Acquisitions Editor: Tim Cox
Assistant Editor: Laura Cheu
Production Editor: Lisa Weber
Marketing Coordinator: Anne Boyd
Manufacturing Manager: Casimira Kostecki
Cover Design and Illustration: Yvo Riezebos
Text Designer: Lisa Jahred
Copyeditor: Nick Murray
Proofreader: Holly McLean-Aldis

Camera-ready copy for this book was prepared using FrameMaker.

Library of Congress Cataloging-in-Publication Data

```
LogicWorks.
     Intermediate Programming and Problem Solving in C++
     name last name/name last name
         p. cm.
     Includes index
     ISBN X-XXXX-XXXX-X
     1. C (Computer program language) 2. Compilers (Computer programs)
I. Last name, first name II. Title
QXXX.XX.XXXXXX 1995
XXX.XXX--XXXXXX-XXXXX
                    CIP
```

ISBN 0-201-89585-4

1 2 3 4 5 6 7 8 9 10 CRS 00 99 98 97 96

Addison Wesley Longman, Inc.
2725 Sand Hill Road
Menlo Park, California 94025

LogicWorks Verilog Modeler

INTERACTIVE CIRCUIT SIMULATION SOFTWARE

FOR WINDOWS® AND MACINTOSH™

VERSION 3

ADDISON-WESLEY

An imprint of Addison Wesley Longman, Inc.

Menlo Park, California • Reading, Massachusetts • Harlow, England
Berkeley, California • Don Mills, Ontario • Sydney • Bonn • Amsterdam • Tokyo • Mexico City

Contents

Preface

Welcome to the LogicWorks Verilog Modeler (LVM) and the world of digital circuit design using hardware description languages! Hardware description languages (HDLs) are now important tools in the design of digital systems. In fact, a visitor to some corporate hardware engineering departments might assume that he had stepped into the Software Department by mistake! Many systems are now designed "behaviorally," that is to say, by describing the operation of the system in high-level terms, rather than by drawing a detailed logic diagram. Designers work with textual descriptions of a circuit that look a lot like software programs, rather than drawing classical circuit schematics.

Verilog is one of the two most widely used HDLs in industry, the other being VHDL. It is well established in the market, and various Verilog-compatible simulation and synthesis tools are available.

The goals of the LogicWorks Verilog Modeler are

- To introduce the student to the concepts of hardware description languages.
- To make it easier to create and test simulation models for a LogicWorks circuit.
- To provide an upward path to industry-standard tools that might be used in more advanced courses or applications.

It is important to note that the LVM does not implement all parts of the Verilog language. Only the behavioral parts of the language are implemented, whereas gate-level modeling, hierarchy, and other structural concepts are not supported in this version.

About This Manual

The purpose of this manual is to get you started using the package quickly and then provide reference information on technical issues and Verilog language support. We assume that you have a general familiarity with LogicWorks 3 operation and with the concepts of the Verilog language. This manual is *not* intended as a primer or application manual for Verilog. If you are new to LogicWorks, try working through at least the "Five-Minute Schematic and Simulation" section of Chapter 4 in the LogicWorks 3 manual.

Acknowledgments

Many people at Capilano Computing and at Addison-Wesley Publishing provided invaluable help in bringing this version of LogicWorks into the world. Particular thanks go out to Tim Cox and Laura Cheu at Addison-Wesley, to Don Gamble and Neil MacKenzie at Capilano, and to Pai Chou at the University of Washington, all of whom remained cheerful and fun to work with despite setbacks and schedule pressures.

Other key contributors include Susan Slater, Lisa Weber, and Anne Boyd at Addison-Wesley.

Special thanks go out also to our many friends and supporters in the academic and industrial worlds who continue to provide valuable feedback and support for the ongoing development of this product.

Chris Dewhurst
Vancouver, B.C., Canada
June, 1996

1 Introduction

Welcome to the LogicWorks Verilog Modeler! The LVM is an extension that runs with LogicWorks 3 and allows you to create device simulation models using a subset of the industry-standard hardware description language Verilog. Using the Verilog Modeler with LogicWorks 3, students and professionals can learn how to program, design, and test their own circuits with a current industry hardware description language. Once the LVM is installed, it becomes part of LogicWorks. There is no switching between applications and no new user interfaces to learn.

This highly interactive package allows you to step, modify, and debug your Verilog code without switching applications or waiting for long compiles. You can simply double-click on a part in the schematic to create or view the Verilog behavioral definition. Variables can be displayed interactively as the simulation progresses, allowing for easy debugging and evaluation. In addition to modeling physical components, the LogicWorks Verilog Modeler is ideal for creating stimulus and test programs for other LogicWorks circuits.

Features

- Fully Integrated Verilog Simulation—Just double-click on a schematic symbol to create and edit a simulation model in Verilog.
- Interactive Control Panels—Allow the user to display the internal variables of the Verilog model as the simulation progresses.
- Unlimited Execution—Any number of Verilog models can execute simultaneously in a single design.

- Flexible Program Output Options—Textual program output from compilation errors, execution errors, or `$display` statements in the Verilog source code can be redirected to the screen or to a text file.
- Fast Compile Times—Whenever you close a source window, or on command, the Verilog source code is compiled in seconds to executable form.
- Tool Sharing—Use LogicWorks Simulation Tools such as the Timing display to create and display simulation data.

How This Manual Is Organized

This manual is divided into two parts reflecting the two different facets of operation of the LVM package. Part I, “General Operation” looks at the mechanics of creating and editing Verilog models and using them in LogicWorks circuits. This part includes a tutorial chapter to help you get started, as well as reference sections on the various menu commands and technical issues.

To get started with the LVM, see Chapter 2 for help in installing the software, and then try out the tutorials in Chapter 3. These will get you up and running quickly. You can use the remaining chapters as a reference when you need specific details.

Part I is divided into the following chapters:

- Chapter 2, Program Installation—How to install the LogicWorks Verilog Modeler package.
- Chapter 3, Tutorial—A quick guide to the features of the package. All the basic procedures you need to get started are covered here.
- Chapter 4, Using the Complete Program—This chapter goes into detail on the interface between the LVM and the LogicWorks simulator, the usage of the various controls and options, and technical issues for advanced users.

Part II, “Verilog Language Support” describes in detail the subset of the Verilog Hardware Description Language implemented in this package.

This section is intended only as a reference and assumes that you have some general familiarity with the Verilog language and its constructs.

- Chapter 5, Structure of a Verilog File—Provides an overview of a Verilog module definition.
- Chapter 6, Language Syntax—Describes the low-level syntax of the Verilog language.
- Chapter 7, Data Types—Gives a complete description of the data types supported in the LVM.
- Chapter 8, Expressions and Assignments—Describes how data values are manipulated and assigned in Verilog.
- Chapter 9, Procedural Constructs—Describes the conditional, looping, and other control structures available.
- Chapter 10, Time Control: Delays and Events—Describes the Verilog delay and synchronization facilities available in the LVM package.
- Appendix A, Differences from OVI Verilog—Enumerates the differences between the full Verilog language and the subset implemented in the LVM.
- Appendix B, Verilog Keywords—Provides a listing of the reserved words in the full Verilog language.

In this manual, text with an arrowhead:

► like this

provides step-by-step instructions for achieving a specific goal. Other text provides background and explanation of the actions being taken.

Platform Differences

This manual applies to both the Macintosh and Windows versions of the LogicWorks Verilog Modeler. Although most of the information provided applies equally to both versions, there are occasional differences in operation. In these cases, you will see each platform described separately in a format like this:

Macintosh–Type ⌘-V to paste the text into the box.

Windows–Type CTRL-V to paste the text into the box.

To avoid constant interruptions for slight differences in terminology, we have been fairly relaxed about using terms like *folder* and *directory* interchangeably. Our apologies to the platform purists among our readers!

Notes Regarding Copyright

Part I
General Operation

This first part of the manual covers the installation and usage of the LVM package, including these topics:

- Installing the LVM package on your machine.
- Creating and editing Verilog models.
- How the LVM simulator works with the main LogicWorks simulator.
- The operation of the LVM controls and options.

Part I also includes the tutorial chapter that will lead you through most of the important procedures required in using the LVM.

2 Program Installation

This chapter gives you the information you need to install the LogicWorks Verilog Modeler package on your machine. After you have installed the package, we suggest that you proceed with the tutorials in Chapter 3. These will help get you up and running with a minimum of reading.

IMPORTANT: Please look for a ReadMe file on the installation diskette provided with this package. This file may contain important information that supersedes the procedures described here.

Although the software is very similar in operation on the Windows and Macintosh versions, installation procedures are quite different for the two systems. For that reason, the rest of this chapter is divided into separate sections for the two machines.

Installing the Macintosh Version

The LogicWorks Verilog Modeler **requires that LogicWorks 3 be already installed** for correct operation.

NOTE: Please see the ReadMe file on the installation disk for version compatibility information. Any attempt to use it with incorrect versions may give very unreliable results.

Quick Installation

The LVM package is easily installed by following these steps:

- Make sure you have the correct version of LogicWorks installed on your hard disk by running the program and consulting the About LogicWorks box. Compare the version number with the one specified in the ReadMe file on the LVM diskette. If you do not have the correct version, there may be an upgrade included with the LVM package, or there may be a free upgrade available over the Internet. Again, please refer to the ReadMe file for details.
- Insert the LogicWorks Verilog Modeler Installer diskette in your machine, if it is not already inserted.
- Double-click on the LVM Installer icon.
- When you are prompted to select the destination for the installer, select any convenient location on your hard disk and click Continue.
- When the installation is complete, locate the LVM Installation folder that was created by the installer and open it.
- Move the files LVM and XEditor to the Tools folder inside the main LogicWorks folder.
- Move the contents of the folder Demos to the Demos folder inside the main LogicWorks folder.
- If LogicWorks is already running, quit and restart the program. If not, just start it up.

Installation is now complete! Test the LVM by opening the circuit file V163.cct in the Demos folder and clicking on the switches. If the LVM is working, you should see a binary count sequence in the output displays. Double-click on the 163 device to display the LVM control panel. If you get the message “This device is not a sub-circuit type and cannot be opened”, or something similar, then the LVM module is not correctly installed.

Files Included in the Installation

This section describes the individual files included with the package. You should review this material if you have a specific need to create some unusual installation setup.

The following files are included:

LVM	This is the main LogicWorks Verilog Modeler tool and includes the compiler, simulator, and interactive control functions. It should be placed in the Tools folder inside the LogicWorks folder. It will be found and loaded automatically when LogicWorks starts up.
XEditor	This is a simple text editor for optional use with the Verilog Modeler. It should be placed in the Tools folder inside the LogicWorks folder. It will be found and loaded automatically when LogicWorks starts up. It is also possible to use an external text editing application and move the text between applications using the clipboard.
Demos	This folder contains some LogicWorks design and library files demonstrating the use of the Verilog Modeler. It can be installed inside the LogicWorks Demos folder or in any convenient location.

NOTE: The LVM tool *does not* appear in the Tools menu even when it is installed and running. Its functions are invoked by opening an associated device symbol.

Memory Usage

The LogicWorks Verilog Modeler occupies a small amount of memory space (about 200 K) regardless of whether it is being used or not. If you are operating on a system with a minimal amount of memory, it may be desirable to keep the module in another folder where it will not be found and loaded by LogicWorks. When it is actually needed, it can be moved temporarily into the LogicWorks folder.

During operation, the LVM module can use a significant amount of memory in creating the tables and simulation data required as part of the

compilation and simulation process. This memory requirement depends completely on the size and complexity of the Verilog source file.

Installing the Windows Version

The LogicWorks Verilog Modeler **requires that LogicWorks 3 be already installed** for correct operation.

NOTE: Please see the ReadMe file on the installation disk for version compatibility information. Any attempt to use it with incorrect versions may give very unreliable results.

Quick Installation

The LVM package is easily installed by following these steps:

- Make sure you have the correct version of LogicWorks installed on your hard disk by running the program and consulting the About LogicWorks box. Compare the version number with the one specified in the ReadMe file on the LVM diskette. If you do not have the correct version, there may be an upgrade included with the LVM package, or there may be a free upgrade available over the Internet. Again, please refer to the ReadMe file for details
- Insert the LogicWorks Verilog Modeler Installer diskette in your machine, if it is not already inserted.
- Locate and execute the file INSTALL.EXE.
- Follow the instructions on the screen for selecting an installation location.

Installation is now complete! Test the LVM by opening the circuit file v163.cct in the Demos directory and clicking on the switches. If the LVM is working, you should see a binary count sequence in the output displays. Double-click on the 163 device to display the LVM control panel. If you get the message “This device is not a sub-circuit type and cannot be

opened", or something similar, then the LVM module is not correctly installed.

Selecting a Text Editor

The LogicWorks Verilog Modeler package relies on having a text editor available for creating and modifying source code. If you don't make any other selection, operations that require a text editor will use the standard NotePad application. You can change this selection at any time using the Preferences command in the Options menu on the LVM control panel. See "Modifying an Existing File" on page 13 for instructions on opening the control panel.

Files Included in the Installation

This section describes the individual files included with the package. You should review this material if you have a specific need to create some unusual installation setup.

The following files are included:

lvm.mda	This is the main LogicWorks Verilog Modeler tool and includes the compiler, simulator, and interactive control functions. It should be placed in the Tools directory inside the main LogicWorks directory. It will be found and loaded automatically when LogicWorks starts up.
Demos	This directory contains some LogicWorks design and library files that demonstrate the use of the Verilog Modeler. It can be installed in any convenient location.

NOTE: The LVM tool *does not* appear in the Tools menu even when it is installed and running. Its functions are invoked by opening an associated device symbol by double-clicking on it.

3 Tutorial

This chapter takes you through the various steps involved in creating and running Verilog simulations using the LogicWorks Verilog Modeler. The following tutorial sections assume that you have installed the LVM package and have a general familiarity with LogicWorks. If you are new to LogicWorks, we suggest that you go through the Tutorial section (Chapter 4) in the LogicWorks 3 manual first.

Modifying an Existing File

To give you a quick introduction to LVM operation, we will first open and modify one of the demonstration files provided with the package. This will give you a place to start in working with your own designs.

- Start up LogicWorks in the usual way, if it is not already running.
- Open the design file named V163.cct in the LVMDemos directory.

This file demonstrates a Verilog model for a 74XX163 counter. Note that the outputs will enter an X state after opening. The Verilog Modeler does not store the internal state of the model with the design file. Only the source code is stored, and it is recompiled when the file is opened. Internal variables will always be reset to their initial states when a design file is opened. If nothing else is specified, this will be X (unknown).

Refer to "Compiling and Initialization" on page 38 for more information on variable initialization.

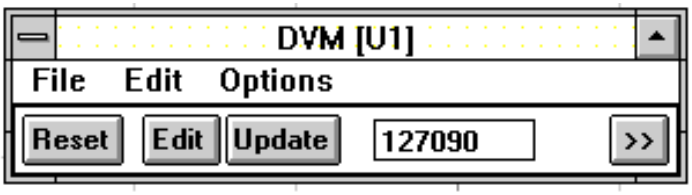
- Try clicking on the CLEAR and LOAD switches and verify that the model is working and the outputs are counting as expected.

Macintosh–Double-click on the 163 device. The LVM control panel will appear.

The pop-up menus on this control panel operate independently of the main program menus; that is, they do not affect any functions of the main program, and none of the main menus affect the LVM.

- Click on the Show Variables control.

Windows–Double-click on the 163 device. The LVM control panel will appear:



This control panel is used to view the internal state of the model, to gain access to the Verilog source code, and to control various options affecting the model.

- Click on the >> button to enlarge the window to display the variable values.

This control increases the size of the control panel and displays a table showing the current values of all internal variables declared in the Verilog source code. These values are updated as the simulation progresses. Note that values are displayed only for the particular device instance that is named in the panel's title bar. In this version there are no display radix or other options. For memory variables (i.e., arrays of `reg`), only the first location is displayed.

See "Debugging a Verilog Model" on page 39 for a description of variable display formats.

Macintosh–Click and hold on the File pop-up menu and select the Open Source to Text Window option. This command will open a new window using the XEditor tool and display the Verilog source code associated with this device type.

Windows–Select the Save Source to File then Edit command in the File menu. This command copies the source text (which is stored internally with the circuit file) to a standard text file and invokes the selected text editor. If you have not specified any other editor, Notepad will be used.

The Verilog source code is treated in much the same way as an internal circuit definition in LogicWorks. The Verilog source is associated with a specific *part type* and is the same for all instances of that type. Changing the source code and recompiling will affect *all devices of the same type*, that is, all devices derived from the same library part.

- Scroll the text window down if necessary to locate the statement close to the bottom of the file:

```
R = R + 1;
```

This statement specifies the normal increment that will occur during counter operation. Change the statement to

```
R = R + 3;
```

Note that `R` is declared as a 4-bit register and that a set of top-level `assign` statements are used to copy the register bits to the corresponding output pins. This version of the Verilog Modeler does not support vectored (i.e., multibit) input or output pins. The internal operation of the model can make use of multibit registers, but you must include statements to interface these registers to the individual input or output pins.

Macintosh–Click on the close box at the upper left corner of the text window. In the close dialog box that appears, select the Save and Compile option. The modified source code will be saved in an attribute field called Verilog.Src associated with the selected part type. The source code will be recompiled, and all affected devices will revert to their default values.

Windows–Double-click on the close box on the text editor window and respond Yes to the Save Changes prompt. When you switch back to LogicWorks, the updated source file will automatically be loaded into the circuit and recompiled.

- Operate the CLEAR and LOAD switches in the circuit again and observe that the counters now count by 3.

Macintosh–Reopen the source code to the text window using the Open Source to Text Window command again.

Windows–Reopen the source code using the Save Source to File and Edit command again.

- Locate the same statement modified above and change it to

```
R = R + blip;
```

- Close the text window again and allow the source to be recompiled.

Since the variable `blip` is not defined, this will create a compilation error. A second text window will be displayed with the error message, which can be used to locate the error. Under the Options menu on the LVM control panel are a number of options for redirecting or disabling messages generated during compilation and execution.

See "Controlling Message Output" on page 37 for more information on message redirection.

Macintosh–Select the text of the error message, either by dragging across the text or by triple-clicking in the line containing the error line number. Press the Enter key on the keyboard to locate the error in the source code.

Windows–The Windows version does not have this error location feature.

- Change the original source back to

```
R = R + 1;
```

- Close and "Save and Compile" the model.

Simulating a Circuit with Multiple Verilog Models

In this section, we will look at a sample file that illustrates having two or more Verilog device models in one circuit.

- Start up LogicWorks in the usual way, if it is not already running.
- Open the design file named 68hc11.cct in the LVMDemos directory.

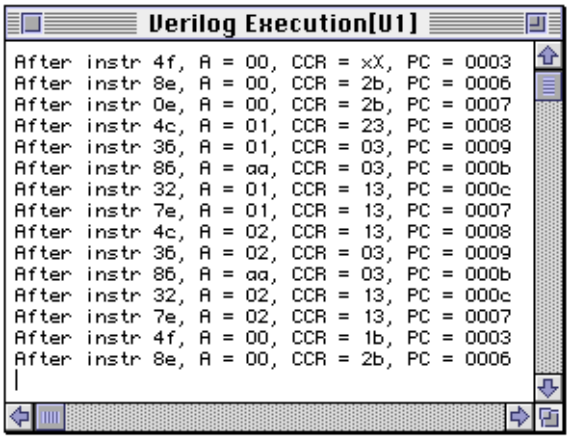
This file demonstrates a Verilog model for a 68HC11 microcontroller and a separate model for an associated memory device. Depending on the speed of your machine, you may notice that a status box shows first one Verilog model, then the other being compiled. The Verilog model is stored with the circuit file as source code only, and it is compiled as soon as it is needed by the simulator after the file is opened.

NOTE: The 68HC11 model provided in this example is not a complete simulation of a commercial 68HC11 device.

- Try clicking on the RESET switch on the diagram to place it in the 1 state.

This action removes the Reset drive from the processor and allows it to perform its initiation sequence. If you watch the timing diagram, you will see the FFFE and FFFF addresses output as the 68HC11 fetches the reset vector. This will be followed by a sequence of instructions being fetched from memory.

Note that after a moment, another text window will pop up.



This window is displayed to show output from `$display` statements in the Verilog source code. `$display` is a very important construct because it allows you to trace and debug your Verilog models and generate text output based on internal variables.

Refer to "The $display System Task" on page 56 for more information on $display.

- Double-click on the 68HC11 device symbol. The LVM control panel will appear.

Macintosh–Click on the Show Variables checkbox.

Windows–Click on the >> button to show the variable display.

You will note that this model has many more internal variables than the 163 in the last example!

- Double-click on the RAM 1Kx8 device.

Note that a *second* LVM control panel appears. This illustrates three important points regarding control panel operation:

- You can have as many control panels open at once as there are Verilog models in the circuit.
- Each control panel is associated only with a single device instance and shows variable values only for the device named in its title bar. Since every device in a simulation can be in a different state, this can be an important feature in tracing the operation of a circuit.
- Each model (i.e., each device instance) in the circuit can produce text output into its own text windows.

Remember that the LVM control panel *does not* show data for the selected device in the circuit, but for the specific device instance that it is associated with, that is, the one named in its title bar.

Creating a Verilog Model Top-Down

In this section, we will look at creating a Verilog model by defining its symbol first and then entering the text for the internal model. This method allows you to take advantage of the automatic Verilog header-generation feature of the LVM.

The most important issue here is to define pins on the device symbol that exactly match the ports that you wish to define for the Verilog model in

terms of name and pin type (i.e., input, output, or inout). We will use the DevEditor tool in LogicWorks to create the symbol and define the pins.

Macintosh–⌘-click in the Parts palette (that is, hold the ⌘ key depressed while clicking) and use the New Lib command to create a new, temporary library to receive the parts that you create in this section.

Windows–Click the right mouse button in the Parts palette and use the New Lib command to create a new, temporary library to receive the parts that you create in this section.

- Use the Open Lib command to open the LVMLib.CLF library provided with the LVM package.

Macintosh–Select the part called VerilogBox in the Parts palette. ⌘-click in the palette and use the Edit Part command to open the DevEditor.

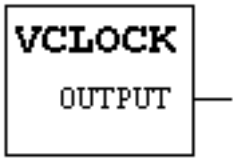
Windows–Select the part called VerilogBox in the Parts palette. Click the right mouse button in the palette and use the Edit command to open this part in the DevEditor.

The VerilogBox part (and the other items in this library) have a special primitive type setting which marks them as Verilog models. We can change the graphics for the symbol and add appropriate pins to suit the specific application for this part.

See "Verilog Primitive Type" on page 28 for more information about Verilog primitive type settings.

- Select the DevEditor's Autocreate Symbol menu command.
- In the Autocreate window, remove any existing text from the Left, Right, Top, Bottom, and Part Name text boxes.
- Type the single word OUTPUT in the Right box and the word VCLOCK in the Part Name box.
- Click the Generate button.

You should now see a part symbol like the following in the DevEditor window:



The remaining (and very important!) task is to set the pin type for the output pin. The Autocreate Symbol command creates all pins with a type setting of Input, and all other pins must be manually changed.

- Double-click on the pin name OUTPUT in the pin list on the left. This will display the pin information palette.
- In the Pin Function pop-up list, select the pin type Output.
- Close the pin information palette.
- Use the DevEditor's File menu commands to save the new part to the library you created, under the name VCLOCK.
- Close the DevEditor window.

For more information on pin information settings, see the section "Pin Information Palette" in Chapter 11 of the LogicWorks 3 manual.

We will now place the new part in a circuit and use the Verilog Modeler to create a simple simulation model for it.

- Use the pop-up library list on the Parts palette to select the library containing the new VCLOCK part, if it isn't selected already.
- Use the New Design command in the File menu to create a new, empty circuit.
- Double-click on the VCLOCK part and place one of them in the circuit.
- Select the pointer cursor in the tool palette.
- Double-click on the device that you just placed in the circuit to open the Verilog control panel.

Macintosh–Select the New Source command in the LVM panel's File menu.

Windows–Select the Generate Templace then Edit command in the File menu on the LVM panel. In some versions of the LVM, it may be necessary to click on the Edit button on the control panel to open the file.

You should now be looking at an autogenerated source file like the following:

```
module VCLOCK(OUTPUT);
output OUTPUT;

endmodule.
```

This is a complete but nonfunctional shell for a Verilog model for this device. Just to complete this example, try adding to this source file until you have a file like the following:

```
module VCLOCK(OUTPUT);
output OUTPUT;
reg OUTPUT;

  initial OUTPUT = 0;

  always #10 OUTPUT = !OUTPUT;

endmodule.
```

- Close the source code and allow it to be compiled.
- Name the output signal of the VCLOCK device using the Pencil tool so that it gets displayed in the Timing window.
- Click on the Reset button on the LVM control panel to restart the model.

You will now see an alternating waveform appear in the timing display. The high and low times should match the 10-unit delay specified in the Verilog source code. Note that pressing the Reset button on the LVM control panel causes the Verilog model to be restarted; that is, any `initial` sections will be executed.

Creating a Verilog Model Bottom-Up

In this section, we will create a Verilog model using an external text editor and then create a LogicWorks symbol to match it. The important issue here is to create a symbol that exactly matches the Verilog source in terms of the naming and functions of pins.

It is also important to note that the LVM always keeps the Verilog source code stored as part of the design file. Any association with an external file is strictly temporary for loading or editing purposes.

NOTE: There is no automatic mechanism in the LVM package that will generate a symbol for you from a source file. One procedure is suggested here, although there are a number of ways to bring source code in for a model by using file and clipboard operations described elsewhere in this manual.

Macintosh–Select the XEditor command in the Tools menu to bring up an empty text window. Use the Open Text command in the File menu to open your Verilog source file. You can use the file named sample.v provided for this example.

Windows–Use the Notepad application, or whichever text editor you are using, to open your Verilog source file. Use the file named sample.v provided for this example.

NOTE: You may wish to make a backup copy of your Verilog source file before proceeding. The following steps require making some temporary edits to the existing file.

- If necessary, edit the `module` line in the file so that all the pin names appear on one line. Select the text for the pin names and use the Copy command in the Edit menu to copy this text to the clipboard.
- Switch to LogicWorks and select the DevEditor tool in the Tools menu.
- Select the DevEditor's Autocreate Symbol command.

Macintosh–Click in the Right Pin Names box and use the ⌘-V key combination to paste the pin names copied from the source.

Windows–Click in the Right Pin Names box and type CTRL-V to paste the pin names copied from the source.

- At this point, you can optionally use standard cut-and-paste techniques to move some of the pin names to alternate sides of the symbol. This has no effect on the association with the Verilog model.
- Click the Generate button on the Autocreate window.

You should now see a symbol with pin names matching your Verilog source. The remaining task is to set the pin function of each pin according to the following table:

Verilog Declaration	LogicWorks Pin Function
input	Input
output	Output
inout	Bidirectional

The Autocreate Symbol command has set all pin functions to be Input, so it is only necessary to change the ones that need to match an "output" or "inout" port in the module header.

- Double-click on the first pin in the list that needs to be changed to open the Pin Information Palette.
- Select the appropriate pin function for that pin.
- Press the Enter key to move to the next pin in the list, or simply click on the next pin in the list that you wish to change.
- When all pin functions have been set, close the Pin Information Palette.
- Save the new part symbol to a suitable library and close the DevEditor window.
- Create a new, empty LogicWorks circuit using the New Design command in the File menu.
- Place one of the new symbols in the circuit and return to the pointer cursor.
- Double-click on the new device to display the LVM control panel.

Macintosh–In the LVM control panel's File menu, select the Load Source from File command and select your original Verilog source file. This will load the contents of the selected file into the device. Note that there is no permanent association with the file. Later modifications to the file will have no effect on the simulation.

Windows–In the LVM control panel's File menu, select the Status of Source/File command. Click on the Browse button and locate the external source file. The status indicator should now show that the file is newer than the source in the part. Click on the Upload <-- from File button to load the

text in the file into the design.

- Select the Compile command in the LVM's File menu to compile the new source. If there is any mismatch between the pins on the symbol and the ports declared in the source, compiler error messages will be generated.

Saving a Verilog Model with a Library Part

In this section we will see how to save a part into a LogicWorks symbol library, complete with its Verilog source code. This will allow other users of the part to just place it in their own circuits and start simulating immediately.

NOTE: Users with access to the DesignWorks professional circuit design package can make use of the Save to Lib command to save a fully defined Verilog part to a symbol library. LogicWorks does not have this command, so we have to use the procedure defined here.

This procedure makes the following assumptions:

- You have your Verilog source code ready, either in an external text file or in an existing LogicWorks circuit.
- You have the symbol with which you wish to associate the source code already created and in a library.

- Copy the Verilog source code that you wish to associate with the part onto the clipboard. If the source code exists in an external text file, you can do this with any text editor. If the source code is already in a symbol in a LogicWorks design, open the source code using the methods described in the first tutorial section. Then select the entire source and select the Copy command in the Edit menu.
- Move to the Parts palette and open or select the library containing the destination part.

Macintosh–Select the part in the Parts palette. ⌘-click in the palette and

use the Edit Part command to open the DevEditor.

Windows–Select the part in the Parts palette. Click the right mouse button in the palette and use the Edit command to open this part in the DevEditor.

- Select the DevEditor's Part Attributes command.
- Select the Verilog.Src attribute field in the field list.

NOTE: The DevEditor gets its list of attribute fields from the current design. If the Verilog.Src attribute field does not appear in the list, close the DevEditor, open one of the Verilog sample circuits provided, or one of your own, and try again.

Macintosh–Type ⌘-V to paste the source code on the clipboard into the Verilog.Src attribute field.

Windows–Type CTRL-V to paste the source code on the clipboard into the Verilog.Src attribute field

- Save the part back to the library.

4

Using the Complete Program

This chapter provides background and reference information on how Verilog models work with the overall LogicWorks simulator, and presents specific procedures for creating and working with Verilog inside LogicWorks.

Design Organization with the Verilog Modeler

In effect, the LogicWorks Verilog Modeler provides an alternate way of defining the "internal circuit" for a device symbol in a LogicWorks schematic. The way the Verilog model definition is stored and updated in a design is analogous to the way subcircuits are created and modified in a design using the "physical hierarchy" mode. In particular, note the following points:

- The Verilog model is associated with a *device type* (i.e., a symbol definition), not an individual device instance. When you double-click on a Verilog device symbol and modify its internal definition, you are affecting *all other devices of the same type* in the design.
- When you modify the Verilog source code for a device type, you have in effect changed the definition of that symbol. It will no longer be considered to be the same as the symbol in the library that was selected when the device was originally placed.
- The Verilog source code is kept in a device attribute field called Verilog.Src. However, the data is actually stored with the type definition, not with each device instance. Modifying the Verilog.Src attribute field for a given instance will have no effect on the Verilog

model. The Verilog source definition can only be changed by using the appropriate LVM commands, or by setting the Verilog.Src attribute field in the symbol itself, using the DevEditor.

- Because the Verilog source code is stored in its entirety in an attribute field, it is limited to 32,000 characters.
- In this version of the LVM, Verilog source code can only be used to create an internal definition of a device. There is no way to make a stand-alone Verilog simulation without a schematic or to simulate (or stimulate) part of a design using Verilog without using a "parent" symbol.
- When a design containing Verilog models is saved to a file, only the Verilog source itself is saved. The compiled data and simulation state is lost when the design is closed. The source code is recompiled when the design file is opened.

Creating a Verilog Device Symbol

This section gives you the background information and procedures you need if you want to create symbols from scratch that will be associated with a Verilog model in a LogicWorks schematic.

Verilog Primitive Type

LogicWorks uses the concept of a *primitive type* to distinguish different types of symbols that receive different internal handling. For example, the SUBCCT primitive type denotes symbols that can have an internal circuit, and the AND primitive type denotes symbols that will be handled by the logical AND simulation model in the Simulator.

The association between the LVM tool and the device symbol on the schematic is also made by primitive type. That is, any user actions and simulator events affecting a symbol with primitive type VERILOG will be intercepted by the LVM.

NOTE: Because the LVM package was developed after the last major release of LogicWorks, the LogicWorks menus and options that deal with primitive types do not yet have an entry for the VERILOG primitive type. For this reason, a raw primitive type number 47, which has been assigned for this purpose, must be used at this time. Future versions of LogicWorks will have this item added as a text entry in the relevant menus and displays. For now, just treat 47 as a magic number that means "Verilog".

To set the primitive type for a Verilog parent symbol, you can do one of the following:

- Use one of the sample symbols provided in the symbol library included with the LVM package as a basis for your symbol. You can freely add, delete, or modify pins or attributes as needed for your application.
- When creating the symbol in the DevEditor, set the primitive type as follows:
  - Select the Subcircuit & Part Type command in the DevEdit menu (Macintosh) or the Subcircuit/Part Type command in the Options menu (Windows).
  - Choose the Primitive Type option.
  - In the pop-up menu that appears, select the Other... option at the bottom of the list. (Note: A VERILOG selection will be added to this list in an upcoming maintenance release of the DevEditor.)
  - Enter the value 47 for the VERILOG primitive type and click OK.
  - Click Done on the Subcircuit and Part Type box.
  - Edit and save the symbol in the usual way.

To test that the primitive type of a symbol has been set correctly, simply place an instance of the symbol on a schematic and double-click on it. The LVM control panel should be displayed instead of the usual response by the Schematic package.

Port Interface

All communication between a Verilog model and the rest of the design is through the pins or "ports" on the parent symbol. Creating a correct linkage between the ports defined in the Verilog source code and the pins on the device symbol is essential to correct operation.

Port-to-Pin Association

The correct port-to-pin association is the single most important issue in attaching a Verilog model to a device symbol. Here are the rules that must be followed to ensure that this is accomplished.

- The ports defined in the Verilog source code are matched with the pins on the parent symbol by name. Definition order is of no significance. This implies the following naming rules:
  - Only valid Verilog name characters can be used in applying pin names to the parent symbol.
  - Pin names must not exceed the maximum length of 16 characters imposed by LogicWorks.
  - Duplicate pin names are not allowed. The DevEditor will allow duplicate pin names when the pins are inside different bus pins, but all pin names must be completely unique for use with the LVM.
- You can use bus pins on the parent symbol to group pins by function, if desired. The bus pin itself is completely ignored by the LVM; pin-to-port association is made strictly by the name of the individual internal pins.
- All ports declared in the Verilog source must be single-bit. Vectored ports are not supported in this version.
- The pin type setting that is used on each pin when the symbol is created in the DevEditor must correspond to its declaration in the Verilog source code, according to the following table:

DevEditor Pin Type Setting	Verilog Declaration
Input	input
Output	output
Tristate	output
Bidirectional	inout
Open Collector	output
Open Emitter	output
Latched Input	input
Latched Output	output
Clocked Input	input
Clocked Output	output
Clock Input	input
All others	ignored

Setting Output Port Values

The Verilog language definition does not allow direct assignment of values to output ports from within behavioral constructs. For this reason, it is always necessary to use one of two techniques to set output port values: *continuous assignment* or *port-register aliasing*.

Continuous Assignment

A continuous assignment statement (i.e., the `assign` keyword) can be used to map the `reg` variables used in the behavioral code to the output ports.

Here is a simple example of an oscillator device with a single output pin. Note how the `reg` variable `osc` is used for all internal calculations, and its value is continuously assigned to the output pin `OSCOUT`:

```
module OSC(OSCOUT);
// Declare the port in the usual way
  outputOSCOUT;

// Declare a "reg" for internal use
  reg osc;

// Continuously assign the reg to the output
  assign OSCOUT = osc;
// Internal processing to generate the values
  initial osc = 0;
  always #1 osc = !osc;
endmodule
```

Port-Register Aliasing

The LVM also supports the concept of *port-register aliasing*, that is, creating an output port and a `reg` variable with the same name. This allows you to use the port name in the behavioral code and creates an automatic transfer of any value changes to the output port.

The previous example, modified to use port-register aliasing, follows:

```
module OSC(OSCOUT);
// Declare the port in the usual way
  outputOSCOUT;

// Declare a "reg" for internal use
  reg OSCOUT;

// Internal processing to generate the values
  initial OSCOUT = 0;
  always #1 OSCOUT = !OSCOUT;
endmodule
```

Mapping Multibit Registers to Ports

This version of the LVM does not support vectored (multibit) ports. Each port in the Verilog `module` header must correspond to a single pin on the parent symbol, which will in turn be connected to a single signal line.

For this reason, the continuous assignment prescribed in the previous section can be extended to map an internal, multibit register to multiple output ports, as in the following example:

```
module OSC2(PHI_A, PHI_B);
// Declare the ports in the usual way
  outputPHI_A;
  outputPHI_B;

// Declare a 2-bit "reg" for internal use
  reg[1:0] osc;

// Continuously assign the reg to the outputs
  assign PHI_A = osc[0];
  assign PHI_B = osc[1];

// Internal processing to general the values
  initial osc = 0;
  always #1 osc = osc + 1;
endmodule
```

For more information on ports from the point of view of the Verilog source code, see "Ports" on page 61.

Operation

This section describes the mechanics of creating, editing, and controlling a Verilog simulation model.

Opening the LVM Control Panel

The LVM control panel can be opened for a specific device instance just by double-clicking on the device on the schematic. Note the following points regarding the association between the LVM control panel and the device in the circuit:

- The name of the device that was opened will be displayed in the title bar of the control panel.
- In general, the commands connected with variable display, message output, and simulation affect only the specific device instance associated with this panel. Commands affecting the Verilog source code and compilation will affect all other instances of the same type, as described earlier in this manual.
- You can have any number of LVM control panels open at once to display and control multiple devices in a design. Closing a control panel has no effect on the simulation of the device.

Opening the Verilog Source Code

Macintosh–Select the Open Source to Text Window command in the File menu on the LVM control panel. This command opens the current contents of the Verilog.Src attribute field to a text editor window. The text can then be edited using the standard features available in the XEditor tool. When the text window is closed, if any changes have been made, you will be prompted for the desired save and recompile actions.

IMPORTANT: The XEditor tool on the Macintosh has the standard Save and Save As commands in the File menu. These commands allow you to save data to disk files, but *do not* affect the Verilog.Src attribute. To update the Verilog.Src attribute field, you must close the window using the close box or Close command and respond to the close dialog box.

Windows–Select the Save Source to File then Edit command in the File menu on the LVM control panel. This command saves the current contents of the Verilog.Src attribute field to a temporary text file and then opens the file using your text editor. When the text editor is closed and you switch back to LogicWorks, you will be prompted for the desired save and recompile actions.

Creating a New Verilog Model

The LVM has the ability to automatically create a "shell" of a Verilog source code file for a device that you have placed in a circuit. This is especially useful when you create a new symbol that has never had a model associated with it, because it saves typing the port names and associated declarations. The generated file will include the `module` line with all the ports defined, declarations for the ports, and an `endmodule` statement. This can be used as a starting point in defining your own model.

Macintosh–Select the New Source command in the LVM control panel. The autogenerated Verilog source file will be displayed in the XEditor. If the device already has source code associated with it, you will be prompted before the old source code is replaced.

Windows–Select the Generate Template then Edit command in the File menu on the LVM panel. If the device already has source code associated with it, you will be warned before the old source is replaced. The new source code will be saved to a temporary file and opened in the text editor.

Selecting a Text Editor (Windows Only)

The LogicWorks Verilog Modeler package relies on having a text editor available for creating and modifying source code. If you don't make any other selection, operations that require a text editor will use the standard

NotePad application. You can change this selection at any time using the Preferences command in the Options menu on the LVM control panel.

Using an External Text Editor

Verilog source code for use with the LVM can be created by using any external text editor, although some manual operations are necessary to transfer the text into the model. The commands in the File and Edit pop-up menus in the LVM panel provide straightforward methods of importing and exporting the source code text.

Macintosh–Source code from an external text file can be loaded into a device by using the Load Source from File command. This command displays a standard file Open box allowing you to select a source text file. This file will be loaded in its entirety into the Verilog.Src attribute and will replace any existing source code. In this version of the LVM, there is no permanent linkage between the device symbol in the design and any external text files. Whenever you load source code into a device model, it is *copied* into the Verilog.Src attribute for compilation. Future changes to the external text file will have *no effect* on the design.

NOTE: The source is *not* automatically recompiled. You must use the Compile command to recompile it.

Windows–In the LVM control panel's File menu, select the Status of Source/File command. Click on the Browse button and locate the external source file. The status indicator should now show that the file is newer than the source in the part. Click on the Upload <-- from File button to load the text in the file into the design.

Saving Source Code to an External File

Macintosh–Select the Save Source to File command. This command displays a standard file Save box allowing you to create a destination file on your disk. The current contents of the Verilog.Src attribute will be saved to this file. The device's Verilog.Src attribute, compilation status, and simulation status are unaffected.

Windows–Select the Save Source to File then Edit command. This command will open the existing source code, using the selected text editor. You can then use the text editor's Save As file command to save the source code to any desired location.

Transferring Source Code Via the Clipboard

In some cases, it may be desirable to move source code quickly into or out of a model without using a text editor. You can do this by using the commands in the Edit menu. The Source to Clipboard command copies the contents of the Verilog.Src attribute field to the clipboard.

The Clipboard to Source command copies the contents of the clipboard to the Verilog.Src attribute field. If the device already has source code associated with it, you will be prompted before the old source code is replaced.

Compiling the Verilog Model

The Verilog source code must be compiled into an internal executable form before any simulation can be performed. Note that the compilation process is performed completely separately for each part type.

The following actions will cause the source code for a given part type to be compiled:

- Selecting the Compile command in the File menu in the LVM control panel.
- Closing a modified source file.
- Performing any simulator action that would cause a device with uncompiled source code to be reevaluated.

Compilation Error Messages

When you compile a Verilog source file, error messages may be produced for illegal syntax, incorrect port-to-pin matching, and so on.

Macintosh–By default, compiler error messages are directed to an XEditor text window. This output can be redirected to a file or message box by

using the Compile Messages command in the Options menu.

Windows–By default, compiler error messages are directed to a scrolling text window. This output can be redirected to a file or message box by using the Messages command in the Options menu.

Locating an Error in the Source Code

Each compilation error is preceded by the line and column number of the offending text.

Macintosh–For any error message currently displayed in an XEditor message window, you can select the text of the message and press the Enter key on the keyboard. The LVM tool will interpret the number following the word line as a line number, open the Verilog source to another XEditor window, and go to the indicated line.

Windows–The line number displayed in the error message can be used with a "Go To Line" feature in many text editors.

Controlling Message Output

The LVM can generate numerous textual messages. These are divided into three groups summarized in the following table.

Message Type	Usage	Default Destination
Compilation messages	Syntactical errors, undefined labels, mismatched pins, etc.	Macintosh: Text editor Windows: Scrolling text box
Execution messages	`$display` statement output	Macintosh: Text editor Windows: Scrolling text box
Internal errors	Code errors, out-of-memory conditions, etc.	Dialog box

A number of commands are available that allow you to control message output.

NOTE: These commands affect only message output associated with the particular device instance that was double-clicked on to open the control panel.

Macintosh–The three Messages commands in the Options pop-up menu in the LVM control panel are used to control redirection of message output to text editor windows, file, dialog box, or none.

Windows–The Messages command in the Options menu in the LVM control panel allows you to control the redirection of message output.

NOTE: If you close a text window that is being actively used to show `$display` output, further output from that device instance will be disabled. To reenable it, use the Execution Messages command in the Options pop-up menu in the LVM panel.

Simulation with the Verilog Modeler

Once a Verilog model is created and compiled, it can be simulated in much the same way as any other LogicWorks symbol. This section covers issues of interaction between the internals of a Verilog model and the LogicWorks Simulator.

Compiling and Initialization

The following points should be noted concerning the initialization of a Verilog model:

- When a design containing Verilog models is saved to a file, only the Verilog source itself is saved. The simulation state is lost when the design is closed. The source code is recompiled when the design file is opened, so all variables will revert to their default state regardless of the state of the rest of the design. When you open a file and start simulating, you should use the Clear Simulation command in LogicWorks' Simulate menu to reset all models to their initial states.
- When a Verilog model is compiled, all internal variables are set to an X (Don't Know) state by default. Other values will only be set as explicitly specified in the source code.
- When the Reset button on the LVM control panel is pressed or a Clear

Simulation command is executed, the following steps are taken:

- All execution threads are terminated. All pending events and assignments are cleared.
- All variables are reset to their default states.
- All `initial` and `always` blocks are restarted.

Debugging a Verilog Model

A number of tools are available to assist in debugging your Verilog code.

Single-Stepping the LogicWorks Simulator

There are no commands available to single-step the internal statements of a Verilog model. However, the LogicWorks Single Step command can be used to execute all the activity scheduled at one time step and then stop. In effect, one Single Step command tells the Verilog model to execute all active "threads" (i.e., parallel processes) until they block (i.e., specify a delay or wait for some other action) or terminate.

IMPORTANT: The LVM uses "zero-delay" events to pass internal values to output pins on the parent symbol. Thus, when using the Single Step command, you will frequently see two steps executed at the same time. This is normal.

`$display` Statements

Standard Verilog `$display` statements can be used to display variable values at any point in the execution of the model. Output generated by `$display` statements is by default displayed in a text editor window. This can be disabled or redirected to a file by using the commands in the Options pop-up menu on the LVM control panel.

NOTE: If you close a text window that is being actively used to show `$display` output, further output from that device instance will be disabled. To reenable it, use the Execution Messages command in the Options pop-up menu in the LVM panel.

See "The $display System Task" on page 56 for a complete description of the `$display` *statement.*

Simulation Value Display

The variable display on the LVM control panel allows you to display all internal variables of the model as it executes. Note that this display shows the values associated with the device instance that was double-clicked on to open the LVM panel. Obviously, each instance of a given device type can be in a different simulation state.

Macintosh–The variable display is opened by checking the Show Variables checkbox. You can control whether input ports and output ports are displayed by using the Show Input Ports and Show Output Ports checkboxes in the display.

Windows–The variable display is opened by clicking on the >> button. You can control whether input ports and output ports are displayed by using the Show Input Ports and Show Output Ports commands in the Options menu.

Here are the format rules for this display:

- The "Scope" column shows the definition scope of the variable. In most cases this will be the module name, but it may also show the name of a named `begin/end` block with local variables.
- The "Value" column is updated immediately when a variable changes value. Integer and Time variables are shown in decimal; all others are in hexadecimal. There are currently no display radix options.
- For memory variables, only the first location is shown. There is no way to display the other locations in this display.
- By default, input and output ports are not shown in the variable list. They can be added by enabling the Show Input Ports or Show Output Ports options.
- There is no way to select display order or to remove variables from the display in this version.

User Interrupt

Using the programming constructs in Verilog, it is quite possible to create an infinite loop in a model. This effectively brings the entire system to a halt because it prevents any other model or any other LogicWorks function from operating. A common example is an `always` loop with no delay specified, as in the following:

```
always clk = !clk;
```

The Verilog simulator executes continuously until it encounters some delay, so this loop will never break.

Macintosh–Press ⌘-. (command-period).

Windows–Press the ESC key.

Depending on the complexity of the model, you may have to hold the key down for a moment before the model reaches a point in its processing where it can break. This effectively terminates execution of the model until a Reset or Clear Simulation is done.

Copying Variable Values to the Clipboard

The LVM allows you to copy the current variable values from the Show Variables display onto the clipboard in text form. You can do this to help create documentation of a design or to provide a “snapshot” of the state of a model for debugging purposes.

The Copy Variables command copies the selected items in the Show Variables display to the clipboard. If Show Variables is disabled, this command will be disabled. Items in the variable display can be selected by clicking on them, and any rectangular group of cells can be selected by holding the Shift key. The Select All Variables command selects all the cells in the Show Variables display. If Show Variables is disabled, this command will be disabled.

Resetting a Single Model

The Reset button on the DVM control panel executes a reset operation on only the specific device instance associated with the control panel.

Reset causes the following steps to be taken:

- All execution threads are terminated. All pending events and assignments are cleared.
- All variables are reset to their default states.
- All `initial` and `always` blocks are restarted.

Selecting the Clear Simulation command in the LogicWorks Simulate menu also executes a Reset operation on all Verilog models in the design.

Part II
Verilog Language Support

The following chapters describe the Verilog language as implemented in the LogicWorks Verilog Modeler. They are intended as a guide for getting started with the language rather than as an exhaustive language reference.

Verilog is a language that was originally intended as an input mechanism for the proprietary Verilog simulator, now sold by Cadence Design Systems, Inc. In order to encourage wider use of the language, Cadence passed it into the public domain, making it possible for a large number of vendors to create products supporting the language. The language is now documented, reviewed, and evangelized by Open Verilog International, an open industry group. Verilog is now widely used for describing hardware systems for simulation and synthesis purposes.

Verilog will look fairly familiar to anyone who has developed a program in the popular Pascal programming language, as much of its syntax was derived from that source. However, Verilog adds important elements needed to describe hardware, such as unknown and high-impedance values, parallel blocks, and so on.

The full Verilog language supports circuit descriptions in both structural (i.e., gates and flip-flops) and behavioral form (i.e., procedure-oriented). Only the behavioral parts of the language are implemented in this version of the LVM. Gate- and switch-level modeling, `wire` variables, module hierarchy, and other structural concepts are not supported.

This manual describes only the supported language features.

For a complete description of the language, refer to the Verilog Hardware Description Language Reference Manual *published by Open Verilog International.*

5 Structure of a Verilog File

This chapter presents a simple example to illustrate the overall structure of a Verilog source file. The term *file* is used loosely here because, in the LVM, the Verilog source code is actually stored in LogicWorks attributes as part of a larger circuit design. In this chapter, we use the term *file* to refer to a single block of Verilog source code associated with a single LogicWorks symbol.

Module Organization

A module block starting with the keyword `module` is the basic unit of organization, similar to a procedure or subroutine in a programming language. In the LVM implementation of Verilog, a single file can contain only one module. The module header specifies the signal interface for the module, that is, the correspondence between the module's inputs and outputs, and the pins on the parent LogicWorks symbol.

For example, consider the module shown here, which implements a simplified D-type flip flop:

```
module DFF(CLK, D, Q);
input CLK;   // Declare clock as input
input D;     // Declare data as input
outputQ;     // Declare Q as output

reg Q;       // Associate Q with a reg

// Give Q an initial value of 0

initial Q = 0;

// Whenever a positive edge is seen on CLK,
// assign D input to Q output

always @(posedge CLK)
  begin
     Q = D;
  end

endmodule
```

The rest of this chapter expands on the points illustrated in this example.

Module Header

A module always begins with the keyword `module`, which is followed by the module name. The module name should be the same as the library type name of the parent symbol (as it appears in the Parts palette) in order to make the association between symbol and model clear, but this is not enforced.

A list of module ports, enclosed in parentheses, follows the type name. There must be a one-to-one correspondence between the pins on the parent device symbol and the ports declared here, except for bus pins. Ports are matched with the pins on the parent symbol by name, not by order.

See "Port Interface" on page 29 for details on port name matching.

The module header is terminated by a semicolon.

Declaration Section

Following the module header is a series of declarations that define the nature of the module's ports and any other variables to be used internally. The complete description of possible declarations is given in "Data Types" on page 59.

In our example, two of the ports are declared as inputs, and one as an output. Ports can also be of type `inout`, corresponding to bidirectional pins on the parent symbol. The port declaration must correspond in type (i.e., input/output/inout) to the pin type of the parent device symbol. Vectored ports (i.e., more than one bit wide) are not supported.

In the example, note that Q is declared twice, once as an `output` and once as a `reg`. This is a special case known as *port-register aliasing*, which is used for output ports that are to be assigned values in behavioral Verilog code. Since behavioral code can only work with `reg` variables, this creates an automatic link between a `reg` and an output port with the same name. Any value assigned to the `reg` will appear at the corresponding output port.

Statement Section

The actual activity of the model is specified in one or more statements that make up the balance of the module source code.

At the topmost nesting level within a module (that is, not inside any other block), the LVM version of Verilog supports only three types of statements:

- The continuous assignment statement. This type of statement starts with the keyword `assign` and is used to apply a value in a `reg` variable to an output or inout port.
- The `initial` statement. This type of statement is executed only once when the module is first instantiated. It is used to set initial values for variables and take any actions that should happen only once during the life of the module, regardless of input conditions.
- The `always` statement. This type of statement is executed repeatedly for the entire life of the module. It is in effect an “infinite loop”, which repeatedly performs assignments and other behavioral actions. Typically, the `always` statements in a module contain the main body of the behavioral code that defines the operation of the module.

Note that `always` and `initial` statements are identical in structure and syntax. The only difference is that the former is applied repeatedly and indefinitely, whereas the latter is applied exactly once. In the LVM implementation, continuous assignment statements are restricted to the specific usage already described.

In this example, a `begin/end` block has been used after the `always`. This is not necessary in this case, because the `begin/end` contains only one statement. It is included here to illustrate how you can use `begin/end` to encapsulate multiple behavioral statements into a single block that can be used with `always` or `initial`.

A module can contain any number of each of these three types of statements, mixed in any order. In effect, all `initial` and `always` blocks are executed in parallel and can be thought of as representing a separate hardware subunit of the module. These subunits can communicate by referring to the same variables or by using *events*, which are discussed in “Events” on page 90.

In the preceding example, the text `@(posedge CLK)` blocks the execution of the `always` statement until the specified event occurs. In this case, when the CLK input changes from a low to a high level, execution proceeds, and the assignment `Q = D` is executed.

Module Termination

The module always ends with the keyword `endmodule`.

Module Structure Summary

In this chapter we have looked at the overall structure of a Verilog behavioral model. You should now see the relationship between the contents of a model and the structure of the real hardware it represents. Each `always` or `initial` block represents a piece of circuitry that can operate independently in parallel, and each `assign` statement represents an output buffer. For example, an `initial` block might be used to express the power-up logic of a circuit, and a couple of `always` blocks might represent a clock generator circuit and a bus interface, which can work independently in parallel.

Especially if you have software programming experience, you may find some of the concepts of parallel operation a bit hard to grasp at first. However, just keep in mind that what we are really doing is describing a digital system that in the end will consist of standard logic elements wired together on a board or chip.

6
Language Syntax

This chapter describes the lowest-level building blocks of the Verilog language, such as lines, constants, and statements. These elements are essential in constructing a complete model using the larger structures described in later chapters.

White Space and Comments

Blanks, tabs, and line terminators (that is, a carriage return or line feed) are considered as “white-space” characters; they act as separators between other items and are otherwise ignored.

Verilog has two forms of comment block. A single-line comment can be inserted using “//” in two ways:

```
// This is a comment line
```

or

```
clk = 1;    // The rest of this line is a comment
```

For comment blocks spanning any number of lines, the “/* */” form can be used:

```
/*
  The following section illustrates how the
  disable keyword is used to terminate execution
  of a block.
*/
```

Statement Separators

Verilog statements can span any number of lines. The semicolon is used to terminate Verilog statements. It should only be used after a single statement, not between compound statements like `begin`/`end` blocks. For example:

```
begin
  a = 1;
  b = 2;
end
begin
  c = 3;
end
```

In general, a semicolon is not used between two Verilog keywords, in this case, between `end` and `begin`.

Numbers

Numbers are assumed to be expressed in decimal unless otherwise specified. Optional size (width in bits), base (binary, octal, decimal, or hexadecimal), and sign specifiers can also be used to specify constant numbers, as shown in these examples:

```
1234     // An unsized decimal constant
16'b011  // A 16-bit number 0000 0000 0000 0011
'b1      // An unsized binary number
12'h3ab  // A 12-bit hexadecimal number
24'o777  // A 24-bit octal number
'D12     // An unsized decimal number
-4'b11   // A negative binary number
```

The base specifier letter and the hexadecimal digits a to f can be uppercase or lowercase. In addition, underscore characters can be inserted between

the digits of a number without affecting its value. This allows you to break up long sequences of digits for readability, as in this example:

```
16'b1011_1111_0001_0101
```

In the LVM implementation, unsized numbers are stored as 32-bit values, and 32 is the maximum size specification.

Unknown and High-Impedance Constants

The letters X and Z (uppercase or lowercase) can be used to express unknown and high-impedance values, respectively. They are valid only in binary, octal, and hexadecimal constants and affect only the bit positions corresponding to their position in the constant. For example,

```
'HX34    // Equals 'bxxxx_0011_0100
'o7z     // Equals 'b111ZZZ
```

A question mark can be used as an alternative to `Z` to indicate a high-impedance value. This is intended primarily for use with the `casez` and `casex` statements to indicate a don't care value. See "Using Don't Care Case Values" on page 81.

Strings

A string is any sequence of ASCII characters enclosed by double quotes. A string must be completely specified on one line. To insert nonprinting characters or double quotes into a string, the escape character \ can be used, as shown in this table.

\n	A new line character. This will be either a carriage return or line feed, as appropriate for the host system.
\t	A tab character.
****	A backslash character.
\"	A double quote character.
\nnn	The ASCII character with the given code, specified as a 3-digit octal number.

Examples of string constants follow:

```
"This is a nice string."
"This string\nwill come out on two lines."
"This is followed by a line feed\012"
"Stuff\tseparated\tby\ttabs."
```

This version of the LVM does not support any string variables, string operations, or string assignments. Strings are used only in the `$display` system function described in "The $display System Task" on page 56.

Identifiers

An identifier is a sequence of letters, digits, dollar sign, and underscore characters used to give a name to a variable, object or program construct. The first character must not be a $ or a digit. Identifiers are case-sensitive; that is, uppercase and lowercase letters are considered to be different. The LVM implementation does not limit the length of identifiers.

Escaped Identifiers

Escaped identifiers are used to specify identifiers that contain any printable ASCII character. This mechanism is provided primarily to assist in translating to and from other hardware description languages that may have different naming standards.

An escaped identifier starts with a backslash character and terminates with any white space. Neither the backslash nor the white space are considered to be part of the identifier.

Here are some examples of escaped identifiers:

```
\**1**
\CLK/
\B[12]
\begin
```

The last example illustrates that a Verilog keyword can be escaped so that it can be used without being recognized as having any special language significance.

IMPORTANT: An escaped identifier must always be followed by white space; otherwise all following characters will be considered part of the identifier.

Keywords

A keyword is a predefined identifier that is part of the Verilog language. All keywords are defined in lowercase letters and must be used in lowercase only. Appendix B lists all the keywords defined in the Verilog language.

System Tasks and Functions

Several built-in system tasks and functions are invoked by name using the `$identifier` construct; for example,

```
$display("We just got a rising clock edge.");
```

The available functions are fully described in the following sections.

NOTE: The full Verilog language also allows user-defined tasks of the form `$identifier`, but this is not implemented in the LVM.

The **$display** System Task

The $display system task is used for displaying textual data as the model executes. You can use it for debugging or to display simulation results that are best displayed in a text format.

The $display call is followed by any number of arguments to be displayed. If no formatting information is provided, all arguments are assumed to be strings and are transmitted verbatim to the text output window.

Formatting information is provided by inserting the character "%" in an argument string followed by a letter indicating the format type. For every "%" format specification in the string, a following argument is removed from the list and displayed in that format. The following table summarizes the available formats (you can also specify the format with an uppercase letter).

Format String	Conversion
%h	Hexadecimal
%d	Decimal
%o	Octal
%b	Binary
%%	Display a "%" sign

Here are some examples that illustrate the use of $display:

```
$display("Mem element %d contains %h",
         mem_addr, mem[mem_addr]);

$display("%d%% completed",
         (count * 100) / max_count);
```

The **$time** System Function

The $time function returns the current system simulation time as a 32-bit integer; for example,

```
$display("Got an event at time %d", $time);
```

The **$finish** System Task

The `$finish` command terminates operation of the Verilog model that executes it. It affects only the current model, not any other Verilog models or the overall LogicWorks simulation. From the LogicWorks simulator's point of view, this model will not generate any further output changes or respond to input changes.

Compiler Directives

The ` mark, referred to as a back-quote, or scribe mark, introduces a compiler directive or text macro. In the LVM implementation, the only compiler directive is `` `define ``, which is described in the next section.

Text Macros

Verilog allows the definition of named text macros to replace commonly used constants or text strings with a meaningful name.

Text macros are defined using the `` `define `` compiler directive; for example,

```
`define T_CLK_TO_D23  // Define data out delay
`define op_ANDI'h86   // Define ANDI instruction
```

Note that the trailing comments are not considered to be part of the macro text.

Macros are invoked by using the same back-quote character followed by the macro name, as in these examples.

```
#`T_CLK_TO_D;
if (opcode == `op_ANDI) ->do_ANDI;
```

Here are some points to remember regarding text macro usage:

- Text macro names may not be the same as those of compiler directives.
- Text macro names may be the same as a Verilog keyword or variable name without confusion.
- Text macro definitions end at the end of the definition line. In other words, they cannot be more than one line.
- Text macros cannot contain partial language objects like strings or identifiers. For example, the following is illegal:

```
`define HEX_16 16'h// Can't have part of a number...
a = `HEX_16 af;// ... in a macro
```

7
Data Types

This chapter introduces the various types of data that you can use to create your Verilog model. In many cases, the data types are similar to those found in a typical programming language, although some types have characteristics intended specifically to mimic the operation of digital circuits.

Register Data Type

The register data type represents a hardware storage device and is the most commonly used data type for Verilog behavioral modeling. Each register can be a single bit or an array of 1–32 bits, and each bit can have one of four states (0, 1, high-Z, and unknown).

The register declaration is introduced by the `reg` keyword, as in these examples:

```
reg   int_en;              // Single bit
reg[7:0]regA;              // 8-bit register, MSB high
reg[1:12]accum;            // 12-bit register, MSB low
reg[`MSB:0]mem_dat;        // Using `define for width
reg[3:0]flags1, flags2;    // Two 4-bit regs
```

When a range is specified, the first number is the bit number of the most significant bit. In the last example above, note that any number of registers of the same width can be declared in a single `reg` statement.

IMPORTANT: When used in an expression, a register variable is treated as an *unsigned*

n-bit integer, regardless of how the value was created. For example, if the value –2 is assigned to a 4-bit register, the resulting binary value will be 4'b1110 using twos-complement arithmetic. This will have an effective value of 14 when used in an arithmetic expression.

When used in expressions, registers can be subscripted to extract single bits or any subrange of the bits in the register. For example, given the declarations above, the following assignments are valid:

```
int_en = regA[3];
accum[12] = regA[3];
regA[7:0] = accum[1:8];
```

You should also refer to the important information on bit lengths in register operations in "Bit Lengths in Expressions" on page 70.

Memories

Memories are implemented as arrays of `reg` variables, as in this example, which creates an array of 16 × 32-bit registers:

```
reg[31:0] regFile[0:15];
```

Note the following points regarding array references:

- Only a single array element can be assigned to at once, as in

  ```
  regFile[3] = 'b1101;
  ```

  Assuming the existence of another similar array called `regFile2`, the following is *not* legal:

  ```
  regFile = regFile2;// NOT LEGAL
  ```

  Similarly, the following is *not* a legal way to initialize an entire array to zero:

  ```
  regFile = 0;// NOT LEGAL
  ```

- The OVI Verilog standard does not specify the value of an array reference if the array index is unknown or out of bounds. In the LVM, such a reference is considered to have an unknown (X) value.

- Because the syntax for selecting a bit out of a multibit register and an element out of an array is identical, it is not possible to select a single bit or a range of bits from an array element. If `mem1` is declared as an array, then `mem1[n]` will always refer to the entire contents of the *n*th element in the array.

Ports

Port variables are used to transfer data into or out of a module. In the LVM, ports are implemented only as single bits and can be of type input, output, or inout.

See "Port Interface" on page 29 for specific rules and restrictions governing the port association with a LogicWorks symbol.

Following are examples of legal port declarations:

```
input    clk;
input    D0, D1, D2, D3;
inout    HALT_;
output   SIZE0, SIZE1;
```

Port-Register Aliasing

The LVM supports the concept of *port-register aliasing*, that is, creating an output port and a `reg` variable with the same name. This allows you to use the port name in the behavioral code and creates an automatic transfer of any value changes to the output port.

See "Port-Register Aliasing" on page 31 for more information on the usage of this feature.

Named Events

Events are used to synchronize parallel processes that may be executing within a module and to communicate between the various hardware units represented. This section describes the declaration and basic usage of named events.

A more complete description of named events and value change events is given in "Events" on page 90.

Events are declared as in these examples:

```
event do_fetch, end_fetch;
event TRIG1;
```

As with other types of declarations, you can declare multiple events after a single `event` keyword. Events have no value or storage associated with them and therefore no size specification. They are activated at an instant and then revert to their inactive state. At the instant of activation, any processes that are blocked and waiting for that particular event will be placed on a list to be executed as soon as the current process blocks.

A behavioral model can wait for a named event by executing a statement like this one:

```
@do_fetch;
```

Upon executing this statement, the procedure will block, or wait, until some other procedure activates the named event. Events are activated by the following type of statement:

```
->do_fetch;
```

Note that this event activation *does not* cause the activating procedure to block. That is, any procedures that are waiting on this event will only start to execute when some subsequent statement causes the activating procedure to block.

Integer Variables

Integer variables are used for general programming purposes in a Verilog behavioral model, such as loop counters, array indexing, and so on. Although a `reg` variable can serve the same purposes, using an `integer` provides a clearer indication that this quantity does not represent hardware storage.

In the LVM implementation, an `integer` is equivalent to 32-bit `reg`, except that the `integer` is treated as a *signed quantity* for arithmetic purposes (unlike a `reg`, which is unsigned). Like a `reg`, each bit can have four states, 0, 1, `X` and `Z`. In addition, you can declare arrays of `integer` variables in the same manner as for `reg`.

Here are two examples of integer declarations:

```
integer  i, j, k;
integer  access_count[0:15];
```

More information on the treatment of `integer` *variables in expressions is provided in "Arithmetic Operators" on page 67.*

Time Variables

Time variables are used to store time values for programming and debugging purposes. In the LVM implementation, a `time` variable is identical to an `integer` variable except that it is treated as an unsigned quantity in arithmetic comparisons.

Here are two examples of time declarations:

```
time  start_time;
time  exec_time_histogram[1:100];
```

Unsupported Data Types

The following types of objects are *not* supported in the LVM:

```
wire
tri0/tri1
supply0/supply1
trireg
real
parameter
```

The first four items are used to express a circuit design in structural form. In LogicWorks, the same function can be achieved by creating multiple Verilog models and wiring them together in a schematic diagram. The `real` and `parameter` data types are used in larger system descriptions that are beyond the scope of application of LogicWorks.

8 Expressions and Assignments

This chapter explains how to compute values using the various arithmetic and Boolean operations available in the Verilog language. Most of these operations will be familiar to users of any standard programming language, but it is important to note how hardware-specific issues influence their use. In particular, the exact bit length of a value and the use of High Impedance or Don't Know states can have a large effect on the results.

Operands

Several different types of objects can supply the values used in computing expressions. The following table summarizes the allowable operand types.

Description	Example
Constant number	8'b1101_0000
`reg`, `integer`, `time`	regA
`reg` bit select	regA[3]
`reg` part select	regA[7:4]
Array element	mem[3]
Call to system function	$time

Operators

The arithmetic, logical, bit-slicing, concatenation, and other operators supported in the LVM are described in the following table.

Precedence	Operator	Description	Comments
1	{}	Concatenation	See next section.
1	~	Bitwise negation	Converts all 0 bits to 1 and 1 bits to 0; X remains X.
1	!	Logical negation	Converts any nonzero operand to zero and zero to 1.
1	+ (unary)	Positive	Takes precedence over + as a binary operator.
1	– (unary)	Negation	Takes precedence over – as a binary operator.
2	%	Modulus	Result takes the sign of the first operand.
2	*	Multiplication	
2	/	Division	Truncates any fractional part.
3	+	Addition	
3	–	Subtraction	
4	<<	Logical shift left	Vacated bits filled with zero.
4	>>	Logical shift right	See preceding entry.
5	<	Less than	Returns 1 if the relationship is true, 0 if it is false, X if either of the operands is unknown.
5	<=	Less than or equal to	See preceding entry.
5	>	Greater than	See preceding entry.
5	>=	Greater than or equal to	See preceding entry.
6	!=	Inequality	See preceding entry.
6	!==	Inequality, including X and Z	Inverse of ===.
6	==	Equality	If either operator contains X or Z, result is X.

6	===	Equality, including X and Z	Result is 1 if each bit has identical value in each operand. X matches X and Z matches Z. Always returns 0 or 1.
7	&	Bitwise AND	
7	&	Reduction AND (unary)	Computes AND of all bits in the operand.
7	~&	Reduction NAND (unary)	Computes AND of all bits in the operand and then inverts the result.
8	^	Exclusive OR	
8	^	Reduction Exclusive OR (unary)	Computes XOR of all bits in the operand.
8	~^ ^~	Reduction Exclusive NOR (unary)	Computes XOR of all bits in the operand and then inverts the result.
9	|	Bitwise OR	
9	|	Reduction OR (unary)	Computes OR of all bits in the operand.
9	~|	Reduction NOR (unary)	Computes OR of all bits in the operand and then inverts the result.
10	&&	Logical AND	Returns 1 if both operands are nonzero, 0 otherwise.
11	||	Logical OR	Returns 1 if either operand is nonzero, 0 if both are 0.
12	? :	Conditional	See next section.

Arithmetic Operators

The arithmetic operators +, –, *, /, and % (modulus) perform their standard binary arithmetic operations, with the following particular properties:

- The modulus operator % gives the remainder left after dividing the left hand operand by the right hand operand using integer division. If either operand is negative, the result is the remainder after dividing the absolute value of the left hand operand by the absolute value of the right hand operand, and then giving the result the sign of the left hand operand.

- The operators + and – can be used in unary form and take precedence over the same operators in binary form.
- With all the arithmetic operators (except unary +), if any bit of either operand is X, the entire result will be X. In the LVM implementation, unary + returns its operand exactly.
- The contents of `reg` variables are always treated as unsigned (i.e., positive) quantities, regardless of what kind of constant or expression was used to assign the value. On the other hand, `integer` variables are treated as signed values.

Comparison Operators

All the comparison operators produce the value 1'b0 (false), 1'b1 (true) or 1'bx (unknown), depending on the values of the operands. The table summarizes the specific response of each of these operators to the presence of X bits in the operands.

< > <= >=	If any bit in either operand is X or Z, result is X.
==	If all bits that are known in both operands are the same, then if any bit in either operand is an X, result is X. If the known bits are different, result is 0.
!=	Inverse of ==.
===	This is called the "case equals" because it operates like the case statement; that is, all bits of both operands must match bit-for-bit: X matches X and Z matches Z. This operator never produces an X result.
!==	Inverse of ===.

Logical Operators

The operators &&, ||, and ! always return 1'b1 (true), 1'b0 (false), or 1'bx regardless of the sizes of the operands. A 0 operand is taken as false, any value containing 1 bits is true, and any value containing only 0 and X bits is

unknown. Their operation with respect to X values is summarized in the following table.

&&	If either operand is zero, the result is false. Otherwise, if either operand is unknown, result is X.
||	If either operand is true, the result is true. Otherwise, if either operand is unknown, result is X.
!	If the operand is unknown, result is X.

Bitwise versus Reduction Operators

The operators ~ (negation), & (and), | (or) and ^ (xor), and their negations, can be used in either binary (bitwise) fashion or unary (reduction) fashion.

When used with two operands, the result is produced by performing the specified operation on each bit of the operands and placing the result in the corresponding bit of the result. The size of the result is equal to the size of the largest operand.

When used with a single operand, the result is 1 bit wide and is obtained by applying the operator successively on each bit of the operand.

Concatenation Operator {}

The concatenation operator takes two or more operands in braces, separated by commas, as in this example:

```
{4'b1111, reg_Sel, accum}
```

If `reg_sel` contained 4'b0101 and `accum` contained 8'b00000000, the result of this operation would be 16'b1111_0101_0000_0000. The bits of the rightmost operand are aligned at the LSB of the result, the bits of the next operand to the left are concatenated on the left, and so on.

IMPORTANT: In order for this operation to be unambiguous, all the operands *must* have an explicit size; unsized constants are not allowed.

The concatenation operator can also take an optional repetition factor, as in

```
{4{4'b1010}}
```

This is equivalent to 16’b1010101010101010. The repetition factor must be a constant.

Conditional Operator ? :

The conditional operator ? : selects one of two operand values based on the truth of a logical expression. For example,

```
a ? q : r
```

returns `q` if `a` is true (nonzero) and `r` if `a` is false (zero). If `a` is unknown, `q` and `r` are compared bit by bit. If the bits in a given position are the same, that value is returned for that bit. Otherwise X is returned for that bit.

Bit Lengths in Expressions

In order to accurately mimic the operation of a hardware system, Verilog truncates all operations to a specific bit length. For example, the multiplication operator can create expression values much larger than either of its operands, but then it trims them to a specific bit length.

Here are the rules that determine the bit length of an operation:

- Integers, time variables, and unsized constants are considered to be 32 bits long.
- All arithmetic operators, bitwise operators, and the conditional operator return a value that is as long as the longest operand.
- The concatenation operator returns a result whose width is the sum of the widths of the operands.
- Comparison operators, reduction operators, and logical operators return a value that is 1 bit long.

- The shift operators return a value that is the same width as the left hand operand.

Continuous Assignments

Continuous assignments are used only at the top level of a Verilog module (i.e., not inside an `initial` or `always` block) and are used to assign values to output ports. They are sometimes referred to as *top-level continuous assignments*, to distinguish them from *procedural continuous assignments*, which we describe later.

Since vectored ports are not supported in the LVM, you can use continuous assignments to break the individual bits of a `reg` variable out to the corresponding output ports. Here are some examples of continuous assignments:

```
assign D0 = d_out[0];
assign RCO = Q == 4'b1111;
```

As the name implies, this type of assignment is continuous and permanent; that is, any change in the variables in the expression results in an immediate reevaluation of expression and assignment of the new value.

NOTE: Drive-strength specifications are not supported in the LVM package.

Procedural Assignments

Procedural assignments are assignments that occur in behavioral Verilog code, that is, inside an `initial` or `always` block. Procedural assignments are similar to assignments in a standard programming language such as C, in that the assignment is instantaneous and takes effect only when the flow of control of the program reaches that point. Once the assignment is made, the variable on the left hand side holds the new value until it is changed by the execution of another procedural assignment. There is no automatic

updating of values as with the continuous assignments described previously.

The left hand side of an assignment must be either a scalar (i.e., nonarray) `reg`, `integer`, or `time` variable, an array element of one of these types, or a bit-select of a `reg`. Some examples of valid left hand sides for assignments follow:

```
reg_A = 8'b0;
reg_A[0] = 0;
reg_A[3:0] = 0;
mem_array[addr] = bus_data;
exec_time[count] = $time;
```

Note that in the example `reg_A[0] = 0;` the left hand side may mean one of two things depending on the declaration of `reg_A`. If the variable is a simple `reg`, then the LHS refers to bit 0 of the `reg`. If `reg_A` is an array, this refers to element 0 of the array.

NOTE: The LVM *does not* support the concatenation operator on the left hand side of an assignment.

Blocking Procedural Assignments

A *blocking procedural assignment* is the standard type of assignment in Verilog, as shown in these examples:

```
pc = oldAddr + 1;      // Simple expression
mem[addr] = reg_B;     // Memory assignment
cond_code[1:0] = 0;    // Bit-select
out_data = #5 accum;   // With delay control
bus = @transfer d_out;// With event control
```

The blocking procedural assignment consists of an expression on the right hand side and a destination on the left hand side, separated by a simple equal sign.

The term *blocking* refers to the fact that execution of the procedure waits, or blocks, until the expression evaluation and assignment are complete. No following statements in this execution path are executed until this one is

complete. This aspect of the blocking procedural assignment is significant *only* if some delay is specified in the assignment statement. If no delay is specified (i.e., no "#" delay control or "@" event control), then the assignment will be completed immediately.

However, if any delay or event control is specified (as in the last two preceding examples), then the execution of the procedure will block until the specified wait conditions are satisfied. Only then will execution continue with the next statement in line.

For more information on delay and event controls, see "Time Control: Delays and Events" on page 87.

Nonblocking Procedural Assignments

Nonblocking procedural assignments differ from blocking procedural assignments in the way they affect the control flow of the procedure containing them. Here are some examples of nonblocking procedural assignments:

```
dbus <= 8'bz;      // Set bus to high-Z
q <= d;            // Transfer data
siz[2:0] <= 3'b010;// Set at end of time step
```

A nonblocking assignment causes the simulator to execute the following steps:

- The simulator immediately evaluates the right hand side expression.
- The expression value is placed on a list for later assignment at the end of the time step.
- Execution proceeds immediately with the next statement.
- The stored value is assigned to the left hand side at the end of the time step.

NOTE: Delay and event control is *not supported* in nonblocking assignments in this version of the LVM.

Procedural Continuous Assignments

Procedural continuous assignments are used within behavioral code to place a continuous assignment on a `reg` variable. Procedural continuous assignments have the following characteristics:

- The continuous assignment overrides any other previous or future procedural assignment to the same variable.
- The right hand side of the assignment is evaluated at the time the `assign` is executed, and only that value is used. Values are not updated automatically as they are with top-level continuous assignments.
- Procedural continuous assignments can be applied and removed at will during the execution of a module. When a continuous assignment is removed (using the `deassign` command), the value of the variable reverts to its former value, and the "forcing" effect of the `assign` is removed. That is, subsequent procedural assignments will again be allowed to affect the value of the variable.
- When a continuous assignment is applied to a variable that already has one in effect, the old assignment is deassigned before applying the new one; that is, continuous assignments are *not* nested.
- The `assign` and `deassign` commands on a given variable do *not* have to be in the same code thread, that is, in the same `initial` or `always` block.

Some examples of the `assign` and `deassign` commands follow:

```
assign q = 0;
assign en = addr[15:12] == 4'b1101;
deassign en;
```

In the first example, the value of `q` will be forced to zero, and standard blocking or nonblocking procedural assignments to `q` will be ignored until a `deassign` is done on `q`. This can be used, for example, to apply a hard reset condition to a system, overriding all other activity.

In the second example, the value of `en` will be set based on the value of `addr` at the time of execution. The third example is the corresponding `deassign`.

9 Procedural Constructs

Verilog includes a variety of programming constructs that allow a system to be defined from a behavioral point of view, rather than as a detailed logic design. This frees the designer from having to think about gates and flipflops while defining the higher-level operation of a system.

`always` and `initial` Blocks

A Verilog module can contain any number of procedures operating in parallel, effectively modeling the parallel operation of subunits in a hardware system. Each of these parallel flows of execution is called a *thread,* or *procedure,* and can start, wait for specified conditions, repeat, and even terminate without affecting the others.

Each `initial` or `always` statement that appears in a module starts a new thread, and all threads start immediately at time zero as soon as the simulation is started. Once started, each thread can independently proceed with executing statements or wait for some specified condition, using delay- and event-control constructs.

`initial` and `always` blocks have exactly the same construction, but differ in behavior, as follows:

- An `initial` block is executed exactly once. After all statements in the block have completed execution, the thread is terminated and is not executed again until the entire simulation process is reset. `initial` blocks are typically used to initialize variables, perform simulated power-up reset functions, specify initial stimulus, and so on.

- An `always` block is executed repeatedly until simulation is terminated: in effect it is an "infinite loop". For this reason, it is essential that an `always` block contain at least one delay- or event-controlled statement. Without some form of delay, an `always` block will execute and repeat in zero time and completely lock up the simulation process. `always` blocks are typically used to express the ongoing behavior of the system.

Execution Order of **`initial`** and **`always`** Blocks

Since the Verilog simulator is running on a standard, sequential CPU, the "parallel" threads are obviously not executing truly concurrently. When the simulation starts at time zero, the `initial` and `always` blocks in a module are started in the order in which they are encountered in the source file. Once a thread is started, it runs until it blocks (i.e., specifies some delay or wait condition), then the next thread is started. This order is important for initialization of variables, as in this example:

```
initial
  clk = 0;

always
  begin
    clk = !clk;
    #`t_clk;
  end

initial
  $display("clk = %h", clk);
```

This code fragment causes the following sequence of actions:

- The first `initial` block will set `clk` to 0 at time zero, and then that thread terminates.
- The `always` block will begin executing immediately, also at simulation time zero. The variable `clk` will then be inverted to 1, and then this thread blocks, waiting for the specified delay.
- Finally, the last `initial` block will be executed, still at simulation time zero, and display the value 1 for `clk`. This thread then terminates.

- For as long as the simulation is run, the `always` block will then repeat after each specified delay.

Note that, in effect, the starting value of `clk` will be 1 because it spends no time in the 0 state, even though it was initialized that way.

Execution Flow Within a Procedure

In contrast to most computer programming languages, Verilog has the concept of simulation time. Simulation time starts at zero when the LogicWorks simulator is reset, then advances in response to delays specified in the Verilog model and in the external LogicWorks design.

The simulation time at which a procedural statement is executed is important because it affects the timing sequence of the system's outputs and its responses to stimulus. The following two rules summarize the passage of time in a Verilog procedure:

- Unless otherwise specified (i.e., using events, delays, or waits), all behavioral statements in Verilog execute in zero simulation time.
- A single thread executes without interruption as long as no time passes.

These rules are especially important when using an `always` statement (or any of the other loop constructs described later in this chapter), since it is in effect an infinite loop. If no delay or event control is specified anywhere in the loop, it will execute forever without allowing any other thread to be executed.

NOTE: The LVM allows execution of a model to be interrupted using ⌘-. (Macintosh) or ESC (Windows). More information appears under "User Interrupt" on page 41.

begin/end Blocks

We use `begin/end` blocks to group statements that are controlled by the `initial`, `always`, `if`, `repeat`, and other behavioral constructs described

in this chapter. Syntactically, each of these constructs consists of a keyword followed by a single statement.

We sometimes refer to begin/end blocks as *sequential blocks* because the statements contained in them are executed sequentially; that is, each successive statement is executed only after the previous one has completed. Consider this example:

```
initial  b = 0;
```

In this case, the entire construct consists of the controlling keyword initial followed by the simple statement b = 0;. In order to allow multiple statements to be grouped into this initial block, you can use begin and end to bracket the group, as follows:

```
initial
  begin
    a = 0;
    b = 0;
    count = 100;
  end
```

The begin, end, and everything in between constitute a single compound statement that can be placed in any construct that calls for a statement.

NOTE: The fork/join *parallel blocks are not implemented in this version of the LVM.*

Named **begin/end** Blocks

Sequential blocks can be named, as in the following example:

```
begin : bus_cycle_clock
  reg    phase;
  phase = 0;
  @cycle_clk;
  phase = 1;
  @cycle_clk;
end
```

In this case, the block is named `bus_cycle_clock`. Naming a block allows you to use the following features:

- The block can contain local variables, like the `reg` variable `phase` in the previous example. The names of these variables are known only within the block, thus reducing the chance of name conflict with other, unrelated parts of the model. Note that all variables are "static"; that is, their storage is permanently allocated, and they retain their values between successive executions of the block.
- A named block can be disabled (i.e., terminated) using the `disable` command described later in this chapter.

Conditional Statements

The conditional statements are the heart of the language; we use them to look at the model's inputs and make decisions about how to proceed. Although these constructs will look familiar to users of any standard programming language, care should be taken to note the effect of unknown and high-impedance values on their operation.

The **`if-else`** Statement

The conditional `if-else` statement allows selection of which statements to execute based on the value of an expression. Here are some examples:

```
if (reset) assign q = 0;
  else deassign q;
if (count >= `MaxCount) count = 0;
if (!_int_mask)
  begin
     ->do_fetch;
     @done_fetch;
  end
```

In all cases, the statement (or `begin`/`end` block) after the `if` is executed if the expression is known and nonzero. If the expression is unknown (i.e., X or Z) or zero, the statement after the `else` (if any) is executed.

Any statement can be used after the `if` or `else`, including another `if`. In this way you can create the following commonly used `if-else-if` structure to make decisions in order of descending priority:

```
if (reset)
  q = 0;
else if (load_en)
  q = in_data;
else if (count_en)
  q = q + 1;
else // do nothing
```

The **case** Statement

The `case` statement is used to choose one statement out of many to execute, based on the value of a single expression. The `case` statement in Verilog is somewhat more general than in many standard programming languages because the case comparison values can themselves be arbitrary expressions, not just constants. Here is an example of a `case` statement used for address decoding:

```
case (addr)
  0 :->do_reset;
  switch_setting :
     enable_pia = `TRUE;
  switch_setting+1 :
     enable_uart = `TRUE;
  switch_setting+2 : ;// Do nothing
  8`hff :
     begin
        enable_pram = `TRUE;
        $display("PRAM enabled");
     end
  default :
     $display("No match at %h", addr);
endcase
```

In effect, the `case` statement is like an `if-else-if` construct that compares the value of the expression following the keyword `case` with the value of the expression preceding each ":". As soon as a match is found, the corresponding statement or `begin/end` block is executed, and no more checking is done. If no match is found, the `default` item is executed. The default group is optional, and if none is provided and no value match is found, nothing is executed.

Take note of these important issues regarding `case` statements:

- When case values are compared, an exact bit-for-bit match is required in order for the corresponding statement to be executed; leading zeroes, X's and Z's must match exactly. For this reason, case values should be carefully specified in terms of bit length. For example, if no length is specified in a constant such as `bx, it will generate a 32-bit X value. This *will not* match an X value in, say, an 8-bit register. The problem applies to negative numbers because of the leading 1 fill.
- Because the case values can be arbitrarily complex expressions, several may produce the same value. However, only the first item that matches is executed. No checking is done for duplicate case values.
- Not all commercial Verilog compilers support the use of arbitrary expressions as case values. If compatibility is an issue in your application, you may wish to check whether your other systems will accept nonconstant case values.

Multiple Case Values in One Statement

The `case` statement syntax allows for multiple case value expressions associated with one case, as in this example:

```
case (instr)
  `ADD :       acc = acc + data;
  `SUB :       acc = acc - data;
  `BEQ, `BNE : if (condition) newAddr = data;
endcase
```

Using Don't Care Case Values

Two variations on the `case` statement allow you to control which bits in the comparison values are used in the matching process. You can use this, for

example, in an instruction decoder, where some bits of some instructions are used for register selection and are not significant to the decoding process.

In the `casez` construct, any bit which has a Z value in either expression is ignored when performing the match. In the following example the case values are given using the character "?", which Verilog allows in place of Z to make it more clear that they are being used as Don't Cares:

```
casez (instr)
// Load immediate with data in low 7 bits
8`b1??????? :acc = instr & 8`b1111111;
// Load from register with reg # in low 4 bits
8'b0101???? :acc = regfile[instr & 8`b1111];
endcase
```

An alternative form of this construct is introduced by the `casex` keyword. This form operates in a similar fashion to `casez`, except that both Z and X bits are ignored.

Looping Statements

The Verilog language defines four types of loop constructs that control repeated execution of a sequence of a statement or `begin/end` block.

`forever`	This is a simple construct that repeats indefinitely.
`repeat`	A simple structure that executes its statement a fixed number of times.
`while`	Executes its statement as long as its test expression is true (nonzero).
`for`	The most general loop structure, it allows for an arbitrary initialization statement, a termination test, and a loop control statement.

The **forever** Loop

The `forever` loop repeats indefinitely until the simulation terminates, or until a `disable` is applied from inside the loop or from another thread.

NOTE: The statement or block controlled by `forever` must contain a disable or some timing controls, or it will loop forever and lock up the simulator.

Here are some examples of `forever` loops:

```
forever #5 clk = !clk;
begin : wait_block
forever
  @step_done
      if (all_steps_done) disable wait_block;
end
```

In the first example, the variable `clk` will be inverted every 5 time units until the simulation terminates. Note that no exit mechanism is provided, so no following statements in this thread will ever be executed.

The second example uses a named `begin`/`end` block inside the `forever`. This allows the use of a `disable` to terminate execution of the loop. Note that the `disable` could also be applied from some other parallel thread elsewhere in the module.

For more information on `disable` *see "Disables" on page 85.*

forever versus **always**

The `forever` and `always` constructs are very similar and you can often use them to implement the same logic. Note, however, the following distinctions between these two statement types:

- `always` is used only at the top level of a module, that is, not inside any other statement or `begin`/`end` block; `forever` is only used *inside* a behavioral block, that is, inside an `always` or `initial`.
- `always` introduces a new parallel thread; `forever` controls the flow of execution within a thread.

The **repeat** Loop

The repeat loop executes its statement a fixed number of times, based on the value of a control variable. Here are some examples of valid repeat loops:

```
repeat (clk_count) #`DELAY clk = !clk;
repeat (8)
begin
  bit_count = bit_count + 1;
  bit_val = bit_val << 1;
  accum = accum + bit_val & reg_A;
end
```

The **while** Loop

The while loop executes the given statement or begin/end block as long as the control expression is true. If the control expression is false the first time the while is encountered, the statement is not executed at all.

Here are some valid examples of while loops:

```
while (clk_en) #`T_CLK clk = !clk;
while ((i < length) && (buf[i] != 0))
  begin
     checksum = checksum + buf[i];
     i = i + 1;
  end
```

The **for** Loop

The for loop is the most general loop construct in Verilog. It can replace any of the preceding loop structures, but due to its greater complexity, it may not be as clear in meaning as the simpler loops. The for loop has the following format:

```
for (initialization statement;
     control expression;
     step statement)  statement
```

The initialization statement is executed once before the loop is started. The control expression is evaluated once before each execution of the loop. If it is true, the loop is executed once; otherwise the loop is terminated. The step statement is executed once after each pass through the loop. The initialization statement and step statement are each optional, although the separating semicolons are not. The control expression is not optional and must produce a valid logical true or false value.

Here are some examples of valid `for` loops:

```
for (i = 0; i < length; i = i + 1) c_array[i] = -1;
for ( ; enabled; ) #5 clk = !clk;
for (t_size = 0 ; thing != 0; thing = thing >> 1)
   t_size = t_size + 1;
```

Disables

The `disable` statement is used to discontinue the execution of a named `begin`/`end` block. You can use it to handle error conditions and asynchronous events such as resets and retriggers.

The format of the `disable` statement is as follows:

```
disable block_name;
```

Here is an example that illustrates the use of a disable to terminate a processor's fetch/execute cycle when a reset input is received:

```
// When _RESET rises, start executing
always @(posedge _RESET)
begin : fetch_exec
  // Execute forever unless terminated
  forever
     begin
        if (instr == `ADD) a = a + data;
        .
        .
        pc = pc + 1;
     end
end

// When _RESET falls, terminate exec loop
always @(negedge _RESET)
  begin
     disable fetch_exec;
     pc = 0;
  end
```

Note that `disable` only terminates operations between the named `begin` and its corresponding `end`. In the preceding example, `disable fetch_exec` *does not* terminate the `always` loop containing the disabled block. It terminates only the current execution of the enclosed `begin/end` block. This means that the procedure immediately loops back to the top of the `always` block and again waits on the _RESET event. In effect, `disable` simply causes a jump to the end of the named block.

Note these additional points regarding `disable`:

- The `disable` statement has no lasting disabling effect on the named block. That is, once the execution of the block is terminated, it can be executed again immediately if an enclosing loop construct causes control to be passed into it again.
- The `disable` statement causes all scheduled nonblocking assignments in the named block to be removed from the execution list.
- A block can disable itself. For example, a `disable` can be used within a loop to discontinue the loop early when some condition arises.
- If named blocks are nested and an outer one is disabled, all inner ones are disabled at the same time.

10 Time Control: Delays and Events

This chapter describes three constructs that are used to specify simulated delays and synchronization of timed processes within a model.

- A *delay control*, introduced by the "#" symbol, can be used to delay the execution of any behavioral statement or assignment by an explicit amount.
- An *event control*, introduced by the "@" symbol, waits indefinitely for some specified value change or named event to take place.
- The `wait` statement provides a third type of delay construct that waits for a specified condition or proceeds immediately if that condition already exists.

Delays

A delay control consists of the "#" character followed by a delay value. The delay value may be specified as a constant, a variable, or an arbitrary expression. The execution of this thread is then blocked for the specified number of time units before execution of the statement proceeds.

NOTE: Min/Typ/Max delays are not supported in this version of the LVM.

Statement Delays

The following example illustrates the use of a simple constant delay:

```
#10 bus = 8'bzzzz_zzzz;
```

NOTE: Intra-assignment delays (e.g., `a = #2 b`) behave somewhat differently than statement delays; they are described in "Assignment Delay Control" on page 89. The discussion here refers to delays used in front of a statement.

The next example uses a `reg`, `time`, or `integer` variable to determine the delay:

```
#new_delay ->do_fetch;
```

The following example uses an expression to determine the delay. Note that the expression must be enclosed in parentheses in order to allow the parser to recognize the end of the expression and the start of the controlled statement.

```
#((`T_MAX + count) / 2) Q0 = serin;
```

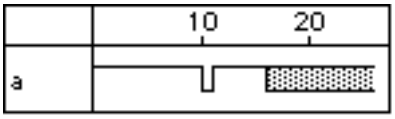
In the next example, the delay control appears by itself without an associated statement. This will simply delay execution of the current thread by the specified amount before proceeding with the next statement. The delay is specified using a constant defined elsewhere in a `` `define `` statement.

```
#`T_CLK_TO_D;
```

With the exception of nonblocking assignments, delays always start at the time of completion of the previous statement. For example, when delays are used to generate a stimulus sequence, each delay in effect specifies the duration of the value set in the previous statement, as in this sequence:

```
a = 1;
#10 a = 0;
#1 a = 1;
#5 a = 'bz;
#10 a = 0;
```

These lines will generate the following waveform:



Assignment Delay Control

As shown in a number of the examples used elsewhere, delay control can be included in an assignment by using the "#" operator. You can use the delay operator in two different ways, as shown in these examples:

```
#5 n_gate = strobe && n_enable;

n_gate = #5 strobe && n_enable;
```

The first example is a standard statement-delay construct, of the kind described earlier. The second example is referred to as an *intra-assignment delay.*

Placing the delay specification in front of the entire statement, as in the first example, indicates that no action should be taken in executing the statement until the delay has expired. The important distinction here is that the expression is not evaluated, and the variables it refers to are not sampled until *after* the delay has expired.

Placing the delay specification after the equal sign, as in the second example, indicates an intra-assignment delay. This means that the expression should be evaluated immediately, then the delay applied, then the new value assigned to the left hand side. Note that in this case the actual assignment takes place at the same time, but the values of the variables in the expression are used immediately, rather than after the delay. The effect of the intra-assignment delay on the execution of the following statement depends on whether the assignment is blocking or nonblocking, as described in "Procedural Assignments" on page 71.

You can use the `define facility to create more readable delay specifications and define all module timing in one part of the source file, as illustrated in the following example:

```
`define t_d_to_q  15
.
.
.
q = #`t_clk_to_q  = d;
```

Events

An event control, introduced by "@", allows the execution of a procedure to be synchronized with some external condition. Events fall into two categories:

- Value-change events – Any change in value of a port or `reg` variable.
- Named events – Events explicitly declared using the syntax outlined in the next section.

The following sections illustrate the various types of event controls available.

Named Events

In the first example, the execution of the thread will block until the named event is activated by some other thread:

```
@do_fetch bus_active = 1;
```

As with delay controls, you can use an event control without an associated statement to document more clearly that the execution flow of the procedure is held up until the named event is activated. The following statements have exactly the same effect as the preceding one:

```
@do_fetch;

bus_active = 1;
```

The named event would be activated by some other procedure using a statement of the form:

```
->do_fetch;
```

NOTE: The declaration of named events is covered in "Named Events" on page 62.

Value-Change Events

An event control can also refer directly to a declared `reg` variable, as in this example:

```
@cc flag = cc[2];
```

In this case, any value change in the variable `cc` will be considered as an event, and execution will proceed. You can also give a specific direction of change using the `posedge` and `negedge` specifiers, in the following format:

```
@(posedge clk) Q = D;
```

The variable or expression after a `posedge` or `negedge` must resolve to a 1-bit binary value.

A `posedge` event control responds to any value change *to 1* or *from 0*, for example, X to 1 or 0 to X. Similarly, the `negedge` specifier responds to 1 to 0, X to 0, or 1 to X.

Event OR Construct

An OR construct can be used to indicate that an occurrence of any of the specified items will be sufficient to unblock the procedure, as illustrated in this example:

```
@(sel1 or sel2) sel_out = sel_array[{sel2,sel1}];
```

In this case, any value change in either variable `sel1` or `sel2` will cause the statement to be executed.

Any combination of different event types can appear in an OR construct:

```
@(posedge pclk or negedge nclk) Q = D;
```

Assignment Event Control

Assignment event control is used in a similar fashion to delay control, as described in “Assignment Delay Control” on page 89. In this case, however, the execution of the statement is delayed until a specific event is activated by some other behavioral code in the module. Here are some examples of assignment event control:

```
q = @(posedge clk) d;
shift_data = @load d_bus;
@en_low D[7:0] = 8'bz;
```

The first example uses *intra-assignment event control*, with the same kind of effect as intra-assignment delay control. In particular, the value of `d` is sampled immediately upon execution of the statement. This value is then placed on a queue of future assignments. When the value of variable `clk` changes from 0 to 1 (i.e., a positive edge), the stored value is assigned to `q`. This may happen at any point in the future, or it may never happen at all, depending on the logic of the module. Since this is a blocking assignment, execution *does not* proceed to the next statement until the event occurs.

The second example differs from the first in that it uses a *named event*, that is, an event variable declared earlier in the module. The event must be explicitly signaled by some other behavioral code using the `->load` syntax described in “Named Events” on page 90.

The third example illustrates the standard statement event control (i.e., not intra-assignment) applied to an assignment. In this case, no action is taken in executing this statement until the named event is activated by another procedure.

Repeat Event Construct

A `repeat` construct can be used in conjunction with intra-assignment event control to specify a wait of some number of occurrences of an event before the assignment is performed. For example,

```
ready = repeat('N_START_CLKS) @(posedge clk) 1;
```

Note that the `repeat` keyword must be followed by an expression in parentheses that can be resolved to an integer value.

The **wait** Statement

The `wait` statement is similar to an event control, except that if the specified condition already exists at the entry to the `wait`, execution proceeds through to the controlled statement. The `wait` statement does not demand that a value change take place before it will consider an event to have occurred.

Here is an example:

```
wait (!_bus_req) _bus_active = 0;
```

This statement will execute the assignment and proceed immediately if `_bus_req` is zero. If not, it will block and wait for a zero value.

Note that although the `while` loop is similar in logic, it does not cause any blocking to occur. For example, the following statements look like they would implement the same logic as the above `wait`, but they are in fact invalid:

```
while (_bus_req);

_bus_active = 0;
```

Since the `while` does not block, if `_bus_req` was 1 on entry, the while loop would *repeat infinitely* and never allow any other procedure to execute and change the value of `_bus_req`. This could be fixed by adding event control to the `while`, as in this example:

```
while (_bus_req) @_bus_req;

_bus_active = 0;
```

With this addition, after each pass through the `while`, the execution of the procedure will block until a change in the value of the variable occurs.

Appendix A– Differences from OVI Verilog

A proper subset of the OVI Verilog language is implemented in the LogicWorks Verilog Modeler. In other words, we have not added any features to the language that would be incompatible with an OVI standard Verilog implementation.

The following concepts defined in the OVI Verilog Language Reference Manual are *not implemented* in the LVM:

- `wire`, `real` and `string` objects
- The { } concatenation operator as a destination for an assignment
- The `fork`/`join` behavioral construct
- Min/Typ/Max delays
- Module hierarchy
- Parameters and path delays
- Gate- and switch-level modeling
- User-defined primitives
- Constant expressions
- `force`/`release` assignments
- Tasks and functions
- The `repeat` keyword in event control
- Delay and event control on nonblocking assignments

In addition, the following limitations exist:

- Ports are implemented as single bits only and can be of type input, output, or inout.
- Time values are stored as 32-bit unsigned values.

Appendix B– Verilog Keywords

The following table lists the reserved keywords in the Verilog language. These reserved words should not be used as variable identifiers in Verilog models. Items marked with an asterisk are not supported in this version of the LogicWorks Verilog Modeler.

always	and*	assign
begin	buf*	bufif0*
bufif1*	case	casex
casez	cmos*	deassign
default	defparam*	disable
edge	else	end
endcase	endfunction*	endmodule
endprimitive*	endspecify*	endtable*
endtask*	event	for
force*	forever	fork*
function*	highz0*	highz1*
if	initial	inout
input	integer	join*
large*	macromodule*	medium*
module	nand*	negedge
nmos*	nor*	not*
notif0*	notif1*	or*
output	parameter*	pmos*
posedge	primitive*	pull0*
pull1*	pulldown*	pullup*
real*	rcmos*	reg
release*	repeat	rnmos*
rpmos*	rtran*	rtranif0*
rtranif1*	scalered*	small*
specify*	specparam*	strong0*

```
strong1*        supply0*        supply1*
table*          task*           time
tran*           tranif0*        tranif1*
tri*            tri0*           tri1*
triand*         trior*          trireg*
vectored*       wait            wand*
weak0*          weak1*          while
wire*           wor*            xnor*
xor*            $display        $fclose*
$fdisplay*      $finish         $fmonitor*
$fopen*         $fstrobe*       $fwrite*
$monitor*       $random*        $stime*
$stop           $strobe*        $time
$write*         $setup*         $hold*
$period*        $width*         $skew*
$recovery*      $setuphold*
```

Index

Addison-Wesley Technical Support

To the Student

Addison-Wesley provides help for students with installation issues, or if you feel you have received a defective product. We do not provide assistance with “how to” questions. Please consult with your instructor if you have a question on how to use the software, or if it appears a particular command or function does not give the expected results.

To the Instructor

We will be happy to provide assistance to adopters of LogicWorks Verilog Modeler with any issue that may arise. Please understand that on some occasions we may need to consult with the software developer for answers to specific problems, but we will make every attempt to obtain a fast answer for you.

Before you call Tech Support

- Make sure that the appropriate version of LogicWorks 3 is installed as specified in the Verilog Modeler README file.
- Please take time to consult the LogicWorks Verilog Modeler manual

and any release notes that came with the software. These items might answer your questions.

- Please document the problem if you are receiving error messages. When do they occur? What is the exact message? Can you recreate it?
- Check that your computer hardware setup meets or exceeds the minimum system requirements printed on the back cover of this book. Please take a moment to compare your hardware against the system requirements. Are your hardware and peripherals set up correctly and are all cables attached securely?
- Verify that your disk drive can read the disk correctly. A quick way for Windows users to check this is to view the directory of the disk. Do you see files?
- Be prepared to state the specifics of your computer hardware setup and the release of LogicWorks Verilog Modeler that you are using so we can answer your questions efficiently.

Reaching Addison-Wesley Tech Support

Voice:	(617) 944-2630	Monday–Friday, 9:00 AM to 4:30, EST
Fax:	(617) 944-9338	anytime
Email:	techsprt@aw.com	anytime

For news on related products and software updates, connect to the LogicWorks Verilog Modeler Home Page at the Addison-Wesley Computer Science & Engineering Web site:

```
http://www.aw.com/cseng/authors/capilano/lwvm/lwvm.html
```