    represents the instance name for the module whose variables will be reported by the PRINT or probe commands.

### Example:

```
SCOPE main.cpu.cache
```

**SIMULATE****5.25 Logic Simulation Specification**

Logic simulation can be performed by entering the SIMULATE command.

To initiate the logic simulation, enter:

**SIMULATE t1 [ TO t2 ]**

**SIMULATE** performs time-response logic simulation that can use both finite and zero delay specifications. Preprocessing (PREPROC) and logic initialization will be automatically performed if they have not been previously.

t1 TO t2 represent the values of the first and last simulation time points (see note 1 below). The keyword “TO” is optional.

**Application Notes:**

- 1) When specifying the simulation time point range, the following items apply:
  - specifying neither t1 nor t2 or setting t2 to an arbitrary large number will cause Silos to simulate until “\$stop” or “\$finish” is encountered in the netlist. You can stop the simulation by clicking-on the “STOP” button or pushing the “Escape” key (Esc) on the keyboard for the PC and “Ctrl-C” on Unix.
  - specifying a single time point indicates that simulation will be incrementally continued for that amount of time. At time=0, this will start the simulation from time=0 for the specified amount of time.
  - Specifying t1 and t2, with t1=0, runs the simulation from time=0 to time=t2.
  - Specifying t1 and t2, with t1 greater than zero, continues the simulation from the last specified time point.
  - Specifying “TO t2” will continue the simulation until time=t2. This can be useful to continue a simulation that was halted due to a breakpoint.

(continued on next page)

## SIMULATE

### (Logic Simulation Specification)

#### (Application Notes:)

- 2) The SIMULATE command will automatically invoke the PREPROC command (if PREPROC has not already been performed) and no further topological data can be entered.
- 3) Simulation can either be continued from the last time point or restarted from time=0.
- 4) The SIMULATE command uses inertial delays, which do not propagate level changes that occur faster than the gate output can change (spike condition).

#### Examples:

```
simul 0 to 22k  
SIM 5KGG 10K  
SIM 5K  
sim 0 15k  
SI TO 5K
```

# SIZES

## 5.26 Size-Of-Data Reprint

The SIZES command reports the memory usage for Silos.

To report the network size information, enter:

**TYPE**  
**STORE**      **SIZES**  
**WTYPE**  
**NSTORE**

**TYPE**            (optional) directs the size information to standard output  
**STORE**           or to a disk file.

...

**SIZES**           generates memory usage information.

### Application Notes:

- 1) Items reported include the total number of devices, network names, etc.
- 2) The memory usage may be different after read-in, preprocessing and simulation.
- 3) The memory usage is also reported in the “Help/About” box.

### Examples:

**NSTO SIZ**  
**TY SIZ**

**SPIKES****5.27 Spike Summary Output**

The “t/s SPIKES” command allows you to view all the nodes on which spikes were made observable during logic simulation (see “Logic Simulation Specification” on page 5-41).

To generate a node spike summary, enter:

**TYPE**  
**STORE      SPIKES [t1 TO t2]**  
**WTYPE**  
**NSTORE**

|                 |  |
|-----------------|--|
| <b>TYPE</b>     | (optional) directs the spike summary to standard output  |
| <b>STORE</b>    | or to a disk file.   |
| ...             |  |
| <b>SPIKES</b>   | lists a summary table of all spikes between two time points.   |
| <b>t1 TO t2</b> | represent the minimum and maximum time point values over which the spike output is to be generated. This time point range must be within the logic simulation time point range. If the time points are not specified, the logic simulation times are used. (The “TO” keyword is optional.) |

**Application Notes:**

- 1) A spike occurs when the gate input level changes faster than the gate output can change.
- 2) Setting the criteria for spike conditions is controlled by the +pulse\_e, +pulse\_r, and +pathpulse command line arguments. For additional information, see “Command-line Options” on page 7-4.
- 3) To enable spike recording during logic simulation for the SPIKE report, use the +silos\_spike command line option. For additional information, see “Command-line Options” on page 7-4.

**Examples:**

```
ty spikes
NSTO SPIKES 4.2K 4.8K
```

**STORE****5.28 Storing Outputs**

When a command that generates a report is preceded by the STORE command, the report output will be directed to a disk file.

To specify the STORE command, enter:

**STORE      command ...**

**STORE**      directs the output to a 132 column disk file. As a default, this file is named “store.out”.

**command**      represents a command structure which defines the type of data to be output. These commands are described within this section of the manual.

**Application Notes:**

- 1) If a command that generates a report is not preceded by the STORE command, then the default output device is specified by the CONTROL command.
- 2) To respecify the page width for the STORE command, use the “FORMAT .STORE” command.
- 3) The default file name for the STORE command may be redefined using the “DISK” command or the “FILE .STO” command.

**Examples:**

store output on change

STO NETWORK /FDD

**STRENGTH****Strength Specification For Gates**

The STRENGTH command allows you to modify the default strength types.

To respecify default strength types for gate devices, use:

**STRENGTH .device/strg .device/strg ...DEFAULT/strg**

**STRENGTH** indicates that default strength-types are to be assigned to unidirectional gate devices.

.device represents a device keyword (see note 1 below). If no “device/N”, “device/P” or “device/strg” keyword is specified for an individual gate device, the program defaults to a “CMOS” strength type.

**DEFAULT** sets the strength for all devices that do not have the strength explicitly specified. The default is “CMOS” strength type.

strg represents any combination of “D”, “R” or “Z”. The first character indicates the strength of the Low level. The second character indicates the strength of the Unknown level. The third character indicates the strength of the High level. “D” represents Strong strength, “R” represents Pull strength, and “Z” represents High-Z strength.

Alternatively, the characters “N”, “P” or “C” can be used by themselves to indicate NMOS-type (DRR), PMOS-type (RRD) or CMOS-type (DRD) defaults.

(continued on next page)

## STRENGTH

### (Strength Specification For Gates)

#### Application Notes:

- 1) The “device” keyword can be any of the combinatorial gate devices.
- 2) Supply-strength cannot be defined.

#### Examples:

```
!strength .nor/n .nand/n .not/c
```

```
!strength default/ddd
```

# SYMBOL

## 5.29 Symbol Modification For Output

The SYMBOL command allows you to use a unique symbol for each possible logic state.

To modify the output state symbols, use:

**SYMBOL sc=char sc=char sc=char ...**

**SYMBOL** specifies that the state code symbols are to be redefined.

**sc** represents one of the state-type codes for the OUTPUT, POUTPUT and probe reports and for the .CLK and .PATTERN stimulus specifications:

| state<br>symbols | state                  | default<br>report symbol | default<br>stimulus<br>char |
|------------------|------------------------|--------------------------|-----------------------------|
| <b>S0</b>        | Supply Low             | <b>0</b>                 | <b>0</b>                    |
| <b>S*</b>        | Supply Unknown         | *                        | *                           |
| <b>S1</b>        | Supply High            | <b>1</b>                 | <b>1</b>                    |
| <b>SHV</b>       | Supply High-Voltage    | <b>1</b>                 |                             |
| <br>             |                        |                          |                             |
| <b>D0</b>        | Driving Low            | <b>0</b>                 |                             |
| <b>D*</b>        | Driving Unknown        | *                        |                             |
| <b>D1</b>        | Driving High           | <b>1</b>                 |                             |
| <b>DHV</b>       | Driving High-Voltage   | <b>1</b>                 |                             |
| <br>             |                        |                          |                             |
| <b>R0</b>        | Resistive Low          | <b>0</b>                 |                             |
| <b>R*</b>        | Resistive Unknown      | *                        |                             |
| <b>R1</b>        | Resistive High         | <b>1</b>                 |                             |
| <b>RHV</b>       | Resistive High-Voltage | <b>1</b>                 |                             |
| <br>             |                        |                          |                             |
| <b>Z0</b>        | High-Z Low             | <b>Z</b>                 |                             |
| <b>Z*</b>        | High-Z Unknown         | <b>Z</b>                 | <b>Z</b>                    |
| <b>Z1</b>        | High-Z High            | <b>Z</b>                 |                             |
| <b>ZHV</b>       | High-Z High-Voltage    | <b>Z</b>                 |                             |

(continued on next page)

# SYMBOL

## (Symbol Modification For Output)

- (sc) The following symbols are used only in the output reports. No symbol can be defined to input these states for .CLK or .PATTERN stimulus:

| state<br>symbols | state                  | default<br>report symbol | default<br>stimulus<br>char |
|------------------|------------------------|--------------------------|-----------------------------|
| *0               | Uncertain Low          | *                        |                             |
| **               | Uncertain Unknown      | *                        |                             |
| *1               | Uncertain High         | *                        |                             |
| *HV              | Uncertain High-Voltage | *                        |                             |
| <b>0D</b>        | Decaying Low           | <b>D</b>                 |                             |
| *D               | Decaying Unknown       | <b>D</b>                 |                             |
| 1D               | Decaying High          | <b>D</b>                 |                             |
| HVD              | Decaying High-Voltage  | <b>D</b>                 |                             |
| <b>*S</b>        | Spike                  | <b>S</b>                 |                             |

char represents the single character you want to be used to indicate the state. The comment character (default a "\$") cannot be used as a "char" symbol.

### Application Notes:

- 1) Enter the SYMBOL command before the ".PATTERN" and ".CLK" specifications are entered to redefine symbols used for stimulus state values.
- 2) The SYMBOL command can redefine node state symbols either before or after simulation for the "t/s OUTPUT", probe, and "t/s POUTPUT" reports.

*(continued on next page)*

# SYMBOL

## (Symbol Modification For Output)

### (Application Notes:)

- 3) When the same symbol is used to represent the different states (as in the defaults of 0,\*,1) for the input stimulus for a .CLK or .PATTERN specification, the program resolves the ambiguity in the following order:
- the most recent symbol specified by the most recently entered SYMBOL command is used.
  - for the default symbols not specified by a SYMBOL command, the higher strength is used and within a strength, the higher level is used.

For example, if “SYMBOL D1== R0==” is entered, then the symbol “+” would mean Resistive Low. If “SYMBOL D1== D\*=+=” is entered, then the symbol “+” would mean Driving Unknown.

- 4) Note that the Unset state symbol cannot be changed; it will always be a question mark “?”.

### Examples:

```

SYMBOL Z0=- Z*=# Z1=+
.symbol r*=U z*=U d*=U r1=H z1=H d1=H d0=L
+ r0=L z0=L
SYM 0D=L *D=U 1D=H S0=O S*=X S1=I
!SYM D0=O D*=X D1=I *S=^
```

## **6. Appendix A: Libraries**

### **6.1 Overview**

Simucad provides many of the popular TTL library models for the SN74LS series. The behavioral source for these parts is provided as four libraries:

- SN74LS series without timing (subdirectory “library\sn74ls”).
- SN74LS series with timing (subdirectory “library\sn74lst”).
- SN74BCT series without timing (subdirectory “library\sn74bct”).
- SN74BCT series with timing (subdirectory “library\sn74bctt”).

# LIBRARY

## 6.2 Library Command

Silos will search library files when module definitions are not found for the module instances in the design. The LIBRARY command can be used to specify library file names and file name extensions for library files.

To specify the library file names, enter:

**LIBRARY [filename] [{.ext}]**

**LIBRARY** defines disk file names for external libraries.

**filename** represents the name of one or more additional library disk files.

**{.ext}** represents the file name extension.

When searching a directory for library files, Silos searches for a file whose root name is the same as the module name on the module instance, and whose extension matches the extension specified for the library search.

### Application Notes:

- 1) Library files can contain module definitions and module instances. Commands are not allowed in library files.
- 2) LIBRARY files are useful for:
  - a method of inputting library parts with specific timing;
  - reducing memory by not loading unreferenced netlist data;
  - encrypting data (see “Encrypting Library Files” on page 5-15).

*(continued on next page)*

# LIBRARY

## (Library Files)

## (Application Notes:)

- 3) The search order for a module definition or macro definition is:
  - a) files entered by the INPUT command or ‘include compiler directive.
  - b) library file names, or directories that contain library files ending with a specified extension, that were specified by the LIBRARY command. The library files will be searched in the order specified to the program.
- 4) Each LIBRARY command will override any previous LIBRARY commands (they are not cumulative). A LIBRARY command can use more than one line by beginning each new line with a “+” sign in column one (see the Examples below). If the file name specified by the LIBRARY command cannot be found during the CHECK command, a warning message is issued. Entering the LIBRARY command after preprocessing (PREPROC command) has no meaning and a warning message is not issued.
- 5) Library files can be encrypted by the CHGLIB program. Once a file has been encrypted, it cannot be entered with the INPUT command or ‘include compiler directive. It is good practice to check the files for errors with the INPUT command before encrypting them.
- 6) Library files can be converted to random-access files by the CHGLIB program. Once a file has been converted to random-access, it cannot be entered with the INPUT command or ‘include compiler directive. It is good practice to check the files for errors with the INPUT command before converting them to random-access.

## Examples:

```
.LIBRARY d:\test\asic{.v}
```

```
!lib fast.lib
+ nmos.lib
+ ecl.dat
```

**TTL - LS****6.3 TTL LS Parts List**

The list below shows the TTL LS library parts provided with Silos. The behavioral source for these parts is provided with unit delays and with full timing:

(Ref. TTL Logic Data book, SDLD001A Revised March 1988)

(Ref. BiCMOS Bus Interface Logic Data book, SCBD001A Revised July 1989)

| <u>Name</u> | <u>Description</u>       |
|-------------|--------------------------|
| 74LS00      | 2 INPUT NAND             |
| 74LS01      | 2 INPUT NAND OC          |
| 74LS02      | 2 INPUT NOR              |
| 74LS03      | 2 INPUT NAND OC          |
| 74LS04      | HEX INVERTER             |
| 74LS05      | HEX INVERTER             |
| 74LS08      | 2 INPUT AND              |
| 74LS09      | 2 INPUT AND OC           |
| 74LS10      | 3 INPUT NAND             |
| 74LS11      | 3 INPUT AND              |
| 74LS12      | 3 INPUT NAND OC          |
| 74LS13      | 4 INPUT NAND SCHM. TRIG. |
| 74LS14      | HEX SCHM. TRIG. INVERTER |
| 74LS15      | 3 INPUT AND OC           |
| 74LS19A     | HEX SCHM. TRIG. INVERTER |
| 74LS20      | 4 INPUT NAND             |
| 74LS21      | 4 INPUT AND              |
| 74LS22      | 4 INPUT NAND OC          |
| 74LS24A     | 2 INPUT NAND SCHM. TRIG. |

(continued on next page)

**TTL - LS****(TTL LS Parts List)**

74LS26 2 INPUT NAND OC  
74LS27 3 INPUT NOR  
74LS28 2 INPUT NOR  
74LS30 8 INPUT NAND  
74LS31 DELAY LINE  
74LS32 2 INPUT OR  
74LS33 2 INPUT NOR OC  
74LS37 2 INPUT NAND  
74LS38 2 INPUT NAND OC  
74LS40 4 INPUT NAND  
74LS42 BCD TO DECIMAL DECODER  
74LS51 3 WIDE AND-OR-INV  
74LS54 4 WIDE AND-OR-INV  
74LS55 2 WIDE AND-OR-INV  
74LS56 FERQUENCY DIVIDER 50:1  
74LS57 FREQUENCY DIVIDER 60:1  
74LS68 DECADE/BINARY COUNTER  
74LS69 DECADE/BINARY COUNTER  
74LS73A JK FLIP-FLOP  
74LS74A D FLIP-FLOP  
74LS75 LATCH  
74LS76A JK FLIP-FLOP  
74LS77 LATCH  
74LS78A JK FLIP-FLOP  
74LS83A ADDER

(continued on next page)

**TTL - LS****(TTL LS Parts List)**

74LS85 COMPARATOR  
74LS86A 2 INPUT XOR  
74LS90 DECADE COUNTER  
74LS91 8 BIT SHIFT REGISTER  
74LS92 DIVIDE BY 12 COUNTER  
74LS93 4 BIT BINARY COUNTER  
74LS95 4 BIT SHIFT REG  
74LS96 5 BIT SHIFT REG.  
74LS107A JK FLIP-FLOP  
74LS109A JK FLIP-FLOP  
74LS112A JK FLIP-FLOP  
74LS113A JK FLIP-FLOP  
74LS114A JK FLIP-FLOP  
74LS122 MONOSTABLE MULTIVIBRATOR  
74LS123 MONOSTABLE MULTIVIBRATOR  
74LS125A TRI-STATE BUFFERS  
74LS126A TRI-STATE BUFFERS  
74LS132 2 INPUT NAND SCHM. TRIG.  
74LS136 2 INPUT XOR OC  
74LS137 3 TO 8 DECODER  
74LS138 3 TO 8 DECODER  
74LS139A 2 TO 4 DECODER  
74LS147 PRIORITY ENCODER  
74LS148 PRIORITY ENCODER

(continued on next page)

**TTL - LS****(TTL LS Parts List)**

74LS151 MUX 8 TO 1  
74LS153 MUX 4 TO 1  
74LS155A 2 TO 4 DECODER  
74LS156 2 TO 4 DECODER OC  
74LS157 2 TO 1 MUX  
74LS158 2 TO 1 MUX  
74LS160A SYNC 4-BIT COUNTER  
74LS161A SYNC 4-BIT COUNTER  
74LS162A SYNC 4-BIT COUNTER  
74LS163A SYNC 4-BIT COUNTER  
74LS164 8 BIT SHIFT REG.  
74LS165A PARALLEL LOAD BIT SHIFT REG.  
74LS166A PARALLEL LOAD BIT SHIFT REG.  
74LS169B UP/DOWN BINARY COUNTER  
74LS170 4\*4 REGISTER FILE OC  
74LS171 D FLIP-FLOP  
74LS173A D FLIP-FLOP WITH 3-STATE  
74LS174 D FLIP-FLOP  
74LS175 D FLIP-FLOP  
74LS181 ALU  
74LS183 CARRY SAVE ADDER  
74LS190 UP/DOWN COUNTER  
74LS191 UP/DOWN COUNTER  
74LS192 UP/DOWN COUNTER  
74LS193 UP/DOWN COUNTER

(continued on next page)

**TTL - LS****(TTL LS Parts List)**

74LS194A 4 BIT SHIFT REGISTER  
74LS195 4 BIT SHIFT REGISTER  
74LS196 BINARY COUNTER  
74LS197 BINARY COUNTER  
74LS221 MONOSTABLE MULTIVIBRATOR  
74LS240 TRI-STATE BUFFERS  
74LS241 TRI-STATE BUFFERS  
74LS242 TRANSCEIVER  
74LS243 TRANSCEIVER  
74LS244 TRI-STATE BUFFERS  
74LS245 TRANSCEIVER  
74LS251 3-STATE MUX  
74LS253 3-STATE MUX  
74LS257B MUX  
74LS258B MUX  
74LS259B LATCH  
74LS261 MULTIPLIER  
74LS266 XNOR OC  
74LS273 D FLIP-FLOPS  
74LS279 SR LATCH  
74LS279A SR LATCH  
74LS280 PARITY GENERATOR/CHECKER  
74LS283 4 BIT ADDER  
74LS290 DECADE COUNTER  
74LS292 PROGRAMMABLE COUNTER

(continued on next page)

**TTL - LS****(TTL LS Parts List)**

74LS293 BINARY COUNTER  
74LS294 PROGRAMMABLE COUNTER  
74LS295B SHIFT REG.  
74LS298 MUX WITH STORAGE  
74LS299 8 BIT SHIFT REGISTER  
74LS322A 8 BIT SHIFT REG.  
74LS323 8 BIT SHIFT REG.  
74LS348 PRIORITY ENCODER  
74LS352 MUX  
74LS353 MUX  
74LS354 MUX  
74LS355 MUX  
74LS356 MUX  
74LS365A BUS DRIVER  
74LS366A BUS DRIVER  
74LS367A BUS DRIVER  
74LS368A BUS DRIVER  
74LS373 LATCH  
74LS374 FLIP-FLOPS  
74LS375 LATCH  
74LS377 D FLIP-FLOPS  
74LS378 D FLIP-FLOPS  
74LS379 D FLIP-FLOPS  
74LS381A ALU  
74LS382A ALU

(continued on next page)

**TTL - LS****(TTL LS Parts List)**

74LS384 MULTIPLIER  
74LS385 ADDER/SUBTRACTOR  
74LS386A XOR  
74LS390 DECADE COUNTER  
74LS393 BINARY COUNTER  
74LS395A SHIFT REG  
74LS396 FLIP-FLOPS  
74LS399 MUX WITH STORAGE  
74LS422 MONOSTABLE MULTIVIBRATOR  
74LS440 TRANSCEIVER  
74LS441 TRANSCEIVER  
74LS442 TRANSCEIVER  
74LS444 TRANSCEIVER  
74LS446 TRANSCEIVER  
74LS449 TRANSCEIVER  
74LS465 BUFFER  
74LS466 BUFFER  
74LS467 BUFFER  
74LS468 BUFFER  
74LS490 DECADE COUNTER  
74LS540 BUFFER  
74LS541 BUFFER  
74LS590 BINARY COUNTER  
74LS591 BINARY COUNTER  
74LS592 BINARY COUNTER

(continued on next page)

**TTL - LS****(TTL LS Parts List)**

74LS593 BINARY COUNTER  
74LS594 SHIFT REG.  
74LS595 SHIFT REG.  
74LS596 SHIFT REG.  
74LS597 SHIFT REG.  
74LS598 SHIFT REG.  
74LS599 SHIFT REG.  
74LS604 LATCH  
74LS606 LATCH  
74LS607 LATCH  
74LS620 TRANSCEIVER  
74LS621 TRANSCEIVER  
74LS623 TRANSCEIVER  
74LS639 TRANSCEIVER  
74LS640 TRANSCEIVER  
74LS641 TRANSCEIVER  
74LS642 TRANSCEIVER  
74LS644 TRANSCEIVER  
74LS645 TRANSCEIVER  
74LS646 TRANSCEIVER/REGISTERS  
74LS647 TRANSCEIVER/REGISTERS  
74LS648 TRANSCEIVER/REGISTERS  
74LS649 TRANSCEIVER/REGISTERS  
74LS651 TRANSCEIVER/REGISTERS  
74LS652 TRANSCEIVER/REGISTERS

(continued on next page)

**TTL - LS****(TTL LS Parts List)**

74LS653 TRANSCEIVER/REGISTERS  
74LS668 UP/DOWN COUNTER  
74LS669 UP/DOWN COUNTER  
74LS671 SHIFT REG.  
74LS672 SHIFT REG.  
74LS673 SHIFT REG.  
74LS674 SHIFT REG.  
74LS681 ALU  
74LS682 COMPARATOR  
74LS684 COMPARATOR  
74LS685 COMPARATOR  
74LS686 COMPARATOR  
74LS687 COMPARATOR  
74LS688 COMPARATOR  
74LS690 COUNTER  
74LS691 COUNTER  
74LS693 COUNTER  
74LS696 COUNTER  
74LS697 COUNTER  
74LS699 COUNTER

**TTL - BCT****6.4 TTL BCT Parts List**

Simucad provides many of the popular TTL library models for the 74 BCT series. The behavioral source for these parts is provided as two libraries:

- SN74BCT series without timing (subdirectory “library\sn74bct”).
- SN74BCT series with timing (subdirectory “library\sn74bctt”).

The list below shows the TTL BCT library parts provided with Silos:

(Ref. TTL Logic Data book, SDLD001A Revised March 1988)

(Ref. BiCMOS Bus Interface Logic Data book, SCBD001A Revised July 1989)

| <u>Name</u> | <u>Description</u>            |
|-------------|-------------------------------|
| 74BCT125    | Quad Buffer Gates             |
| 74BCT126    | Quad Buffer Gates             |
| 74BCT240    | Octal buffers, line drivers   |
| 74BCT241    | Octal buffers, line drivers   |
| 74BCT244    | Octal buffers, line drivers   |
| 74BCT245    | Octal bus transceivers        |
| 74BCT373    | Octal D-type latches          |
| 74BCT374    | Octal D-type FFs              |
| 74BCT540    | Octal buffers, line drivers   |
| 74BCT541    | Octal buffers, line drivers   |
| 74BCT543    | Octal registered transceivers |
| 74BCT534    | Octal D-type FFs              |
| 74BCT620A   | Octal bus transceivers        |
| 74BCT623    | Octal bus transceivers        |
| 74BCT640    | Octal bus transceivers        |

(continued on next page)

**TTL - BCT****(TTL BCT Parts List)**

|             |                              |
|-------------|------------------------------|
| 74BCT652    | Octal bus transceivers       |
| 74BCT760    | Octal buffers, line drivers  |
| 74BCT2240   | Octal buffers, line drivers  |
| 74BCT2241   | Octal buffers                |
| 74BCT2244   | Octal buffers                |
| 74BCT2827A  | 10-bit bus/memory drivers    |
| 74BCT2828A  | 10-bit bus/memory drivers    |
| 74BCT29827A | 10-bit buffers               |
| 74BCT29828A | 10-bit buffers               |
| 74BCT29833  | 8 to 9bit Parity Transceiver |
| 74BCT29834  | 8 to 9bit Parity Transceiver |
| 74BCT29843  | 9bit Bus Interface DLatches  |
| 74BCT29844  | 9bit Bus Interface DLatches  |
| 74BCT29845  | 8bit Bus Interface DLatches  |
| 74BCT29846  | 8bit Bus Interface DLatches  |
| 74BCT29853  | 8 to 9-bit Parity Xsciever   |
| 74BCT29854  | 8 to 9-bit Parity Xsciever   |
| 74BCT29861A | 10bit bus transceivers       |
| 74BCT29862A | 10bit bus transceivers       |
| 74BCT29863A | 9bit bus transceivers        |
| 74BCT29864A | 9bit bus transceivers        |

(continued on next page)

**TTL - BCT****(TTL BCT Parts List)**

The following are from Preliminary Data sheets. There is a total of 10 parts in 74BCT series which are not released products in the TI 1989 data book.

| <u>Name</u> | <u>Description</u>            |
|-------------|-------------------------------|
| 74BCT544    | Octal registered transceivers |
| 74BCT756    | Octal buffers, line drivers   |
| 74BCT8244   | SCAN Test with Octal Buffer   |
| 74BCT8245   | SCAN Test & Octal Xscievers   |
| 74BCT29821  | 10bit Bus Interface FF        |
| 74BCT29822  | 10bit Bus Interface FF        |
| 74BCT29823  | 9bit Bus Interface FF         |
| 74BCT29824  | 9bit Bus Interface FF         |
| 74BCT29841  | 10bit Bus - D Latches         |
| 74BCT29842  | 10bit Bus - D Latches         |

## **7. Appendix B: Command Line Arguments**

### **7.1 Batch Execution Overview**

Silos can be run on the host computer as:

- an interactive session for debugging a design;
- a batch session for running regression tests.

Running Silos as a batch execution may be useful for:

- Running regression tests.

This section explains how to run Silos as a batch execution in the Windows 2000, Windows 98 operating system, Windows NT operating system or the Unix operating system. Examples are provided for common tasks such as using Silos commands from a file to input and simulate the netlist, and report simulation results.

Before reading the sections on running as a batch execution, you may want to review the section on “Commands Overview” on page 5-1 to gain a better understanding of how to use Silos commands.

## Commands In Files

### 7.1.1 Commands in Files

Silos commands can be entered in the Command window for the Main toolbar (for more information, see “Commands Overview” on page 5-1) or from a file. Commands entered from a file must be directly preceded (without any white space) by a “!” or a “.”. Preceding a command by “!” will cause the command to be echoed to standard output as it is executed. Usually, only the first two characters are required when specifying a command. A few commands (e.g. PRE, PRO) require three letters to prevent ambiguity. For more information on the commands available for Silos, see the “Silos Commands” on page 5-1.

An is shown below. For this example, file “test.v” will automatically simulate to the \$finish and report any error messages. Enclosing Silos commands with a “‘ifdef silos” compiler directive allows you to maintain Verilog compatibility (the keyword “silos” is defined as true by default in Silos).

File “test.v”:

```
//title simple circuit
module foo;
    reg clock;
    initial
        begin
            clock = 0;
            #10 clock = 1;
            #10 $finish;
        end
    endmodule
'ifdef silos
    !sim
    !errors
'endif
```

(continued on next page)

## Commands In Files

Commands are executed immediately upon being encountered in the data. Therefore, the order in which the commands are placed may be important (e.g., !PREP before !SIM ).

You should use the ‘include compiler directive when inputting a file from another file. In the previous example, remove the Silos commands from the file “test.v” and put them in file “test1.v” (shown below) with an ‘include compiler directive to include file “test.v” (for additional information see ‘include in the Verilog HDL Reference on-line help file).

File “test1.v”

```
'include "test.v"
'ifdef silos
!sim 200
!errors
'endif
```

## Command-line Options

### 7.1.2 Command-line Options

You can use Verilog style command line arguments for Silos. The command line arguments can be entered from the Command Line Arguments box in the Project Settings dialog box (see “Project Settings Menu Selection” on page 3-17), or, from the command line if you are running a batch simulation. Available command line arguments are:

- -c: This option compiles the source files and then exits.
- -f file\_name: This option instructs Silos to get the command line arguments from a file. Silos has the ability to nest the command files. For example, command file “logicsim”,

```
silos -f logicsim
```

could contain the name of another command file “logicsim1” that has additional “-“ command line arguments.

SILOS style commands can be used in a command file by enclosing the command with double quotes -“!silos\_command”, i.e. -“!control .sav=2”.

- -k file\_name: This option saves the text that has been entered from standard input to a file.
- -l file\_name: This option writes the standard output from Silos to a log file. An “exit” or “quit” command must be encountered for Silos to terminate.
- -la : This option appends the standard output from Silos to a log file instead of over writing the log file, and also to standard output. This option must appear before the "-l <fn>" option.
- -r save\_file\_name: This option restores Silos to the last saved simulation state from a previous Silos save command.
- -s: This option causes Silos to enter the interactive mode after executing any commands that have been input to Silos.
- -u : This option converts every name to upper case.
- -v file\_name: This option specifies a library file name.
- -w : Specifying -w means that Silos will not display any warning messages.
- -y directory\_path: This option specifies a directory of library files.
- +libext+extension1+extension2...: This option names the file name extensions for library files in the directory specified by the “-y” option. An example of specifying the library path and extension would be:

```
-y c:\sse\examples\library +libext+.v
```

(continued on next page)

## Command-line Options

- `+libnonamehide`: This option causes Silos to read in the module and UDP definition names as they are written in the file without appending character strings.
- `+librescan`: Search all the library files again for undefined modules.
- `+libverbose`: This option prints information about the opening of files and the resolution of module and UDP definitions during the scanning of libraries.
- `+define+text_macro_name=macro_text` : This option allows you to specify `define macros from the command line. The “text\_macro\_name” is the macro identifier, and the “macro\_text” is the text substitution. Double quotes (“ ”) must be used around the macro\_text if the macro\_text contains whitespace. For example:

```
silos.exe +define+sdf=test.sdf
```

is equivalent to:

```
'define sdf test.sdf
```

and:

```
silos.exe +define+declare="reg a;"
```

is equivalent to:

```
'define declare reg a;
```

- `+delay_mode_distributed` command line argument specifies the distributed delay mode for all modules in the source description. This means that the distributed delays for gates connecting the module input to the module output will always be used as the pin-to-pin delay for the module input to output. For examples of distributed delays, see Chapter 13 on specify blocks in the Verilog LRM on-line help file.
- `+delay_mode_path` command line argument specifies the path delay mode for all modules in the source description. This means that the path delays specified in the specify blocks for delays from the module input to the module output will always be used as the pin-to-pin delay for the module input to output. For examples of path delays, see Chapter 13 on specify blocks in the Verilog LRM on-line help file.
- `+delay_mode_unit` command line argument sets all gate and specify block delays to one.
- `+delay_mode_zero` command line argument sets all gate and specify block delays to zero.
- `+inmdir+directory1+directory2...:` If Silos can not find a file name that is specified on the user’s ‘include in the current directory, then Silos will search the directories specified by the `+inmdir` command line option for the file.

(continued on next page)

## Command-line Options

- `+ignore_sdf_interconnect_delay`: specifies that SDF INTERCONNECT delays will not be used. This can be useful for reducing the runtime and memory usage for fault simulation.
- `+ignore_sdf_port_delay`: specifies that SDF PORT delays will not be used. This can be useful for reducing the runtime and memory usage for fault simulation.
- `+libcodecoverage`: This option enables code coverage for libraries.
- `+linecov`: This option enables code coverage for line reporting.
- `+mindelays`: This option selects the minimum delay specification for delays (**min:typ:max**).
- `+typdelays`: This option selects the typical delay specification for delays (**min:typ:max**). (Default: `+typdelays`)
- `+maxdelays`: This option selects the maximum delay specification for delays (**min:typ:max**).
- `+no_default_variables`: This option issues an error message when a variable is used without first specifying the type of variable that it is. Since "wire" is the default type for variables, this assists the user with finding variables that were automatically set to wire where a register was intended..
- `+nodoldisplay`: This option suppresses all messages from \$display, \$write, etc. system tasks to standard output. This can be used to prevent these messages from cluttering the log file during logic simulation.
- `+nolibfaults`: Automatically inserts `suppress\_faults and `enable\_portfaults, and `nosuppress\_faults and `disable\_portfaults around every module in a library file. The library file can be specified using the -y and -v command line options, the !library command, or the "Project Files" dialog box.
- `+no_pulse_msg`: The command line option “`+no_pulse_msg`” turns off the `+pulse` messages. This does the same thing as the “`+pulse_quiet`” command line option.
- `+notimingchecks`: This option disables all timing checks, improving speed and reducing memory used.
- `+no_tchk_msg`: This option suppresses timing check violation messages. Timing checks are still processed, but no messages are printed to standard output if there is a timing check violation.
- `+nowarntfmpc`: This option suppresses the warning message for a mismatch in the number of port connections.

(continued on next page)

## Command-line Options

- **+oprcov:** This option enables code coverage for operator reporting. This option may slow simulation speed, you may not want to use it unless you wish to obtain a code coverage report for operators.
- **+plusargs:** You can enter “+” command-line arguments that are project specific, such “+compare”, “+sdf”, .etc. For example, suppose you wanted to specify the SDF file only when you entered “+sdf” in the “plusargs” box for the Project Settings dialog box. Then your test bench may look like:

```
module test_bench;
initial
    if ( $test$plusargs( "sdf" ))
        $sdf_annotate("test.sdf"); // only execute if "+sdf" is an argument
endmodule
```

- **+pulse\_r/<n>** and **+pulse\_e/<n>** command line arguments specify a range of pulse widths that will propagate to the path destination. For **+pulse\_r/<n>**, “n” specifies a number in the range 0-100. This will reject any pulse whose width is less than “n” percent of the module path delay. For **+pulse\_e/<n>**, “n” specifies a number in the range 0-100. This will flag as an error and drive unknown (“x”) any path pulse whose width is less than “n” percent of the module path delay, but whose width is greater than **pulse\_r/<n>**. For more information, see PATHPULSE\$ in the IEEE 1364 Verilog HDL manual.
- **+pulse\_quiet** command line argument suppresses warning messages generated by **pulse\_e** command line argument.
- **+suppressredef:** This option will suppress the warning message for redefinition of `define macros.
- **+suppressfloat:** This option will suppress the warning message for floating nodes, which may be caused by a gate input not having a driver, or by declaring a variable as a wire and then never assigning a value to it.
- **+timing\_checks:** This option turns on all timing checks for fault simulation. This will slow down the fault simulation and increase the memory used by fault simulation. Note: “**+no\_tchk\_msg**” turns off the printed message when there is a timing violation.
- **+width\_mismatches:** This option will issue a warning message if the widths for variables in an expression or assignment are inconsistent, i.e.: .

```
wire [3:0] a4;
wire [2:0] a2;
assign a4 = a2 & a4; // caught
assign a4 = a2; // caught
assign a4 = a2 & a2; // caught
```

(continued on next page)

## Command-line Options

- `+xl_order` specifies a switch so that the order of evaluation for always blocks is the same order as for Verilog-XL. This switch may be useful for obtaining the same simulation results as Verilog-XL. This option must be parsed before any modules are parsed. This option automatically enters ``define xl\_order 1''. An example would be the order of evaluation for:

```
always @posedge clock .....
```

```
always @posedge clock .....
```

Silos also supports the following command-line option::

- `-nospec`

The `-nospec` command-line option eliminates all specify blocks (see "Windows Batch Execution" on page 7-10 and "Unix Batch Execution" on page 7-12 for specifying the `-nospec` option). Eliminating the specify blocks will reduce the memory used and increase the simulation speed. However, eliminating the specify block delays may cause race conditions and non-convergence due to zero delays. If this happens, the rise and fall delays for all gates (whose delays are not explicitly specified) can be set to "1" with the following Silos command:

- `-“!delay .default =1,1”`

Silos also allows system commands to be entered from the command line, i.e. `-“!system “ls -lt””`  
**(continued on next page)**

## Command-line Options

For library searching, Silos also supports the `uselib compiler directive. The format for `uselib is:

```
'uselib file=filename dir=directory_name
```

where:

filename is the full path name for a file containing one or more module definitions that are searched to complete unresolved instantiations.

directory\_name is the full path name for a directory of files whose names are a concatenation of the name of a module definition and a file extension, such as “dff.v”.

Some examples of `uselib are:

The below example uses `define to specify macros for the `uselib. This makes it easier to change the library paths.

```
`define asic1 dir=c:\actel\lib\vlog libext=.v  
`define asic2 file=d:\library\udp.v  
'uselib `asic1 `asic2
```

The below example uses specifies the same `uselib without using a `define. Notice that the “libext” keyword for the library file name extensions is required when specifying a directory “dir” specification for a directory of library files.

```
'uselib file=\test\lib\udp.v dir=\test\lib2 libext=.v
```

## Windows Batch

### 7.1.3 Windows Batch Execution

Silos can be run as a batch execution from the Windows 2000, Windows 98, and Windows NT, operating systems.

The command-line syntax for running Silos as a batch execution on the Windows 2000, Windows 98, and Windows NT operating system is:

```
silos.exe -options +plusargs filename1 ... filenamen -!command1 ... -  
!commandn
```

where:

“**silos.exe**” is the path to the “silos.exe” executable on Windows.

“**-options**” is one or more Verilog HDL style command-line options (for more information, see "Command-line Options" on page 7-4). An example of using command line options would be:

```
silos.exe -v exam1.udp -v exam1.lib -y library +libext+.v exam1.v exam1.tst
```

For the above example, Silos will scan library files exam1.udp and exam1.lib and the “.v” files in directory library, and then input files exam1.v and exam1.tst. Then Silos will automatically simulate the circuit, report any errors, and exit when there are no further commands to be executed from the command line or from a file (the “sim”, “error”, and “exit” commands do not have to be specified).

The above examples could also have used the “-f” command-line option to specify the file that has the command-line options. For example:

```
silos.exe -f design.vc
```

File “design.vc” would then contain the following commands and file names for the above example:

```
-v exam1.udp  
-v exam1.lib  
-y library  
+libext_.v  
exam1.v  
exam1.tst
```

“**+plusargs**” is one or more “+” arguments for the \$testplusargs system task in Verilog HDL.

(continued on next page)

## Windows Batch

### (Windows Batch Execution)

“filename<sub>1</sub> ... filename<sub>n</sub>” is the name of one or more input files for Silos. The files can contain Verilog HDL at the behavioral, gate, and switch levels. The files can also contain Silos commands. Any Silos commands in the files will be executed as they are encountered.

“-!command<sub>1</sub> ... -!command<sub>n</sub>” is one or more optional Silos commands. For information on how to use Silos commands in a file, which may be simpler than from the command line, see “Commands in Files” on page 7-2. The “!” is required for all Silos commands that are on the command line. There must be no whitespace between the “!” and the Silos command. Silos commands that contain an embedded space must be enclosed by quotes, such as:

-“!bat exam1.log”

## Unix Batch

### 7.1.4 Unix Batch Execution

Silos can be run as a batch execution from the Unix operating system.

The command-line syntax for running Silos as a batch execution on the Unix operating system is:

```
silos -options +plusargs filename1 ... filenamen -\!command1 ... -\!commandn
```

where:

“**silos**” is the path to the “silos” executable on Unix. If you are running from a directory other than the installation directory you can set a link to silos on Unix:

```
ln -s installation_path/silos
```

“**-options**” is one or more command-line options supported by Silos (for more information, see "Command-line Options" on page 7-4). An example of using command line options would be:

```
silos -v exam1.udp -v exam1.lib -y library +libext+.v exam1.v exam1.tst
```

For the above example, Silos will scan library files exam1.udp and exam1.lib and the “.v” files in directory library, and then input files exam1.v and exam1.tst. Then Silos will automatically simulate the circuit, report any errors, and exit when there are no further commands to be executed from the command line or from a file (the “sim”, “error”, and “exit” commands do not have to be specified).

“**+plusargs**” is one or more “+” arguments for the \$testplusargs system task in Verilog HDL.

“**filename<sub>1</sub> ... filename<sub>n</sub>**” is the name of one or more input files for Silos. The files can contain Verilog HDL at the behavioral, gate, and switch levels. The files can also contain Silos commands. Any Silos commands in the files will be executed as they are encountered.

(continued on next page)

## Unix Batch

**“-\\!command<sub>1</sub> ... -\\!command<sub>n</sub>”** is one or more optional Silos commands. For information on how to use Silos commands in a file, which may be simpler than from the command line, see “Commands in Files” on page 7-2. The “\\!” is required for all Silos commands that are on the command line, however, the “!” has a special meaning on UNIX and must be escaped as “\\!”. There must be no whitespace between the “!” and the Silos command. Silos commands that contain an embedded space must be enclosed by quotes, such as:

**-”\\!batch exam1.log”**

In Unix, there is an additional method for running Silos in the batch mode that is similar to the interactive mode. To setup a batch session, edit a file and enter the same Silos commands as you would have used for an interactive session. Next, submit the file as a batch run on your computer. The following file would run Silos as a batch session on a UNIX operating system using “C” shell:

```
#/bin/csh
silos << mark
    batch exam1.log // redirects standard output to file "exam1.log"
    library exam1.lib exam1.udp library{.v}
    input exam1.v exam1.tst
    sim
    disk run01.out
    store probe q[4:1]
    exit
mark
```

## 8. Index

### \$

\$resetstarts save system task 4-23  
\$stop save system task 4-23

### ,

`celldefine  
  saving variables 3-19  
'define  
  Project Setting menu 3-17  
'uselib 7-9

### +

+define command line option 7-5  
+delay\_mode\_distributed command line option 7-5  
+delay\_mode\_path command line option 7-5  
+delay\_mode\_unit command line option 7-5  
+delay\_mode\_zero command line option 7-5  
+ignore\_sdf\_interconnect\_delay command line option 7-6  
+ignore\_sdf\_port\_delay command line option 7-6  
+inmdir command line option 7-5  
+libcodecoverage command line option 7-6  
+libext command line option 7-4  
+libnonamehide command line option 7-5  
+librescan command line option 7-5  
+libverbose command line option 7-5  
+linecov command line option 7-6  
+maxdelays command line option 7-6  
+mindelays command line option 7-6  
+no\_default\_variables command line option 7-6  
+no\_pulse\_msg command line option 7-6  
+no\_tchk\_msg command line option 7-6  
+nodoldisplay command line option 7-6  
+nolibfaults command line option 7-6  
+notimingchecks command line option 7-6  
+nowarntfmpc command line option 7-6  
+oprccov command line option 7-7  
+pulse\_e command line option 7-7  
+pulse\_quiet command line option 7-7  
+pulse\_r command line option 7-7  
+suppressfloat 7-7  
+suppressredef 7-7  
+timing\_checks command line option 7-7  
+typdelays command line option 7-6  
+width\_mismatches command line option 7-7  
+xl\_order command line option 7-8

## A

abort  
  Escape key 3-2  
  STOP button 3-2  
activity  
  Activity menu selection 3-30  
  Activity Report For Nodes 5-2  
Add One Bit 3-74  
Add Signal/Expression menu 3-65  
Add Signals to Analyzer menu selection 3-58  
Add Zero Bit 3-74  
Add/Remove Breakpoint menu 3-77  
analog behavioral modeling  
  analog extensions 4-20  
  analog waveforms 2-101, 3-51  
  Tutorial example 2-100  
Analog Integer Display menu 3-45  
Analyzer Symbol Table File 3-17  
Arrange Icons menu 3-47  
ASCII radix 2-26

## B

batch execution 7-1  
  Unix 7-12  
  Windows 7-10  
batch logic simulation example 2-95  
Black Background menu 3-45, 3-46  
blank lines  
  Data Analyzer 3-74  
bookmark  
  Tutorial example 2-43  
Break Simulation menu 3-40  
breakpoints 2-53  
  Breakpoints menu 3-42  
bundling  
  creating buses and vectors in Data Analyzer 2-30  
bus  
  BUSCON command  
    syntax definition 5-10  
bus  
  contention report 5-10  
  creating in Data Analyzer 2-30

## C

-c command line option 7-4  
Cancel button  
  definition 3-2  
Cascade menu 3-47  
case  
  -u command line option 7-4  
ccexport command  
  syntax definition 5-12

CCMEXCLUDE command  
     syntax definition 5-13  
 ccmkeep command  
     syntax definition 5-14  
 celldefine  
     saving variables 5-18  
 CHGLIB command  
     syntax definition 5-15  
 Clear Signal List menu 3-75  
 click-on  
     explanation 2-4  
 Close button definition 3-2  
 Close menu 3-15  
 code coverage  
     enable libraries 3-18  
     example 2-55  
     excluding module instance code coverage 5-13  
     exporting results 5-12  
     line example 2-55  
     merge example 2-64  
     operator example 2-60  
     saving code coverage by module instance 5-14  
 Code Coverage 3-30  
 Code Coverage explanation 3-21  
 color  
     signal strength 3-50  
 color coding  
     Data Analyzer 3-75  
 colored traces  
     Data Analyzer 3-75  
 colored waveforms 2-15  
 command line  
     Unix 7-10, 7-12  
 command line arguments 7-4  
 Command Line Arguments  
     GUI 3-17  
     relative path names 3-17  
 command line options 7-8  
 commands  
     input  
         from file 7-2  
     input from the command line menu 5-1  
 comments  
     respecifying comment character 5-16  
 common logarithm function 4-21  
 conditional search 2-33  
 context menus  
     main menus 3-57  
 continuous assignment  
     registers 4-26  
 Control command  
     syntax definition 5-16  
 CONTROL command  
     .COM option 5-16  
     .CUSTREPORT option 5-16  
     .DISABLECACHE option 5-16  
     .DISK option 5-16  
     .DMAX option 5-16  
     .DMIN option 5-16  
     .EUNK option 5-16  
     .HCHAR option 5-16  
     .HIST option 5-16  
     .MXDCI option 5-16  
     .MXITR option 5-16  
     .NONCON option 5-16  
     .NSCHAR option 5-16  
     .RECIRCULATE option 5-16  
     .SAVCELL option 5-16  
     .SAVSIM option 5-16  
     .SYNONYM option 5-16  
     .TPS option 5-16  
 convergence 3-32  
 Copy Image to Clipboard menu 3-8  
 Copy menu 3-6, 3-76, 3-78  
 Copy Scope menu selection 3-61  
 cosine function 4-21  
 custom report savefile 5-16  
 Cut menu 3-6, 3-76

**D**

Data Analyzer  
     Add Bookmark 3-69  
     analog & digital signals 3-51  
     copy waveforms to Word 2-19  
     Delete Bookmark 3-69  
     different simulations 3-52  
     Fonts menu 3-44  
     Goto Timepoint 3-67  
     memory caching 5-16  
     multiple copies 3-52  
     No Saved Data 3-53  
     Print menu 3-4  
     Scope 3-49  
     signal thickness 3-50  
     strength color codes 3-50  
     Time Scale Dialog Box 3-68  
     Trace Signal Inputs 3-70  
     Value 3-49  
 Data Analyzer menu 3-49  
 Data Tip Radix menu 3-77  
 data tips 2-45  
 Data Tips menu 3-46, 3-77  
 debug  
     Break Simulation menu 3-40  
     Breakpoints menu 3-42  
     Data Analyzer menu 3-49  
     Display menu 3-48  
     Enable Single Step/Breakpoints menu 3-39  
     Explorer menu 3-48

Finish Current Timepoint menu 3-41

Go menu 3-39

Restart Simulation menu 3-41

Step menu 3-41

Debug menu 3-39

delay

    command

        syntax definition 5-21

delay

    default specification 5-21

    maximum/minimum delay command 5-17

    preprocessing calculation 5-34

delay

    min

        typ

            max setting 3-18

Delete Group 2-29, 3-73

Delete Item/s menu 3-75

Delete menu 3-8

digital signals

    Data Analyzer 3-51

disable cache

    Project Setting menu 3-18

disk

    command

        syntax definition 5-22

Display Iteration Data 3-71

Display menu 3-48

## E

Edit menu 3-6

Enable Code Coverage for library modules menu 3-18

Enable Code Coverage menu 3-21

Enable Log File menu 3-18

Enable Operator Coverage menu 3-22

Enable Single Step/Breakpoints menu 3-39

encryption

    CHGLIB command 5-15

error

    command

        syntax definition 5-23

    Error menu selection 3-31

    preprocessing check 5-34

    reporting 5-23

    reset all errors 5-38

    turn off port connection warning 7-6

    Tutorial example 2-97

exclude

    command

        syntax definition 5-24

    excluding module instance node values 5-28

    excluding simulation node states 5-24

exit

    command

syntax definition 5-25

Exit menu 3-5

expected values 4-6

Explorer

    Add Signals to Analyzer 3-58

    Explorer menu 3-37

    Go to Definition 3-58

    Go to Module Source menu 3-38

    Open Explorer menu 3-37

    Explorer menu 3-48

    Explorer Window 2-12

    exponential function 4-21

    Export menu 3-4

    Expression Mode menu 3-29

    expressions

        Data Analyzer 3-70, 3-74

        Watch Window 3-65

    extensions

        save file size 4-23

        Verilog HDL extensions 4-24

    extensions to Verilog HDL 3-18

## F

-f command line option 7-4

Fault report 3-31

fault simulation

    Activity menu selection 3-30

    activity report 5-2

    fault exclusion 5-2

file

    FILE command

        syntax definition 5-26

file

    file name specification 5-26

    output naming 5-22

File menu 3-3

Files menu 3-13

filter

    expressions 3-58

Filters menu 3-20

Finish Current Timepoint menu 3-41

Finite State Machine 2-66

FLEXlm license manager

    PC 1-4

floating license

    computer name on PC 1-5

    host name on Linux 1-12

    host name on Unix 1-9

    hostid on Linux 1-12

    hostid on PC 1-5

    hostid on Unix 1-9

    PC to UNIX 1-5

floating node messages

    Project Setting menu 3-18

Fonts menu 3-44  
 Full Path Title menu 3-46  
 Functional Simulation menu 3-18  
 functions  
     global 4-24  
     multiple outputs 4-25  
     no inputs 4-25  
     ports 4-25

**G**

gate  
     MOS  
         Unknown levels on enables 5-17  
     XFR  
         Unknown levels on enables 5-17  
 gate  
     strength specification 5-46  
 global  
     functions 4-24  
     tasks 4-24  
     variables 4-24  
 Go menu 3-39  
 Go to Definition menu selection 3-58  
 Go to Instance menu 3-77  
 Go to Module Source menu 3-38  
 Go To Scope menu 3-61  
 Goto Timepoint  
     Tutorial example 2-43  
 Goto Timepoint menu 3-67  
 group  
     creating buses and vectors in Data Analyzer 2-30  
 groups in Data Analyzer 2-28, 3-72

**H**

hanging  
     behavioral level designs 3-34  
 HDL Code Coverage 3-30  
 help  
     on-help 2-7  
 Help menu 3-55  
 hyperbolic cosine function 4-21  
 hyperbolic sine function 4-21  
 hyperbolic tangent function 4-21

**I**

I/O Pad  
     stimulustable 4-13  
 implicit node  
     created during preprocessing 5-34  
 input  
     commands input from file 7-2  
     commands within the data file 7-2

input file order 7-3  
 relative paths to files 3-14  
 symbol specification for state values 5-48  
 Insert Group 2-29, 3-73  
 installation  
     requirements 1-2  
     windows 1-2  
 Instance Name Filter menu 3-62  
 integer and real waveform display 3-51  
 interactive debug  
     report 5-35  
     scope 5-40  
 inverse cosine function 4-21  
 inverse sine function 4-21  
 inverse tangent function 4-21  
 Iteration menu selection 3-31

**K**

-k command line option 7-4  
 KEEP command  
     syntax definition 5-27  
 Keeping simulation node states 5-27

**L**

-l command line option 7-4  
 -la command line option 7-4  
 library  
     example 3-13  
     file encryption 5-15  
     overview 6-1  
     specify from menus 3-13  
     TTL BCT parts 6-13  
     TTL LS parts 6-4  
 license  
     computer name on PC 1-5  
     host name on Linux 1-12  
     host name on Unix 1-9  
     hostid on Linux 1-12  
     hostid on PC 1-5  
     hostid on Unix 1-9  
     PC node locked 1-3  
     PC to UNIX 1-5  
 license manager  
     PC 1-4  
 Linux installation 1-11  
 Load/Reload Input Files menu 3-16  
 lockup  
     behavioral level designs 3-34  
 log file  
     enable/disable 3-18  
 logic initialization  
     maximum iteration limit 5-17  
 logic simulation

batch example 2-95  
 control specification 5-16  
 excluding selected modules for code coverage 5-13  
 excluding selected nodes 5-24  
 excluding selected nodes by module 5-28  
 exporting code coverage results 5-12  
 functional simulation 3-18  
 iteration limit 5-17  
 maximum/minimum delay command 5-17  
 node synonym names 5-18  
 preprocessing 5-34  
 renaming save file 5-26  
 running logic simulation 5-41  
 save file limit 5-16  
 save file reset 5-38  
 save file size 5-18  
 save variables in 'celldefine' 5-18  
 smaller save file example 2-90  
 specification 5-41  
 spike report 5-44  
 Unknown levels on enables 5-17  
 xl\_order command 5-19  
 xl\_order command line option 7-8

**M**

math functions 4-21  
 Maximum File Size 3-18  
 memory  
   size of RAM on PC 1-5  
   size report 5-43  
   stimulustable 4-11  
   usage display 2-7  
 memory available on Unix 1-10  
 memory usage  
   Data Analyzer caching 5-16  
 menus  
   Analyzer Toolbar 3-10  
   Arrange Icons menu 3-47  
   Break Simulation menu 3-40  
   Breakpoints menu 3-42  
   Cascade menu 3-47  
   Close menu 3-15  
   Code Coverage 3-21  
   context menus 3-57  
   Data Analyzer menu 3-49  
   Debug menu 3-39  
   Display menu 3-48  
   Edit menu 3-6  
   Enable Code Coverage 3-21  
   Enable Operator Coverage 3-22  
   Enable Single Step/Breakpoints menu 3-39  
   Explorer menu 3-37, 3-48  
   Expression Mode menu 3-29  
   File menu 3-3

File/Exit menu 3-5  
 File/Export menu 3-4  
 File/Open menu 3-3  
 File/Print menu 3-4, 3-5  
 File/Print Preview menu 3-5  
 File/Save As menu 3-4  
 File/Save menu 3-3  
 Files menu 3-13  
 Filters menu 3-20  
 Finish Current Timepoint menu 3-41  
 Fonts menu 3-44  
 FSM Toolbar 3-10  
 Go menu 3-39  
 Go to Module Source menu 3-38  
 Help menu 3-55  
 Load/Reload Input Files menu 3-16  
 Main Toolbar 3-10  
 New menu 3-12  
 New menu  
   State Machine 3-28  
 Note Mode menu 3-29  
 Open Explorer menu 3-37  
 Open menu 3-13  
 Open menu  
   State Machine 3-28  
 Options menu 3-44  
 Project List Size menu 3-20  
 Project menu 3-12  
 Project Settings menu 3-17  
 Reload and Go menu 3-16  
 Reports menu 3-30  
 Restart Simulation menu 3-41  
 Save As menu 3-15  
 Save As menu  
   State Machine 3-28  
 Save menu  
   State Machine 3-28  
 Save Project State menu 3-15  
 Select Files to Merge 3-22, 3-23  
 Select Mode menu 3-28  
 State Machine 3-27  
 State Mode menu 3-29  
 Status Bar 3-11  
 Step menu 3-41  
 Tabs menu 3-44  
 Tile menu 3-47  
 Title Tips menu 3-45  
 Transition Mode menu 3-29  
 View menu 3-9  
 Window menu 3-47  
 zoom selections 3-9  
 MEXCLUDE command  
   syntax definition 5-28  
 min : typ  
 max

Project Setting menu 3-18

min:typ:max

command 5-17

MKEEP command

syntax definition 5-29

## N

Name Filter menu selection 3-58

Name List Box

resizing 2-18

natural logarithm function 4-21

netlist

preprocessing 5-34

size 5-43

New Group 2-29, 3-72

New menu 3-3, 3-12

New menu

State Machine 3-28

no convergence 3-32

no saved data

project settings menu 3-19

No Saved Data 3-53

NOCONV command

syntax definition 5-30

nonconvergence

behavioral level designs 3-34

choosing a possible solution 5-17

command for report 5-30

gate level designs 3-32

initialization iterations 5-17

iteration limit 5-17

Nonconvergence menu selection 3-32

-nospec command line option 7-8

Note Mode menu 3-29

NSTORE command 5-22

syntax definition 5-33

## O

OK button

definition 3-2

Open Explorer menu 3-37

Open menu 3-3, 3-13

Open menu

State Machine 3-28

Options menu 3-44

order of execution switch 3-19

output

file naming 5-22

saving additional nodes 5-27

saving additional nodes by macro 5-29

symbol specification for state values 5-48

## P

Pan to Last View menu 3-67

Pan to T1 menu 3-67

Pan to T2 menu 3-67

Paste menu 3-7, 3-76

PLI on PC 4-1

PLI on Unix 4-2

PLI overview 4-1

plusargs

command line option 7-7

Project Settings menu 3-19

ports

set missing ports to gnd 5-19

power function 4-21

preprocess

PREPROC command

syntax definition 5-34

Print menu 3-4

Data Analyzer 3-4

Print Preview menu 3-5

Print Setup menu 3-5

probe

syntax definition 5-35

probing node states 5-35

project

Filters menu 3-20

Project List Size menu 3-20

Project List Size menu 3-20

Project menu 3-12

Project Settings menu 3-17

Properties menu 3-63

## Q

Quitting execution 5-37

## R

-r command line option 7-4

race conditions

Display Iteration Data 3-71

radix

ASCII 2-26

Data Analyzer 3-73

symbol names 2-22

real and integer waveform display 3-51

rearranging signal names 2-20

recirculating save file for batch 5-17

recirculating save file for GUI 3-18

regular expressions 3-58

relative path names 3-17

relative paths to files 3-14

Reload and Go menu 3-16

Reload Groups 3-75

reports  
 default state symbols 5-48  
 narrow store to disk file 5-33  
 nonconvergence 5-30  
 Reports menu 3-30  
 size information 5-43  
 spike report 5-44  
 stopping a report 5-1  
 storing wide output to disk file 5-45  
 symbol definition 5-48  
 reset  
 command 5-38  
 syntax definition 5-38  
 errors and warnings 5-38  
 everything (all) 5-38  
 preprocessing errors 5-34  
 Restart Simulation menu 3-41  
 Restore Project State menu 3-15  
 Retain simulation data file 3-19  
 Reverse Bit Order 3-74

**S**

-s command line option 7-4  
 save  
 automatically delete simulation history file 3-19  
 by hierarchy 3-63  
 control command saving 5-18  
 recirculating save file for batch 5-17  
 recirculating save file for GUI 3-18  
 save all variables 3-19  
 save file path 3-19  
 saving node states 5-27  
 saving node states by macro 5-29  
 simulation results 5-25  
 variables in `celldefine 3-19  
 save  
 turn-off, turn-on, reset 4-23  
 Save As menu 3-4, 3-15  
 Save As menu  
 State Machine 3-28  
 save file  
 renaming 5-26  
 resetting simulation results 5-38  
 save file  
 turn-off, turn-on, reset 4-23  
 save file size 3-18  
 save file size  
 smaller 2-90  
 Save menu 3-3  
 Save menu  
 State Machine 3-28  
 Save Project State menu 3-15  
 scanning to edge 2-37  
 SCOPE  
 syntax definition 5-40  
 Scope for Data Analyzer 3-49  
 scoping node states 5-40  
 SDF  
 \$sdf\_annotate system task 4-4  
 search on condition  
 Tutorial 2-33  
 security code  
 computer name on PC 1-5  
 host name on Linux 1-12  
 host name on Unix 1-9  
 hostid on Linux 1-12  
 hostid on PC 1-5  
 hostid on Unix 1-9  
 license manager PC 1-4  
 node locked 1-3  
 PC to UNIX 1-5  
 sharing node locked 1-3  
 Select Files to Merge menu 3-22, 3-23  
 Select Mode menu 3-28  
 Show Groups 2-29, 3-73  
 signal  
 color 3-50  
 Signal List Box  
 rearranging names 2-20  
 signal names  
 hierarchical expansion of name 3-45  
 Silos  
 analog behavioral modeling 2-100, 2-101  
 blank lines in Data Analyzer 3-74  
 breakpoints 2-53  
 errors 2-97  
 Explorer Window 2-12  
 expressions in Data Analyzer 3-74  
 radix in Data Analyzer 3-73  
 scanning to edge 2-37  
 Simulation Environment 1-1  
 single stepping 2-50  
 Toolbar  
 breakpoints 2-53  
 single stepping 2-50  
 Toolbar text 2-4  
 Toolbar zoom buttons 2-40  
 silos keyword 4-23  
 SIMULATE command  
 syntax definition 5-41  
 simulation  
 saving results 5-25  
 Simulation Data File Path 3-19  
 sine function 4-21  
 single stepping 2-50  
 size limits  
 report 5-43  
 SIZE command  
 syntax definition 5-43

Size menu selection  
     memory usage 3-36  
 snap to edge  
     menu selection 3-44, 3-68  
 Sort by Name menu selection 3-61  
 Sort by Type menu selection 3-61  
 specify blocks  
     eliminating specify blocks 7-8  
     variable delays 4-28  
 spike  
     output report 5-44  
 SPIKES command  
     syntax definition 5-44  
 square root function 4-21  
 sse keyword 4-23  
 State Machine  
     Expression Mode menu 3-29  
     Note Mode menu 3-29  
     Select Mode menu 3-28  
     State Mode menu 3-29  
     Transition Mode menu 3-29  
 State Machine menu 3-27  
 state machine symbols 3-17  
 State Mode menu 3-29  
 Status Bar  
     View menu 3-11  
 Step menu 3-41  
 stimulus  
     converting behavioral to tabular values 4-18  
     inputting tabular values 4-6  
 stimulustable 4-6, 4-7  
     extension setting 4-29  
     I/O Pad 4-13  
 stop  
     a process (Ctrl-C) 5-1  
 STORE command 5-22  
     syntax definition 5-45  
 Storing outputs 5-33, 5-45  
 strength  
     mixed signal strength 3-50  
 strength  
     default specification for gates 5-46  
 Strength Color Coding menu 3-45  
 STRENGTH command  
     syntax definition 5-46  
 SYMBOL command  
     syntax definition 5-48  
 symbol names for vectors 2-22  
 Syntax Color Coding menu 3-46

**T**

Tabs menu 3-44  
 tabular outputs  
     probe command 5-35

tangent function 4-21  
 tasks  
     global 4-24  
     ports 4-25  
 technical support  
     how to access 1-14  
 test pattern  
     activity report 5-2  
 Tile menu 3-47  
 time  
     Add Bookmark 3-69  
     Delete Bookmark 3-69  
     Goto Timepoint 3-67  
     Time Scale dialog box 3-68  
 Timescale dialog box 2-43  
 timing checks  
     turn off messages 7-6  
     turn off timing checks 7-6  
     turn on timing checks for fault simulation 7-7  
 Title Tips menu 3-45  
 Toolbar  
     Analyzer Toolbar 3-10  
     explanatory text 2-4  
     FSM Toolbar 3-10  
     Main Toolbar 3-10  
 trace  
     Trace Signal Inputs 3-70  
 Trace Color Coding menu 3-45  
 Trace Signal Inputs 2-81  
 transcendental math functions 4-21  
 Transition Mode menu 3-29  
 TTL library  
     BCT parts 6-13  
     LS parts 6-4

**U**

-u command line option 7-4  
 UDP  
     default value 4-27  
     High-Z 4-27  
 Undo menu 3-6, 3-76  
 Unset state symbol 5-50  
 upper case  
     -u command line option 7-4  
 uselib 7-9  
 User Registration menu 3-56

**V**

-v command line option 7-4  
 Value for Data Analyzer 3-49  
 variables  
     displaying values in text window 2-45  
     global 4-24

vector  
    creating in Data Analyzer 2-30

vectors  
    ASCII radix 2-26  
    expanding and hiding bits 2-21  
    symbol names 2-22

Verilog HDL  
    extensions  
        analog 4-20  
        expected values 4-6  
        Silos extensions 4-24  
        stimulustable 4-6, 4-7  
        tabular stimulus 4-18  
        save file size 4-23

Verilog HDL extensions  
    enabling 3-18

View menu 3-9

## **W**

-w command line option 7-4

warning  
    +suppressfloat 7-7  
    +suppressredef 7-7  
    reset all warnings 5-38  
    turn off port connection warning 7-6

Watch Window 3-65

waveform color 2-15

waveforms  
    color and signal strength 3-50  
    copy to Word 2-19  
    signal thickness 3-50

width mismatches  
    in expressions and assignments 7-7

Window menu 3-47

wire  
    procedural assignment 4-26

## **X**

xl\_order command 5-19  
xl\_order command line option 7-8

## **Y**

-y command line option 7-4

## **Z**

zoom  
    buttons 2-40  
    View menu 3-9

# safariexamples.informit.com - /0130449113/Help/

---

[\[To Parent Directory\]](#)

Wednesday, May 23, 2001 2:40 PM

5446878 [sse.hlp](#)

---



DESIGN AUTOMATION

[Site Map](#) | [Log-In](#) | [Contact](#)
[Corporate](#) | [Products & Solutions](#) | [Customers & Partners](#) | [Resources & Support](#) | [News & Events](#)

# The Leader in Assertion-Based Verification

**Verification Resources:**

- > Request web based demos and evaluation licenses
- > Verification white papers and other tools


[Register for quick access.](#)
**Customers:**
**business partner**

**Contact Us**

To learn how 0-In's ABV products eliminate bugs

**800-919-3640**


Verification Process Automation

▷ **Resources:** > Designers > Press > Partners > Jobs

[Corporate](#) | [Products & Solutions](#) | [Customers & Partners](#) | [Resources & Support](#) | [News & Events](#) | [Legal Notices](#) | [Privacy](#) | [Site Map](#)

**See how 0-In's assertion-based verification (ABV) tool suite helps leading technology companies debug multi-million gate ASIC and SOC designs.**

**Announcements:**


**0-In** introduces Archer Verification™ system, a complete verification solution that is equal parts tools and engineered methodology for the most critical verification issues facing design teams developing multi-million gate ASIC and system-on-chip (SoC) devices.

> [Learn more about Archer Verification system](#)

> [See our V2.x partner quotes](#)

The following three people each won an Apple iPod mini at the 0-In drawing during DAC2004:

Jim Brewer, HP  
Steve Mecham, LSI Logic  
Jay Lee, Marvell

Congratulations!

**Highlights:**

August 24, 2004 0-In Welcomes FishTail Design Automation to its Check-In Partner Program

July 29, 2004 Carbon Design Systems Joins 0-In's Check-In Partner Program

June 7, 2004 Mentor Graphics Enters Into Agreement to Acquire 0-In Design Automation

June 7, 2004 0-In Announces SystemVerilog and VHDL Products

## Embedded Secure Document

The file *file:///C|/eMule/incoming/1qxl@N/Prentice%20Hall%20-%20Verilog%20HDL%20-%20A%20Guide%20To%20Digital%20Design%20And%20Synthesis,%202nd%20Edition%20(2004).pdf* is a secure document that has been embedded in this document. Double click the pushpin to view.

