
Contents

Volume 1

Introduction

Introduction	1–1
The Verilog Hardware Description Language	1–1
The Verilog–XL Logic Simulator	1–3
Major Features of Verilog–XL	1–3
Verilog–XL Licenses	1–4

Lexical Conventions

Lexical Conventions	2–1
Operators	2–1
White Space and Comments	2–2
Numbers	2–2
Strings	2–5
String Variable Declaration	2–5
String Manipulation	2–6
Special Characters in Strings	2–7
Identifiers, Keywords, and System Names	2–7
Escaped Identifiers	2–8
Keywords	2–9
Text Substitutions	2–9

Data Types

Data Types	3–1
Value Set	3–1
Registers and Nets	3–2

Return to MAIN MENU

Registers	3-2
Declaration Syntax	3-3
Declaration Examples	3-4
Vectors	3-5
Specifying Vectors	3-5
Vector Net Accessibility	3-5
Strengths	3-6
Charge Strength	3-6
Drive Strength	3-7
Implicit Declarations	3-7
Net Initialization	3-7
Net Types	3-8
wire and tri Nets	3-8
Wired Nets	3-8
triereg Net	3-9
tri0 and tri1 Nets	3-13
supply Nets	3-13
Memories	3-14
Integers and Times	3-16
Real Numbers	3-17
Declaration Syntax for Real Numbers	3-17
Specifying Real Numbers	3-17
Operators and Real Numbers	3-18
Conversion	3-18
Parameters	3-19

Expressions

Expressions	4-1
Operators	4-2
Binary Operator Precedence	4-4
Numeric Conventions in Expressions	4-5
Arithmetic Operators	4-5
Arithmetic Expressions with Registers and Integers ..	4-6
Relational Operators	4-7

Equality Operators	4-8
Logical Operators	4-9
Bit-Wise Operators	4-11
Reduction Operators	4-12
Syntax Restrictions	4-14
Shift Operators	4-15
Conditional Operator	4-15
Concatenations	4-16
Operands	4-17
Net and Register Bit Addressing	4-17
Memory Addressing	4-18
Strings	4-19
String Operations	4-20
String Value Padding and Potential Problems ..	4-20
Null String Handling	4-22
Minimum, Typical, Maximum Delay Expressions ..	4-22
Expression Bit Lengths	4-23
An Example of an Expression Bit Length Problem ...	4-24
Verilog Rules for Expression Bit Lengths	4-24

Assignments

Assignments	5-1
Continuous Assignments	5-2
The Net Declaration Assignment	5-3
The Continuous Assignment Statement	5-3
Delays	5-5
Strength	5-9
Procedural Assignments	5-9
Accelerated Continuous Assignments	5-10
The Restrictions on Accelerated	
Continuous Assignments	5-10
How to Control the Acceleration of	
Continuous Assignments	5-22
The Effects of Accelerated Continuous Assignments	5-24

**Gate and Switch
Level Modeling**

Gate and Switch Level Modeling	6-1
Gate and Switch Declaration Syntax	6-2
and, nand, nor, or, xor, and xnor Gates	6-6
buf and not Gates	6-8
bufif1, bufif0, notif1, and notif0 Gates	6-9
MOS Switches	6-10
Bidirectional Pass Switches	6-12
cmos Gates	6-13
pullup and pulldown Sources	6-14
Implicit Net Declarations	6-15
Logic Strength Modeling	6-16
Strengths and Values of Combined Signals	6-18
Combined Signals of Unambiguous Strength ...	6-18
Ambiguous Strengths: Sources and Combinations	6-20
Ambiguous Strength Signals and	
Unambiguous Signals	6-26
Wired Logic Net Types	6-30
Mnemonic Format	6-33
Strength Reduction by Non-Resistive Devices	6-33
Strength Reduction by Resistive Devices	6-33
Strengths of Net Types	6-34
tri0 and tri1 Net Strengths	6-34
trireg Strength	6-34
supply0 and supply1 Net Strengths	6-34
Gate and Net Delays	6-34
min/typ/max Delays	6-37
trireg Net Charge Decay	6-39
Gate and Net Name Removal	6-43

**User-Defined
Primitives (UDPs)**
De

User-Defined Primitives (UDPs)	7-1
Memory Usage and Performance Considerations	7-2
Syntax	7-3

UDP Definition	7-4
UDP Terminals	7-5
UDP Declarations	7-5
Sequential UDP initial Statement	7-5
UDP State Table	7-5
Combinational UDPs	7-6
Level-Sensitive Sequential UDPs	7-8
Edge-Sensitive UDPs	7-9
Sequential UDP Initialization	7-10
UDP Instances	7-14
Compilation	7-14
Symbols to Enhance Readability	7-15
Mixing Level-Sensitive and Edge-Sensitive Descriptions	7-16
Reducing Pessimism	7-17
Level-Sensitive Dominance	7-19
Processing of Simultaneous Input Changes	7-19
Summary of Symbols	7-21
Examples	7-22

Behavioral Modeling

Behavioral Modeling	8-1
Behavioral Model Overview	8-1
Procedural Assignments	8-3
Blocking Procedural Assignments	8-4
The Non-Blocking Procedural Assignment	8-4
How the Simulator Processes Blocking and Non-Blocking Procedural Assignments	8-11
Conditional Statement	8-11
if-else-if Construct	8-14
Example	8-15
Case Statement	8-16
Case Statement with Don't-Cares	8-19
Looping Statements	8-20
forever Loop	8-21
repeat Loop Example	8-22

while Loop Example	8-23
for Loop Examples	8-23
Procedural Timing Controls	8-25
Delay Control	8-26
Zero-Delay control	8-26
Event Control	8-27
Named Events	8-28
Event OR Construct	8-29
Level-Sensitive Event Control	8-29
Intra-Assignment Timing Controls	8-30
Block Statements	8-35
Sequential Blocks	8-35
Parallel Blocks	8-37
Block Names	8-39
Start and Finish Times	8-39
Structured Procedures	8-41
initial Statement	8-42
always Statement	8-43
Examples	8-43

Tasks and Functions

Tasks and Functions	9-1
Distinctions Between Tasks and Functions	9-1
Tasks and Task Enabling	9-2
Defining a Task	9-3
Task Enabling and Argument Passing	9-4
Task Example	9-6
Effect of Enabling an Already Active Task	9-7
Functions and Function Calling	9-8
Defining a Function	9-8
Returning a Value from a Function	9-9
Calling a Function	9-9
Function Rules	9-10
Function Example	9-11

1

Figure 1-0
Example 1-0
Syntax 1-0
Table 1-0

Introduction

This reference manual describes the features of the Verilog-XL™ digital logic simulator and the Verilog Hardware Description Language you use to model a design for simulation by Verilog-XL. There are three volumes in this reference manual. Section 1.3 consists of brief descriptions of all the chapters and appendices in this reference manual and shows the divisions between volumes.

The volumes of the reference manual are not separate documents. The reference manual has a table of contents, located at the front of each volume, and an index, located at the back of each volume.

1.1 The Verilog Hardware Description Language

The Verilog Hardware Description Language (HDL) describes a hardware design or part of a design. Descriptions of designs in the Verilog HDL are Verilog models. The Verilog HDL is both a behavioral and structural language. Models in the Verilog HDL can describe both the function of a design and the components and connections to the components in a design.

Verilog models can be developed for different levels of abstraction. These levels of abstraction and their corresponding model types are as follows:

algorithmic	a model that implements a design algorithm in high-level language constructs
RTL	a model that describes the flow of data between registers and how a design processes that data
gate-level	a model that describes the logic gates and the connections between logic gates in a design
switch-level	a model that describes the transistors and storage nodes in a device and the connections between them

The basic building block of the Verilog HDL is the module. The module format facilitates top-down and bottom-up design. A module contains a model of a design or part of a design. Modules can incorporate other modules to establish a model hierarchy that describes how parts of a design are incorporated in an entire design. The constructs of the Verilog HDL, such as its declarations and statements, are enclosed in modules.

The Verilog HDL behavioral language is structured and procedural like the C programming language. The behavioral language constructs are for algorithmic and RTL models. The behavioral language provides the following capabilities:

- structured procedures for sequential or concurrent execution
- explicit control of the time of procedure activation specified by both delay expressions and by value changes called event expressions
- explicitly named events to trigger the enabling and disabling of actions in other procedures
- procedural constructs for conditional, if-else, case, and looping operations
- procedures called tasks that can have parameters and non-zero time duration
- procedures called functions that allow the definition of new operators
- arithmetic, logical, bit-wise, and reduction operators for expressions

The Verilog HDL structural language constructs are for gate-level and switch-level models. The structural language provides the following capabilities:

- a complete set of combinational primitives
- primitives for bidirectional pass and resistive devices
- the ability to model dynamic MOS models with charge sharing and charge decay

Verilog structural language models can accurately model signal contention. In the Verilog HDL, structural modeling accuracy is enhanced by primitive delay and output strength specification. Signal values can have different strengths and a full range of ambiguous values to reduce the pessimism of unknown conditions.

1.2 The Verilog-XL Logic Simulator

The Verilog-XL digital logic simulator is a software tool that allows you to perform the following tasks in the design process without building a hardware prototype:

- determine the feasibility of new design ideas
- try more than one approach to a design problem
- functional verification
- identify design errors

To use Verilog-XL, you develop models that describe your design and its environment in the Verilog Hardware Description Language (the Verilog HDL) and then supply Verilog-XL with the file names that contain these models. To use Verilog-XL, you also need a Verilog-XL license. This section describes the major features of Verilog-XL and the Verilog-XL license.

1.2.1 Major Features of Verilog-XL

Verilog-XL provides you with the following simulation capabilities:

- set break points during simulation that stop the simulation and allow you to enter an interactive mode to examine and debug your design
- display information about the current state of the design and to specify the format of that information
- apply stimulus during simulation
- circuit patching during simulation
- tracing the execution flow of the statements in your model
- traversing the model hierarchy to various regions of your design to examine the state of the simulation in that region
- stepping through the statements of a design and executing them one at a time
- displaying the active statements in a design
- displaying and disabling the operations you entered in interactive mode

- reading data from a file and writing data to that file
- saving the current state of a simulation in a file and restoring that simulation at another time
- investigating the performance ramifications of architectural decision—stochastic modeling

1.2.2

Verilog-XL Licenses

To get permission to simulate with the Verilog-XL logic simulator you need a license. SoftShare™ handles all licenses for Verilog-XL 1.6c. Highland's FLEXIm™ (Flexible License Manager) software is the core of SoftShare technology.

Licenses for Verilog-XL 1.6c are all floating XL licenses, but it is possible to lock a license to a node. A Verilog-XL invocation causes SoftShare processes to seek a license file, and check out a license for you if one is available.

1.3

The Contents of the Reference Manual

Volume 1

- **Chapter 1 – Introduction**
This chapter discusses the major features of the Verilog HDL and the Verilog-XL logic simulator. It also discusses the contents of the reference manual.
- **Chapter 2 – Lexical Conventions**
This chapter describes how the language interprets and how to specify lexical tokens. A lexical token is one or more characters. Lexical tokens include white space, comments, numbers, character strings, identifiers, keywords, and operators. The chapter also describes the text macro substitution facility.
- **Chapter 3 – Data Types**
This chapter describes the Verilog HDL data types. The Verilog HDL has two main groups of data types: registers and nets. Registers and nets model storage devices and physical connections. The chapter also discusses the parameter data type for constant values and describes drive and charge strength of the values on nets.
- **Chapter 4 – Expressions**
This chapter describes the operators and operands that can be used in expressions.

- **Chapter 5 – Assignments**
This chapter compares the two main types of assignment statements in the Verilog HDL—continuous assignments and procedural assignments. It describes the continuous assignment statement that drives values onto nets.
- **Chapter 6 – Gate and Switch Level Modeling**
This chapter describes the gate and switch level primitive and their declarations and specifications.
- **Chapter 7 – User-Defined Primitives (UDPs)**
This chapter describes how a primitive can be defined in the Verilog HDL and how these primitives are included in Verilog models.
- **Chapter 8 – Behavioral Modeling**
This chapter describes procedural assignments and the behavioral language statements.
- **Chapter 9 – Tasks and Functions**
This chapter describes tasks and functions—procedures that can be called from more than one place in a behavioral model. It describes how tasks can be used like subroutines and how functions can be used to define new operators.
- **Chapter 10 – Disabling of Named Blocks and Tasks**
This chapter describes how to disable the execution of a task and a block of statements that has a specified name.
- **Chapter 11 – Procedural Continuous Assignments**
This chapter describes a type of procedural assignment called a procedural continuous assignment.
- **Chapter 12 – Hierarchical Structures**
This chapter describes how model hierarchies are created in the Verilog HDL and how parameter values declared in a module can be overridden. The chapter also discusses macro modules—a construct that saves memory and port collapsing—a technique that improves simulator efficiency.

Volume 2

- **Chapter 13 – Specify Blocks (SDPDs)**
This chapter describes the Verilog HDL constructs that belong in a construct called a specify block. Specify blocks contain pin-to-pin delays and timing checks.
- **Chapter 14 – State-dependent Path Delays**
This chapter describes pin-to-pin delays whose validity is conditioned by the values at pins.
- **Chapter 15 – Module Input Port Delays (MIPDs)**
This chapter describes modeling delays between certain drivers and their loads with Module Input Port Delays (MIPDs). It discusses the use of PLI access routines to insert MIPDs.

- **Chapter 16 – Timescales**
This chapter describes how you can use models that were developed with different time units together in a simulation.
- **Chapter 17 – Delay Mode Selection**
This chapter describes how you use command line options and compiler directives to alter the delay values in your models.
- **Chapter 18 – The Behavior Profiler**
This chapter describes how you can identify the behavioral modules and statements in your design that use the most CPU time during simulation.
- **Chapter 19 – Value Change Dump File**
This chapter describes how you can produce a file that contains information about value changes during simulation nets and registers that you select.
- **Chapter 20 – Source Protection**
This chapter describes how to protect proprietary Verilog HDL source descriptions from being accessed or modified.
- **Chapter 21 – System Tasks and Functions**
This chapter describes the general purpose system tasks and functions that are built into Verilog-XL.
- **Chapter 22 – Programmable Logic Arrays**
This chapter describes the system tasks that you can use to model PLA devices.
- **Chapter 23 – Stochastic Analysis**
This chapter describes the system tasks that you can use for stochastic analysis—investigating the performance ramifications of architectural decisions.
- **Chapter 24 – Compilation and Execution**
This chapter describes the compiler directives and command line options that control how Verilog-XL compiles and simulates your model.

Volume 3

- **Chapter 25 – Library Management**
This chapter describes two different schemes that enable you to save compilation time and memory by controlling what modules, in the source description file or directory, Verilog-XL compiles. The newer of the two schemes provides greater control over library scanning.
- **Chapter 26 – Interactive Control and Debugging**
This chapter describes the how to use the features of Verilog-XL's interactive mode.
- **Chapter 27 – XL Usage and Performance**
This chapter describes the high-speed XL algorithm that accelerates the simulation of gate and switch-level primitives and certain continuous assignments in a model. It discusses how you

invoke the XL algorithm, the primitives and continuous assignments whose simulation it accelerates, and performance expectations.

- **Chapter 28 – Switch-level Simulation**

This chapter describes three algorithms that simulate channel-connected switch networks, one of which is default. The first non-default algorithm, named the Switch-XL algorithm, employs the XL algorithm to simulate bidirectional switches and a strength model that allows you to specify a wide range of capacitances and conductances. The other non-default algorithm, named the Switch-RC algorithm, enables you to simulate with real resistances and capacitances and to describe the electrical characteristics of manufacturing technologies.

- **Chapter 29 – Software Behavior and Recommendations**

This chapter discusses software behavior that it is helpful to be aware of and some methods for dealing with it.

- **Appendix A – Formal Syntax Definition**

This appendix describes in the Baccus-Naur Format (BNF), the syntax of the Verilog HDL.

- **Appendix B – The Switch-RC Algorithm**

This appendix discusses the equations in the Switch-RC algorithm.

- **Appendix C – Switch-RC Technology Characterization**

This appendix describes how to derive values that describe Switch-RC technologies.

2

Figure 2-0
Example 2-0
Syntax 2-0
Table 2-0

Lexical Conventions

Verilog language source text files are a stream of lexical tokens. A token consists of one or more characters, and each single character is in exactly one token. The layout of tokens in a source file is free format—that is, spaces and newlines are not syntactically significant. However, spaces and newlines are very important for giving a visible structure and format to source descriptions. A good style of format, and consistency in that style, are an essential part of program readability.

The types of lexical tokens in the language are:

- operator
- white space
- comment
- number
- string
- identifier
- keyword

The rest of this chapter defines these tokens.

This manual uses a syntax formalism based on the Backus-Naur Form (BNF) to define the Verilog language syntax. Appendix A contains the complete set of syntax definitions in this format, plus a description of the BNF conventions used in the syntax definitions.

2.1 Operators

Operators are single, double, or triple character sequences and are used in expressions. Chapter 4 discusses the use of operators in expressions.

Unary operators appear to the left of their operand. Binary operators appear between their operands. A ternary operator has two operator characters that separate three operands. The Verilog language has one ternary operator the—conditional operator. See Section 4.1.12 for an explanation of the conditional operator.

2.2

White Space and Comments

White space can contain the characters for blanks, tabs, newlines, and formfeeds. The Verilog language ignores these characters except when they serve to separate other tokens. However, blanks and tabs are significant in strings.

The Verilog language has two forms to introduce comments. A one-line comment starts with the two characters `//` and ends with a newline. A block comment starts with `/*` and ends with `*/`. Block comments cannot be nested, but a one-line comment can be nested within a block comment.

2.3

Numbers

Constant numbers can be specified in decimal, hexadecimal, octal, or binary format. The Verilog language defines two forms to express numbers. The first form is a simple decimal number specified as a sequence of the digits 0 to 9 which can optionally start with a plus or minus. The second takes the following form:

```
<size><base_format><number>
```

The `<size>` element contains decimal digits that specify the size of the constant in terms of its exact number of bits. For example, the `<size>` specification for two hexadecimal digits is 8, because one hexadecimal digit requires four bits. The `<size>` specification is optional. The `<base_format>` contains a letter specifying the number's base, preceded by the single quote character (`'`). Legal base specifications are one of `d`, `h`, `o`, or `b`, for the bases decimal, hexadecimal, octal, and binary respectively. (Note that these base identifiers can be upper- or lowercase.)

The `<number>` element contains digits that are legal for the specified `<base_format>`. The `<number>` element must physically follow the `<base_format>`, but can be separated from it by spaces. No spaces can separate the single quote and the base specifier character.

Alphabetic letters used to express the `<base_format>` or the hexadecimal digits `a` to `f` can be in upper- or lowercase.

Example 2-1 shows *unsized* constant numbers.

```
659          // is a decimal number
'h 837FF     // is a hexadecimal number
'o7460       // is an octal number
4af          // is illegal (hexadecimal format requires 'h)
```

Example 2-1: Unsized constant numbers

Example 2-2 shows *sized* constant numbers.

```
4'b1001 // is a 4-bit binary number
5 'D 3   // is a 5-bit decimal number
3'b01x   // is a 3-bit number with the least
          // significant bit unknown
12'hx    // is a 12-bit unknown number
16'hz    // is a 16-bit high impedance number
```

Example 2-2: Sized constant numbers

In the Verilog language a plus or minus preceding the size constant is a sign for the constant number—the size constant does not take a sign. A plus or minus between the `<base_format>` and the `<number>` is illegal syntax. In Example 2-3, the first expression is a syntax error. The second expression legally defines an 8-bit number with a value of minus 6.

```
8 'd -6     // this is illegal syntax
-8 'd 6     // this defines the two's complement of 6,
              // held in 8 bits—equivalent to -(8'd 6)
```

Example 2-3: A plus or minus between the <base_format> and the <number> is illegal

The number of bits that make up an un-sized number (which is a simple decimal number or a number without the `<size>` specification) is the host machine word size—for most machines this is 32 bits.

In the Verilog language, an `x` expresses the unknown value in hexadecimal, octal, and binary constants. A `z` expresses the high impedance value. See Section 3.1 for a discussion of the Verilog value set. An `x` sets four bits to unknown in the hexadecimal base, three bits in the octal base, and one bit in the binary base. Similarly, a `z` sets four, three, and one bit, respectively, to the high impedance value. If the most significant specified digit of a constant number is an `x` or a `z`, then Verilog-XL automatically extends the `x` or `z` to fill the higher order bits of the constant. This makes it easy to specify complete vectors of the unknown and high impedance values. Example 2-4 illustrates this value extension:

```
reg [11:0] a;
initial
begin
    a = 'h x;      // yields xxx
    a = 'h 3x;     // yields 03x
    a = 'h 0x;     // yields 00x
end
```

Example 2-4: Automatic extension of x values

The question mark (`?`) character is a Verilog HDL alternative for the `z` character. It sets four bits to the high impedance value in hexadecimal numbers, three in octal, and one in binary. Use the question mark to enhance readability in cases where the high impedance value is a don't-care condition. See the discussion of `casez` and `casex` in Section 8.4.1 and the discussion on personality files in Section 22.5.

The underline character is legal anywhere in a number except as the first character. Use this feature to break up long numbers for readability purposes. Example 2-5 illustrates this.

```
27_195_000
16'b0011_0101_0001_1111
32 'h 12ab_f001
```

Example 2-5: Use of underline in constant numbers

Underline characters are also legal in numbers in text files read by the \$readmemb and \$readmemh system tasks.

Please note: A sized negative number is not sign-extended when assigned to a register data type.

2.4 Strings

A string is a sequence of characters enclosed by double quotes and must all be contained on a single line. Verilog treats strings used as operands in expressions and assignments as a sequence of eight-bit ASCII values, with one eight-bit ASCII value representing one character.

Examples of strings:

```
"this is a string"
```

```
"print out a message\n"
```

```
"bell!\007"
```

2.4.1 String Variable Declaration

To declare a variable to store a string, declare a register large enough to hold the maximum number of characters the variable will hold. Note that no extra bits are required to hold a termination character; Verilog does not store a string termination character.

For example, to store the string "Hello world!" requires a register 8*12, or 96 bits wide, as shown in Example 2-6.

```
reg [8*12:1] stringvar;  
initial  
begin  
    stringvar = "Hello world!";  
end
```

Example 2-6: Storage needed for strings

2.4.2 String Manipulation

Verilog permits strings to be manipulated using the standard Verilog HDL operators. Keep in mind that the value being manipulated by an operator is a sequence of 8-bit ASCII values, with no special termination character.

The code in Example 2-7 declares a string variable large enough to hold 14 characters and assigns a value to it. The code then manipulates this string value using the concatenation operator.

Note that when a variable is larger than required to hold a value being assigned, Verilog pads the contents on the left with zeros after the assignment. This is consistent with the padding that occurs during assignment of non-string values.

```
module string_test;
  reg [8*14:1] stringvar;
  initial
  begin
    stringvar = "Hello world";
    $display("%s is stored as %h",
      stringvar,stringvar);
    stringvar = {stringvar,"!!!"};
    $display("%s is stored as %h",
      stringvar,stringvar);
  end
endmodule
```

Example 2-7: String manipulation

The following strings display as a result of executing Verilog-XL on Example 2-7:

```
Hello world is stored as 00000048656c6c6f20776f726c64
Hello world!!! is stored as 48656c6c6f20776f726c64212121
```

2.4.3 Special Characters in Strings

Certain characters can only be used in strings when preceded by an introductory character called an *escape character*. Table 2-1 lists these characters in the right-hand column with the escape sequence that represents the character in the left-hand column.

EscapeString	Character Produced by Escape String
\n	new line character
\t	tab character
\\	\ character
\"	" character
\ddd	a character specified in 1-3 octal digits (0 <= d <= 7)
%%	% character

Table 2-1: Specifying special characters in strings

2.5 Identifiers, Keywords, and System Names

An identifier is used to give an object, such as a register or a module, a name so that it can be referenced from other places in a description. An identifier is any sequence of letters, digits, dollar signs (\$), and the underscore (_) symbol.

The first character must NOT be a digit or \$; it can be a letter or an underscore.

Upper- and lowercase letters are considered to be different (unless the upper case option is used when compiling). Identifiers can be up to 1024 characters long.

Examples of identifiers follow:

```
shiftreg_a
busa_index
error_condition
merge_ab
_bus3
n$657
```

2.5.1 Escaped Identifiers

Escaped identifiers start with the backslash character (\) and provide a means of including any of the printable ASCII characters in an identifier (the decimal values 33 through 126, or 21 through 7E in hexadecimal). An escaped identifier ends with white space (blank, tab, newline). Neither the leading back-slash character nor the terminating white space is considered to be part of the identifier.

The primary application of escaped identifiers is for translators from other hardware description languages and CAE systems, where special characters may be allowed in identifiers. Escaped identifiers should not be used under normal circumstances.

Examples of escaped identifiers follow:

```
\busa+index
\ -clock
\***error-condition***
\net1/\net2
\{a,b}
\a*(b+c)
```

Please note: Remember to terminate escaped identifiers with white space, otherwise characters that should follow the identifier are considered as part of it.

2.5.2

Keywords

Keywords are predefined non-escaped identifiers that are used to define the language constructs. A Verilog HDL keyword preceded by an escape character is not interpreted as a keyword.

All keywords are defined in lowercase only and therefore must be typed in lowercase in source files (unless the upper case option is used when compiling).

2.6

Text Substitutions

A text macro substitution facility has been provided so that meaningful names can be used to represent commonly used pieces of text. For example, in the situation where a constant number is repetitively used throughout a description, a text macro would be useful in that only one place in the source description would need to be altered if the value of the constant needed to be changed. Text macros can also be defined and used in the interactive mode where they can be helpful for predefining those interactive commands that you use often.

The syntax for text macro definitions is as follows:

```
<text_macro_definition>  
 ::= 'define <text_macro_name> <MACRO_TEXT>  
<text_macro_name>  
 ::= <IDENTIFIER>
```

Syntax 2-1: Syntax for <text_macro_definition>

<MACRO_TEXT> is any arbitrary text specified on the same line as the <text_macro_name>. If a one-line comment (that is, a comment specified with the characters //) is included in the text, then the comment does not become part of the text substituted. The text for <MACRO_TEXT> can be blank, in which case the text macro is defined to be empty and no text is substituted when the macro is used.

The syntax for using a text macro is as follows:

```
<text_macro_usage>  
 ::= '<text_macro_name>
```

Syntax 2-2: Syntax for <text_macro_usage>

Once a text macro name has been defined (that is, assigned <MACRO_TEXT>), it can be used anywhere in a source description or in an interactive command; that is, there are no scope restrictions. However, to use a text macro the compiler directive symbol ``` (open quote, also known as “accent grave”) must precede the text macro name. Example 2-8 shows two definitions of macro text and a use of each of the defined macros.

```
`define wordsize 8
reg [1:`wordsize] data;

`define typ_nand nand #5 //define a nand w/typical delay
`typ_nand g121 (q21, n10, n11);
```

Example 2-8: Using macro text

The text specified for <MACRO_TEXT> must not be split across the following lexical tokens:

- comments
- numbers
- strings
- identifiers
- keywords
- double or triple character operators

For example, the following is illegal syntax in the Verilog language because it is split across a string:

```
`define first_half "start of string
$display(`first_half end of string");
```

Text macro names can re-use names being used as ordinary identifiers. For example, `signal_name` and ``signal_name` are different. Redefinition of text macros is allowed; the latest definition of a particular text macro read by the compiler prevails when the macro name is encountered in the source text.

3

Figure 3-0
Example 3-0
Syntax 3-0
Table 3-0

Data Types

The set of Verilog HDL data types is designed to represent the data storage and transmission elements found in digital hardware.

3.1 Value Set

The Verilog HDL value set consists of four basic values:

- 0 - represents a logic zero, or false condition
- 1 - represents a logic one, or true condition
- x - represents an unknown logic value
- z - represents a high-impedance state

The values 0 and 1 are logical complements of one another.

When the z value is present at the input of a gate, or when it is encountered in an expression, the effect is usually the same as an x value. Notable exceptions are the MOS primitives, which can pass the z value.

Almost all of the data types in the Verilog language store all four basic values. The exceptions are the `event` type, which has no storage, and the `triereg` net data type, which retains its first state when all of its drivers go to the high impedance value, and z. All bits of vectors can be independently set to one of the four basic values.

The language includes strength information in addition to the basic value information for scalar net variables. This is described in detail in Chapter 6, *Gate and Switch Level Modeling*.

3.2 Registers and Nets

There are two main groups of data types: the register data types and the net data types. These two groups differ in the way that they are assigned and hold values. They also represent different hardware structures.

3.2.1 Nets

The net data types represent physical connections between structural entities, such as gates. A net does not store a value (except for the `triereg` net, discussed in Section 3.7.3). Instead, it must be driven by a driver, such as a gate or a continuous assignment. See Chapter 6, *Gate and Switch Level Modeling*, and Chapter 5, *Assignments*, for definitions of these constructs. If no driver is connected to a net, its value will be high-impedance (`z`)—unless the net is a `triereg`.

3.2.2 Registers

A register is an abstraction of a data storage element. The keyword for the register data type is `reg`. A register stores a value from one assignment to the next. An assignment statement in a procedure acts as a trigger that changes the value in the data storage element. The Verilog language has powerful constructs that allow you to control when and if these assignment statements are executed. These control constructs are used to describe hardware trigger conditions, such as the rising edge of a clock, and decision-making logic, such as a multiplexer. Chapter 8, *Behavioral Modeling*, describes these control constructs.

The default initialization value for a `reg` data type is the unknown value, `x`.

CAUTION

Registers can be assigned negative values, but, when a register is an operand in an expression, its value is treated as an unsigned (positive) value. For example, a minus one in a four-bit register functions as the number 15 if the register is an expression operand. See Section 4.1.2 for more information on numeric conventions in expressions.

3.2.3 Declaration Syntax

The syntax for net and register declarations is as follows:

```

<net_declaration>
    ::= <NETTYPE> <expandrange>? <delay>? <list_of_variables> ;
    | = trireg <charge_strength>? <expandrange>? <delay>? <list_of_variables> ;
    | = <NETTYPE> <drive_strength>?
        <expandrange>? <delay>? <list_of_assignments> ;

<reg_declaration>
    ::= reg <range>? <list_of_register_variables> ;

<list_of_variables>
    ::= <name_of_variable> <,<name_of_variable>>*>

<name_of_variable>
    ::= <IDENTIFIER>

<list_of_register_variables>
    ::= <register_variable> <,<register_variable>>*>

<register_variable>
    ::= <name_of_register>

<name_of_register>
    ::= <IDENTIFIER>

<expandrange>
    ::= <range>
    | = scaled <range>
    | = vectored <range>
    iff [the data type is not a trireg] the following syntax is available:

<range>
    ::= [ <constant_expression> : <constant_expression> ]

<list_of_assignments>
    ::= <assignment> <,<assignment>>*>

<charge_strength>
    ::= ( <CAPACITOR_SIZE> )

<drive_strength>
    ::= ( <STRENGTH0> , <STRENGTH1> )
    | = ( <STRENGTH1> , <STRENGTH0> )

```

Syntax 3-1: Syntax for <net_declaration>

<NETTYPE> is one of the following keywords:

wire tri tri1 supply0
wand triand tri0 supply1
wor trior trireg

<IDENTIFIER> is the name of the net that is being declared. See Chapter 2, *Lexical Conventions*, for a discussion of identifiers.

<delay> specifies the propagation delay of the net (as explained in Chapter 6, *Gate and Switch Level Modeling*), or, when associated with a <list_of_assignments>, it specifies the delay executed before the assignment (as explained in Chapter 5, *Assignments*).

<CAPACITOR_SIZE> is one of the following keywords:

small medium large

<STRENGTH0> is one of the following keywords:

supply0 strong0 pull0 weak0 highz0

<STRENGTH1> is one of the following keywords:

supply1 strong1 pull1 weak1 highz1

Syntax 3-2: Definitions for <net_declaration> syntax

3.2.4 Declaration Examples

The following are examples of register and net declarations:

```
reg a;           // a scalar register
wand w;         // a scalar net of type 'wand'
reg[3:0] v;      // a 4-bit vector register made up of
                // (from most to least significant)
                // v[3], v[2], v[1] and v[0]

tri [15:0] busa; // a tri-state 16-bit bus
reg [1:4] b;     // a 4-bit vector register
triereg (small) storeit; // a charge storage node
                        // of strength small
```

Example 3-1: Register and net declarations

If a set of nets or registers shares the same characteristics, they can be declared in the same declaration statement. The following is an example:

```
wire w1, w2;           // declares 2 wires
reg [4:0] x, y, z;      // declares 3 5-bit registers
```

3.3 Vectors

A net or reg declaration without a <range> specification is one bit wide; that is, it is scalar. Multiple bit net and reg data types are declared by specifying a <range>, and are known as vectors.

3.3.1 Specifying Vectors

The <range> specification gives addresses to the individual bits in a multi-bit net or register. The most significant bit (msb) is the left-hand value in the <range> and the least significant bit (lsb) is the right-hand value in the <range>.

The range is specified as follows:

```
[ msb_expr : lsb_expr ]
```

Both `msb_expr` and `lsb_expr` are non-negative constant expressions. There are no restrictions on the values of the indices. The `msb` and `lsb` expressions can be any value, and `lsb_expr` can be a greater value than `msb_expr`, if desired.

Vector nets and registers obey laws of arithmetic modulo 2 to the power n , where n is the number of bits in the vector. Vector nets and registers are treated as unsigned quantities.

3.3.2 Vector Net Accessibility

A vector can be used as a single entity or as a group of n scalars, where n is the number of bits in the vector. The keyword `vectored` allows you to specify that a vector can be modified *only* as an indivisible entity. The keyword `scalared` explicitly allows access to bit and parts. This is also the default case. The Verilog-XL process of accessing bits within a vector is known as vector *expansion*.

Only when a net is *not* specified as vectored can bit selects and part selects be driven by outputs of gates, primitives, and modules—or be on the left-hand side of continuous assignments. You cannot declare a `triereg` with the `vectored` keyword.

The following are examples of vector net declarations:

```
tril scalared [63:0] bus64; //a bus that will be expanded
tri vectored [31:0] data; //a bus that will not be expanded
```

Example 3-2: Vector net declarations

3.4 Strengths

There are two types of strengths that can be specified in a net declaration. They are as follows:

- **charge strength** used when declaring a net of type `triereg`
- **drive strength** used when placing a continuous assignment on a net in the same statement that declares the net

Gate declarations can also specify a drive strength. See Chapter 6, *Gate and Switch Level Modeling*, for more information on gates and for important information on strengths.

3.4.1 Charge Strength

The `<charge_strength>` specification can be used only with `triereg` nets. A `triereg` net is used to model charge storage; `<charge_strength>` specifies the relative size of the capacitance. The `<CAPACITOR_SIZE>` declaration is one of the following keywords:

- `small`
- `medium`
- `large`

When no size is specified in a `triereg` declaration, its size is `medium`.

The following is a syntax example of a strength declaration:

```
triereg (small) st1 ;
```

A `triereg` net can model a charge storage node whose charge decays over time. The simulation time of a charge decay is specified in the `triereg` net's delay specification (see Section 6.16.2).

3.4.2 Drive Strength

The `<drive_strength>` specification allows a continuous assignment to be placed on a net in the same statement that declares that net. See Chapter 5, *Assignments*, for more details.

Net strength properties are described in detail in Chapter 6, *Gate and Switch Level Modeling*.

3.5 Implicit Declarations

The syntax shown in Section 3.2.3, *Declaration Syntax*, is used to explicitly declare variables. In the absence of an explicit declaration of a variable, statements for gate, user-defined primitive, and module instantiations assume an *implicit* variable declaration. This happens if you do the following: in the terminal list of an instance of a gate, a user-defined primitive, or a module, specify a variable that has not been explicitly declared previously in one of the declaration statements of the instantiating module.

These implicitly declared variables are scalar nets of type wire.

3.6 Net Initialization

The default initialization value for a net is the value `z`. Nets with drivers assume the output value of their drivers, which defaults to `x`. The `triereg` net is an exception to these statements. The `triereg` defaults to the value `x`, with the strength specified in the net declaration (`small`, `medium`, or `large`).

3.7 Net Types

There are several distinct types of nets. Each is described in the sections that follow.

3.7.1 wire and tri Nets

The `wire` and `tri` nets connect elements. The net types `wire` and `tri` are identical in their syntax and functions; two names are provided so that the name of a net can indicate the purpose of the net in that model. A `wire` net is typically used for nets that are driven by a single gate or continuous assignment. The `tri` net type might be used where multiple drivers drive a net.

Logical conflicts from multiple sources on a `wire` or a `tri` net result in unknown values unless the net is controlled by logic strength.

Table 3-1 is a truth table for `wire` and `tri` nets. Note that it assumes equal strengths for both drivers. Please refer to Section 6.10 for a discussion of *logic strength modeling*.

wire/ tri	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

Table 3-1: Truth table for `wire` and `tri` nets

3.7.2 Wired Nets

Wired nets are of type `wor`, `wand`, `trior`, and `triand`, and are used to model wired logic configurations. Wired nets resolve the conflicts that result when multiple drivers drive the same net. The `wor` and `trior` nets create wired `or` configurations, such that when any of the drivers is 1, the net is 1. The `wand` and `triand` nets create wired `and` configurations, such that if any driver is 0, the net is 0.

The net types `wor` and `trior` are identical in their syntax and functionality—as are the `wand` and `triand`. Table 3-2 gives the truth tables for wired nets. Note that it assumes equal strengths for both drivers. Please refer to Section 6.10 for a discussion of logic strength modeling.

wand/ triand	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

wor/ trior	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

Table 3-2: Truth tables for `wand/triand` and `wor/trior` nets

3.7.3 `triereg` Net

The `triereg` net stores a value and is used to model charge storage nodes. A `triereg` can be one of two states:

The Driven State	When at least one driver of a <code>triereg</code> has a value of 1, 0, or x, that value propagates into the <code>triereg</code> and is the <code>triereg</code> 's driven value.
Capacitive State	When all the drivers of a <code>triereg</code> net are at the high impedance value (z), the <code>triereg</code> net retains its last driven value; the high impedance value does not propagate from the driver to the <code>triereg</code> .

The strength of the value on the `triereg` net in the capacitive state is small, medium, or large, depending on the size specified in the declaration of the `triereg`. The strength of a `triereg` in the driven state is strong, pull, or weak depending on the strength of the driver. You cannot declare a `triereg` with the vectored keyword.

Figure 3-1 shows a schematic that includes the following items: a trireg net whose size is medium, its driver, and the simulation results.

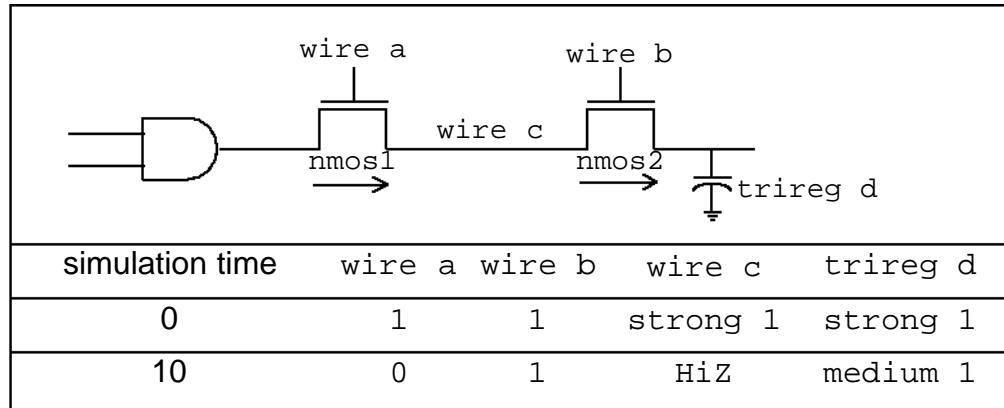


Figure 3-1: Simulation values of a trireg and its driver

Simulation of the design in Figure 3-1 reports the following results:

1. At simulation time 0, wire a and wire b have a value of 1. A value of 1 with a strong strength propagates from the AND gate through the NMOS switches connected to each other by wire c, into trireg d.
2. At simulation time 10, wire a changes value to 0, disconnecting wire c from the AND gate. When wire c is no longer connected to the AND gate, its value changes to HiZ. The wire b's value remains 1 so wire c remains connected to trireg d through the NMOS2 switch. The HiZ value does not propagate from wire c into trireg d. Instead, trireg d enters the capacitive state, storing its last driven value of 1. It stores the 1 with a medium strength.

Capacitive networks

A capacitive network is a connection between two or more triregs. In a capacitive network whose trireg's are in the capacitive state, logic and strength values can propagate between triregs. Figure 3-2 shows a capacitive network in which the logic value of some triregs change the logic value of other triregs of equal or smaller size.

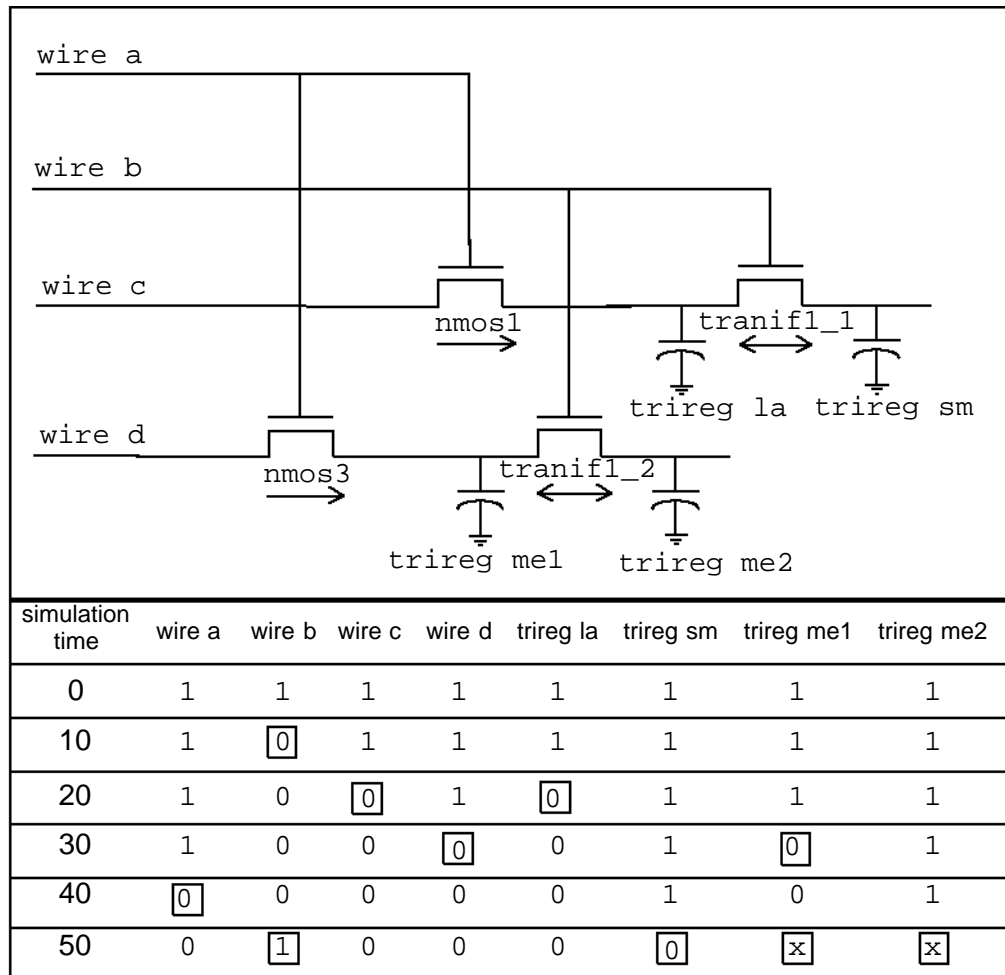


Figure 3-2: Simulation results of a capacitive network

In Figure 3-2, trireg la's size is large, triregs me1 and me2 are size medium, and trireg sm's size is small. Simulation reports the following sequence of events:

1. At simulation time 0, wire a and wire b have a value of 1. The wire c drives a value of 1 into triregs la and sm, wire d drives a value of 1 into triregs me1 and me2.
2. At simulation time 10, wire b's value changes to 0, disconnecting trireg sm and me2 from their drivers. These triregs enter the capacitive state and store the value 1, their last driven value.
3. At simulation time 20, wire c drives a value of 0 into trireg la.
4. At simulation time 30, wire d drives a value of 0 into trireg me1.
5. At simulation time 40, wire a's value changes to 0, disconnecting trireg la and me1 from their drivers. These triregs enter the capacitive state and store the value 0.

6. At simulation time 50, the wire b's value changes to 1. This change of value in wire b connects trireg sm to trireg la; these triregs have different sizes and stored different values. This connection causes the smaller trireg to store the larger trireg's value and trireg sm now stores a value of 0. This change of value in wire b also connects trireg me1 to trireg me2; these triregs have the same size and stored different values. The connection causes both trireg me1 and me2 to change value to x.

In a capacitive network, charge strengths propagate from a larger trireg to a smaller trireg. Figure 3-3 shows a capacitive network and its simulation results.

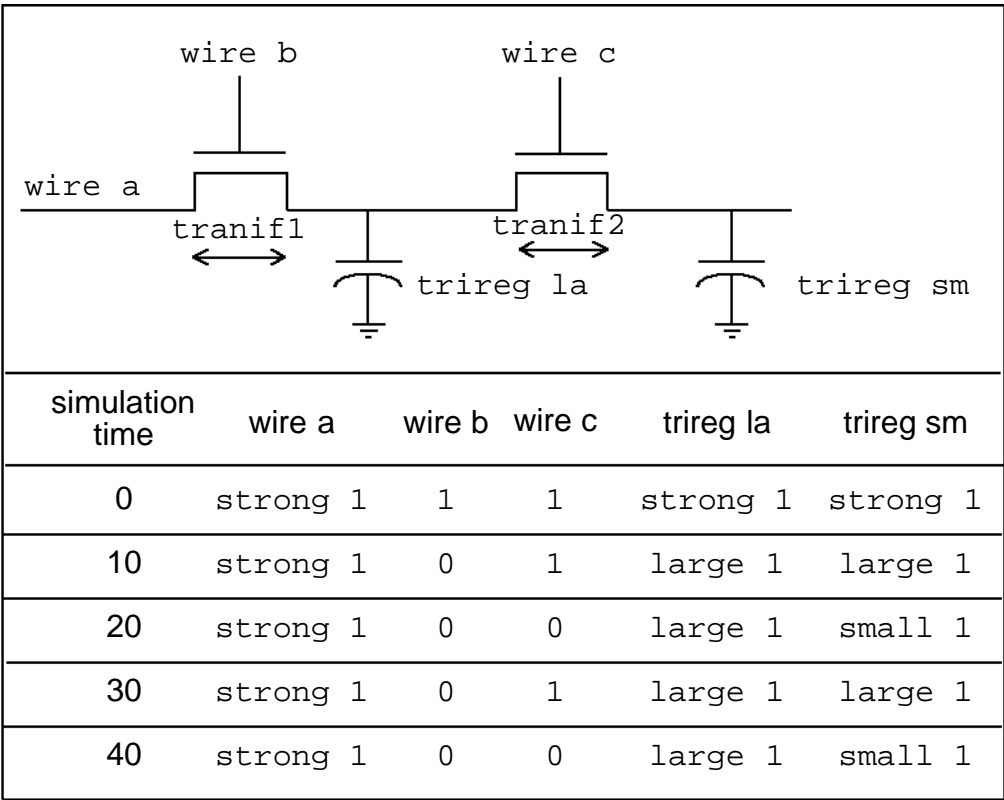


Figure 3-3: Simulation results of charge sharing

In Figure 3-3, `triereg la`'s size is large and `triereg sm`'s size is small. Simulation reports the following results:

1. At simulation time 0, the value of wire `a`, `b`, and `c` is 1 and wire `a` drives a strong 1 into `triereg la` and `sm`.
2. At simulation time 10, wire `b`'s value changes to 0, disconnecting `triereg la` and `sm` from wire `a`. The `triereg`s `la` and `sm` enter the capacitive state. Both `triereg`s share the large charge of `triereg la` because they remain connected through `tranif2`.
3. At simulation time 20, wire `c`'s value changes to 0, disconnecting `triereg sm` from `triereg la`. The `triereg sm` no longer shares `triereg la`'s large charge and now stores a small charge.
4. At simulation time 30, wire `c`'s value changes to 1, connecting the two `triereg`s. These `triereg`s now share the same charge.
5. At simulation time 40, wire `c`'s value changes again to 0, disconnecting `triereg sm` from `triereg la`. Once again, `triereg sm` no longer shares `triereg la`'s large charge and now stores a small charge.

Ideal capacitive state and charge decay

A `triereg` net can retain its value indefinitely or its charge can decay over time. The simulation time of charge decay is specified in the `triereg` net's delay specification.

3.7.4

`tri0` and `tri1` Nets

The `tri0` and `tri1` nets model nets with resistive pulldown and resistive pullup devices on them. When no driver drives a `tri0` net, its value is 0. When no driver drives a `tri1` net, its value is 1. The strength of this value is `pull`. See Chapter 6, *Gate and Switch Level Modeling*, for a description of strength modeling.

3.7.5

`supply` Nets

The `supply0` and `supply1` nets model the power supplies in a circuit. The `supply0` nets are used to model `Vss` (ground) and `supply1` nets are used to model `Vdd` or `Vcc` (power). These nets should never be connected to the output of a gate or continuous assignment, because the strength they possess will override the driver. They have `supply0` or `supply1` strengths.

3.8 Memories

The Verilog HDL models memories as an array of register variables. These arrays can be used to model read-only memories (ROMs), random access memories (RAMs), and register files. Each register in the array is known as an *element* or *word* and is addressed by a single array index. There are no multiple dimension arrays in the Verilog Language.

Memories are declared in register declaration statements by specifying the element address range after the declared identifier. Syntax 3-3 gives the syntax for a register declaration statement. Note that this syntax extends the <register_variable> definition given in Section 3.2.3, *Declaration Syntax*.

```
<register_variable>
    ::= <name_of_register>
       | |= <name_of_memory> [ <constant_expression> : <constant_expression> ]

<constant_expression>
    ::= <expression>

<name_of_memory>
    ::= <IDENTIFIER>
```

Syntax 3-3: Syntax for <register_variable>

The following example illustrates a memory declaration:

```
reg[7:0] mema[0:255];
```

This example declares a memory called `mema` consisting of 256 eight-bit registers. The indices are 0 through 255. The expressions that specify the indices of the array must be constant expressions.

Note that within the same declaration statement both registers and memories can be declared. This makes it convenient to declare both a memory and some registers that will hold data to be read from and written to the memory in the same declaration statement, as in Example 3-3.

```

parameter                // parameters are run-time
                        // constants - see Section 3.11

wordsize = 16,
memsize = 256;

// Declare 256 words of 16-bit memory plus two registers
reg [wordsize-1:0]       // equivalent to [15:0]
    mem [memsize-1:0], // equivalent to [255:0]
    writereg,
    readreg;

```

Example 3-3: Declaring memory

Note that a memory of n 1-bit registers is different from an n -bit vector register, as shown in the following example:

```

reg [1:n] rega;
reg mema [1:n];

```

an n -bit register is not the same as a memory of n 1-bit registers

An n -bit register can be assigned a value in a single assignment, but a complete memory cannot; thus the following assignment to `rega` is legal and the succeeding assignment that attempts to clear all of the memory `mema` is illegal, as shown in the following example:

```

rega = 0;
mema = 0;

```

legal syntax

illegal syntax

To assign a value to a memory element, an index must be specified, as shown in the following example:

```

mema[1] = 0;

```

assigns 0 to the first element of `mema`

The index can be an expression. This option allows you to reference different memory elements, depending on the value of other registers and nets in the circuit. For example, a program counter register could be used to index into a RAM.

3.9 Integers and Times

In addition to modeling hardware, there are other uses for variables in an HDL model. Although you can use the `reg` variables for general purposes such as counting the number of times a particular net changes value, the `integer` and `time` register data types are provided for convenience and to make the description more self-documenting.

The syntax for declaring `integer` and `time` variables is as follows:

```
<time_declaration>  
    ::= time <list_of_register_variables> ;  
  
<integer_declaration>  
    ::= integer <list_of_register_variables> ;
```

Syntax 3-4: Syntax for time and integer declarations

The `<list_of_register_variables>` item is defined in Section 3.2.3, *Declaration Syntax*.

A `time` variable is used for storing and manipulating simulation time quantities in situations where timing checks are required and for diagnostics and debugging purposes. This data type is typically used in conjunction with the `$time` system function. The size of a `time` variable is 64 bits.

An `integer` is a general purpose variable used for manipulating quantities that are not regarded as hardware registers. The size of an `integer` variable is 32 bits.

Arrays of `integer` and `time` variables are allowed. They are declared in the same manner as arrays of `reg` variables, as in the following example:

```
integer a[1:64]; // an array of 64 integers  
time change_history[1:1000]; // an array of 1000 times
```

The `integer` and `time` variables are assigned values in the same manner as `reg` variables. Procedural assignments are used to trigger their value changes.

Time variables behave the same as 64 bit reg variables. They are unsigned quantities, and unsigned arithmetic is performed on them. In contrast, integer variables are signed quantities. Arithmetic operations performed on integer variables produce 2's complement results.

3.10 Real Numbers

The Verilog HDL supports real number constants and variables in addition to integers and time variables. The syntax for real numbers is the same as the syntax for register types, and is described in Section 3.10.1. Except for the following restrictions, real number variables can be used in the same places that integers and time variables are used.

- Not all Verilog HDL operators can be used with real number values. See the tables in Section 4.1 for lists of valid and invalid operators for real numbers.
- Ranges are not allowed on real number variable declarations.
- Real number variables default to an initial value of zero.

3.10.1 Declaration Syntax for Real Numbers

The syntax for declaring real number variables is as follows:

```
<real_declaration>  
    ::=real<list_of_variables>;
```

Syntax 3-5: Syntax for real number variable declarations

The <list_of_variables> item is defined in Section 3.2.3, *Declaration Syntax*.

3.10.2 Specifying Real Numbers

Real numbers can be specified in either decimal notation (for example, 14.72) or in scientific notation (for example, 39e8, which indicates 39 multiplied by 10 to the 8th power). Real numbers expressed with a decimal point must have at least one digit on each side of the decimal point.

The following are some examples of valid real numbers in the Verilog language:

```
1.2
0.1
2394.26331
1.2E12 (the exponent symbol can be e or E)
1.30e-2
0.1e-0
23E10
29E-2
236.123_763_e-12 (underscores are ignored)
```

The following are invalid real numbers in the Verilog HDL because they do not have a digit to the left of the decimal point:

```
.12
.3E3
.2e-7
```

3.10.3 Operators and Real Numbers

The result of using logical or relational operators on real numbers is a single-bit scalar value. Not all Verilog operators can be used with real number expressions. Table 4-2 in Section 4.1 lists the valid operators for use with real numbers. Real number constants and real number variables are also prohibited in the following contexts:

- edge descriptors (posedge, negedge) applied to real number variables
- bit-select or part-select references of variables declared as `real`
- real number index expressions of bit-select or part-select references of vectors
- real number memories (arrays of real numbers)

3.10.4 Conversion

The Verilog language converts real numbers to integers by rounding a real number to the nearest integer, rather than by truncating it. For example, the real numbers 35.7 and 35.5 both become 36 when converted to an integer, and 35.2 becomes 35. Implicit conversion takes place when you assign a real to an integer.

3.11 Parameters

Verilog parameters do not belong to either the register or the net group. Parameters are not variables, they are constants. The syntax for parameter declarations is as follows:

```
<parameter_declaration>  
 ::= parameter <list_of_assignments> ;
```

Syntax 3-6: Syntax for <parameter_declaration>

The <list_of_assignments> is a comma-separated list of assignments, where the right-hand side of the assignment must be a constant expression, that is, an expression containing only constant numbers and previously defined parameters. Example 3-4 shows examples of parameter declarations:

```
parameter msb = 7; // defines msb as a constant value 7  
  
parameter e = 25, f = 9; // defines two constant numbers  
  
parameter average_delay = (r + f) / 2;  
  
parameter byte_size = 8, byte_mask = byte_size - 1;  
  
parameter  r = 5.7;      //declares r as a 'real'  
                        // parameter
```

Example 3-4: Parameter declarations

Even though they represent constants, Verilog parameters can be modified at compilation time to have values that are different from those specified in the declaration assignment. This allows you to customize module instances. You can modify the parameter with the `defparam` statement, or you can modify the parameter in the module instance statement. Typical uses of parameters are to specify delays and width of variables. See Chapter 12, *Hierarchical Structures*, for complete details on parameter value assignment.

4

Figure 4-0
Example 4-0
Syntax 4-0
Table 4-0

Expressions

This chapter describes the operators and operands available in the Verilog HDL, and how to use them to form expressions.

An expression is a construct that combines operands with operators to produce a result that is a function of the values of the operands and the semantic meaning of the operator. Alternatively, an expression is any legal operand—for example, a net bit-select. Wherever a value is needed in a Verilog HDL statement, an expression can be given. However, several statement constructs limit an expression to a constant expression. A constant expression consists of constant numbers and predefined parameter names only, but can use any of the operators defined in Table 4-1.

For their use in expressions, integer and time data types share the same traits as the data type reg. Descriptions pertaining to register usage apply to integers and times as well.

An operand can be one of the following:

- number (including real)
- net
- register, integer, time
- net bit-select
- register bit-select
- net part-select
- register part-select
- memory element
- a call to a user-defined function or system defined function that returns any of the above

4.1 Operators

The symbols for the Verilog HDL operators are similar to those in the C language. Table 4-1 lists these operators.

Verilog Language Operators	
{ }	concatenation
+ - * /	arithmetic
%	modulus
> >= < <=	relational
!	logical negation
&&	logical and
	logical or
==	logical equality
!=	logical inequality
===	case equality
!==	case inequality
~	bit-wise negation
&	bit-wise and
	bit-wise inclusive or
^	bit-wise exclusive or
^~ or ~^	bit-wise equivalence
&	reduction and
~&	reduction nand
	reduction or
~	reduction nor
^	reduction xor
~^ or ^~	reduction xnor
<<	left shift
>>	right shift
? :	conditional

Table 4-1: Operators for Verilog language

Not all of the operators listed above are valid with real expressions. Table 4-2 is a list of the operators that are legal when applied to real numbers.

Operators for Real Expressions	
unary + unary -	unary operators
+ - * /	arithmetic
> >= < <=	relational
! &&	logical
== !=	logical equality
?:	conditional
or	logical

Table 4-2: Legal operators for use in real expressions

The result of using logical or relational operators on real numbers is a single-bit scalar value.

Table 4-3 lists operators that are *not allowed* to operate on real numbers.

Disallowed Operators for Real Expressions	
{ }	concatenate
%	modulus
=== !==	case equality
~ &	bit-wise
^ ^~ ~^	reduction
& ~& ~	
<< >>	shift

Table 4-3: Operators not allowed for real expressions

See Section 3.10.3 for more information on use of real numbers.

4.1.1 Binary Operator Precedence

The precedence order of binary operators (and the ternary operator `?:`) is the same as the precedence order for the matching operators in the C language. Verilog has two equality operators not present in C; they are discussed in Section 4.1.6. Table 4-4 summarizes the precedence rules for Verilog's binary and ternary operators.


Operator Precedence Rules	
<code>! ~</code> <code>* / %</code> <code>+ -</code> <code><< >></code> <code>< <= > >=</code> <code>== != === !==</code> <code>&</code> <code>^ ^~</code> <code> </code> <code>&&</code> <code> </code> <code>?:</code> (ternary operator)	<p>highest precedence</p>  <p>lowest precedence</p>

Table 4-4: Precedence rules for operators

Operators on the same line in Table 4-4 have the same precedence. Rows are in order of decreasing precedence, so, for example, `*`, `/`, and `%` all have the same precedence, which is higher than that of the binary `+` and `-` operators.

All operators associate left to right. Associativity refers to the order in which a language evaluates operators having the same precedence. Thus, in the following example B, is added to A and then C is subtracted from the result of A+B.

A + B - C

When operators differ in precedence, the operators with higher precedence apply first. In the following example, B is divided by C (division has higher precedence than addition) and then the result is added to A.

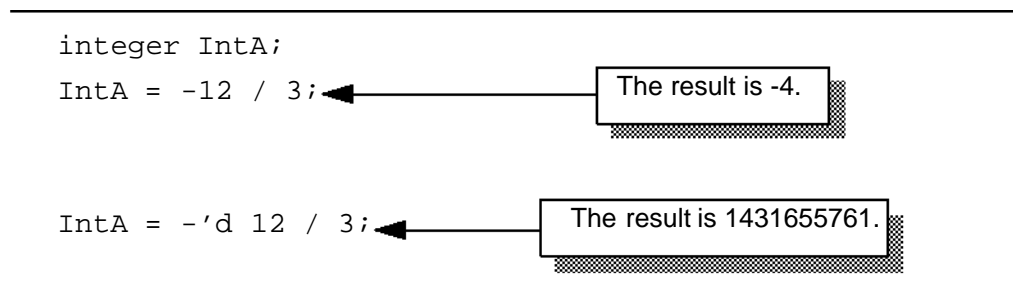
`A + B / C`

Parentheses can be used to change the operator precedence.

`(A + B) / C` // not the same as `A + B / C`

4.1.2 Numeric Conventions in Expressions

Operands can be expressed as based and sized numbers—with the following restriction: The Verilog language interprets a number of the form `sss 'f nnn`, *when used directly in an expression*, as the *unsigned* number represented by the two's complement of nnn. Example 4-1 shows two ways to write the expression “minus 12 divided by 3.” Note that `-12` and `-d12` both evaluate to the same bit pattern, but in an expression `-d12` loses its identity as a signed, negative number.



Example 4-1: Number format in expressions

4.1.3 Arithmetic Operators

The binary arithmetic operators are the following:

`+` `-` `*` `/` `%` (the modulus operator)

Integer division truncates any fractional part. The modulus operator—for example, $y \% z$, gives the remainder when the first operand is divided by the second, and thus is zero when z divides y exactly. The result of a modulus operation takes the sign of the first operand. Table 4-5 gives examples of modulus operations.

Modulus Expression	Result	Comments
$10 \% 3$	1	$10/3$ yields a remainder of 1
$11 \% 3$	2	$11/3$ yields a remainder of 2
$12 \% 3$	0	$12/3$ yields no remainder
$-10 \% 3$	-1	the result takes the sign of the first operand
$11 \% -3$	2	the result takes the sign of the first operand
$-4'd12 \% 3$	1	$-4'd12$ is seen as a large, positive number that leaves a remainder of 1 when divided by 3

Table 4-5: Examples of modulus operations

The unary arithmetic operators take precedence over the binary operators. The unary operators are the following:

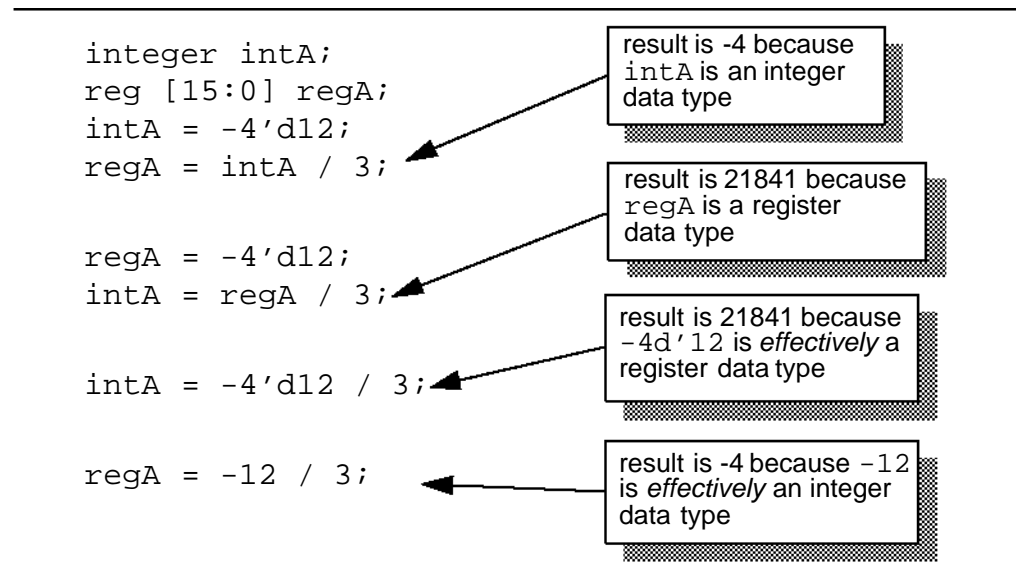
+ -

For the arithmetic operators, if any operand bit value is the unknown value x , then the entire result value is x .

4.1.4 Arithmetic Expressions with Registers and Integers

An arithmetic operation on a register data type behaves differently than an arithmetic operation on an integer data type. The Verilog language sees a register data type as an unsigned value and an integer type as a signed value. As a result, when you assign a value of the form `-<size><base_format><number>` to a *register* and then use that register as an expression operand, you are actually using a positive number that is the two's complement of nnn . In contrast, when you assign a value of

the form `-<size><base_format><number>` to an *integer* and then use that integer as an expression operand, the expression evaluates using signed arithmetic. Example 4-2 shows various ways to divide minus twelve by three using integer and register data types in expressions.



Example 4-2: Modulus operation with registers and integers

4.1.5 Relational Operators

Table 4-2 lists and defines the relational operators.

Relational Operators	
a < b	a less than b
a > b	a greater than b
a <= b	a less than or equal to b
a >= b	a greater than or equal to b

Table 4-6: The relational operators defined

The relational operators in Table 4-2 all yield the scalar value 0 if the specified relation is false, or the value 1 if the specified relation is true. If, due to unknown bits in the operands, the relation is ambiguous, then the result is the unknown value (x).

Please note: If Verilog-XL tests a value that is x or z, then the result of that test is *False*.

All the relational operators have the same precedence. Relational operators have lower precedence than arithmetic operators. The following examples illustrate the implications of this precedence rule:

```
a < size - 1      // this construct is the same as
a < (size - 1)    // this construct, but . . .
size - (1 < a)     // this construct is not the same
size - 1 < a      // as this construct
```

Note that when `size - (1 < a)` evaluates, the relational expression evaluates first and then either zero or one is subtracted from `size`. When `size - 1 < a` evaluates, the `size` operand is reduced by one and then compared with `a`.

4.1.6 Equality Operators

The equality operators rank just lower in precedence than the relational operators. Table 4-2 lists and defines the equality operators.

Equality Operators	
<code>a == b</code>	a equal to b, including x and z
<code>a != b</code>	a not equal to b, including x and z
<code>a === b</code>	a equal to b, result may be unknown
<code>a !== b</code>	a not equal to b, result may be unknown

Table 4-7: The equality operators defined

All four equality operators have the same precedence. These four operators compare operands bit for bit, with zero filling if the two operands are of unequal bit-length. As with the relational operators, the result is 0 if false, 1 if true.

For the `==` and `!=` operators, if either operand contains an `x` or a `z`, then the result is the unknown value (`x`).

For the `===` and `!==` operators, the comparison is done just as it is in the procedural case statement. Bits that are `x` or `z` are included in the comparison and must match for the result to be true. The result of these operators is always a known value, either 1 or 0.

4.1.7 Logical Operators

The operators logical AND (`&&`) and logical OR (`||`) are logical connectives. Expressions connected by `&&` or `||` are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is known. The result of the evaluation of a logical comparison is one (defined as *true*), zero (defined as *false*), or, if the result is ambiguous, then the result is the unknown value (`x`). For example, if register `alpha` holds the integer value 237 and `beta` holds the value zero, then the following examples perform as described:

```
regA = alpha && beta; // regA is set to 0
regB = alpha || beta; // regB is set to 1
```

The precedence of `&&` is greater than that of `||`, and both are lower than relational and equality operators. The following expression ANDs three sub-expressions without needing any parentheses:

```
a < size-1 && b != c && index != lastone
```

However, it is recommended for readability purposes that parentheses be used to show very clearly the precedence intended, as in the following rewrite of the above example:

```
(a < size-1) && (b != c) && (index != lastone)
```

A third logical operator is the unary logical negation operator `!`. The negation operator converts a non-zero or true operand into 0 and a zero or false operand into 1. An ambiguous truth value remains as `x`. A common use of `!` is in constructions like the following:

```
if (!inword)
```

In some cases, the preceding construct makes more sense to someone reading the code than the equivalent construct shown below:

```
if (inword == 0)
```

Constructions like `if (!inword)` read quite nicely (“if not inword”), but more complicated ones can be hard to understand. The first form is slightly more efficient in simulation speed than the second.

4.1.8 Bit-Wise Operators

The bit operators perform bit-wise manipulations on the operands—that is, the operator compares a bit in one operand to its equivalent bit in the other operand to calculate one bit for the result. The logic tables in Table 4-8 show the results for each possible calculation.

bit-wise unary negation

~	
0	1
1	0
x	x

bit-wise binary AND operator

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

bit-wise binary inclusive
OR operator

	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

bit-wise binary exclusive
OR operator

^	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

bit-wise binary exclusive
NOR operator

^~	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

Table 4-8: Bit-wise operators logic tables

Care should be taken to distinguish the bit-wise operators `&` and `|` from the logical operators `&&` and `||`. For example, if `x` is 1 and `y` is 2, then `x & y` is 0, while `x && y` is 1. When the operands are of unequal bit length, the shorter operand is zero-filled in the most significant bit positions.

4.1.9 Reduction Operators

The unary reduction operators perform a bit-wise operation on a single operand to produce a single bit result. The first step of the operation applies the operator between the first bit of the operand and the second—using the logic tables in Table 4-9. The second and subsequent steps apply the operator between the one-bit result of the prior step and the next bit of the operand—still using the same logic table.

reduction unary
AND operator

<code>&</code>	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

reduction unary inclusive
OR operator

<code> </code>	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

reduction unary exclusive
OR operator

<code>^</code>	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

Table 4-9: Reduction operators logic tables

Note that the reduction unary NAND and reduction unary NOR operators operate the same as the reduction unary AND and OR operators, respectively, but with their outputs negated. The effective results produced by the unary reduction operators are listed in Table 4-10 and Table 4-11.

Results of Unary &, , ~&, and ~ Reduction Operations				
Operand	&		~&	~
no bits set	0	0	1	1
all bits set	1	1	0	0
some bits set, but not all	0	1	1	0

Table 4-10: AND, OR, NAND, and NOR unary reduction operations

Results of Unary ^ and ~^ Reduction Operators		
Operand	^	~^
odd number of bits set	1	0
even number of bits set (or none)	0	1

Table 4-11: Exclusive OR and exclusive NOR unary reduction operations

4.1.10 Syntax Restrictions

The Verilog language imposes two syntax restrictions intended to protect description files from a typographical error that is particularly hard to find. The error consists of transposing a space and a symbol. Note that the constructs on line 1 below do *not* represent the same syntax as the similar constructs on line 2.

1. a & &b	a b
2. a && b	a b

In order to protect users from this type of error, Verilog requires the use of parentheses to separate a reduction or or and operator from a bit-wise or or and operator. Table 4-12 shows the syntax that requires parentheses:

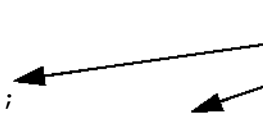
Invalid Syntax	Equivalent Syntax
a & &b a b	a & (&b) a (b)

Table 4-12: Syntax equivalents for syntax restriction

4.1.11 Shift Operators

The shift operators, << and >>, perform left and right shifts of their left operand by the number of bit positions given by the right operand. Both shift operators fill the vacated bit positions with zeroes. Example 4-3 illustrates this concept.

```
module shift;
  reg [3:0] start, result;
  initial
  begin
    start = 1;
    result = (start << 2);
  end
endmodule
```

- 
- 1.) Start is set to 0001.
 - 2.) Result is set to 0100.

Example 4-3: Use of shift operator

In this example, the register `result` is assigned the binary value 0100, which is 0001 shifted to the left two positions and zero filled.

4.1.12 Conditional Operator

The conditional operator has three operands separated by two operators in the following format:

```
cond_expr ? true_expr : false_expr
```

If `cond_expr` evaluates to false, then `false_expr` is evaluated and used as the result. If the conditional expression is true, then `true_expr` is evaluated and used as the result. If `cond_expr` is ambiguous, then both `true_expr` and `false_expr` are evaluated and their results are compared, bit by bit, using Table 4-13 to calculate the final result. If the lengths of the operands are different, the shorter operand is lengthened to match the longer and zero filled from the left (the high-order end).

?:	0	1	x	z
0	0	x	x	x
1	x	1	x	x
x	x	x	x	x
z	x	x	x	x

Table 4-13: Conditional operator ambiguous condition results

The following example of a tri-state output bus illustrates a common use of the conditional operator.

```
wire [15:0] busa = drive_busa ? data : 16'bz;
```

The bus called data is driven onto busa when drive_busa is 1. If drive_busa is unknown, then an unknown value is driven onto busa. Otherwise, busa is not driven.

4.1.13 Concatenations

A concatenation is the joining together of bits resulting from two or more expressions. The concatenation is expressed using the brace characters { and }, with commas separating the expressions within. The next example concatenates four expressions:

```
{a, b[3:0], w, 3'b101}
```

The previous example is equivalent to the following example:

```
{a, b[3], b[2], b[1], b[0], w, 1'b1, 1'b0, 1'b1}
```

Unsize constant numbers are not allowed in concatenations. This is because the size of each operand in the concatenation is needed to calculate the complete size of the concatenation.

Concatenations can be expressed using a repetition multiplier as shown in the next example.

```
{4{w}} // This is equivalent to {w, w, w, w}
```

The next example illustrates nested concatenations.

```
{b, {3{a, b}}}
```

 // This is equivalent to
// {b, a, b, a, b, a, b}

The repetition multiplier must be a constant expression.

4.2 Operands

As stated before, there are several types of operands that can be specified in expressions. The simplest type is a reference to a net or register in its complete form—that is, just the name of the net or register is given. In this case, all of the bits making up the net or register value are used as the operand.

If just a single bit of a vector net or register is required, then a bit-select operand is used. A part-select operand is used to reference a group of adjacent bits in a vector net or register.

A memory element can be referenced as an operand.

A concatenation of other operands (including nested concatenations) can be specified as an operand.

A function call is an operand.

4.2.1 Net and Register Bit Addressing

Bit-selects extract a particular bit from a vector net or register. The bit can be addressed using an expression. The next example specifies the single bit of `acc` that is addressed by the operand `index`.

```
acc[index]
```

The actual bit that is accessed by an address is, in part, determined by the declaration of `acc`. For instance, each of the declarations of `acc` shown in the next example causes a particular value of `index` to access a *different* bit:

```
reg [15:0] acc;  
reg [1:16] acc;
```

If the bit select is out of the address bounds or is `x`, then the value returned by the reference is `x`.

Several contiguous bits in a vector register or net can be addressed, and are known as **part-selects**. A part-select of a vector register or net is given with the following syntax:

```
vect[ms_expr:ls_expr]
```

Both expressions must be constant expressions. The first expression must address a more significant bit than the second expression. Compiler errors result if either of these rules is broken.

The next example and the bullet items that follow it illustrate the principles of bit addressing. The code declares an 8-bit register called `vect` and initializes it to a value of 4. The bullet items describe how the separate bits of that vector can be addressed.

```
reg [7:0] vect;  
vect = 4;
```

- if the value of `addr` is 2, then `vect[addr]` returns 1
- if the value of `addr` is out of bounds, then `vect[addr]` returns `x`
- if `addr` is 0, 1, or 3 through 7, `vect[addr]` returns 0
- `vect[3:0]` returns the bits 0100
- `vect[5:1]` returns the bits 00010
- `vect[<expression that returns x>]` returns `x`
- `vect[<expression that returns z>]` returns `x`
- if any bit of `addr` is `x/z`, then the value of `addr` is `x`

4.2.2 Memory Addressing

Section 3.8 discussed the declaration of memories. This section discusses memory addressing. The next example declares a memory of 1024 8-bit words:

```
reg [7:0] mem_name[0:1023];
```

The syntax for a memory address consists of the name of the memory and an expression for the address—specified with the following format:

```
mem_name[addr_expr]
```

The `addr_expr` can be any expression; therefore, memory indirections can be specified in a single expression. The next example illustrates memory indirection:

```
mem_name[mem_name[3]]
```

In the above example, `mem_name[3]` addresses word three of the memory called `mem_name`. The value at word three is the index into `mem_name` that is used by the memory address `mem_name[mem_name[3]]`. As with bit-selects, the address bounds given in the declaration of the memory determine the effect of the address expression. If the index is out of the address bounds or is `x`, then the value of the reference is `x`.

There is no mechanism to express bit-selects or part-selects of memory elements directly. If this is required, then the memory element has to be first transferred to an appropriately sized temporary register.

4.2.3 Strings

String operands are treated as constant numbers consisting of a sequence of 8-bit ASCII codes, one per character, with no special termination character.

Any Verilog HDL operator can manipulate string operands. The operator behaves as though the entire string were a single numeric value.

Example 4-4 declares a string variable large enough to hold 14 characters and assigns a value to it. The example then manipulates the string using the concatenation operator.

Note that when a variable is larger than required to hold the value being assigned, the contents after the assignment are padded on the left with zeros. This is consistent with the padding that occurs during assignment of non-string values.

```
module string_test;
    reg [8*14:1] stringvar;
    initial
        begin
            stringvar = "Hello world";
            $display("%s is stored as %h",
                    stringvar,stringvar);
            stringvar = {stringvar,"!!!"};
            $display("%s is stored as %h",
                    stringvar,stringvar);
        end
    endmodule
```

Example 4-4: Concatenation of strings

The result of running Verilog on the above description is:

```
Hello world is stored as 00000048656c6c6f20776f726c64
Hello world!!! is stored as 48656c6c6f20776f726c64212121
```

4.2.4 String Operations

The common string operations *copy*, *concatenate*, and *compare* are supported by Verilog operators. Copy is provided by simple assignment. Concatenation is provided by the concatenation operator. Comparison is provided by the equality operators. Example 4-4 and Example 4-5 illustrate assignment, concatenation, and comparison of strings.

When manipulating string values in vector variables, at least $8 \cdot n$ bits are required in the vector, where n is the number of characters in the string.

4.2.5 String Value Padding and Potential Problems

When strings are assigned to variables, the values stored are padded on the left with zeros. Padding can affect the results of comparison and concatenation operations. The comparison and concatenation operators do not distinguish between zeros resulting from padding and the original string characters.

Example 4-5 illustrates the potential problem.

```

reg [8*10:1] s1, s2;
initial
begin
    s1 = "Hello";
    s2 = " world!";
    if ({s1,s2} == "Hello world!")
        $display("strings are equal");
end

```

Example 4-5: Comparing string variables

The comparison in the example above fails because during the assignment the string variables get padded as illustrated in the next example:

```

s1 = 000000000048656c6c6f
s2 = 00000020776f726c6421

```

The concatenation of s1 and s2 includes the zero padding, resulting in the following value:

```
000000000048656c6c6f00000020776f726c6421
```

Since the string "Hello world" contains no zero padding, the comparison fails, as shown below:

The diagram illustrates the comparison of two padded strings, s1 and s2, against the target string "Hello world!".

- s1** is shown as `000000000048656c6c6f`. A bracket below it indicates the original string `"Hello"` is padded with 10 zeros.
- s2** is shown as `00000020776f726c6421`. A bracket below it indicates the original string `" world!"` is padded with 6 zeros.
- The comparison is shown as `{s1,s2} == "Hello world!"`. The result is `0`, indicating the comparison fails.

The above comparison yields a result of zero, which is equivalent to false.

4.2.6

Null String Handling

The null string (" ") is equivalent to the value zero (0).

4.3

Minimum, Typical, Maximum Delay Expressions

Verilog HDL delay expressions can be specified as three expressions separated by colons. This triple is intended to represent minimum, typical, and maximum values—in that order. The appropriate expression is selected by the compiler when Verilog-XL is run. The user supplies a command-line option to select which of the three expressions will be used on a global basis. In the absence of a command-line option, Verilog-XL selects the second expression (the “typical” delay value). The syntax is as follows:

```
<mintypmax_expression>  
 ::= <expression>  
 || = <expression1> : <expression2> : <expression3>
```

Syntax 4-1: Syntax for <mintypmax_expression>

The three expressions follow these conventions:

- expression1 is less than or equal to expression2
- expression2 is less than or equal to expression3

Verilog models typically specify three values for delay expressions. The three values allow a design to be tested with minimum, typical, or maximum delay values. In the following example, one of the three specified delays will be executed before the simulation executes the assignment; if the user does not select one, the simulator will take the default.

```
always @A  
X = #(3:4:5) A;
```


The command-line option `+mindelays` selects the minimum expression in all expressions where `min:typ:max` values have been specified. Likewise, `+typdelays` selects all the typical expressions and `+maxdelays` selects all the maximum expressions. Verilog-XL defaults to the second value when a two or three-part delay expression is specified.

Values expressed in `min:typ:max` format can be used in expressions. The next example shows an expression that defines a single triplet of delay values. The minimum value is the sum of `a+d`; the typical value is `b+e`; the maximum value is `c+f`, as follows:

```
a:b:c + d:e:f
```

The next example shows some typical expressions that are used to specify `min:typ:max` format values:

```
val - 32'd 50: 32'd 75: 32'd 100
```

The `min:typ:max` format can be used wherever expressions can appear, both in source text files and in interactive commands.

4.4 Expression Bit Lengths

Controlling the number of bits that are used in expression evaluations is important if consistent results are to be achieved. Some situations have a simple solution, for example, if a bit-wise AND operation is specified on two 16-bit registers, then the result is a 16-bit value. However, in some situations it is not obvious how many bits are used to evaluate an expression, what size the result should be, or whether signed or unsigned arithmetic should be used.

For example, when is it necessary to perform the addition of two 16-bit registers in 17 bits to handle a possible carry overflow? The answer depends on the context in which the addition takes place. If the 16-bit addition is modeling a real 16-bit adder that loses or does not care about the carry overflow, then the model must perform the addition in 16 bits. If the addition of two 16-bit unsigned numbers can result in a significant 17th bit, then assign the answer to a 17-bit register.

4.4.1

An Example of an Expression Bit Length Problem

During the evaluation of an expression, interim results take the size of the largest operand (in the case of an assignment, this also includes the left-hand side). You must therefore take care to prevent loss of a significant bit during expression evaluation. This section describes an example of the problems that can occur.

The expression $(a + b \gg 1)$ yields a 16-bit result, but cannot be assigned to a 16-bit register without the potential loss of the high-order bit. If a and b are 16-bit registers, then the result of $(a+b)$ is 16 bits wide—unless the result is assigned to a register wider than 16 bits. If `answer` is a 17-bit register, then $(\text{answer} = a + b)$ yields a full 17-bit result. But in the expression $(a + b \gg 1)$, the sum of $(a + b)$ produces an interim result that is only 16 bits wide. Therefore, the assignment of $(a + b \gg 1)$ to a 16-bit register loses the carry bit *before* the evaluation performs the one-bit right shift.

There are two solutions to a problem of this type. One is to assign the sum of $(a+b)$ to a 17-bit register before performing the shift and then shift the 17-bit answer into the 16-bits that your model requires. An easier solution is to use the following trick.

The problem:

Evaluate the expression $(a+b)\gg 1$, assigning the result to a 16-bit register without losing the carry bit. Variables a and b are both 16-bit registers.

The solution:

Add the integer zero to the expression. The expression evaluates as follows:

1. $0 + (a+b)$ evaluates—the result is as wide as the widest term, which is the 32-bit zero
2. the 32-bit sum of $0 + (a+b)$ is shifted right one bit

This trick preserves the carry bit until the shift operation can move it back down into 16 bits.

4.4.2

Verilog Rules for Expression Bit Lengths

In the Verilog language, the rules governing the expression bit lengths have been formulated so that most practical situations have a natural solution.

The number of bits of an expression (known as the size of the expression) is determined by the operands involved in the expression and the context in which the expression is given.

A self-determined expression is one where the bit length of the expression is solely determined by the expression itself—for example, an expression representing a delay value.

A context-determined expression is one where the bit length of the expression is determined by the bit length of the expression *and* by the fact that it is part of another expression. For example, the bit size of the right-hand side expression of an assignment depends on itself and the size of the left-hand side.

Expressions

Expression Bit Lengths

Table 4-14 shows how the form of an expression determines the bit lengths of the results of the expression. In Table 4-14, i , j , and k represent expressions of an operand, and $L(i)$ represents the bit length of the operand represented by i .

Expression	Bit length	Comments
unsized constant number	same as integer (usually 32)	
sized constant number	as given	
$i \text{ op } j$ where op is: + - * / % & ^ ~	$\max(L(i), L(j))$	
+i and -i	$L(i)$	
~i	$L(i)$	
$i \text{ op } j$ where op is === !== == != && > >= < <=	1 bit	all operands are self-determined
op i where op is & ~& ~ ^ ~^	1 bit	all operands are self-determined
$i >> j$ $i << j$	$L(i)$	j is self-determined
$i ? j : k$	$\max(L(j), L(k))$	i is self-determined
$\{i, \dots, j\}$	$L(i) + \dots + L(j)$	all operands are self-determined
$\{i \{j, \dots, k\}\}$	$i * (L(j) + \dots + L(k))$	all operands are self-determined

Table 4-14: Bit lengths resulting from expressions

5

Figure 5-0
Example 5-0
Syntax 5-0
Table 5-0

Assignments

The assignment is the basic mechanism for getting values into nets and registers. There are two basic forms of the assignment:

- the *continuous assignment*, which assigns values to *nets*
- the *procedural assignment*, which assigns values to *registers*

An assignment consists of two parts, a left-hand side and a right-hand side, separated by the equal (=) character. The right-hand side can be any expression that evaluates to a value. The left-hand side indicates the variable that the right-hand side is to be assigned to. The left-hand side can take one of the following forms, depending on whether the assignment is a continuous assignment or a procedural assignment.

Statement type	Left-hand side
continuous assignment	net (vector or scalar) constant bit-select of a vector net constant part-select of a vector net concatenation of any of the above 3
procedural assignment	register (vector or scalar) bit-select of a vector register constant part-select of a vector register memory element concatenation of any of the above 4

Table 5-1: Legal left-hand side forms in assignment statements

5.1 Continuous Assignments

Continuous assignments drive values onto nets, both vector and scalar. The significance of the word “continuous” is that the assignment occurs whenever simulation causes the value of the right-hand side to change. Continuous assignments provide a way to model combinational logic without specifying an interconnection of gates. Instead, the model specifies the logical expression that drives the net. The expression on the right-hand side of the continuous assignment is not restricted in any way. It can even contain a reference to a function. Thus, the result of a case statement, if statement, or other procedural construct can drive a net.

The syntax for continuous assignments is as follows:

```
<net_declaration>
    ::= <NETTYPE> <expandrange>? <delay>? <list_of_variables> ;
    || = trireg <charge_strength>? <expandrange>? <delay>? <list_of_variables> ;
    || = <NETTYPE> <drive_strength>? <expandrange>? <delay>?
        <list_of_assignments> ;

<continuous_assign>
    ::= assign <drive_strength>? <delay>? <list_of_assignments> ;

<expandrange>
    ::= <range>
    || = scaled <range>
    || = vectored <range>

<range>
    ::= [ <constant_expression> : <constant_expression> ]

<list_of_assignments>
    ::= <assignment> <,<assignment>>*>

<charge_strength>
    ::= ( small )
    || = ( medium )
    || = ( large )

<drive_strength>
    ::= ( <STRENGTH0> , <STRENGTH1> )
    || = ( <STRENGTH1> , <STRENGTH0> )
```

Syntax 5-1: Syntax for <net_declaration>

5.1.1 The Net Declaration Assignment

The first two alternatives in the <net_declaration> are discussed in Chapter 3, *Data Types* (see Section 3.2.3). The third alternative, the net declaration assignment, allows a continuous assignment to be placed on a net in the same statement that declares that net. The following is an example of the <net_declaration> form of a continuous assignment:

```
wire (strong1, pull0) mynet = enable;
```

Please note: Because a net can be declared only once, only one net declaration assignment can be made for a particular net. This contrasts with the continuous assignment statement; one net can receive multiple assignments of the continuous assignment form.

5.1.2 The Continuous Assignment Statement

The <continuous_assign> statement places a continuous assignment on a net that has been previously declared, either explicitly by declaration or implicitly by using its name in the terminal list of a gate, a user-defined primitive or module instance. The following is an example of a continuous assignment to a net that has been previously declared:

```
assign (strong1, pull0) mynet = enable;
```

Assignments on nets are continuous and automatic. This means that whenever an operand in the right-hand side expression changes value during simulation, the whole right-hand side is evaluated and assigned to the left-hand side.

The following is an example of the use of a continuous assignment to model a four bit adder with carry. Note that the assignment could not be specified directly in the declaration of the nets because it requires a concatenation on the left-hand side.

```
module adder (sum_out, carry_out, carry_in, ina, inb) ;
output [3:0]sum_out;
input [3:0]ina, inb;
output carry_out;
input carry_in;
wire carry_out, carry_in;
wire[3:0] sum_out, ina, inb;
    assign
        {carry_out, sum_out} = ina + inb + carry_in;
endmodule
```

Example 5-1: Use of continuous assign statement

The following example describes a module with one 16-bit output bus. It selects between one of four input busses and connects the selected bus to the output bus.

```
module select_bus(busout, bus0, bus1, bus2, bus3, enable, s);
    parameter n = 16;
    parameter Zee = 16'bz;
    output [1:n] busout;
    input [1:n] bus0, bus1, bus2, bus3;
    input enable;
    input [1:2] s;

    tri [1:n] data;                                // net declaration.
    tri [1:n] busout = enable ? data : Zee; // net declaration with
                                           // continuous assignment.

    assign                                     // assignment statement with
        data = (s == 0) ? bus0 : Zee, // 4 continuous assignments.
        data = (s == 1) ? bus1 : Zee,
        data = (s == 2) ? bus2 : Zee,
        data = (s == 3) ? bus3 : Zee;
endmodule
```

Example 5-2: Net declaration assignment and continuous assign statement

The following sequence of events is experienced during simulation of the description in Example 5-2:

1. The value of *s*, a bus selector input variable, is checked in the assign statement; based on the value of *s*, the net data receives the data from one of the four input busses.
2. The setting of data triggers the continuous assignment in the net declaration for busout; if enable is *set*, the contents of data are assigned to busout; if enable is *clear*, the contents of Zee are assigned to busout.

Note that the parameter Zee has an *explicit* width specification on the high impedance value. This is recommended practice, because it avoids mistakes where extra bits of a value would cause erroneous results. The default width of the high-impedance value (*z*) is the word size of the host machine, typically 32 bits.

Please note: There is a functional difference between a net declaration assignment and a continuous assignment statement. In net declaration assignments, all changes during a time unit in the expression on the right-hand side of the assignment operator (=) propagate to the net. In continuous assignment statements, the value in the expression on the right-hand side of the assignment operator (=) propagates to the net after the final change to the value of the expression.

5.1.3 Delays

A delay given to a continuous assignment specifies the time duration between a right-hand side operand value change and the assignment made to the left-hand side. If the left-hand side references a scalar net, then the delay is treated in the same way as for gate delays—that is, different delays can be given for the output rising, falling, and changing to high impedance (see Chapter 6, *Gate and Switch Level Modeling*).

If the left-hand side references a vector net, then up to three delays can also be applied. The following rules determine which delay controls the assignment:

- If the right-hand side LSB is non-zero or becomes zero, then the falling delay is used.
- If the right-hand side LSB is *z* or becomes *z*, then the turn-off delay is used.
- If the right-hand side LSB is a one or becomes a one, then the rising delay is used.
- If the right-hand side LSB is an *x* or becomes an *x*, then the lesser of the delay values is used.

When different rise and fall delays are specified for a vector net, the actual delay chosen is based on the value or value change of the least significant bit. An example of this is shown in Example 5-3.

```
module least_significant_bit (out);
output [3:0] out;
reg [3:0] a;
wire [3:0] b;

    assign #(10,20) b = a;

    initial
        begin
            a = 'b0000;
            #100 a = 'b1101;
            #100 a = 'b0111;
            #100 a = 'b1110;
        end

    initial
        begin
            $monitor($time, , "a=%b, b=%b",a, b);
            #1000 $finish;
        end
endmodule
```

Compiling source file
Highest level modules:
least_significant_bit

```
    0 a=0000, b=xxxx
    20 a=0000, b=0000
   100 a=1101, b=0000
   110 a=1101, b=1101 // LSB is high so uses a rise delay
   200 a=0111, b=1101
   210 a=0111, b=0111 // LSB is high so uses a rise delay
   300 a=1110, b=0111
   320 a=1110, b=1110 // LSB is low so uses a fall delay
```

Example 5-3: Delay based on the value or value change of the least significant bit

In order to model rise and fall delay for individual bits, you need to expand the register expression to a single bit expression as shown in Example 5-4.

```
module least_significant_bit (out);
output [3:0] out;
reg [3:0] a;
wire [3:0] b;
    assign #(10,20) b[0] = a[0],
                b[1] = a[1],
                b[2] = a[2],
                b[3] = a[3];

    initial
    begin
        a = `b0000;
        #100 a = `b1101;
        #100 a = `b0111;
        #100 a = `b1110;
    end

    initial
    begin
        $monitor($time, , "a=%b, b=%b",a, b);
        #1000 $finish;
    end
endmodule
```

```
Compiling source file
Highest level modules:
least_significant_bit

0 a=0000, b=xxxx
20 a=0000, b=0000
100 a=1101, b=0000
110 a=1101, b=1101 // rise delay of 10 time units
200 a=0111, b=1101
210 a=0111, b=1111 // rise delay of 10 time units
220 a=0111, b=0111 // fall delay of 20 time units
300 a=1110, b=0111
310 a=1110, b=1111 // rise delay of 10 time units
320 a=1110, b=1110 // fall delay of 20 time units
```

Example 5-4: Delay based on the rise and fall delay for individual bits

Note that specifying the delay in a continuous assignment that is part of the net declaration is different from specifying a net delay and then making a continuous assignment to the net. A delay value can be applied to a net in a net declaration, as in the following example:

```
wire #10 wireA;
```

This syntax, called a *net delay*, means that any value change that is to be applied to wireA by some other statement is delayed for ten time units before it takes effect. When there is a continuous assignment in a declaration, the delay is part of the continuous assignment and is *not* a net delay. Thus, it is not added to the delay of other drivers on the net. Furthermore, if the assignment is to an expanded vector net (a net not specified with the keyword *vectored*), then the rising and falling delays are not applied to the individual bits if the assignment is included in the declaration.

In situations where a right-hand side operand changes before a previous change has had time to propagate to the left-hand side, then the latest value change is the only one to be applied. That is, only one assignment occurs. This effect is known as *inertial delay*.

The following example implements a vector exclusive OR. The size and delay of the operation are controlled by parameters, which can be changed when instances of this module are created. See Section 12.2 for details on *Overriding Module Parameter Values*.

```
module modxor(axorb, a, b);  
    parameter size = 8, delay = 15;  
    output [size-1:0] axorb;  
    input [size-1:0] a, b;  
    wire [size-1:0] #delay axorb = a ^ b;  
endmodule
```

Example 5-5: Use of delays with assignments

5.1.4 Strength

The driving strength of a continuous assignment can be specified by the user. This applies only to assignments to scalar nets of the types listed below:

wire	wand	tri	triereg
	wor	triand	tri0
		trior	tri1

Continuous assignments driving strengths can be specified in either a net declaration or in a stand-alone assignment, using the `assign` keyword. The strength specification, if provided, must immediately follow the keyword (either the keyword for the net type or the `assign` keyword) and must precede any delay specified. Whenever the continuous assignment drives the net, the strength of the value will simulate as specified.

A `<drive_strength>` specification contains one strength value that applies when the value being assigned to the net is 1 and a second strength value that applies when the assigned value is 0. The following keywords specify the strength value for an assignment of 1:

`supply1 strong1 pull1 weak1 highz1`

The following keywords specify the strength value for an assignment of 0:

`supply0 strong0 pull0 weak0 highz0`

The order of the two strength specifications is arbitrary. The following two rules constrain the use of drive strength specifications:

- The strength specifications (`highz1, highz0`) and (`highz0, highz1`) are illegal language constructs.
- When the keyword `vectored` is specified together with a specification of strength on a continuous assignment, the keyword `vectored` is ignored.

5.2 Procedural Assignments

The primary discussion of procedural assignments is in Section 8.2. However, a description of the basic ideas here will highlight the differences between continuous assignments and procedural assignments.

As stated above, continuous assignments drive nets in a manner similar to the way gates drive nets. The expression on the right-hand side can be thought of as a combinatorial circuit that drives the net continuously. The word continuous is important; continuous assignments cannot be disabled.

In contrast, procedural assignments put values in registers. The assignment does not have duration; instead, the register holds the value of the assignment until the next procedural assignment to that register.

Procedural assignments occur within procedures such as `always`, `initial`, `task`, and `function` (these procedures are described in later chapters), and can be thought of as "triggered" assignments. The trigger occurs when the flow of execution in the simulation reaches an assignment within a procedure. Reaching the assignment can be controlled by conditional statements. Event controls, delay controls, `if` statements, `case` statements, and looping statements can all be used to control whether assignments get evaluated. Chapter 8, *Behavioral Modeling*, gives details and examples.

5.3 Accelerated Continuous Assignments

This section describes how you can accelerate continuous assignments to make your designs simulate faster. This chapter also explains the following:

- the restrictions on accelerated continuous assignments
- how to accelerate continuous assignments
- the kinds of designs that simulate faster with this feature and the kind of design that simulates slower
- how accelerated continuous assignments affect simulation

5.3.1 The Restrictions on Accelerated Continuous Assignments

You can accelerate continuous assignments only if they meet the restrictions described in this section. These restrictions apply to the following syntax elements of a continuous assignment statement:

- the types of nets on the left-hand side of the assignment operator
- the operators and operands in the expressions on the right-hand side of the assignment operator
- the contents and use of a delay expression

Left-hand side restrictions

You can accelerate a continuous assignment if it assigns a value to one of the following types of nets:

- a scalar net
- a expanded vector net that contains less than 64 bits
- a bit-select of an expanded vector net
- a part-select that is less than 64 bits of an expanded vector net

You can also accelerate a continuous assignment if it assigns a value to a concatenation of these types of nets, provided that the concatenation contains fewer than 64 bits.

An expanded vector net is a vector net that Verilog-XL converts to a group of scalar nets. This group contains one scalar net for each bit of the vector net. Verilog-XL automatically converts or “expands” a vector net for a number of reasons, which include the following:

- to handle bit-selects and part-selects
- to improve performance and to accelerate continuous assignments

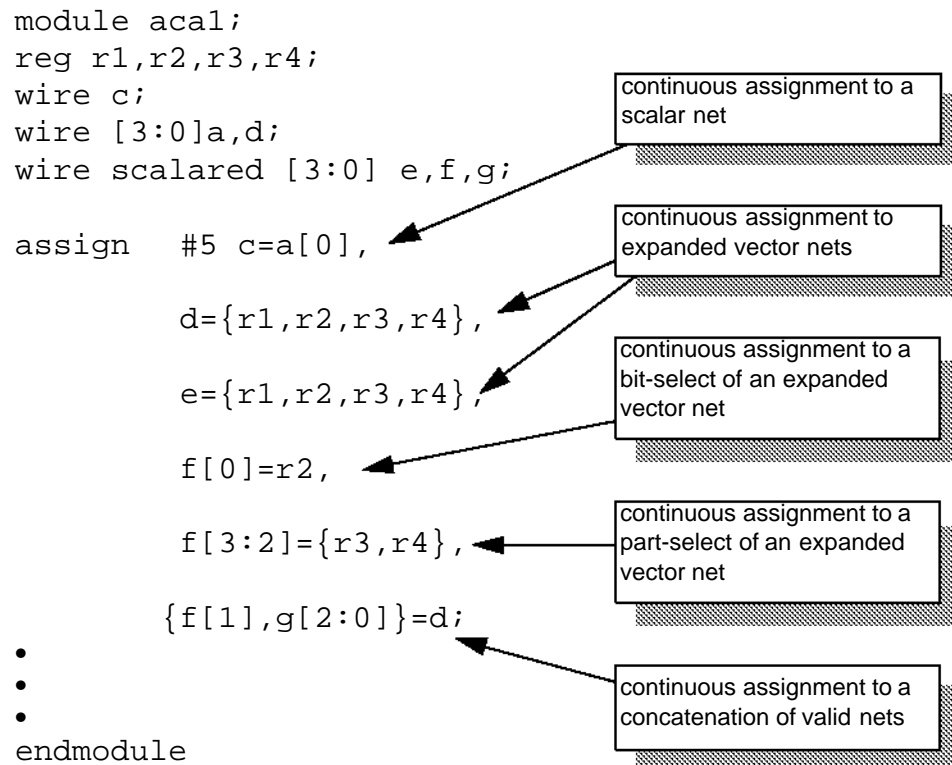
You can require Verilog-XL to expand a vector net by including the keyword `scalared` in the net’s declaration.

An unexpanded vector net is a vector net that Verilog-XL does not convert to scalar nets. You can prevent Verilog-XL from expanding a vector net by including the keyword `vectored` in its declaration.

Example 5-6 shows continuous assignments that you can accelerate because the left-hand side meets these restrictions.

Assignments

Accelerated Continuous Assignments



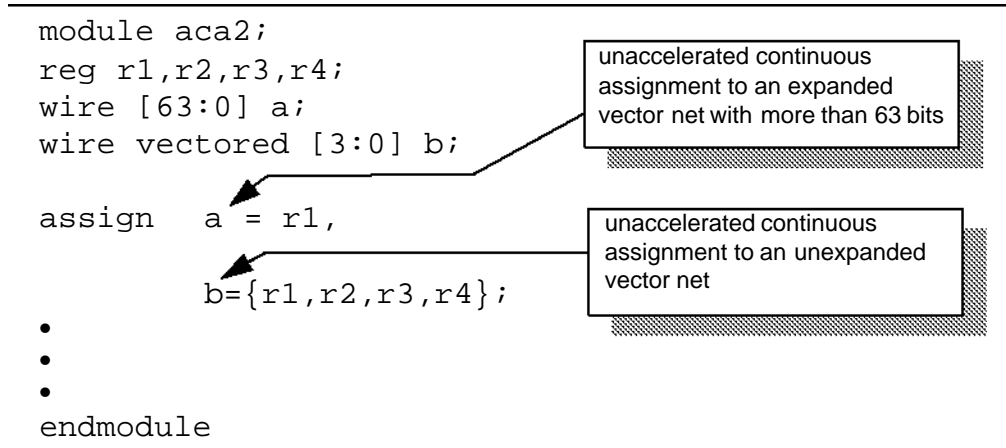
Example 5-6: Left-hand sides of continuous assignments that can be accelerated

Verilog-XL cannot accelerate a continuous assignment to the following types of vector nets:

- vector nets with 64 or more bits
- vector nets declared with the keyword `vectored`

To accelerate a continuous assignment to a vector net, Verilog-XL must expand that vector net. If you declare a vector net with the keyword `vectored`, Verilog-XL cannot accelerate a continuous assignment to it.

Example 5-7 shows continuous assignments that you cannot accelerate because the left-hand side does not meet these restrictions.



Example 5-7: Left-hand side of continuous assignments that cannot be accelerated

Right-hand side restrictions

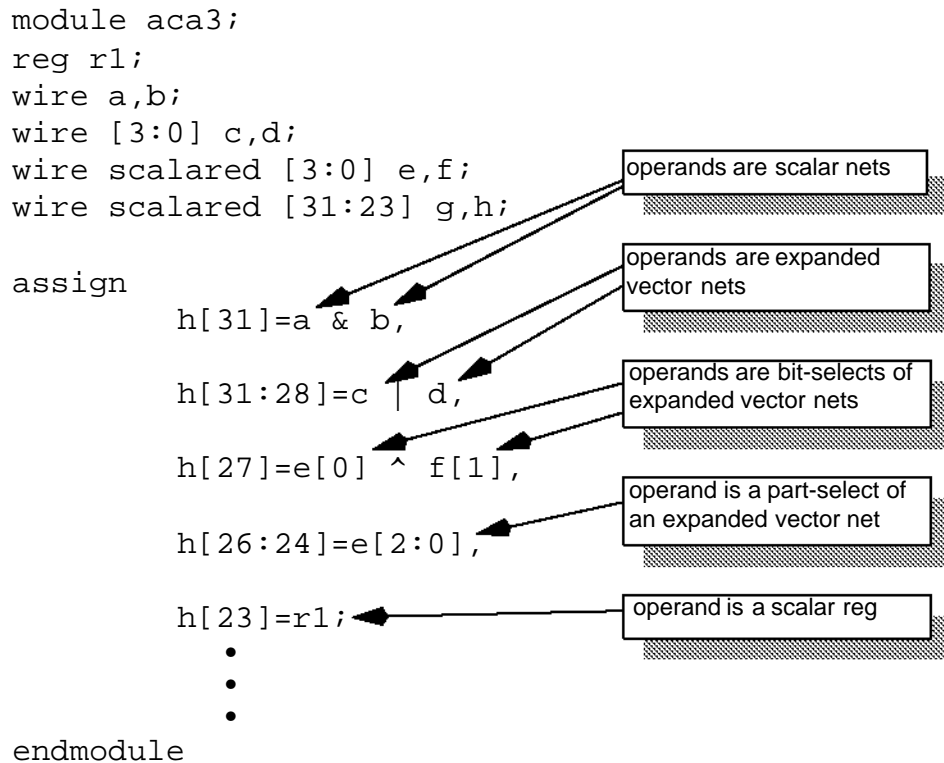
You can accelerate a continuous assignment only if the expression on the right-hand side contains certain operands and operators.

The right-hand expression of a continuous assignment can contain any of the following operands:

- scalar nets
- expanded vector nets that contain less than 64 bits
- bit-selects of expanded vector nets
- part-selects that are less than 64 bits of expanded vector nets
- scalar registers
- constants

You can also accelerate a continuous assignment where the right-hand side is a concatenation of these types of nets, provided that the concatenation contains fewer than 64 bits.

Example 5-8 shows continuous assignments that you can accelerate because the operands in the expression on the right-hand side meet these restrictions.



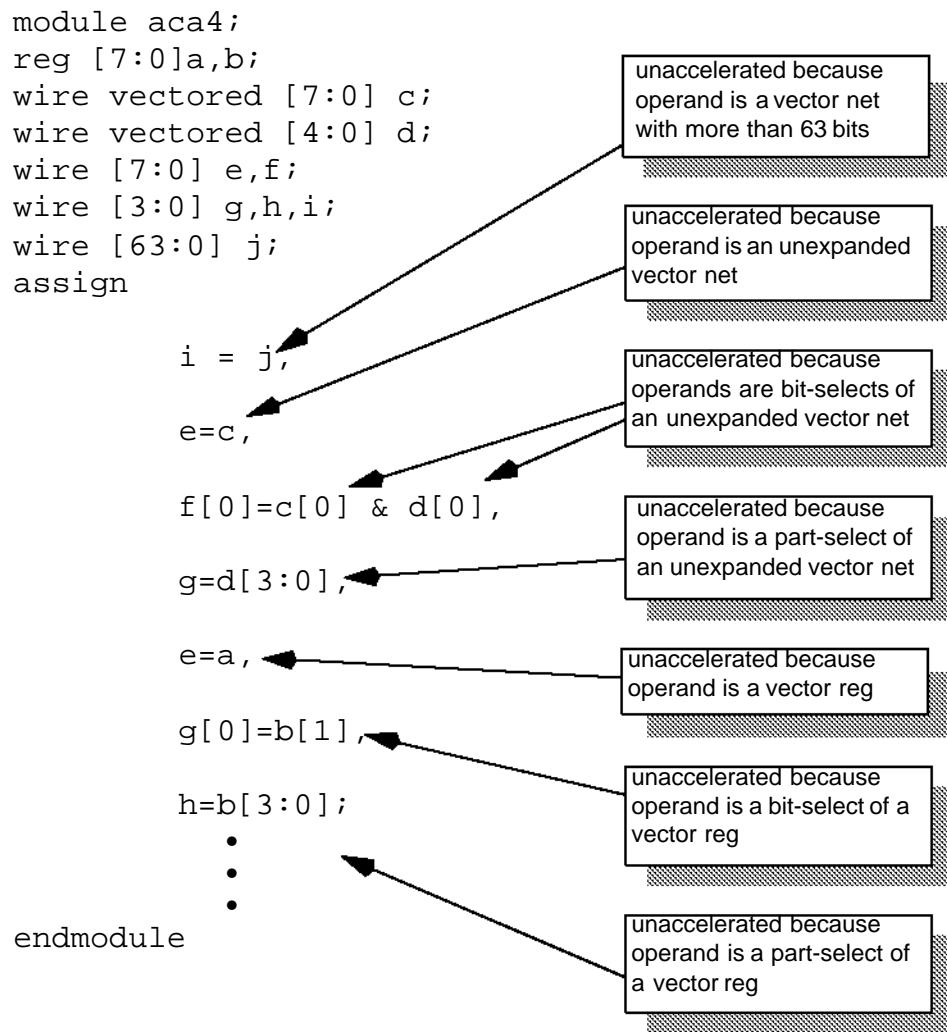
Example 5-8: Operands in continuous assignments that can be accelerated

In Example 5-8, all operands are less than 64 bits.

The prohibited operands are as follows:

- expanded vector nets that contain more than 63 bits
- unexpanded vector nets
- bit-selects of unexpanded vector nets
- part-selects of unexpanded vector nets
- vector registers
- bit-selects of vector registers
- part-selects of vector registers

Example 5-9 shows continuous assignments that you cannot accelerate because the operands in the expression of the right-hand side do not meet these restrictions.



Example 5-9: Operands in continuous assignments that cannot be accelerated

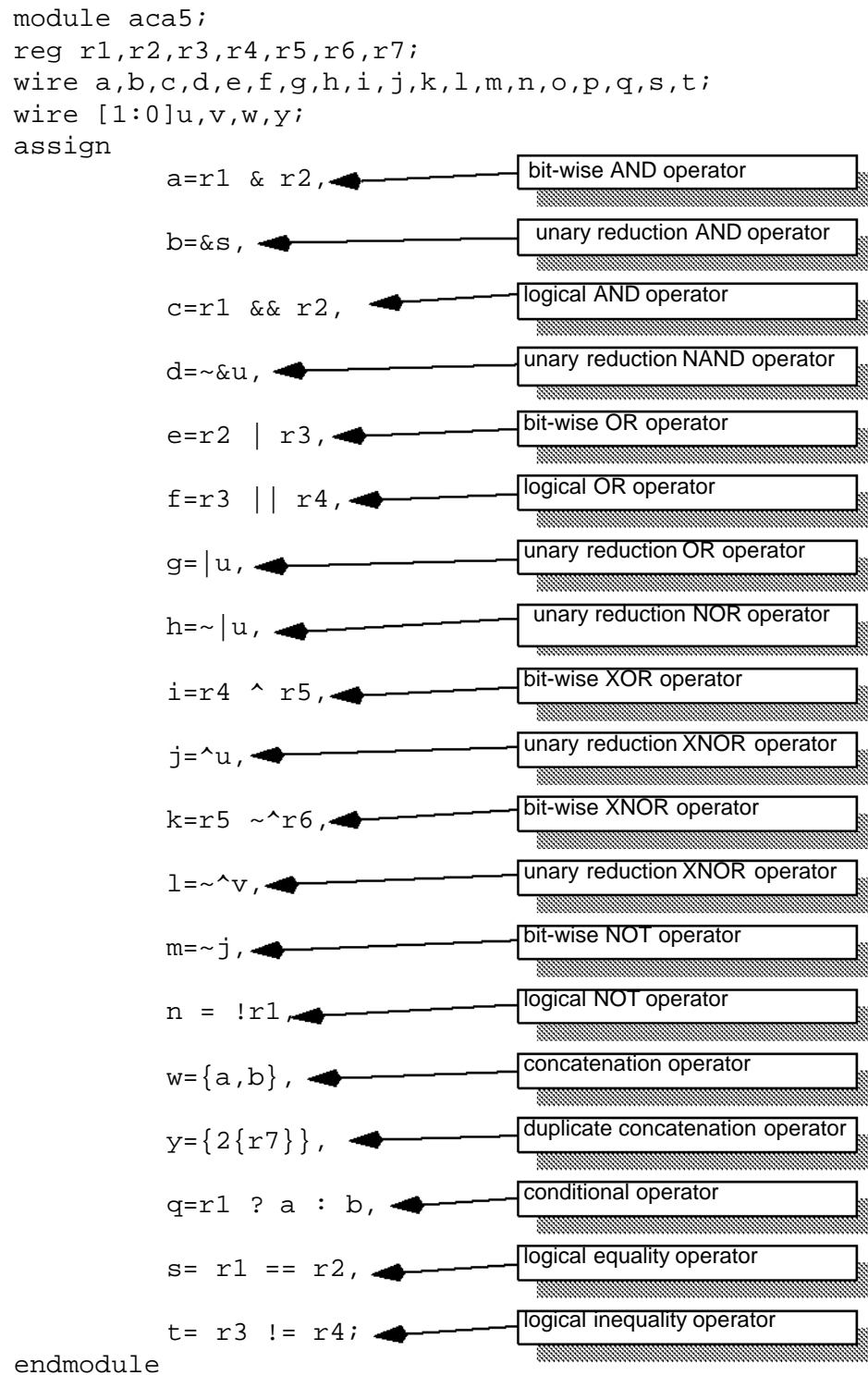
Assignments

Accelerated Continuous Assignments

The expression on the right-hand side of a continuous assignment can only contain the following operators:

&	bit-wise and reduction AND
&&	logical AND
~&	reduction NAND
	bit-wise and reduction OR
	logical OR
~	reduction NOR
^	bit-wise and reduction XOR
~^	bit-wise and reduction XNOR
~	bit-wise NOT
!	logical NOT
{ }	concatenation
{ { } }	duplicate concatenation
? :	conditional
==	logical equality
!=	logical inequality

Example 5-10 shows continuous assignments that you can accelerate because the operators in the expression on the right-hand side meet these restrictions.

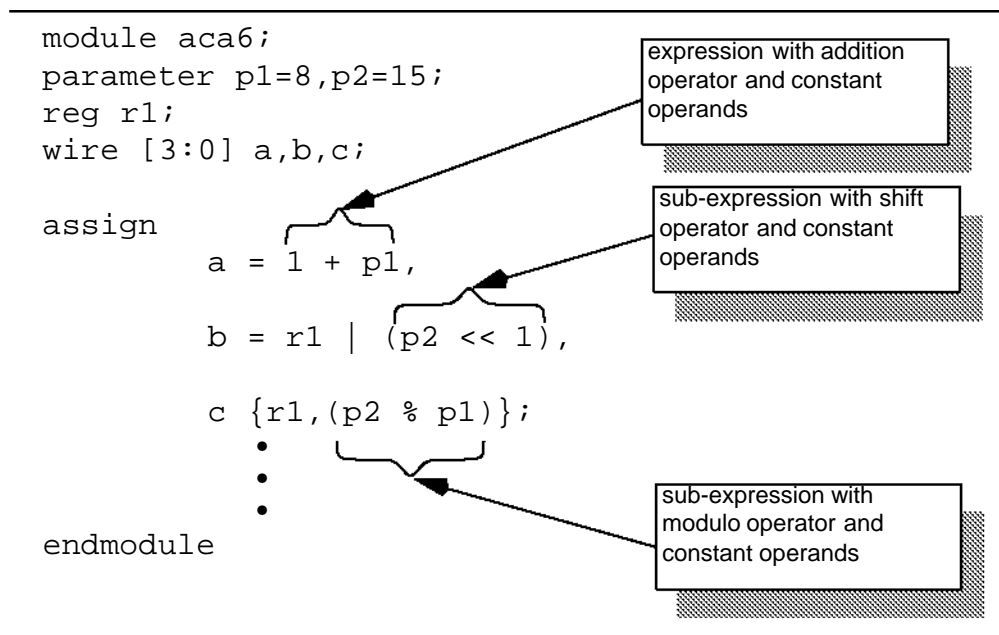


Example 5-10: Operators in continuous assignments that can be accelerated

Assignments

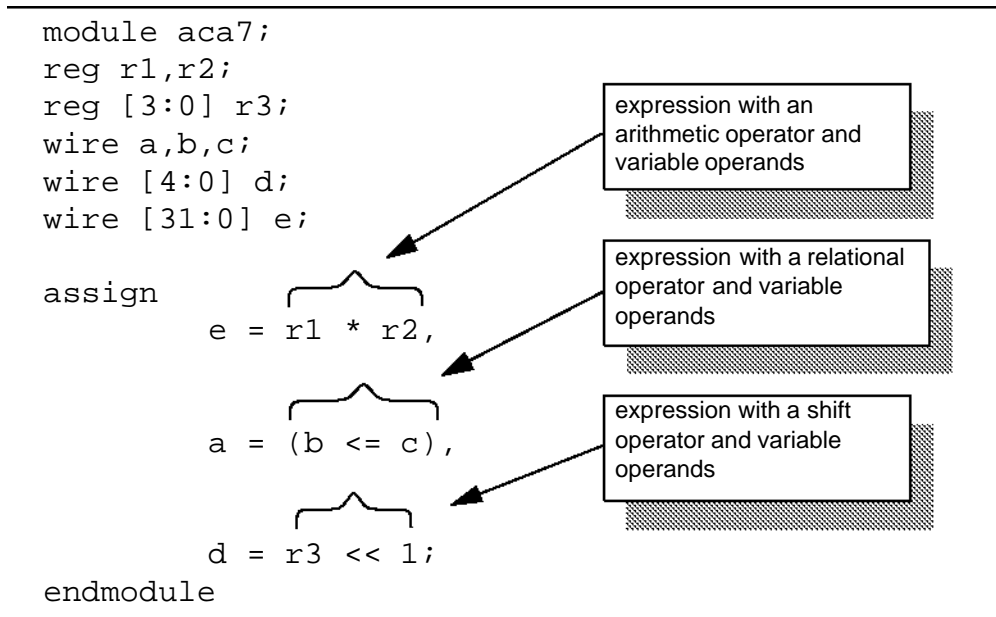
Accelerated Continuous Assignments

You can enter other operators in the right-hand side of accelerated continuous assignments, but only in an expression or sub-expression whose operands are constants. (A sub-expression is a part of an expression that Verilog-XL can evaluate separately.) The prohibition against other operators does not apply in these expressions or sub-expressions because Verilog-XL evaluates them at compile time. Example 5-11 shows how you can use other operators in accelerated continuous assignments.



Example 5-11: Other operators in accelerated continuous assignments

Example 5-12 shows continuous assignments that you cannot accelerate because they use other operators with variable operands.



Example 5-12: Operators in continuous assignments that cannot be accelerated

Delay expression restrictions

You can accelerate a continuous assignment that includes a delay only if that delay is a constant or an expression whose operands are constants.

Example 5-13 shows continuous assignments that you can accelerate because the delay expression meets this restriction.

```
module aca8;
  reg r1,r2;
  wire a,b,q,qb;
  parameter p=10;

  assign #p q = ~(a & qb);
  assign #(p+1) qb = ~(b & q);
  .
  .
  .
endmodule
```

Annotations:

- delay is a constant (points to `#p`)
- delay expression with constant operands (points to `#(p+1)`)

Example 5-13: Delay expressions in continuous assignments that can be accelerated

Example 5-14 shows continuous assignments that you cannot accelerate because the delay expression does not meet this restriction.

```
module aca9;
  wire a,b,c,d;

  reg r1,r2;

  assign #r1 a=c;
  assign #(a & r2) b=d;
  .
  .
  .
endmodule
```

Annotations:

- delay is not a constant (points to `#r1`)
- operands in delay expression are variables (points to `a` and `r2` in `#(a & r2)`)

Example 5-14: Delay expressions in continuous assignments that cannot be accelerated

Restriction summary

Figure 5-1 summarizes the valid syntax elements in accelerated continuous assignments.

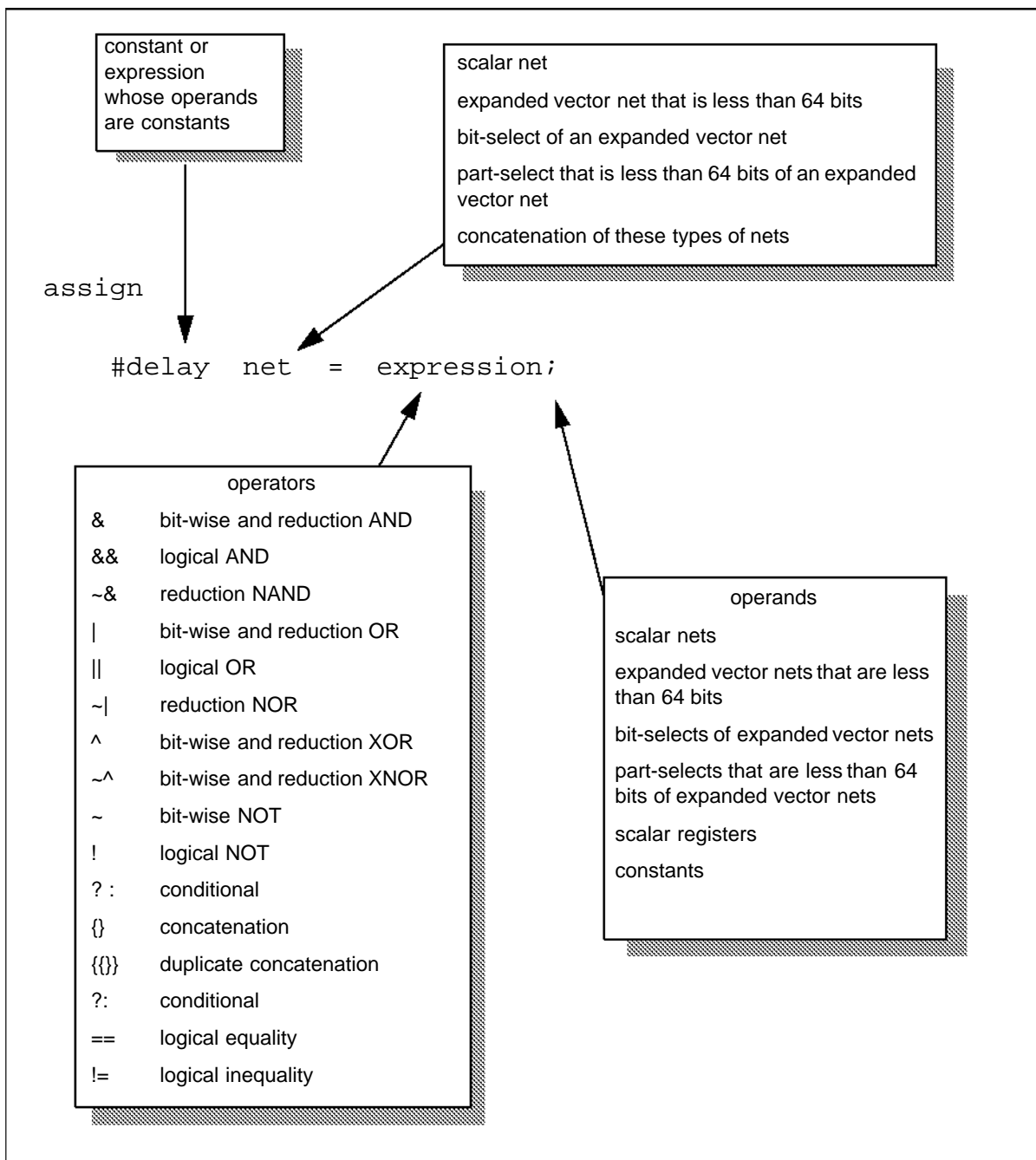


Figure 5-1: Syntax elements of an accelerated continuous assignment

5.3.2

How to Control the Acceleration of Continuous Assignments

Accelerate continuous assignments in your design by entering the `+caxl` command line option. When you enter this option, you accelerate the continuous assignments in the regions of your design that can contain accelerated primitives. You specify these regions with the following mechanisms:

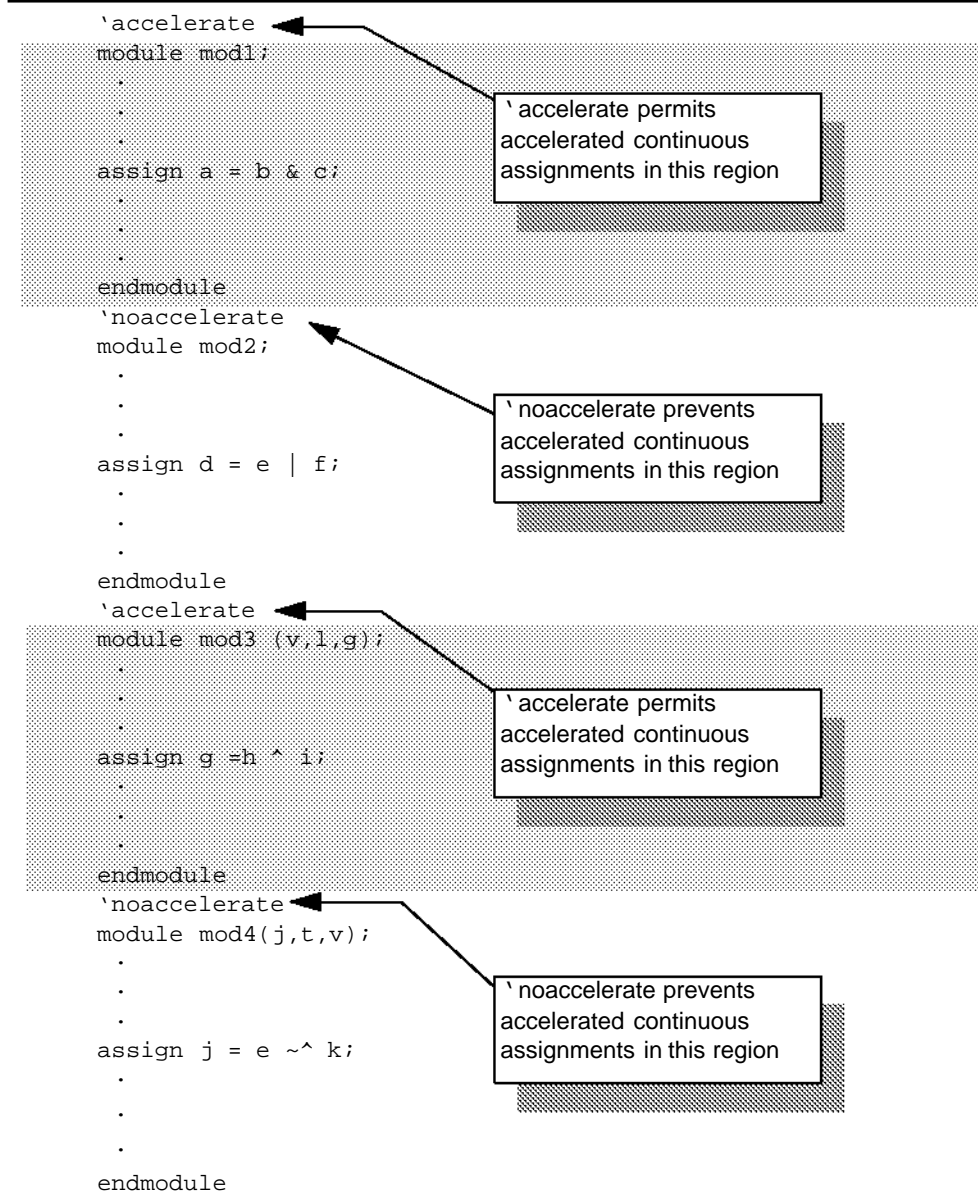
- `-a` command line option
- `'accelerate` compiler directive
- `'noaccelerate` compiler directive

The following command line shows how the `-a` option works with the `+caxl` option:

```
verilog source.v -a +caxl
```

This command line tells Verilog-XL to accelerate all the primitives and continuous assignments in `source.v` that it can.

Example 5-15 shows the regions of a sample design, delimited by `'accelerate` and `'noaccelerate`, whose continuous assignments you can accelerate if you enter the `+caxl` option, without the `-a` option, on the command line. In Example 5-15, the continuous assignments in the grey regions can be accelerated, and the other continuous assignments cannot be accelerated.



Example 5-15: Design regions that you can accelerate

5.3.3

The Effects of Accelerated Continuous Assignments

Accelerating continuous assignments can have the following effects on your simulation:

- faster simulation
- slightly slower compilation
- slightly more memory use
- simulation results that are different from the results when you do not accelerate continuous assignments

These effects are described in the following subsections.

Simulation speed

Accelerating continuous assignments does not increase the simulation speed of all designs. The types of designs that simulate faster, and the one type that simulates slower, are described in this subsection.

Designs that simulate faster

The following is a list of the kinds of designs that simulate faster when you accelerate continuous assignments:

- designs that consist entirely of accelerated continuous assignments to scalar nets
- designs that are a combination of gate-level and accelerated continuous assignments
- gate-level designs that are stimulated by accelerated continuous assignments
- designs that consist of accelerated continuous assignments to large vector nets

The following are examples of these designs and an explanation of how accelerated continuous assignment increases their simulation speed.

1. Accelerating continuous assignments is what most increases the simulation speed of designs that consist entirely of accelerated continuous assignments to scalar nets. These designs simulate approximately eight times faster when you accelerate all their continuous assignments. The following source description shows a design of a multiplexer that consists of accelerated continuous assignments to scalar nets:

```
module aca10 (op1,op2,s1,s2,out,cr);
input  op1,op2,s1,s2;
output out,cr;
wire nop1,nop2,mx1,mx2;
assign
    nop1 = ~op1,
    nop2 = ~op2,
    mx1  = ((op1 & s1)|(nop1 & ~s1)),
    mx2  = ((op2 & s2)|(nop2 & ~s2)),
    out  = mx1 ^ mx2,
    cr   = mx1 & mx2;
endmodule
```

*Example 5-16: Design that consists entirely of
accelerated continuous assignments*

In this source description, data flows through a path of accelerated continuous assignments.

Assignments

Accelerated Continuous Assignments

2. Accelerating continuous assignments also increases the simulation speed of designs whose logic is a combination of gate-level and accelerated continuous assignments. How much the acceleration of the continuous assignments increases the simulation speed depends on the proportion of continuous assignments to gate instances. The following source description shows a design that is a combination of accelerated continuous assignments and gate instances:

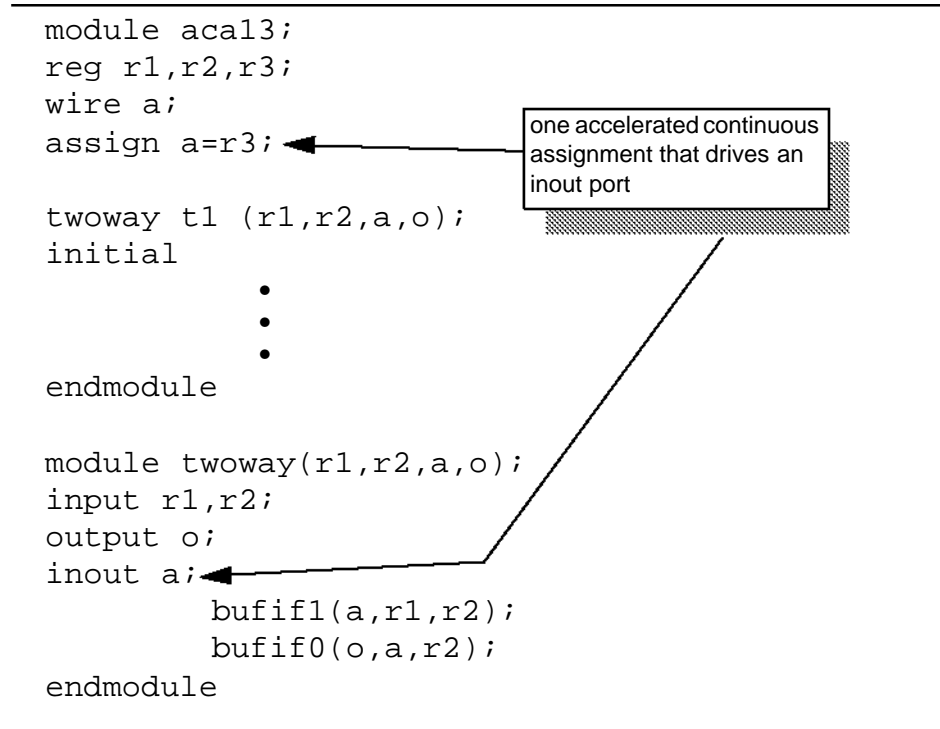
```
module acall1 (op1,op2,s1,s2,out,cr);
input op1,op2,s1,s2;
output out,cr;
wire nop1,nop2,mx1,mx2;
assign
    mx1  = ((op1 & s1)|(nop1 & ~s1)),
    mx2  = ((op2 & s2)|(nop2 & ~s2));

    not nt1 (nop1,op1),
       nt2 (nop2,op2);
    xor xr1 (out,mx1,mx2);
    and ad1 (cr,mx1,mx2);
endmodule
```

Example 5-17: Design that consists of accelerated continuous assignments and gate instances

In this source description, data flows from gates to continuous assignments and back to gates.

3. Accelerating continuous assignments also increases the simulation speed of gate-level designs that are stimulated by accelerated continuous assignments. How much the acceleration of the continuous assignments increases the simulation speed of these designs also depends on the proportion of continuous assignments to gate instances, as in the following source description:



*Example 5-18: Design that contains only one
accelerated continuous assignment*

This design includes one accelerated continuous assignment. Accelerating this continuous assignment does little to increase the design's simulation speed because the accelerated continuous assignment is such a small proportion of this design.

Assignments

Accelerated Continuous Assignments

4. Accelerating the continuous assignments in designs that consist of continuous assignments to large vector nets results in the smallest increase in simulation speed. Continuous assignments to vector nets 64 bits wide and larger cannot be accelerated. The closer the left-hand side of a continuous assignment comes to this limit of 63 bits, the more time the XL algorithm needs to simulate the continuous assignment, as in the following source description:

```
module aca14;
wire [30:0]
a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t;
wire [61:0] m1,m2,m3,m4,m5;
assign  m1=~(( {a,b}&{d,e}) | ({c,d}^ {e,f})),
        m2={e,f}&{h,i},
        m3=~{i,j},
        m4=~( {m,n} | {a,b} ),
        m5=((q & r)^(p | t)~^{q,r});
endmodule
```

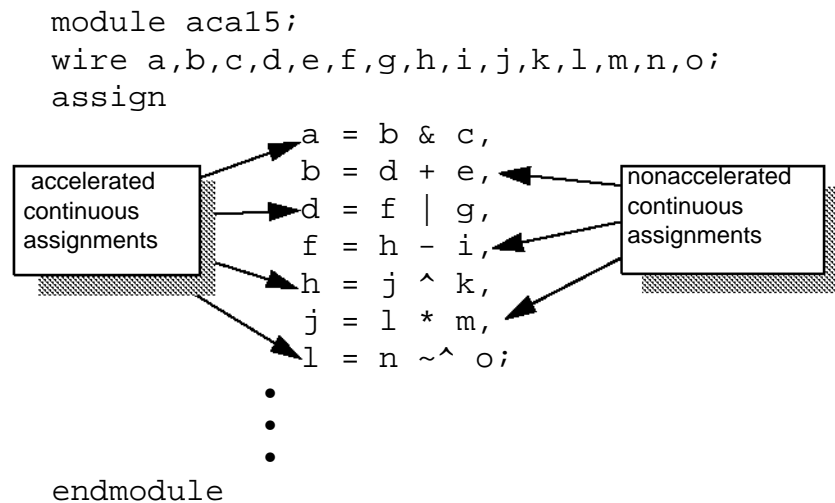
Example 5-19: Design that consists of continuous assignments to large vector nets

This source description shows the continuous assignment of expressions with large operands and several operators to very large vector nets. The greater the complexity of the expression on the right-hand side and the larger the vector net on the left-hand side, the more time the XL algorithm needs to simulate the continuous assignment.

Designs that simulate slower

Not all designs with continuous assignments that can be accelerated simulate faster with the XL algorithm. XL speeds up the simulation when it processes a continuous assignment, but transitions between the XL and non-XL algorithms slow down the simulation. A large number of transitions can make a simulation run slower than if no part of it is simulated by the XL algorithm. The following is a list of designs that contain continuous assignments that you can accelerate, but which simulate faster without accelerating these continuous assignments.

1. Designs whose data flows many times from accelerated to nonaccelerated continuous assignments simulate at a slower speed than if you did not accelerate any continuous assignment. This slower speed is caused by the performance cost of a large number of transitions between algorithms. The following source description shows data flowing through a path of continuous assignments that cause Verilog-XL to transition frequently between algorithms.



Example 5-20: Design whose data flows from accelerated to nonaccelerated continuous assignments

Assignments

Accelerated Continuous Assignments

2. Designs whose data flows many times from accelerated continuous assignments to procedural assignments also simulate at a slower speed than if you did not accelerate any continuous assignment. This slower speed is also caused by transitions between algorithms. In the following source description, data flows between both kinds of assignments.

```
module aca16;
  reg r1,r2,r3,r4,r5;
  wire a,b,c,d,e;
  assign
    a = r1,
    b = r2,
    c = r3,
    d = r4,
    e = r5;

  always
  begin
    #10 r1 = b;
    #10 r2 = c;
    #10 r3 = d;
    #10 r4 = e;
    #10 r5 = ~r5;
  end

  initial
  begin
    r5=1;
    •
    •
    •
  end

endmodule
```

The diagram illustrates the flow of data between two types of assignments. A box labeled "accelerated continuous assignments" is connected by a line to a box labeled "procedural assignments". The "accelerated continuous assignments" box is connected to the "assign" block in the code, which assigns values to wires a, b, c, d, and e. The "procedural assignments" box is connected to the "always" block, which updates registers r1, r2, r3, r4, and r5. The "initial" block is also shown, which initializes register r5 to 1.

Example 5-21: Design whose simulation causes transitions between algorithms

In this source description, a value of 1 propagates through several wires and registers. Data flow begins with a procedural assignment to reg r5, then through a path of registers and wires that are driven by alternating continuous and procedural assignments.

Compilation speed

During compilation, Verilog-XL processes accelerated continuous assignments so that they can be simulated by the XL algorithm. Therefore, compilation time increases as the number of accelerated continuous assignments increases. A design that consists entirely of continuous assignments that can be accelerated takes approximately twice as long to compile if you accelerate these continuous assignments. (In a typical worst-case design, compilation without accelerated continuous assignments took 19 seconds; compilation with accelerated continuous assignments took 41 seconds.)

Memory usage

Accelerated continuous assignments cause Verilog-XL to use more memory at compile time, but less memory at run time.

Verilog-XL needs more memory to compile a design with accelerated continuous assignments. A design that consists entirely of accelerated continuous assignments needs 20% more memory to compile.

Accelerated continuous assignments reduce Verilog-XL's memory requirements during simulation.

The possibility of different results

Accelerating continuous assignments to vector nets when these continuous assignments include delay expressions can produce simulation results that differ from the results produced without accelerating these continuous assignments. This possible difference is caused by the difference between how the XL and non-XL algorithm simulate these continuous assignments.

In both the XL and non-XL algorithms, when a continuous assignment statement includes a delay expression, Verilog-XL evaluates the right-hand side and schedules the assignment to occur after the delay elapses. In the non-XL algorithm, if any of the bits of the right-hand side change before the delay elapses, Verilog-XL re-evaluates the entire right-hand side and reschedules the assignment. In the XL algorithm, if any of the bits of the right-hand side change before the delay elapses, Verilog-XL schedules a subsequent assignment to those bits.

Example 5-22 and Example 5-23 show how accelerating continuous assignments can produce different simulation results.

Example 5-22 shows a module that contains accelerated and unaccelerated continuous assignments that assign the same values and include the same delay expression. The accelerated continuous assignments propagate value changes at simulation times when the unaccelerated continuous assignments do not propagate these value changes.

Assignments

Accelerated Continuous Assignments

```
module dif;
wire [1:0] a1, a2;
wire vectored [1:0] b1;
reg c1,c2;
reg [1:0] d1;

assign    #10 a1 = {c1,c2};

assign    #10 b1 = {c1,c2},
          a2 = d1;

initial
begin
$monitor("At simulation time %0d\n",
$time,
"  accelerated a1=%b\n",a1,
"unaccelerated b1=%b a2=%b\n\n",b1,a2);
#25 c1 = 0;
    d1[1] = 0;
#5  c2 = 0;
    d1[0] = 0;
end
endmodule
```

an accelerated continuous assignment

unaccelerated continuous assignments

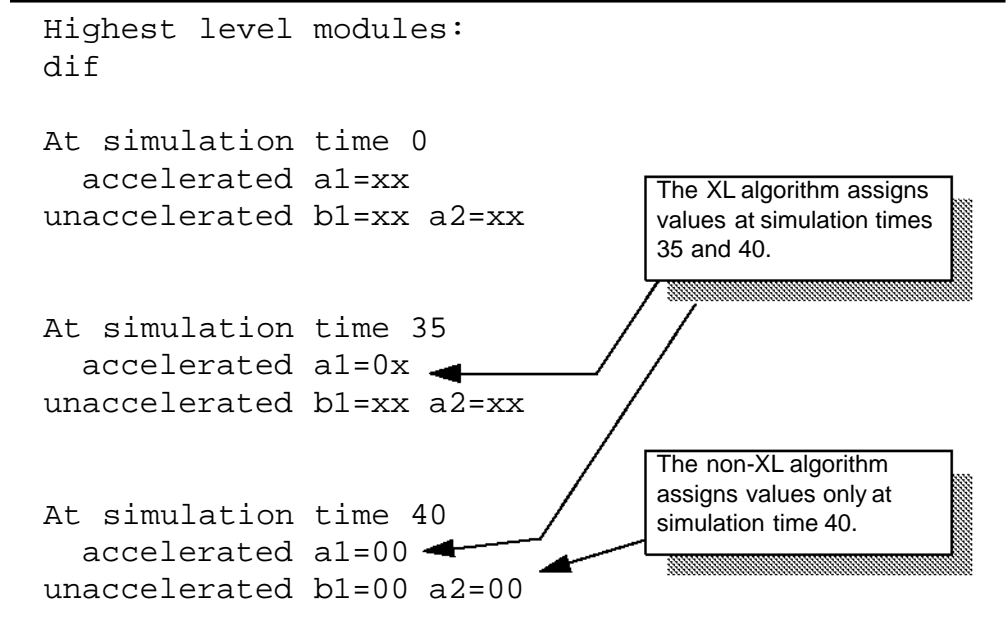
procedural assignments of the same values to the bits of the right-hand side of all three continuous assignments

Example 5-22: Module with accelerated and unaccelerated continuous assignments

In Example 5-22, the continuous assignment to wire `a1` of the concatenation of scalar registers `c1` and `c2` can be accelerated. The continuous assignment to wire `b1` cannot be accelerated because it assigns a value to an unexpanded vector net; the continuous assignment to wire `a2` cannot be accelerated because its operand is a vector reg. The delay expression in these continuous assignments is 10 time units.

Procedural assignments assign the same values to the right-hand sides of these continuous assignments. These procedural assignments specify a five time unit interval between bit changes of the right-hand sides of the continuous assignments.

The XL algorithm schedules the propagation of all bit changes as they occur; the non-XL algorithm does not. The difference in simulation results between the accelerated and unaccelerated continuous assignments is shown in Example 5-23.



Example 5-23: Different simulation results

In Example 5-23, the XL algorithm assigns values to a1 at simulation times 35 and 40. The non-XL algorithm waits until simulation time 40 to assign values.

6

Figure 6-0
Example 6-0
Syntax 6-0
Table 6-0

Gate and Switch Level Modeling

A logic network can be modeled using continuous assignments or switches and logic gates. Gates and continuous assignments serve different modeling purposes and it is important to appreciate the differences between them in order to achieve the right balance between accuracy and efficiency in Verilog-XL. Modeling with switches and logic gates has the following advantages:

- Gates provide a much closer one to one mapping between the actual circuit and the network model.
- There is no continuous assignment equivalent to the bidirectional transfer gate.

A limitation of those nets declared with the keyword `vectored` affects gates and switches as well as continuous assignments. Individual bits of vectored nets cannot be driven; thus, gates and switches can only drive scalar output nets. If you declare a multi-bit net as `vectored` and you drive individual bits of it, Verilog-XL will display a compilation error message. If you do not declare a multi-bit net as `vectored`, Verilog-XL handles it as a vector except in the following cases. A multi-bit net is handled as a scalar if:

- part of the vector is driven by a gate or switch.
- part of the vector is assigned a value with a continuous assignment.

In Verilog-XL, gate and switch level modeling is superior to continuous assignment modeling for the following two reasons:

1. Because gates and switches have fixed functions, Verilog-XL can optimize its data structure so as to reduce the amount of memory needed to simulate large circuits.
2. For a random network of nets, it is likely that the use of gates and switches for modeling gives a shorter simulation run time than the use of continuous assignments.

6.1

Gate and Switch Declaration Syntax

A gate or switch declaration names a gate or switch type and specifies its output signal strengths and delays. It contains one or more gate instances. Gate instances include an optional instance name and a required terminal connection list. The terminal connection list specifies how the gate or switch connects to other components in the model. All the instances contained in a gate or switch declaration have the same output strengths and delays.

Syntax 6-1 presents the gate or switch declaration syntax.

```

<gate_declaration>
    ::= <GATETYPE> <drive_strength>? <delay>? <gate_instance>
        <,<gate_instance>>* ;

<GATETYPE> is one of the following keywords:
    and nand or nor xor xnor buf bufif0 bufif1 not notif0 notif1
    pulldown pullup nmos rmos pmos rmos cmos rmos tran
    rtran tranif0 rtranif0 tranif1 rtranif1

<drive_strength>
    ::= ( <STRENGTH0> , <STRENGTH1> )
    || = ( <STRENGTH1> , <STRENGTH0> )

<delay>
    ::= # <number>
    || = # <identifier>
    || = # ( <mintypmax_expression> <,<mintypmax_expression>>?
        <,<mintypmax_expression>>?)

<gate_instance>
    ::= <name_of_gate_instance>? ( <terminal> <,<terminal>>* )

<name_of_gate_instance>
    ::= <IDENTIFIER>

<terminal>
    ::= <IDENTIFIER>
    || = <expression>

```

Syntax 6-1: Syntax for gate instantiation

This section describes the following parts of a gate or switch declaration:

- the keyword that names the type of gate or switch primitive
- the drive strength specification
- the delay specification
- the identifier that names each gate or switch instance in gate or switch declarations
- the terminal connection list in primitive gate or switch instances

The gate type specification

A gate declaration begins with the <GATETYPE> keyword. The keyword specifies the gate or switch primitive that is used by the instances that follow in the declaration. Table 6-1 lists the keywords that can begin a gate or switch declaration.

Gate Type Keywords				
and	buf	nmos	tran	pullup
nand	not	pmos	tranif0	pulldown
nor	bufif0	cmos	tranif1	
or	bufif1	rnmos	rtran	
xor	notif0	rpmos	rtranif0	
xnor	notif1	rcmos	rtranif1	

Table 6-1: Keywords for the <GATETYPE> syntax item

Explanations of the keywords in Table 6-1 begin in Section 6.2.

The drive strength specification

The drive strength specifications specify the strengths of the values on the output terminals of the instances in the gate declaration. It is possible to specify the strength of the output signals from the gate primitives in Table 6-2.

Gate Types That Support Driving Strength				
and	buf			pullup
nand	not			pulldown
nor	bufif0			
or	bufif1			
xor	notif0			
xnor	notif1			

Table 6-2: Gate types that accept strength specifications

The drive strength specification in Syntax 6-1 has two parts. A gate declaration must contain both parts or no parts, with the exception of pullup and pulldown sources. One of the parts specifies the strength of signals with a value of 1, and the other specifies the strength of signals with a value of 0.

The **STRENGTH1** specification, which specifies the strength of an output signal with a value of 1, is one of the following keywords:

`supply1 strong1 pull1 weak1 highz1`

Specifying `highz1` causes the gate to output a logic value of Z in place of a 1.

The **STRENGTH0** specification, which specifies the strength of an output signal with a value of 0, is one of the following:

`supply0 strong0 pull0 weak0 highz0`

Specifying `highz0` causes the gate to output a logic value of Z in place of a 0.

The strength specifications must follow the gate type keyword and precede any delay specification. The **STRENGTH0** specification can precede or follow the **STRENGTH1** specification. In the absence of a strength specification, the instances have the default strengths `strong1` and `strong0`.

The strength specifications, (`highz0`, `highz1`) and (`highz1`, `highz0`), are invalid and produce the following compiler error message:

Error! Illegal strength specification

The following example shows a drive strength specification in a declaration of an open collector `nor` gate:

```
nor(highz1,strong0)(out1,in1,in2);
```

In this example, the `nor` gate outputs a Z in place of a 1.

Sections 6.10 through 6.15 discuss logic strength modeling in more detail.

The delay specification

The delay specifies the propagation delay through the gates and switches in a declaration. Gates and switches in declarations with no delay specification have no propagation delay. A delay specification can contain up to three delay values, depending on its gate type. Section 6.2 begins discussions of each type of gate that detail the applicable delays. Section 6.16 discusses delays in more detail. The `pullup` and `pulldown` source declarations do not include delay specifications.

The primitive instance identifier

The <IDENTIFIER> in Syntax 6-1 is an optional name given to a gate or switch instance. The name is useful in tracing the operation of the circuit during debugging. Verilog-XL can generate names for unnamed gate instances in the source description. See Section 12.6 for information about automatic naming. Compiler directives discussed in Section 6.17 remove optional gate and net names to reduce virtual memory requirements at simulation time.

Primitive instance connection list

The <terminal>s at the end of Syntax 6-1 are the terminal list. The terminal list describes how the gate or switch connects to the rest of the model. The gate or switch type limits these expressions. The output or bidirectional terminals always come first in the terminal list, followed by the input terminals.

6.2

and, nand, nor, or, xor, and xnor Gates

Declarations of these gates begin with one of these keywords:

and nand nor or xor xnor

The delay specification can be zero, one, or two delays. If there is no delay, there is no delay through the gate. One delay specifies the delays for all output transitions. If the specification contains two delays, the first delay determines the rise delay, the second delay determines the fall delay, and the smaller of the two delays applies to transitions to X.

These six gates have one output and one or more inputs. The first terminal in the terminal list connects to the gate's output and all other terminals connect to its inputs.

Gate and Switch Level Modeling and, nand, nor, or, xor, and xnor Gates

The truth tables for these gates, showing the result of two input values, appear in Table 6-3.

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

nor	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

xnor	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

Table 6-3: Logic tables for and, nand, or, nor, xor, and xnor gates

Versions of these six gates having more than two inputs behave identically with cascaded 2-input gates in producing logic results, but the number of inputs does not alter propagation delays.

The following example declares a two input and gate:

```
and (out,in1,in2);
```

The inputs are in1 and in2. The output is out.

6.3

buf and not Gates

Declarations of these gates begin with one of the following keywords:

buf

not

The delay specification can be zero, one, or two delays. If there is no delay, there is no delay through the gate. One delay specifies the delays for all output transitions. If the specification contains two delays, the first delay determines the rise delay, the second delay determines the fall delay, and the smaller of the two delays applies to transitions to X.

These two gates have one input and one or more outputs. The last terminal in the terminal list connects to the gate's input, and the other terminals connect outputs.

Truth tables for versions of these gates with one input and one output appear in Table 6-4.

buf		not	
inputs	outputs	inputs	outputs
0	0	0	1
1	1	1	0
x	x	x	x
z	x	z	x

Table 6-4: Logic tables for buf and not gates

The following example declares a two output buf:

```
buf (out1,out2,in);
```

The input is in. The outputs are out1 and out2.

6.4

bufif1, bufif0, notif1, and notif0 Gates

Declarations of these gates begin with one of the following keywords:

```
bufif0    bufif1    notif1    notif0
```

A strength specification follows the keyword and a delay specification follows the strength specification. The next item is the optional identifier. A terminal list completes the declaration.

These four gates model three-state drivers. In addition to values of 1 and 0, these gates output Z.

The delay specification can be zero, one, two, or three delays. If there is no delay, there is no delay through the gate. One delay specifies the delay of all transitions. If the specification contains two delays, the first delay determines the rise delay, the second delay determines the fall delay, and the smaller of the two delays specifies the delay of transitions to X and Z. If the specification contains three delays, the first delay determines the rise delay, the second delay determines the fall delay, the third delay determines the delay of transitions to Z, and the smallest of the three delays applies to transitions to X.

Some combinations of data input values and control input values cause these gates to output either of two values, without a preference for either value. These gates' logic tables include two symbols representing such unknown results. The symbol L represents a result which has a value of 0 or Z. The symbol H represents a result which has a value of 1 or Z. Delays on transitions to H or L are the same as delays on transitions to X.

These four gates have one output, one data input, and one control input. The first terminal in the terminal list connects to the output, the second connects to the data input, and the third connects to the control input.

Table 6-5 presents these gates' logic tables:

bufif0		CONTROL			
		0	1	x	z
D A T A	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	x	z	x	x

bufif1		CONTROL			
		0	1	x	z
D A T A	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

notif0		CONTROL			
		0	1	x	z
D A T A	0	1	z	H	H
	1	0	z	L	L
	x	x	z	x	x
	z	x	z	x	x

notif1		CONTROL			
		0	1	x	z
D A T A	0	z	1	H	H
	1	z	0	L	L
	x	z	x	x	x
	z	z	x	x	x

Table 6-5: Logic tables for bufif0, bufif1, notif0, and notif1 gates

The following example declares a bufif1:

```
bufif1 (outw, inw, controlw);
```

The output is outw, the input is inw, and the control is controlw.

6.5 MOS Switches

Models of MOS networks consist largely of the following four primitive types:

```
nmos    pmos    rnmos    rpmos
```

The `pmos` keyword stands for PMOS transistor and the `nmos` keyword stands for NMOS transistor. PMOS and NMOS transistors have relatively low impedance between their sources and drains when they conduct. The `rpmos` keyword stands for resistive PMOS transistor and the `rnmos` keyword stands for resistive NMOS transistor. Resistive PMOS and resistive NMOS transistors have significantly higher impedance between their sources and drains when they conduct than PMOS and NMOS transistors have. The load devices in static MOS networks are examples of `rpmos` and `rnmos` gates. These four gate types are unidirectional channels for data similar to the `bufif` gates.

Declarations of these gates begin with one of the following keywords:

`pmos` `nmos` `rpmos` `rnmos`

A delay specification follows the keyword. The next item is the optional identifier. A terminal list completes the declaration.

The delay specification can be zero, one, two, or three delays. If there is no delay, there is no delay through the switch. A single delay determines the delay of all output transitions. If the specification contains two delays, the first delay determines the rise delay, the second delay determines the fall delay, and the smaller of the two delays specifies the delay of transitions to Z and X. If there are three delays, the first delay specifies the rise delay, the second delay specifies the fall delay, the third delay determines the delay of transitions to Z, and the smallest of the three delays applies to transitions to X. Delays on transitions to H and L are the same as delays on transitions to X.

These four switches have one output, one data input, and one control input. The first terminal in the terminal list connects to the output, the second terminal connects to the data input, and the third terminal connects to the control input.

The `nmos` and `pmos` switches pass signals from their inputs and through their outputs with a change in the signals' strengths in only one case, discussed in Section 6.13. The `rnmos` and `rpmos` gates reduce the strength of signals that propagate through them, as discussed in Section 6.14.

Some combinations of data input values and control input values cause these switches to output either of two values, without a preference for either value. These switches' logic tables include two symbols representing such unknown results. The symbol L represents a result which has a value of 0 or Z. The symbol H represents a result which has a value of 1 or Z.

Table 6-6 presents these switches' logic tables:

pmos		CONTROL			
rmos		0	1	x	z
D A T A	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	z	z	z	z

nmos		CONTROL			
rmos		0	1	x	z
D A T A	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	z	z	z

Table 6-6: Logic tables for pmos, rmos, nmos, and rmos gates

The following example declares a pmos switch:

```
pmos (out,data,control);
```

The output is out, the data input is data, and the control input is control.

6.6 Bidirectional Pass Switches

Declarations of bidirectional switches begin with one of the following keywords:

```
tran      tranif1    tranif0
rtran     rtranif1   rtranif0
```

A delay specification follows the keywords in declarations of tranif1, tranif0, rtranif1, and rtranif0; the tran and rtran devices do not take delays. The next item is the optional identifier. A terminal list completes the declaration.

The delay specifications for tranif1, tranif0, rtranif1, and rtranif0 devices can be zero, one, or two delays. If there is no delay, the device has no turn-on or turn-off delay. If the specification contains one delay, that delay determines both turn-on and turn-off delays. If there are two delays, the first delay specifies the turn-on delay, and the second delay specifies the turn-off delay.

These six devices do not delay signals propagating through them. When these devices are turned off they block signals, and when they are turned on they pass signals.

The `tranif1`, `tranif0`, `rtranif1`, and `rtranif0` devices have three items in their terminal lists. Two are bidirectional terminals that conduct signals to and from the devices, and the other terminal connects to a control input. The terminals connected to inouts precede the terminal connected to the control input in the terminal list.

The `tran` and `rtran` devices have terminal lists containing two bidirectional terminals.

The bidirectional terminals of all six of these devices connect only to scalar nets or bit-selects of expanded vector nets.

The `tran`, `tranif0`, and `tranif1` devices pass signals with an alteration in their strength in only one case, discussed in Section 6.13. The `rtran`, `rtranif0`, and `rtranif1` devices reduce the strength of signals passing through them according to rules discussed in Section 6.14.

The following example declares a `tranif1`:

```
tranif1 (inout1,inout2,control);
```

The bidirectional terminals are `inout1` and `inout2`. The control input is `control`.

6.7 cmos Gates

Declarations of these gates begins with one of these keywords:

`cmos`

`rcmos`

The delay specification can be zero, one, two, or three delays. If there is no delay, there is no delay through the gate. A single delay specifies the delay for all transitions. If the specification contains two delays, the first delay determines the rise delay, the second delay determines the fall delay, and the smaller of the two delays is the delay of transitions to Z and X. If the specification contains three delays, the first delay controls rise delays, the second delay controls fall delays, the third delay controls transitions to Z, and the smallest of the three delays applies to transitions to X. Delays in transitions to H or L are the same as delays in transitions to X.

The `cmos` and `rcmos` gates have a data input, a data output, and two control inputs. In the terminal list, the first terminal connects to the data output, the second connects to the data input, the third connects to the n-channel control input, and the last connects to the p-channel control input.

The `cmos` gate passes signals with an alteration in their strength in only one case, discussed in Section 6.13. The `rcmos` gate reduces the strength of signals passing through it according to rules that appear in Section 6.14.

The `cmos` gate is the combination of a `pmos` gate and an `nmos` gate. The `rcmos` gate is the combination of an `rpmos` gate and an `rnmos` gate. The combined gates in these configurations share data input and data output terminals, but they have separate control inputs. These combined configurations simulate more efficiently than the equivalent networks of two gates.

The equivalence of the `cmos` gate to the pairing of an `nmos` gate and a `pmos` gate is detailed in the following explanation:

```
cmos (w, datain, ncontrol, pcontrol);
```

is equivalent to:

```
nmos (w, datain, ncontrol);
```

```
pmos (w, datain, pcontrol);
```

6.8

pullup and pulldown Sources

Declarations of these sources begin with one of the following keywords:

```
pullup           pulldown
```

A strength specification follows the keyword and an optional identifier follows the strength specification. A terminal list completes the declaration.

A `pullup` source places a logic value of 1 on the nets listed in its terminal list. A `pulldown` source places a logic value of 0 on the nets listed in its terminal list. The signals that these sources place on nets have `pull` strength in the absence of a strength specification. There are no delay specifications for these sources because the signals they place on nets continue throughout simulation without variation.

The following example declares two pullup instances:

```
pullup (strong1, strong0)(neta),(netb);
```

In this example, one gate instance drives `neta`, the other drives `netb`.

6.9 Implicit Net Declarations

Including a previously unused identifier in a terminal list implicitly declares a new net of the `wire` type with zero delay.

If the `wire` type is unsuitable for implicitly declared nets, the compiler directive ``default_nettype` can change the type acquired by implicitly declared nets.

The following is the directive's syntax:

```
`default_nettype <type_of_net>
```

The first character in the directive is an accent grave.

The `<type_of_net>` can be one of the following net types:

<code>wire</code>	<code>tri</code>	<code>tri0</code>
<code>wand</code>	<code>triand</code>	<code>tril</code>
<code>wor</code>	<code>trior</code>	<code>triereg</code>

This directive must occur outside of module definitions. All the modules between any two ``default_nettype` directives are affected by the first ``default_nettype` directive. The effect of the directive crosses source file boundaries in the order in which they appear on the command line. The ``resetall` compiler directive ends the effect of a preceding ``default_nettype` directive. A source description can contain any number of these directives. Implicit nets are of type `wire` in the absence of a ``default_nettype` directive.

Each implicitly declared net must connect to one or more of the following:

- gate output
- tranif bidirectional terminal
- module output port

If an implicitly declared net does not connect to one of the listed items, the compiler produces an error message with this form:

```
"warning! implicit wire (<name>) has no fanin"
```

If nothing drives a net, Verilog-XL assigns a value of Z to the net.

6.10 Logic Strength Modeling

The Verilog HDL provides for accurate modeling of signal contention, bidirectional pass gates, resistive MOS devices, dynamic MOS, charge sharing, and other technology dependent network configurations by allowing scalar net signal values to have a full range of unknown values and different levels of strength or combinations of levels of strength. This multiple level logic strength modeling resolves combinations of signals into known or unknown values to represent the behavior of hardware with maximum accuracy.

A strength specification has two components:

1. the strength of the 0 portion of the net value, designated <STRENGTH0> in Syntax 6-1
2. the strength of the 1 portion of the net value, designated <STRENGTH1> in Syntax 6-1

Despite this division of the strength specification, it is helpful to consider strength as a property occupying regions of a continuum in order to predict the results of combinations of signals.

Table 6-7 demonstrates the continuum of strengths. The left column lists the keywords that specify strength levels of `triereg` or gate output. The middle column in Table 6-7 shows relative strength levels correlated with the keywords. The abbreviations Verilog-XL reports are in the right column in Table 6-7.

strength name	strength level	abbreviation
supply0	7	Su0
strong0	6	St0
pull0	5	Pu0
large0	4	La0
weak0	3	We0
medium0	2	Me0
small0	1	Sm0
highz0	0	HiZ0
highz1	0	HiZ1
small1	1	Sm1
medium1	2	Me1
weak1	3	We1
large1	4	La1
pull1	5	Pu1
strong1	6	St1
supply1	7	Su1

Table 6-7: Strength levels for scalar net signal values

In the preceding table, there are four driving strengths:

supply strong pull weak

Signals with driving strengths propagate from gate outputs and continuous assignment outputs.

In the preceding table, there are three charge storage strengths:

large medium small

Signals with the charge storage strengths originate in the `triereg net` type.

It is possible to think of the strengths of signals in the preceding table as locations on the scale in Figure 6-1.

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Figure 6-1: Scale of strengths

Discussions of signal combinations later in this document will employ graphics similar to Figure 6-1.

A net signal can have one or more strength levels associated with it. If a net signal value is known, its strength levels are all in either the 0 strength part of the scale represented by Figure 6-1, or they are all in its 1 strength part. If a net signal value is unknown, it has strength levels in both the 0 strength and the 1 strength parts. A signal with a value of Z has a strength level only in the HiZ0 or HiZ1 subdivisions of the scale.

6.11 Strengths and Values of Combined Signals

In addition to a value, a signal has either a single unambiguous strength level or it has an ambiguous strength, consisting of more than one level. When signals combine, their strengths and values determine the strength and value of the resulting signal in accord with the principles in the four sections that follow.

6.11.1 Combined Signals of Unambiguous Strength

This section deals with combinations of signals in which each signal has a known value and a single strength level.

If two signals of unequal strength combine in a wired net configuration, the stronger signal is the result. This case appears in Figure 6-2.

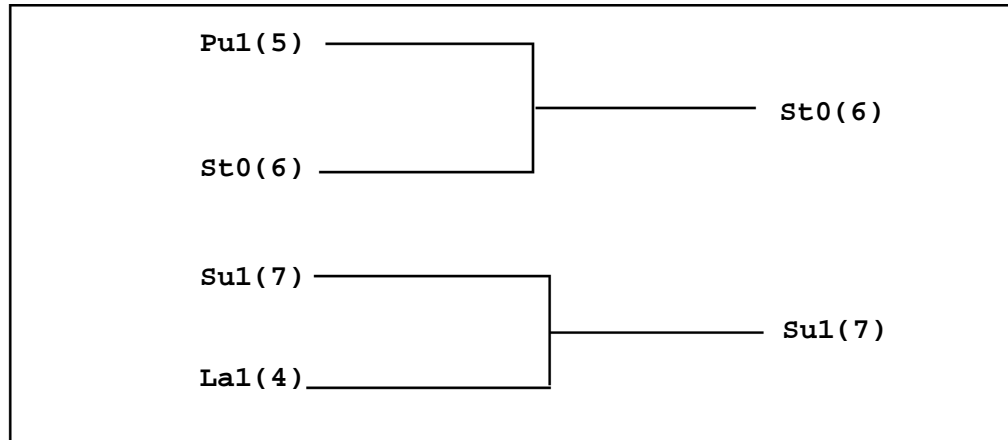


Figure 6-2: Combining unequal strengths

In Figure 6-2, the numbers in parentheses indicate the relative strengths of the signals. The combination of a pull 1 and a strong 0 results in a strong 0, which is the stronger of the two signals. The combination of two signals of like value results in the same value with the greater of the two strengths.

The combination of signals identical in strength and value results in the same signal.

The combination of signals with unlike values and the same strength has three possible results. Two of the results occur in the presence of wired logic and the third occurs in its absence. Section 6.11.4 discusses wired logic. The result in the absence of wired logic is the subject of the first figure in the next section.

6.11.2

Ambiguous Strengths: Sources and Combinations

The classifications of signals possessing ambiguous strengths are the following:

- signals with known values and multiple strength levels
- signals with a value of X, which have strength levels consisting of subdivisions of both the strength 1 and the strength 0 parts of the scale of strengths in Figure 6-1
- signals with a value of L, which have strength levels that consist of high impedance joined with strength levels in the 0 strength part of the scale of strengths in Figure 6-1
- signals with a value of H, which have strength levels that consist of high impedance joined with strength levels in the 1 strength part of the scale of strengths in Figure 6-1

Many configurations can produce signals of ambiguous strength. When two signals of equal strength and opposite value combine, the result has a value of X and the strength levels of both signals and all the smaller strength levels. Figure 6-3 shows the combination of a weak signal with a value of 1 and a weak signal with a value of 0 yielding a signal with weak strength and a value of X.

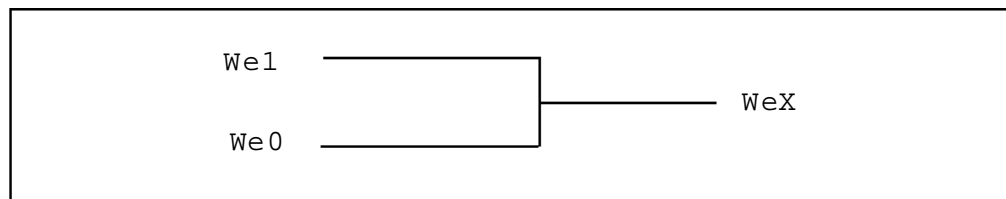


Figure 6-3: Combination of signals of equal strength and opposite values

This signal is described in Figure 6-4.

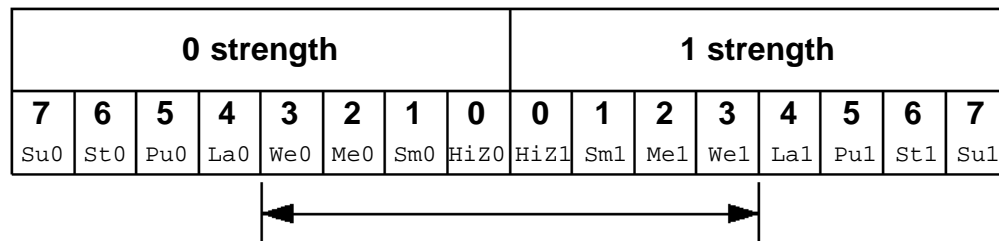


Figure 6-4: Weak X signal strength

An ambiguous signal strength can be a range of possible values. An example is the strength of the output from the tristate drivers with unknown control inputs in Figure 6-5.

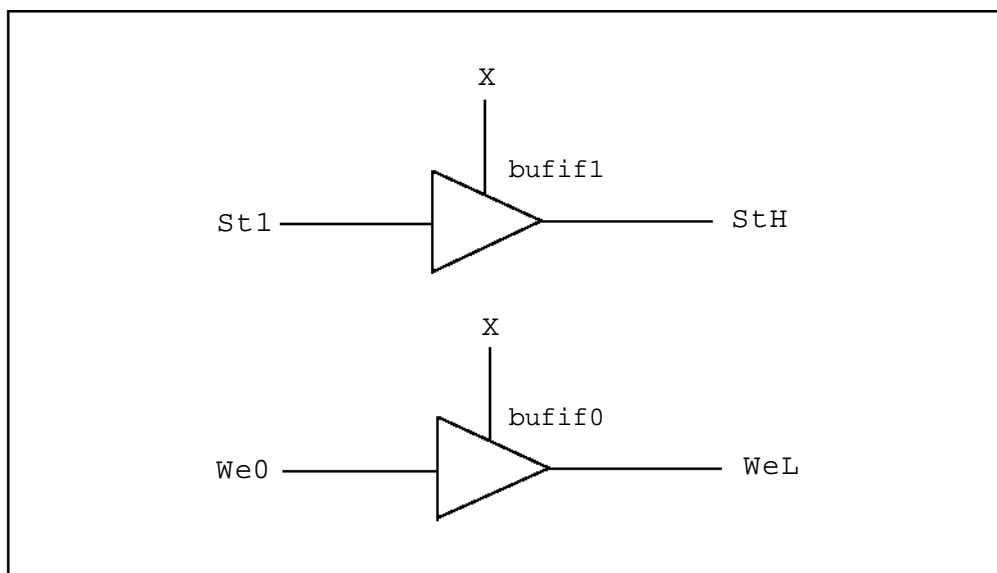


Figure 6-5: Bufifs with control inputs of X

The output of the `bufif1` in Figure 6-5 is a strong H, composed of the range of values described in Figure 6-6.

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Figure 6-6: StrongH range of values

The output of the `bufif0` in Figure 6-5 is a weak L, composed of the range of values described in Figure 6-7.

Gate and Switch Level Modeling

Strengths and Values of Combined Signals

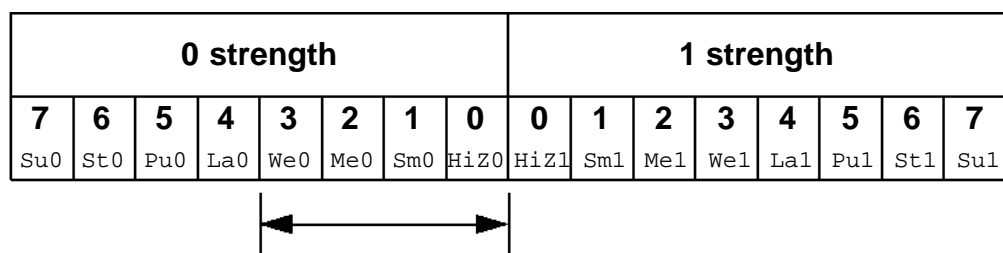


Figure 6-7: Weak L range of values

The combination of two signals of ambiguous strength results in a signal of ambiguous strength. The resulting signal has a range of strength levels that includes the strength levels in its component signals. The combination of outputs from two tristate drivers with unknown control inputs, shown in Figure 6-8, is an example.

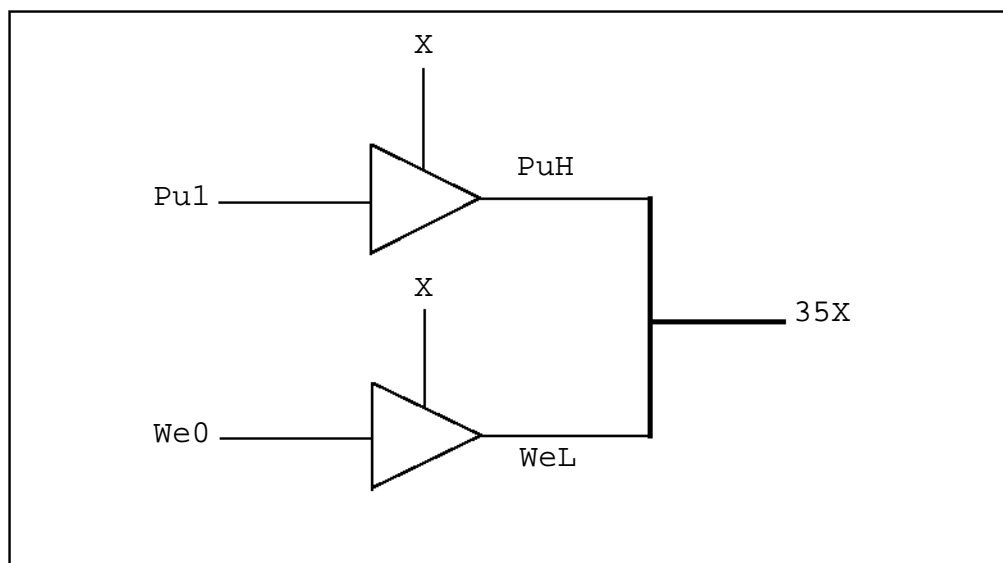


Figure 6-8: Combined signals of ambiguous strength

In Figure 6-8, the combination of signals of ambiguous strengths produces a range which includes the extremes of the signals and all the strengths between them, as described in Figure 6-9.

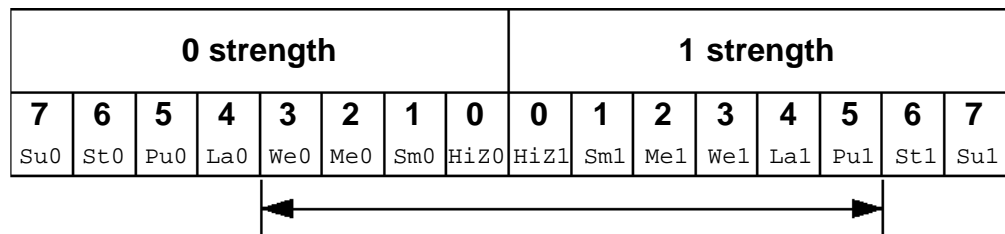


Figure 6-9: An unknown signal's range of strengths

The result is an X because values of both H and L are being driven onto the output net with ambiguous strengths. The number 35, which precedes the X, is a concatenation of two digits. The first is the digit 3, which corresponds to the highest strength level for the result's value of 0. The second digit, 5, corresponds to the highest strength level for the result's value of 1.

Switch networks can produce a range of strengths of the same value, such as the signals from the upper and lower configurations in Figure 6-10.

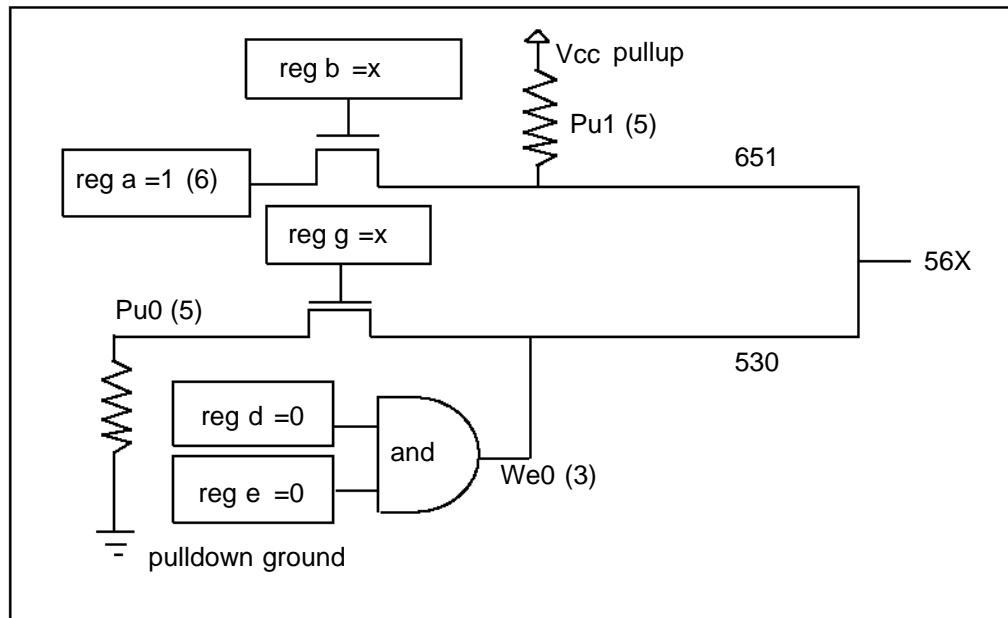


Figure 6-10: Ambiguous strengths from switch networks

In Figure 6-10, the upper combination of a register, a gate controlled by a register of unspecified value, and a pullup produces a signal with a value of 1 and a range of strengths (651) described in Figure 6-11.

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

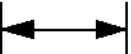


Figure 6-11: Range of two strengths of a defined value

In Figure 6-10 the lower combination of a pulldown, a gate controlled by a register of unspecified value, and an and gate produces a signal with a value of 0 and a range of strengths (530) described in Figure 6-12.

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

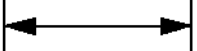


Figure 6-12: Range of three strengths of a defined value

When the signals from the upper and lower configurations in Figure 6-10 combine, the result is an unknown with a range (56X) determined by the extremes of the two signals shown in Figure 6-13.

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

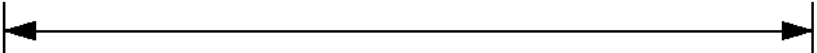


Figure 6-13: Unknown value with a range of strengths

In Figure 6-10, replacing the pulldown in the lower configuration with a supply0 would change the range of the result to the range (StX) described in Figure 6-14.

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

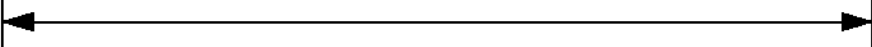


Figure 6-14: Strong X range

The range in Figure 6-14 is strong X, because it is unknown and both of its components' extremes are strong. The extreme of the output of the lower configuration is strong because the lower pmos reduces the strength of the supply0 signal. Section 6.13 discusses this modeling feature.

Logic gates produce results with ambiguous strengths as well as tristate drivers. Such a case appears in Figure 6-15.

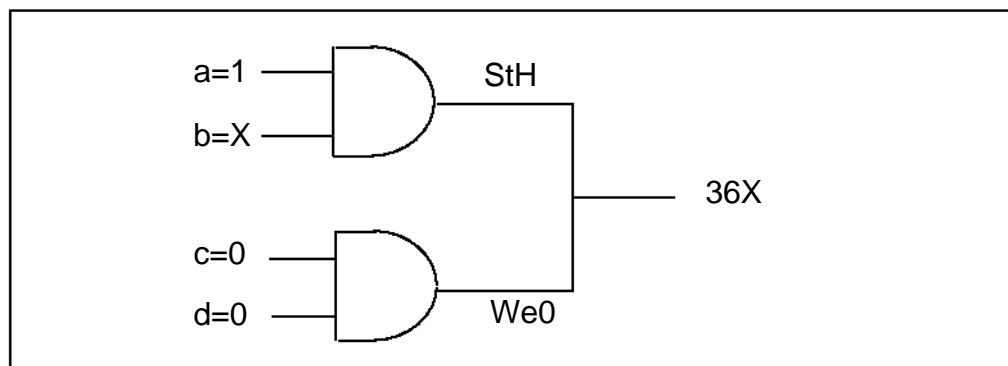


Figure 6-15: Ambiguous strength from gates

In Figure 6-15, register b has an unspecified value, so its input to the upper and gate is strong X. The upper and gate has a strength specification including highz0. The signal from the upper and gate is a strong H composed of the values described in Figure 6-16.

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

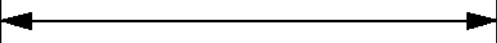


Figure 6-16: Ambiguous strength signal from a gate

HiZ0 is part of the result, because the strength specification for the gate in question specified that strength for an output with a value of 0. A strength specification other than high impedance for the 0 value output results in a gate output of X. The output of the lower and gate is a weak 0 described in Figure 6-17.

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1




Figure 6-17: Weak 0

When the signals combine, the result is the range (36X) described in Figure 6-18.

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

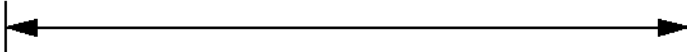


Figure 6-18: Ambiguous strength in combined gate signals

This figure presents the combination of an ambiguous signal and an unambiguous signal. Such combinations are the topic of Section 6.11.3.

6.11.3 Ambiguous Strength Signals and Unambiguous Signals

The combination of a signal with unambiguous strength and known value with another signal of ambiguous strength presents several possible cases. To understand a set of rules governing this type of combination, it is necessary to consider the strength levels of the ambiguous strength signal separately from each other and relative to the unambiguous strength signal. When a signal of known value and unambiguous strength combines with a component of a signal of ambiguous strength, these are the effects:

Rule 1:

The strength levels of the ambiguous strength signal that are greater than the strength level of the unambiguous signal remain in the result.

Rule 2:

The strength levels of the ambiguous strength signal that are smaller than or equal to the strength level of the unambiguous signal disappear from the result, subject to Rule 3.

Rule 3:

If the operation of Rule 1 and Rule 2 results in a gap in strength levels because the signals are of opposite value, the signals in the gap are part of the result.

The following figures show some applications of the rules.

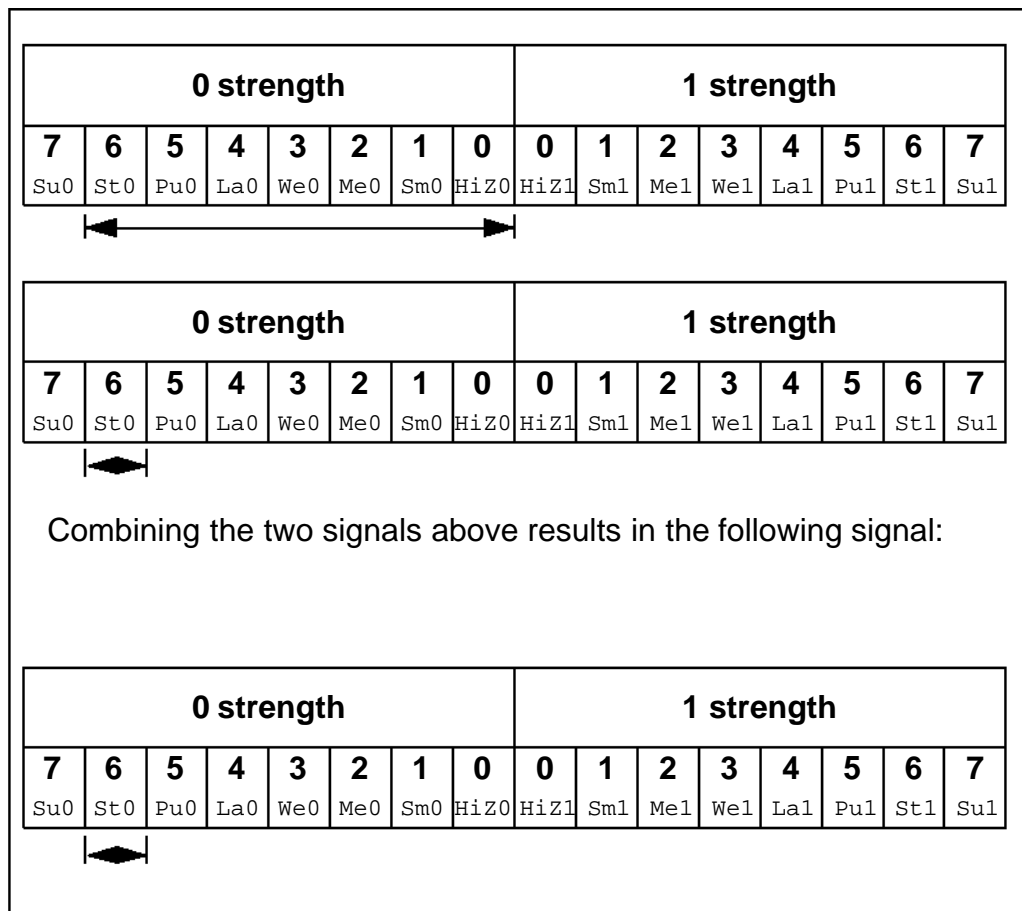


Figure 6-19: Elimination of strength levels

In Figure 6-19, the strength levels in the ambiguous strength signal that are smaller than or equal to the strength level of the unambiguous strength signal disappear from the result, demonstrating Rule 2.

Gate and Switch Level Modeling

Strengths and Values of Combined Signals

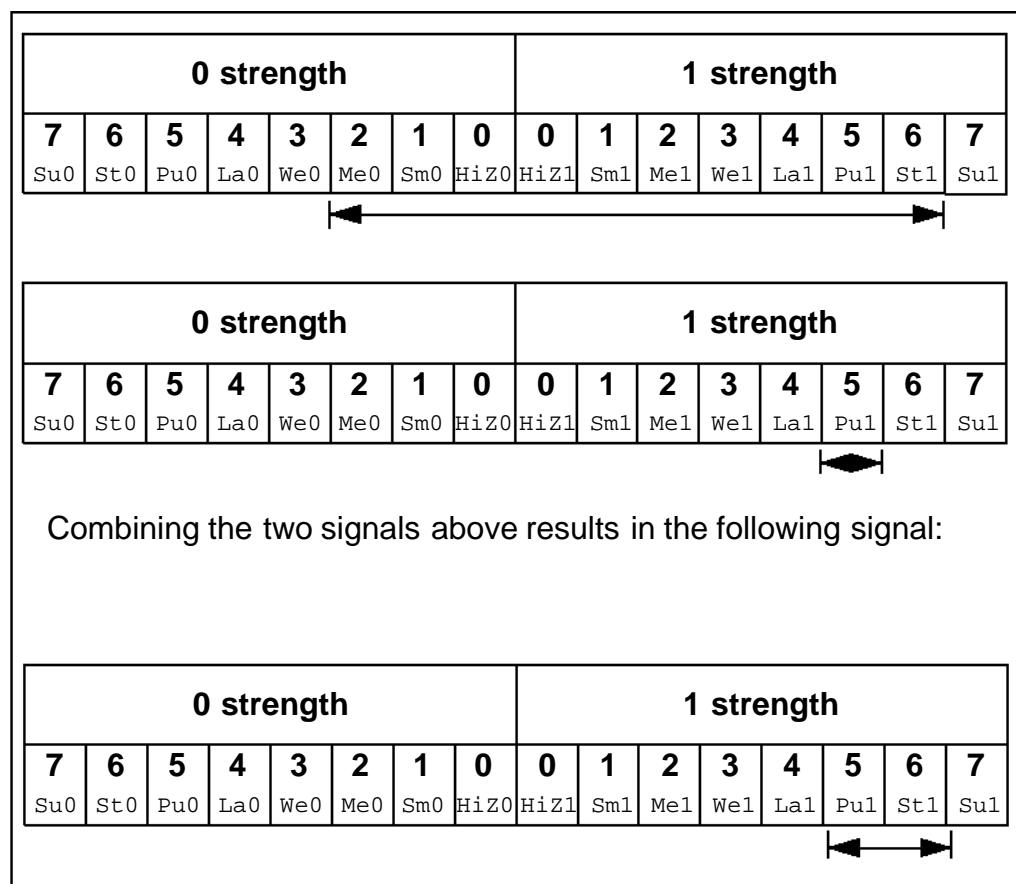


Figure 6-20: Result demonstrating a range and the elimination of strength levels of two values

In Figure 6-20, Rule 1, Rule 2, and Rule 3 apply. The strength levels of the ambiguous strength signal that are of opposite value and lesser strength than the unambiguous strength signal disappear from the result. The strength levels in the ambiguous strength signal that are less than the strength level of the unambiguous strength signal, and of the same value, disappear from the result. The strength level of the unambiguous strength signal and the greater extreme of the ambiguous strength signal define a range in the result.

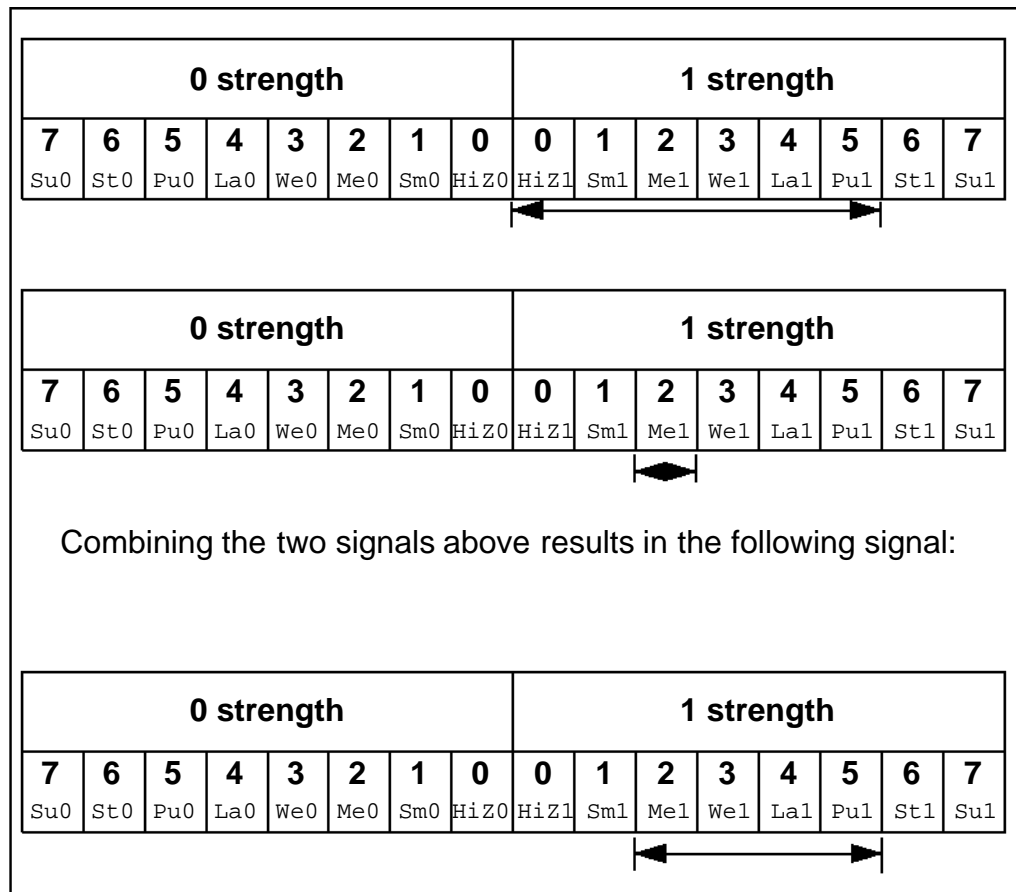


Figure 6-21: Result demonstrating a range and the elimination of strength levels of one value

In Figure 6-21, Rule 1 and Rule 2 apply. The strength levels in the ambiguous strength signal that are less than the strength level of the unambiguous strength signal disappear from the result. The strength level of the unambiguous strength signal and the strength level at the greater extreme of the ambiguous strength signal define a range in the result.

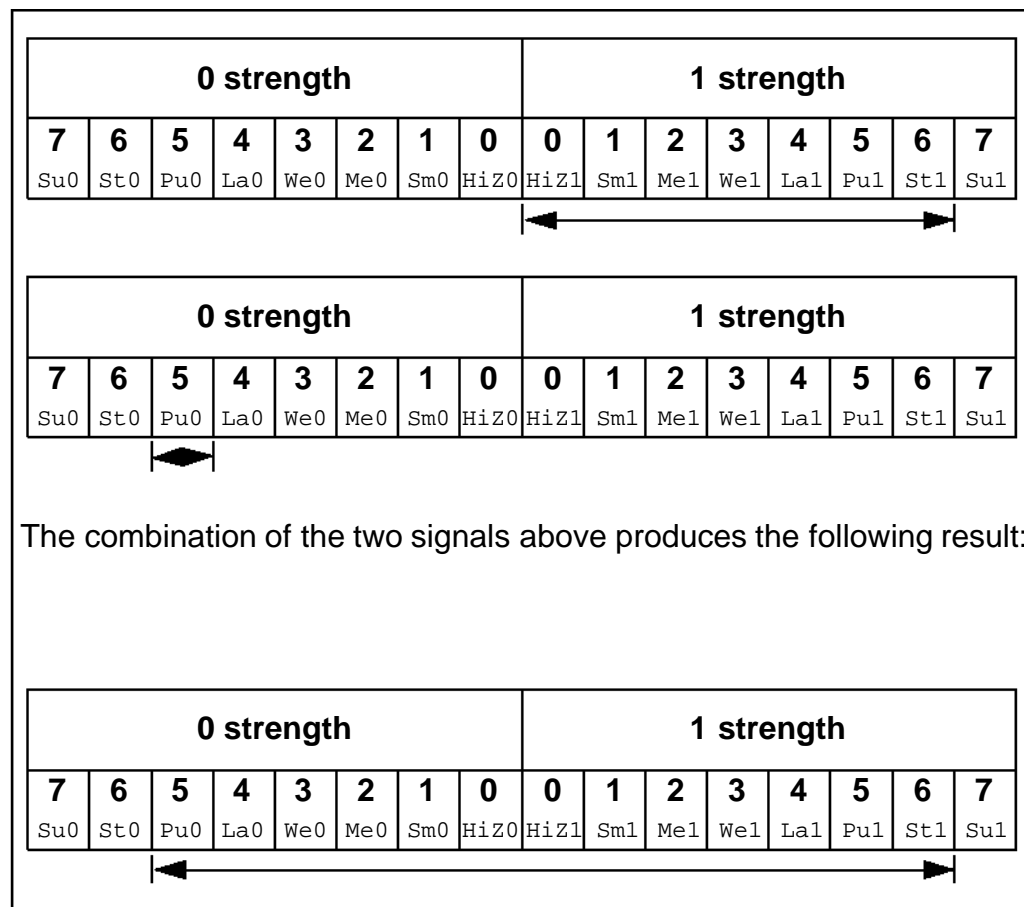


Figure 6-22: A range of both values

In Figure 6-22, Rule 1, Rule 2, and Rule 3 apply. The greater extreme of the range of strengths for the ambiguous strength signal is larger than the strength level of the unambiguous strength signal. The result is a range defined by the greatest strength in the range of the ambiguous strength signal and by the strength level of the unambiguous strength signal.

6.11.4 Wired Logic Net Types

The net types `triand`, `wand`, `trior`, and `wor` resolve conflicts when multiple drivers are at the same level of strength. These net types resolve signal values by treating signals as inputs of logic functions.

For example, consider the combination of two signals of unambiguous strength in Figure 6-23.

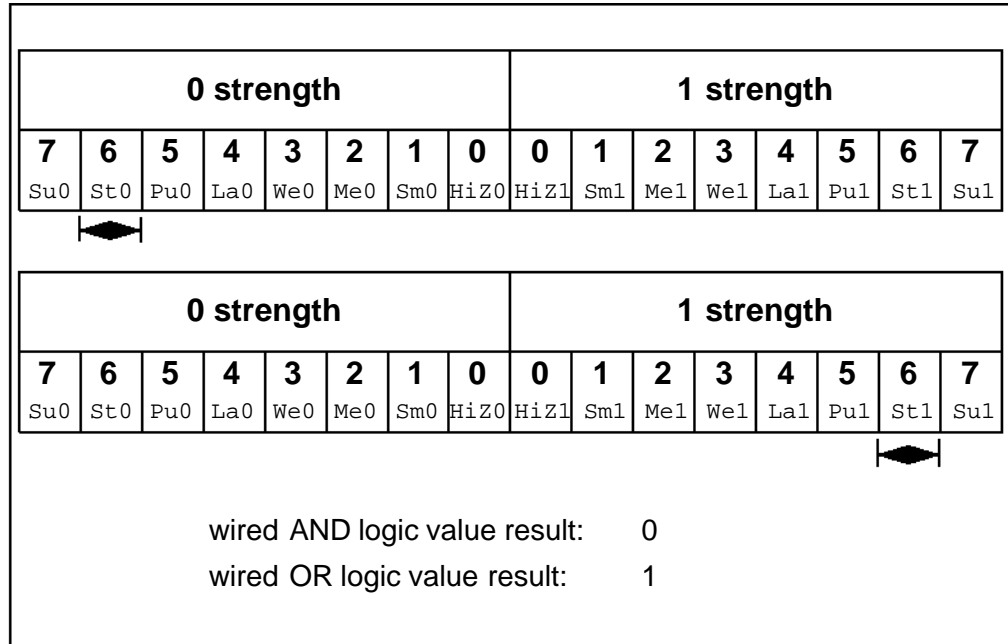


Figure 6-23: Wired logic with unambiguous strength signals

The combination of the signals in Figure 6-23, using wired AND logic, produces a result with the same value as the result produced by an AND gate with the two signals' values as its inputs. The combination of signals using wired OR logic produces a result with the same value as the result produced by an OR gate with the two signals' values as its inputs. The strength of the result is the same as the strength of the combined signals in both cases. If the value of the upper signal changes so that both signals in Figure 6-23 possess a value of 1, then the results of both types of logic have a value of 1.

When ambiguous strength signals combine in wired logic, it is necessary to consider the results of all combinations of each of the strength levels in the first signal with each of the strength levels in the second signal, as shown in Figure 6-24.

Gate and Switch Level Modeling

Strengths and Values of Combined Signals

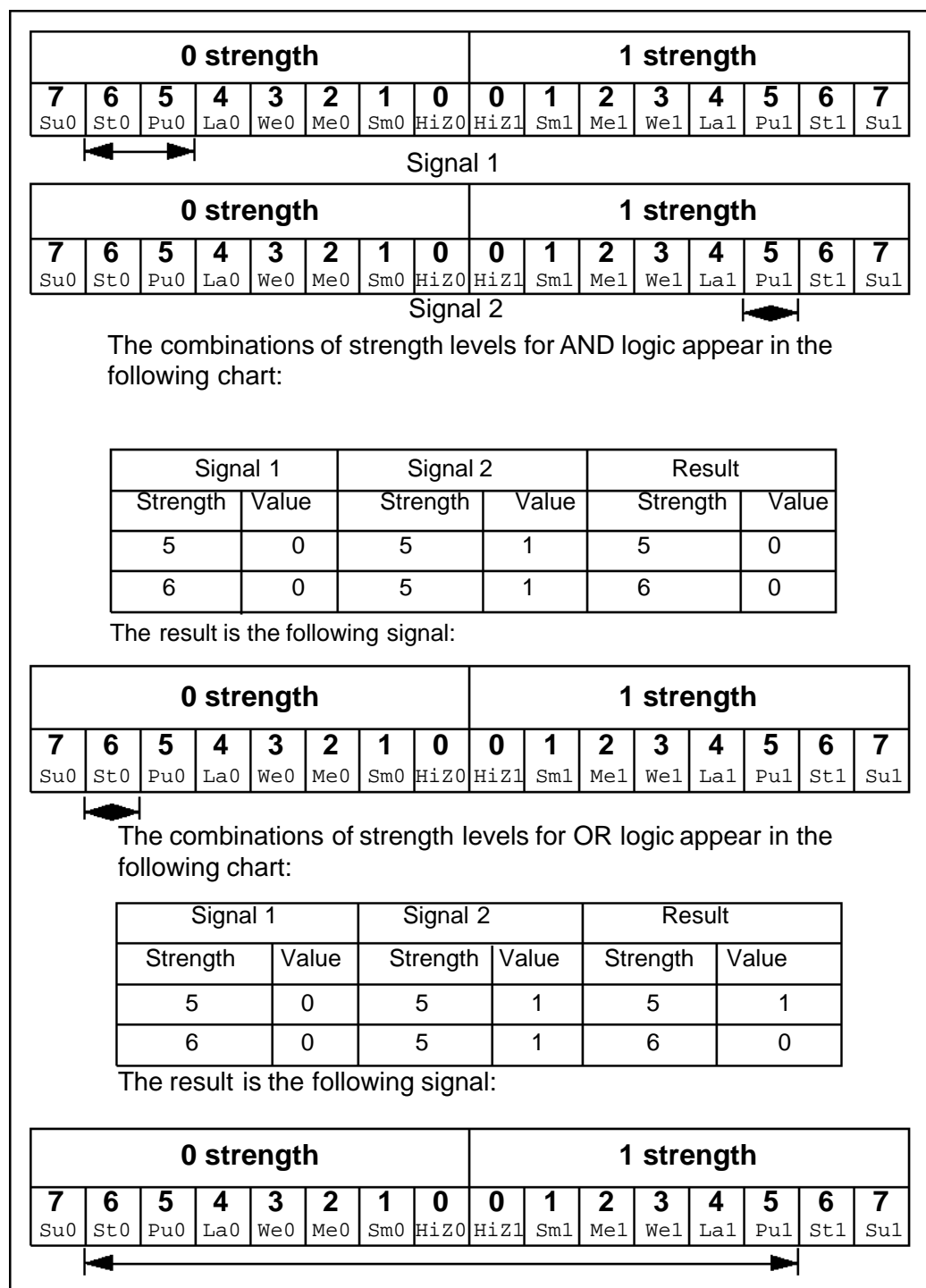


Figure 6-24: Wired logic and ambiguous strengths

6.12 Mnemonic Format

Trace messages giving signal strength information are compatible with the %v format option in the \$display system task. See Section 21.2 for more information on this mnemonic strength notation.

6.13 Strength Reduction by Non-Resistive Devices

The nmos, pmos, and cmos gates pass through the strength from the data input to the output, except that a supply strength is reduced to a strong strength.

The tran, tranif0, and tranif1 gates do not affect signal strength across the bidirectional terminals, except that a supply strength is reduced to a strong strength.

6.14 Strength Reduction by Resistive Devices

The rnmos, rpmos, rcmos, rtran, rtranif1, and rtranif0 devices reduce the strength of signals that pass through them according to Table 6-8.

input strength	reduced strength
supply drive	pull drive
strong drive	pull drive
pull drive	weak drive
weak drive	medium capacitor
large capacitor	medium capacitor
medium capacitor	small capacitor
small capacitor	small capacitor
high impedance	high impedance

Table 6-8: Strength reduction rules

6.15

Strengths of Net Types

The `tri0`, `tri1`, `supply0`, and `supply1` net types generate signals with specific strength levels. The `triereg` declaration can specify either of two signal strength levels other than a default strength level.

6.15.1

`tri0` and `tri1` Net Strengths

The `tri0` net type models a net connected to a resistive pulldown device. Its signal has a value of 0 and a pull strength in the absence of an overriding source. The `tri1` net type models a net connected to a resistive pullup device: its signal has a value of 1 and a pull strength in the absence of an overriding source.

6.15.2

`triereg` Strength

The `triereg` net type models charge storage nodes. The strength of the drive resulting from a `triereg` net that is in the charge storage state (that is, a driver charged the net and then went to high impedance) is one of these three strengths: `large`, `medium`, or `small`. The specific strength associated with a particular `triereg` net is specified by the user in the net declaration. The default is `medium`. The syntax of this specification is described in Section 3.4.1.

6.15.3

`supply0` and `supply1` Net Strengths

The `supply0` net type models ground connections. The `supply1` net type models connections to power supplies. The `supply0` and `supply1` net types have `supply` driving strengths.

6.16

Gate and Net Delays

Gate and net delays provide a means of accurately describing delays through a circuit. The gate delays specify the signal propagation delay from any gate input to the gate output. Up to three values per output can be specified. The descriptions in this chapter of each gate type give the rules for which gates can take how many delays—see Section 6.2 through Section 6.7.

Net delays refer to the time it takes from any driver on the net changing value to the time when the net value is updated and propagated further. Up to three delay values per net can be specified.

Please note: Verilog-XL treats two nets connected by a bidirectional switch as one net and simulates the delays on both nets in parallel.

For both gates and nets, the default delay is zero when no delay specification is given. When one delay value is given, then this value is used for all propagation delays associated with the gate or net. The following is an example of a delay specification with one delay:

```
and #(10) (out, in1, in2);
```

The following is an example of a delay specification with two delays:

```
and #(10, 12) (out, in1, in2);
```

When two delays are given, the first specifies the rise delay and the second specifies the fall delay. The delay when the signal changes to high impedance or to unknown is the lesser of the two delay values.

The following is an example of a delay specification with three delays:

```
and #(10, 12, 11) (out, in1, in2);
```

For a three delay specification:

- the first delay refers to the transition to the 1 value (rise delay)
- the second delay refers to the transition to the 0 value (fall delay)
- the third delay refers to the transition to the high impedance value

When a value changes to the unknown (X) value, the delay is the smallest of the three delays.

Table 6-9 summarizes the from-to propagation delay choice for the two and three delay specifications.

from value:	to value:	delay used if there are:	
		2 delays	3 delays
0	1	d1	d1
0	x	min(d1, d2)	min(d1, d2, d3)
0	z	min(d1, d2)	d3
1	0	d2	d2
1	x	min(d1, d2)	min(d1, d2, d3)
1	z	min(d1, d2)	d3
x	0	d2	d2
x	1	d1	d1
x	z	min(d1, d2)	d3
z	0	d2	d2
z	1	d1	d1
z	x	min(d1, d2)	min(d1, d2, d3)

Table 6-9: Rules for propagation delays

The following example specifies a simple latch module with tri-state outputs, where individual delays are given to the gates. The propagation delay from the primary inputs to the outputs of the module will be cumulative, and depends on the signal path through the network.

```
module tri_latch(qout, nqout, clock, data, enable);
    output qout, nqout;
    input clock, data, enable;

    tri qout, nqout;
    not #5
        (ndata, data);
    nand #(3, 5)
        (wa, data, clock),
        (wb, ndata, clock);
    nand #(12, 15)
        (q, nq, wa),
        (nq, q, wb);
    bufif1 #(3, 7, 13)
        q_drive (qout, q, enable),
        nq_drive (nqout, nq, enable);
endmodule
```

Example 6-1: Using delay values

6.16.1 min/typ/max Delays

The syntax for delays on gate primitives (including user-defined primitives), nets, and continuous assignments allows three values each for the rising, falling, and turn-off delays. The minimum, typical, and maximum values for each are specified as constant expressions separated by colons. The following example shows min/typ/max values for rising, falling, and turn-off delays:

```
module iobuf(io1, io2, dir);
    •
    •
    •
    bufif0 #(5:7:9, 8:10:12, 15:18:21) (io1, io2, dir);
    bufif1 #(6:8:10, 5:7:9, 13:17:19) (io2, io1, dir);
    •
    •
    •
endmodule
```

Example 6-2: Syntax example for delay expressions

The syntax for delay controls in procedural statements also allows minimum, typical, and maximum values. These are specified by expressions separated by colons. Example 6-3 illustrates this concept.

```
parameter
    min_hi = 97, typ_hi = 100, max_hi = 107;
reg clk;
always
    begin
        #(95:100:105) clk = 1;
        #(min_hi:typ_hi:max_hi) clk = 0;
    end
```

Example 6-3: Delay controls in procedural statements

The delay used during simulation will be one of the three—either minimum, typical, or maximum. One delay choice is used throughout a simulation run; it cannot be changed dynamically.

Selection of which delays will be used is done using one of three command options. The `+maxdelays` option selects all of the maximum delays; the `+typdelays` option selects all of the typical delays; the `+mindelays` option selects all of the minimum delays. For example, the following command line runs Verilog-XL with only the values specified for the maximum delay:

```
verilog source1.v +maxdelays
```

Please note: If only one delay is specified, then Verilog-XL uses it regardless of whether minimum, typical, or maximum delays are selected. If more than one delay is desired, then all three delays must be specified; for example, it is not possible to specify minimum and maximum without typical.

CAUTION

There is currently no syntax checking on plus command options. Be very careful in specifying them to avoid confusing results. If you misspell “`maxdelays`”, “`mindelays`” or “`typdelays`”, the option will be ignored.

6.16.2

triereg Net Charge Decay

Like all nets, a `triereg` declaration's delay specification can contain up to three delays. The first two delays specify the simulation time that elapses in a transition to the 1 and 0 logic states when the `triereg` is driven to these states by a driver. The third delay specifies the charge decay time instead of the time that elapses in a transition to the z logic state. The charge decay time specifies the simulation time that elapses between when a `triereg`'s drivers turn off and when its stored charge can no longer be determined.

A `triereg` needs no turn-off delay specification because a `triereg` never makes a transition to the z logic state. When a `triereg`'s drivers make transitions from the 1, 0, or x logic states to off, the `triereg` retains the previous 1, 0, or x logic state that was on its drivers. The z value does not propagate from a `triereg`'s drivers to a `triereg`. A `triereg` can only hold a z logic state when z is the `triereg`'s initial logic state or when it is forced to the z state with a `force` statement.

A delay specification for charge decay models a charge storage node that is not ideal, a charge storage node whose charge leaks out through its surrounding devices and connections.

This section describes the charge decay process and the delay specification for charge decay.

The charge decay process

Charge decay is the cause of transition of a 1 or 0 that is stored in a `triereg` to an unknown value (x) after a specified number of time units. The charge decay time is that specified number of time units.

The charge decay process begins when the `triereg`'s drivers turn off and the `triereg` starts to hold charge. The charge decay process ends under the following two conditions:

1. The specified number of time units elapse and the `triereg` makes a transition from 1 or 0 to x.
2. The `triereg`'s drivers turn on and propagate a 1, 0 or x into the `triereg`.

When charge decay causes a `triereg`'s value to change to x, Verilog-XL issues a warning message. This message takes the following form:

```
Warning! Time = simulation_time: Charge on node hierarchical_name_of_triereg has
decayed                               [Verilog-DECAY]
"source_file_name", line_number: triereg_identifier
```

You can tell Verilog-XL not to issue this warning with the `$disable_warnings` system task.

The delay specification for charge decay time

The third delay in a `triereg` declaration specifies the charge decay time. A three-valued delay specification in a `triereg` declaration has the following form:

```
#(d1, d2, d3)
// three delays -
// (rising_delay, falling_delay, charge_decay_time)
```

The specification in a `triereg` declaration of the charge decay time must be preceded by a rise and fall delay specification. The following example shows a specification of the charge decay time in a `triereg` declaration:

```
triereg (large) #(0,0,50) cap1;
```

This example declares a `triereg` with the identifier `cap1`. This `triereg` stores a `large` charge. The delay specifications for the rise delay is 0, the fall delay is 0, and the charge decay time specification is 50 time units.

Please note: A charge decay time is not a propagation delay like a rising delay or a falling delay. A charge decay time greater than 0 does not prevent the acceleration of the `triereg`.

Example 6-4 presents a source description file that contains a `triereg` declaration with a charge decay time specification. Figure 6-25 assists you in reading the source description file.

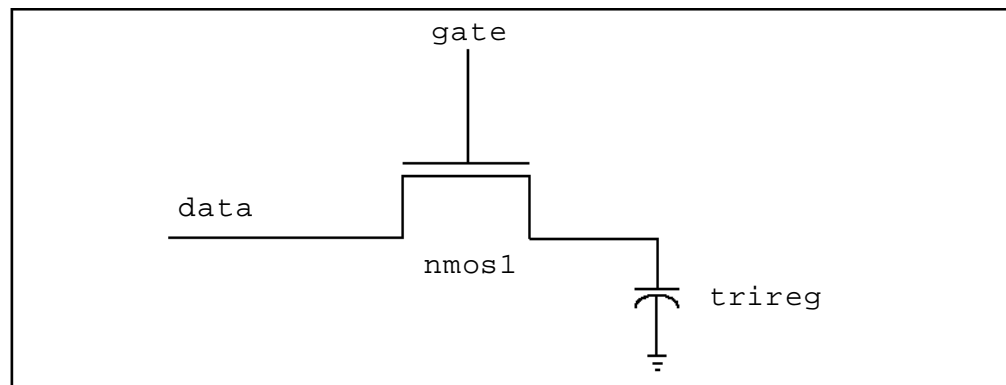


Figure 6-25: This figure accompanies the example below

```
module capacitor;
reg data,gate;

triereg (large) #(0,0,50) cap1;

nmos nmos1 (cap1,data,gate);

initial
begin
    $monitor("%0d data = %v gate = %v cap1 = %v",
            $time,data,gate,cap1);
    data = 1;
    gate = 1;
    #10 gate = 0;
    #30 gate = 1;
    #10 gate = 0;
    #100 $finish;
end

endmodule
```

triereg declaration with a charge decay time of 50 time units

nmos switch that drives the triereg

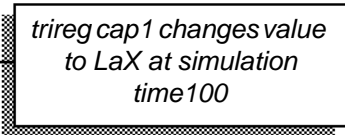
toggles the driver of the control input to the nmos switch

Example 6-4: triereg with a charge decay

Example 6-5 shows the simulation results of the model in Example 6-4.

```
0 data = St1 gate = St1 cap1 = St1
10 data = St1 gate = St0 cap1 = La1
40 data = St1 gate = St1 cap1 = St1
50 data = St1 gate = St0 cap1 = La1
```

```
Warning! Time = 100: Charge on node capacitor.cap1 has
    decayed                                     [Verilog-DECAY]
    "trireg1.v", 4: cap1
100 data = St1 gate = St0 cap1 = LaX
```



*trireg cap1 changes value
to LaX at simulation
time 100*

Example 6-5: Charge decay simulation results

The results show the following sequence of events:

1. At simulation time 0, data drives a strong 1 into trireg cap1.
2. At simulation time 10, gate's value changes to 0, disconnecting trireg cap1 from data; trireg cap1 enters the capacitive state, storing its value of 1 with a large strength. The charge decay process begins for trireg cap1; its value is scheduled to change to x at simulation time 60.
3. At simulation time 40, gate's value changes to 1, connecting trireg cap1 to data; trireg cap1 enters the driven state, and data drives a strong 1 into trireg cap1. The charge decay process stops for trireg cap1 because it is no longer in the capacitive state.
4. At simulation time 50, reg gate's value changes to 0, disconnecting trireg cap1 from reg data again; trireg cap1 enters the capacitive state, storing its value of 1 with a large strength. The charge decay process begins again for trireg cap1; its value is scheduled to change to x at simulation time 100.
5. At simulation time 100, the charge decay process changes the stored value in trireg cap1 from 1 to x.

Please note: Specifying a charge decay time can affect performance. You may see a performance degradation caused by specifying trireg charge decay time in a design—such as a dynamic circuit, whose triregs frequently enter the capacitive state.

6.17

Gate and Net Name Removal

Four compiler directives have been provided that control the removal of gate and/or net names in order to reduce the virtual memory requirements at the gate and switch level. The names are removed from the second and all subsequent module instances so that removing gate and net names saves the most memory in designs containing gate-level modules that are instantiated many times.

The compiler directives are the following:

```
`remove_gatenames  
`noremove_gatenames  
`remove_netnames  
`noremove_netnames
```

The first two directives control the removal of gate names, and the latter two control the removal of net names. For both controls, the default is to NOT remove the names.

These directives can only be specified outside modules. The control applies to all modules following a directive until the end of the source description (going across source files if necessary) or until another of these directives is given or until a ``resetall` directive is given. Any number of these compiler directives can be given in a source description.

The removal of gate names is more useful than the removal of net names because gate names at the present are only used for the tracing of value changes across the gates.

Net names cannot be removed if they have been referenced in a hierarchical name. An example of a hierarchical referencing would be a monitoring task, or nets that will need to be referenced interactively.

As shown in the following partial description, all gate names from modules a and b, and net names from all the instances of module b are removed.

```
      •
      •
      •

'remove_gatenames
module a;

      •
      •
      •

      b b1(), b2(), b3();

      •
      •
      •

endmodule

'remove_netnames
module b;

      •
      •
      •

      c c1(), c2();

      •
      •
      •

endmodule

'noremove_gatenames
'noremove_netnames
module c;

      •
      •
      •

endmodule
```

Example 6-6: Gate and net name removal

Note that it is not possible to selectively remove the gate and/or net names from particular instances of a module.

7

Figure 7-0
Example 7-0
Syntax 7-0
Table 7-0

User-Defined Primitives (UDPs)

This chapter describes a modeling technique whereby the user can effectively augment the set of predefined gate primitives by designing and specifying new primitive elements called user-defined primitives (UDPs). Instances of these new UDPs can then be used in exactly the same manner as the gate primitives to represent the circuit being modeled. This technique can reduce the amount of memory that a description needs and can improve simulation performance. Evaluation of these UDPs is accelerated by the Verilog-XL algorithm.

The following two types of behavior can be represented in a user-defined primitive:

- *combinational*—modeled by a combinational UDP
- *sequential*—modeled by a sequential UDP

A sequential UDP uses the value of its inputs and the current value of its output to determine the next value of its output. Sequential UDPs provide an easy and efficient way to model sequential circuits such as flip-flops and latches. A sequential UDP can model both level-sensitive and edge-sensitive behavior.

The maximum number of inputs to a combinational UDP is ten. The maximum number of inputs to a sequential UDP is limited to nine because the internal state counts as an input. Each UDP has exactly one output, which can be in one of three states: 0, 1, or x. The tri-state value z is not supported. In sequential UDPs, the output always has the same value as the internal state.

7.1

Memory Usage and Performance Considerations

The user should be aware of the amount of memory required for the internal tables created for evaluation of these UDPs during simulation. Although only one such table is required per UDP definition, and not for each instance, the UDPs with 8, 9, and 10 inputs do consume a large amount of memory. The trade-off here is one of speed versus memory. If many instances of a large UDP are needed, then it is easily possible to gain back the memory used by the definition, because each UDP instance can take less memory than that required for the group of gates it replaces.

The memory required for a UDP definition is given below:

Number of variables	Memory required (K bytes)
1-5	<1
6	5
7	17
8	56
9	187
10	623

Table 7-1: UDP memory requirements

Note that the number of variables is the number of inputs for combinational UDPs and the number of inputs plus one for sequential UDPs.

7.2 Syntax

The formal syntax of the UDP definition is as follows:

```

<UDP>
    ::= primitive <name_of_UDP> ( <output_terminal_name>,
        <input_terminal_name> <,<input_terminal_name>>* ) ;
        <UDP_declaration>+
        <UDP_initial_statement>?
        <table_definition>
        endprimitive
<name_of_UDP>
    ::= <IDENTIFIER>
<UDP_declaration>
    ::= <UDP_output_declaration>
        || = <reg_declaration>
        || = <UDP_input_declaration>
<UDP_output_declaration>
    ::= output <output_terminal_name>;
<reg_declaration>
    reg <output_terminal_name> ;
<UDP_input_declaration>
    ::= input <input_terminal_name>
        <,<input_terminal_name>>* ) ;
<UDP_initial_statement>
    ::= initial <output_terminal_name> = <init_val> ;
<init_val>
    ::= 1'b0
        || = 1'b1
        || = 1'bx
        || = 1
        || = 0
<table_definition>
    ::= table
        <table_entries>
        endtable
<table_entries>
    ::= <combinational_entry>+
        || = <sequential_entry>+

```

—continued

Syntax 7-1: Syntax for user-defined primitives

```
<combinational_entry>
    ::= <level_input_list> : <OUTPUT_SYMBOL> ;
<sequential_entry>
    ::= <input_list> : <state> : <next_state> ;
<input_list>
    ::= <level_input_list>
    || = <edge_input_list>
<level_input_list>
    ::= <LEVEL_SYMBOL>+
<edge_input_list>
    ::= <LEVEL_SYMBOL>* <edge> <LEVEL_SYMBOL>*
<edge>
    ::= ( <LEVEL_SYMBOL> <LEVEL_SYMBOL> )
    || = <EDGE_SYMBOL>
<state>
    ::= <LEVEL_SYMBOL>
<next_state>
    ::= <OUTPUT_SYMBOL>
    || = - (This is a literal hyphen—
           see Section 7.15 for more details.)

Lexical tokens:
<OUTPUT_SYMBOL> is one of the following:
    0 1 x X
<LEVEL_SYMBOL> is one of the following:
    0 1 x X ? b B
<EDGE_SYMBOL> is one of the following:
    r R f F p P n N *
```

Syntax 7-1 continued: Syntax for user-defined primitives

7.3 UDP Definition

UDP definitions are independent of modules; they are at the same level as module definitions in the syntax hierarchy. They can appear anywhere in the source text, either before or after they are used inside a module. They **may not** appear between the keywords `module` and `endmodule`.

A UDP definition begins with the keyword `primitive`. This is followed by an identifier, which is the name of the UDP. This in turn is followed by a comma separated list of terminals enclosed in parentheses, which is followed by a semicolon.

The UDP definition header described previously is followed by terminal declarations and a state table. The UDP definition is terminated by the keyword `endprimitive`.

7.3.1 UDP Terminals

UDPs have multiple input terminals and exactly one output terminal; they cannot have bidirectional inout terminals.

The output terminal **MUST** be the first terminal in the terminal list.

All UDP terminals are scalar. No vector terminals are allowed.

The output terminal of a sequential UDP requires an additional declaration as type `reg`. It is illegal to declare a `reg` for the output terminal of a combinational UDP.

7.3.2 UDP Declarations

UDPs must contain input and output terminal declarations. The output terminal declaration begins with the keyword `output`, followed by one output terminal name. The input terminal declaration begins with the keyword `input`, followed by one or more input terminal names.

Sequential UDPs must contain a `reg` declaration for the output terminal. Combinational UDPs cannot contain a `reg` declaration. The initial value of the output terminal `reg` can be specified in an `initial` statement in a sequential UDP.

7.3.3 Sequential UDP initial Statement

The sequential UDP `initial` statement specifies the value of the output terminal when simulation begins. This statement begins with the keyword `initial`. The statement that follows must be an assignment statement that assigns a single bit literal value to the output terminal `reg`.

7.3.4 UDP State Table

The state table which defines the behavior of a UDP begins with the keyword `table` and is terminated with the keyword `endtable`.

Each row of the table is created using a variety of characters that indicate input and output states. Three states—0, 1, and x—are supported. The z state is explicitly excluded from consideration in

user-defined primitives. A number of special characters are defined to represent certain combinations of state possibilities. These are detailed in this chapter, in Section 7.10, *Symbols to Enhance Readability*.

The order of the input state fields of each row of the state table is taken directly from the terminal list in the UDP definition header. It is NOT related to the order of the input declarations.

Combinational UDPs have one field per input and one field for the output. The input fields are separated from the output field by a colon.

Sequential UDPs have an additional field inserted between the input fields and the output field. This additional field represents the current state of the UDP and is considered equivalent to the current output value. It is delimited by colons.

Each row defines the output for a particular combination of input states. If all inputs are specified as *x*, then the output must be specified as *x*. All combinations that are not explicitly specified result in a default output state of *x*. Each row of the table is terminated by a semicolon.

Consider the following entry from a UDP state table:

```
0      1      : ? :      1      ;
```

In this entry the *?* represents a don't-care condition—it is replaced by cases of the entry when the *?* is replaced by 1, 0, and *x*. This specifies that when the inputs are 0 and 1, no matter what the value of the current state, the output is 1.

It is not necessary to explicitly specify every possible input combination. All combinations that are not explicitly specified result in a default output state of *x*.

It is illegal to have the same combination of inputs, including edges, specified for different outputs.

7.4 Combinational UDPs

In combinational UDPs, the output state is determined solely as a function of the current input states. Whenever an input changes state, the UDP is evaluated and one of the state table rows is matched. The output state is set to the value indicated by that row.

Consider the following example, which defines a multiplexer with two data inputs, a control input. Remember, there can only be a single output.

```

primitive multiplexer(mux, control, dataA, dataB ) ;
    output mux ;
    input control, dataA, dataB ;
    table
        // control dataA dataB  mux
            0      1      0  :  1  ;
            0      1      1  :  1  ;
            0      1      x  :  1  ;
            0      0      0  :  0  ;
            0      0      1  :  0  ;
            0      0      x  :  0  ;

            1      0      1  :  1  ;
            1      1      1  :  1  ;
            1      x      1  :  1  ;
            1      0      0  :  0  ;
            1      1      0  :  0  ;
            1      x      0  :  0  ;

            x      0      0  :  0  ;
            x      1      1  :  1  ;

    endtable
endprimitive

```

Example 7-1: Combinational form of user-defined primitive

The first entry in the table above can be explained as follows: when control equals 0 and dataA equals 1 and dataB equals 0, then output mux equals 1.

All combinations of the inputs that are not explicitly specified will drive the output to the unknown value x. For example, in the table for multiplexer above (Example 7-1), the input combination 0xx(control=0, dataA=x, dataB=x) is not specified. If this combination occurs during simulation, the value of output mux will become x.

To improve the readability, and to ease writing of the table, several special symbols are provided. A ? represents iteration of the table entry over the values 0, 1, and x — a ? generates cases of that entry where the ? is replaced by a 0, 1, or x. It represents a don't-care condition on that input. Using ?, the description of a multiplexer given in Example 7-1 can be abbreviated as implemented in Example 7-2.

```

primitive multiplexer(mux,control,dataA,dataB ) ;
  output mux ;
  input control, dataA, dataB ;
  table
    // control dataA dataB mux
        0      1      ?  : 1  ;    // ? = 0,1,x
        0      0      ?  : 0  ;
        1      ?      1  : 1  ;
        1      ?      0  : 0  ;

        x      0      0  : 0  ;
        x      1      1  : 1  ;
  endtable
endprimitive

```

Example 7-2: Special symbols in user-defined primitive

7.5 Level-Sensitive Sequential UDPs

Level-sensitive sequential behavior is represented the same way as combinational behavior, except that the output is declared to be of type `reg`, and there is an additional field in each table entry. This new field represents the current state of the UDP.

The output field in a sequential UDP represents the next state.

Consider the example of a latch in Example 7-3.

```

primitive latch(q, clock, data) ;
  output q; reg q ;
  input clock, data;
  table
    // clock data  q    q+
        0      1  : ? :  1  ;
        0      0  : ? :  0  ;
        1      ?  : ? :  -   ; // - = no change
  endtable
endprimitive

```

Example 7-3: UDP for a latch

This description differs from a combinational UDP model in two ways. First, the output identifier `q` has an additional `reg` declaration to indicate that there is an internal state `q`. The output value of the UDP is always the same as the internal state. Second, a field for the current state, which is separated by colons from the inputs and the output, has been added.

7.6 Edge-Sensitive UDPs

In level-sensitive behavior, the values of the inputs and the current state are sufficient to determine the output value. Edge sensitive behavior differs in that changes in the output are triggered by specific transitions of the inputs. This makes the state table a transition table as illustrated in Example 7-4.

```
primitive d_edge_ff(q, clock, data);
output q; reg q;
input clock, data;

table
// obtain output on rising edge of clock
// clock  data  q      q+
(01)     0   :  ?   :  0  ;
(01)     1   :  ?   :  1  ;
(0?)     1   :  1   :  1  ;
(0?)     0   :  0   :  0  ;
// ignore negative edge of clock
(?0)     ?   :  ?   :  -  ;
// ignore data changes on steady clock
?        (??) :  ?   :  -  ;
endtable
endprimitive
```

Example 7-4: UDP for an edge-sensitive D-type flip-flop

Example 7-4 has terms like (01) in the input fields. These terms represent transitions of the input values. Specifically, (01) represents a transition from 0 to 1. The first line in the table of the previous UDP

definition (Example 7-4) can be interpreted as follows: when clock changes value from 0 to 1 and data equals 0, the output goes to 0 no matter what the current state.

Please note: Each table entry can have a transition specification on, at most, one input. Entries such as the one shown below are illegal:

(01)(01)0 : 0 : 1

As in the combinational and the level-sensitive entries, a ? implies iteration of the entry over the values 0, 1, and x. A dash (-) in the output column indicates no value change.

All unspecified transitions default to the output value x. Thus, in the previous example, transition of clock from 0 to x with data equal to 0 and current state equal to 1 will result in the output q going to x.

All transitions that should not affect the output **must** be explicitly specified. Otherwise, they will cause the value of the output to change to x. If the UDP is sensitive to edges of any input, the desired output state must be specified for *all* edges of *all* inputs.

7.7 Sequential UDP Initialization

The value on the output terminal of a sequential UDP can be specified with an `initial` statement that contains a procedural assignment statement. The `initial` statement is optional.

Like `initial` statements in modules, the `initial` statement in UDPs begin with the keyword `initial`. The valid contents of `initial` statements in UDPs and the valid left and right-hand sides of their procedural assignment statements differ from `initial` statements in modules. The difference between these two types of `initial` statements is described in Table 7-2.

initial statements in UDPs	initial statements in modules
contents limited to one procedural assignment statement	contents can be one procedural statement of any type or a block statement that contains more than one procedural statement
the procedural assignment statement must assign a value to a reg whose identifier matches the identifier of an output terminal	procedural assignment statements in initial statements can assign values to a reg whose identifier does not match the identifier of an output terminal
the procedural assignment statement must assign one of the following values: 1'b1 1'b0 1'bx 1 0	procedural assignment statements can assign values of any size, radix, and value

Table 7-2: Initial statements in UDPs and modules

Example 7-5 shows a sequential UDP that contains an initial statement.

```

primitive srff (q,s,r);
output q;
input s,r;
reg q;
initial q = 1'b1;
table
//  s  r   q   q+
   1  0 : ? : 1 ;
   f  0 : 1 : - ;
   0  r : ? : 0 ;
   0  f : 0 : - ;
   1  1 : ? : 0 ;
endtable
endprimitive

```

sequential UDP initial statement specifies that output terminal q has a value of 1 at the start of the simulation

Example 7-5: Sequential UDP initial statement

User-Defined Primitives (UDPs)

Sequential UDP Initialization

In Example 7-5, the output *q* has an initial value of 1 at the start of the simulation; a delay specification in the UDP instance does not delay the simulation time of the assignment of this initial value to the output. When simulation starts, this value is the current state in the state table.

Please note: Verilog-XL does not have an initialization or power-up phase. The initial value on the output to a sequential UDP does not propagate to the design output before simulation starts. All nets in the fanout of the output of a sequential UDP begin with a value of *x* even when that output has an initial value of 1 or 0.

The following example and figure show how values are applied in a module that instantiates a sequential UDP with an initial statement. Example 7-6 shows the source description for the module and UDP.

```
primitive dff1 (q,clk,d);
input clk,d;
output q;
reg q;
initial
    q = 1'b1;

table
// clkd    q    q+
p  0  :  ?  :  0  ;
p  1  :  ?  :  1  ;
n  ?  :  ?  :  -  ;
?  *  :  ?  :  -  ;
endtable
endprimitive

module dff (q,qb,clk,d);
input clk,d;
output q,qb;
    dff1  g1 (qi,clk,d);
    buf #3 g2 (q,qi);
    not #5 g3 (qb,qi);
endmodule
```

initial statement

UDP instance output is qi

q and qb are in the fanout of qi

Example 7-6: Instance of a sequential UDP with an initial statement

In Example 7-6, UDP dff1 contains an initial statement that sets the initial value of its output to 1. Module dff contains an instance of UDP dff1. In this instance, the UDP output is *qi*; the output's fanout includes nets *q* and *qb*.

Figure 7-1 shows the schematic of the module in Example 7-6 and the simulation times of the propagation of the initial value of the output of the UDP.

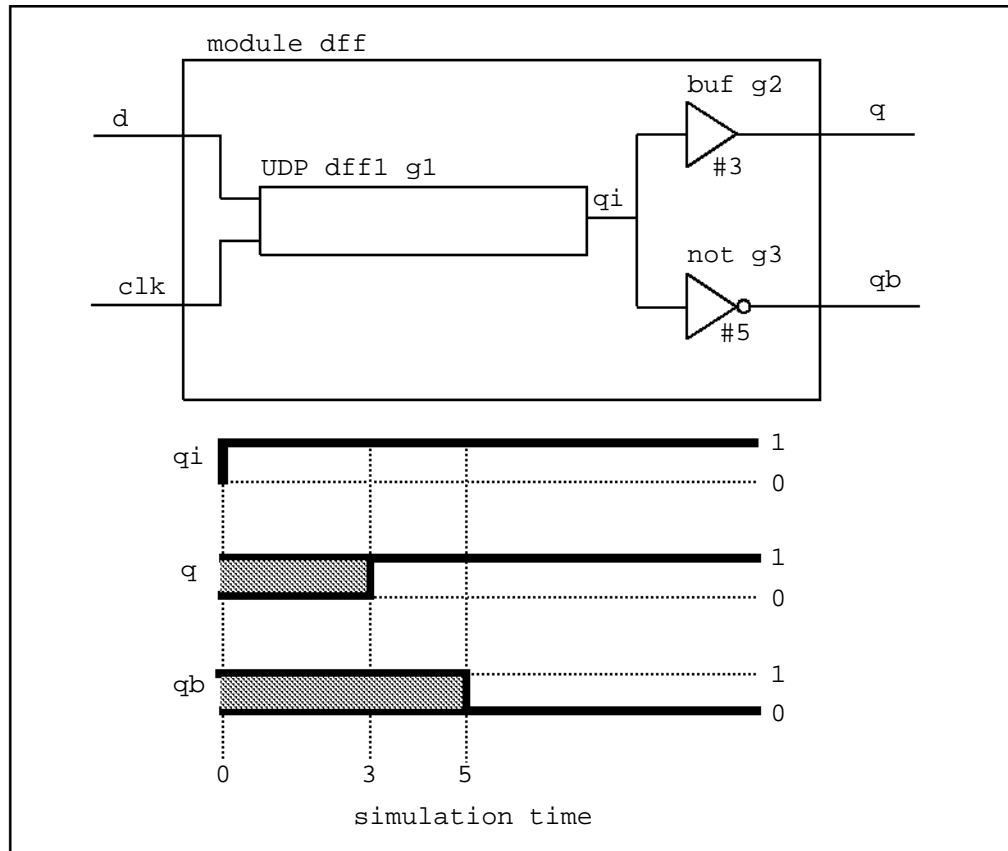


Figure 7-1: Module schematic and the simulation times of initial value propagation

In Figure 7-1, the fanout from the UDP output *qi* includes nets *q* and *qb*. At simulation time 0, *qi* changes value to 1. That initial value of *qi* does not propagate to net *q* until simulation time 3, and does not propagate to net *qb* until simulation time 5.

7.8 UDP Instances

Instances of user-defined primitives are specified inside modules in the same manner as for gates. The instance name is optional, just as for gates. The terminal order is as specified in the UDP definition. Only two delays can be specified, because *z* is not supported for UDPs.

The system can generate names for unnamed instances of UDPs. See Section 12.6 for more information on automatic naming.

Example 7-7 creates an instance of the D-type flip-flop *d_edge_ff* (defined in Example 7-4).

```
module flip;
    reg clock , data ;
    parameter p1 = 10 ;
    parameter p2 = 33;
    d_edge_ff #(5,7) d_inst( q, clock, data);
initial
begin
    data = 1;
    clock = 1;
end
always #p1 clock = ~clock;
always #p2 data = ~data;
endmodule
```

Example 7-7: UPD for a D-type flip-flop

7.9 Compilation

Several checks are applied to user-defined primitive definitions as they are compiled.

The table entries are checked for consistency. This means that if two entries specify different outputs for the same combination of inputs, including edges, an error will result. Special care should be taken when using the *?*, *b*, ***, *p*, and *n* symbols which are described in the next section.

The table entries are checked for redundancy. If two or more table entries specify the same output for the same combination of inputs, including edges, a warning will result. The message indicates the entry that duplicates what is specified in previous lines.

7.10 Symbols to Enhance Readability

Like `?`, there are several symbols that can be used in UDP definitions to make the description more readable. The symbols described in Table 7-3 are used in Example 7-8.

Symbol	Interpretation	Explanation
<code>b</code>	0 or 1	like <code>?</code> , except x is excluded
<code>r</code>	(01)	rising edge on an input
<code>f</code>	(10)	falling edge on an input
<code>p</code>	(01) or (0x) or (x1) or (1z) or (z1)	rising edges, including unknown
<code>n</code>	(10) or (1x) or (x0) or (0z) or (z0)	falling edges, including unknown
<code>*</code>	(??)	all transitions

Table 7-3: Symbols for readability

7.11 Mixing Level-Sensitive and Edge-Sensitive Descriptions

UDP definitions allow a mixing of the level-sensitive and the edge-sensitive constructs in the same description. An edge-triggered JK flip-flop with asynchronous preset and clear needs such a mixture. Example 7-8 illustrates this concept.

```

primitive jk_edge_ff(q, clock, j, k, preset, clear);
  output q; reg q;
  input clock, j, k, preset, clear;

  table
    //clock jk pc state output/next state
    ?  ?? 01 : ? : 1 ; //preset logic
    ?  ?? *1 : 1 : 1 ;
    ?  ?? 10 : ? : 0 ; //clear logic
    ?  ?? 1* : 0 : 0 ;

    r  00 00 : 0 : 1 ; //normal clocking cases
    r  00 11 : ? : - ;
    r  01 11 : ? : 0 ;
    r  10 11 : ? : 1 ;
    r  11 11 : 0 : 1 ;
    r  11 11 : 1 : 0 ;
    f  ?? ?? : ? : - ;

    b  *? ?? : ? : - ; //j and k transition cases
    b  ?* ?? : ? : - ;

  endtable
endprimitive

```

Example 7-8: Sequential UDP for level-sensitive and edge-sensitive behavior

In this example, the preset and clear logic is level-sensitive. Whenever the preset and clear combination is 01, the output has value 1. Similarly, whenever the preset and clear combination has value 10, the output has value 0.

The remaining logic is sensitive to edges of the clock. In the normal clocking cases, the flip-flop is sensitive to the rising clock edge as indicated by an *r* in the clock field in those entries. The insensitivity to the falling edge of clock is indicated by a hyphen (-) in the output field (see Section 7.15) for the entry with an *f* as the value of clock. Remember that the desired output for this input transition must be specified to

avoid unwanted x values at the output. The last two entries show that the transitions in j and k inputs do not change the output on a steady low or high clock.

7.12 Reducing Pessimism

Three-valued logic tends to make pessimistic estimates of the output when one or more inputs are unknown. User-defined primitives can be used to reduce this pessimism. The following is an extension of the previous latch example illustrating reduction of pessimism.

```

primitive latch(q, clock, data)
    output q; reg q ;
    input clock, data ;
    table
//      clock data state output/next state
        0     1 : ? : 1 ;
        0     0 : ? : 0 ;
        1     ? : ? : - ; // - = no change
//      ignore x on clock when data equals state
        x     0 : 0 : - ;
        x     1 : 1 : - ;
    endtable
endprimitive

```

Example 7-9: Latch UDP illustrating pessimism

The last two entries specify what happens when the clock input has value x. If these are omitted, the output will go to x whenever the clock is x. This is a pessimistic model, as the latch should not change its output if it is already 0 and the data input is 0. Similar analysis is true for the situation when the data input is 1 and the current output is 1.

Consider the jk flip-flop with preset and clear in Example 7-10.

```

primitive jk_edge_ff(q, clock, j, k, preset, clear);
    output q; reg q;
    input clock, j, k, preset, clear;

    table
    //clock jk pc state output/next state
    //preset logic
        ?  ?? 01 : ? : 1 ;
        ?  ?? *1 : 1 : 1 ;
    //clear logic
        ?  ?? 10 : ? : 0 ;
        ?  ?? 1* : 0 : 0 ;
    //normal clocking cases
        r  00 00 : 0 : 1 ;
        r  00 11 : ? : - ;
        r  01 11 : ? : 0 ;
        r  10 11 : ? : 1 ;
        r  11 11 : 0 : 1 ;
        r  11 11 : 1 : 0 ;
        f  ?? ?? : ? : - ;

    //j and k cases
        b  *? ?? : ? : - ;
        b  ?* ?? : ? : - ;

    //cases reducing pessimism
        p  00 11 : ? : - ;
        p  0? 1? : 0 : - ;
        p  ?0 ?1 : 1 : - ;
        (?0)?? ?? : ? : - ;
        (1x)00 11 : ? : - ;
        (1x)0? 1? : 0 : - ;
        (1x)?0 ?1 : 1 : - ;
        x   *0 ?1 : 1 : - ;
        x   0* 1? : 0 : - ;
    endtable
endprimitive

```

Example 7-10: UDP for a JK flip-flop with preset and clear

This example has additional entries for the positive clock (p) edges, the negative clock edges (?0 and 1x), and with the clock value x. In all of these situations, the output is deduced to remain unchanged rather than going to x. Thus, this model is less pessimistic than the previous example.

7.13 Level-Sensitive Dominance

In the Verilog HDL, edge-sensitive cases are processed first, followed by level-sensitive cases. When level-sensitive and edge-sensitive cases specify different output values, the result is specified by the level-sensitive case. The following table shows level-sensitive and edge-sensitive entries in Example 7-10, their level-sensitive or edge-sensitive behavior, and a case that each includes.

entry	included case	behavior
? ?? 01: ?: 1;	0 00 01: 0: 1;	level-sensitive
f ?? ??: ?: -;	f 00 01: 0: 0;	edge-sensitive

Table 7-4: The level-sensitive and edge-sensitive entries in Example 7-10

The included cases specify opposite next state values for the same input and current state combination.

The level-sensitive included case specifies that when the inputs `clock`, `jk` and `pc` values are 0 00 01, and the current state is 0, the output changes to 1.

The edge-sensitive included case specifies that when `clock` falls from 1 to 0, and the other inputs `jk` and `pc` are 00 01, and the current state is 0, the output changes to 0.

When the edge-sensitive case is processed first, followed by the level-sensitive case, the output changes to 1.

7.14 Processing of Simultaneous Input Changes

When multiple UDP inputs change at the same simulation time the UDP will be evaluated multiple times, once per input value change. This situation cannot be detected by any form of table entry. This fact has important implications for modeling sequential circuits where the order of input changes and subsequent UDP evaluations can have a profound effect on the results of the simulation.

Consider the D-type flip-flop in Example 7-11.

```
primitive d_edge_ff(q, clock, data);
output q; reg q;
input clock, data;

table
// obtain output on rising edge of clock
// clock  data  q    q+
(01)     0   : ?   : 0   ;
(01)     1   : ?   : 1   ;
(0?)     1   : 1   : 1   ;
(0?)     0   : 0   : 0   ;
// ignore negative edge of clock
(?0)     ?   : ?   : -   ;
// ignore data changes on steady clock
?        (??) : ?   : -   ;
endtable
endprimitive
```

Example 7-11: D-type flip-flop

If the current state of the flip-flop is 0 and the clock and data inputs make transitions from 0 to 1 at the same simulation time, then the state of the output at the next simulation time is unpredictable because it cannot be predicted which of these transitions is processed first.

If the clock input transition is processed first and the data input transition is processed second, then the next state of the output will be 0. Likewise, if the data input transition is processed first and the clock transition is processed second, then the next state of the output will be 1.

This fact should be taken into consideration when constructing models. Keep in mind that gate-level models have the same sort of unpredictable behavior given particular input transition sequences; event-driven simulation is subject to idiosyncratic dependence on the order in which events are processed.

Timing checks can be used to detect simultaneous input transitions, provide a warning, and affect the simulation results; see Chapter 13, *Specify Blocks*.

7.15 Summary of Symbols

The following table summarizes the meaning of all the value symbols that are valid in the table part of a UDP definition.

Symbol	Interpretation	Notes
0	logic 0	
1	logic 1	
x	unknown	
?	iteration of 0, 1, and x	cannot be given in output field
b	iteration of 0 and 1	cannot be given in output field
-	no change	can only be given in the output field of a sequential UDP
(vw)	value change from v to w	v and w can be any one of 0, 1, x, ? or b
*	same as (??)	any value change on input
r	same as (01)	rising edge on input
f	same as (10)	falling edge on input
p	iteration of (01), (0x), and (x1)	potential positive edge on the input
n	iteration of (10), (1x), and (x0)	potential Negative edge on the input

Table 7-5: UDP table symbols

7.16 Examples

The following examples show UDP modeling for an and-or gate, a majority function for carry, and a 2-channel multiplexor with storage.

```
// Description of an AND-OR gate.
// out = (a1 & a2 & a3) | (b1 & b2).
primitive and_or(out, a1,a2,a3, b1,b2);
    output out;
    input a1,a2,a3, b1,b2;
    table
    //   a  b  : out ;
        111 ?? : 1  ;
        ??? 11 : 1  ;
        0?? 0? : 0  ;
        0?? ?0 : 0  ;
        ?0? 0? : 0  ;
        ?0? ?0 : 0  ;
        ??0 0? : 0  ;
        ??0 ?0 : 0  ;
    endtable
endprimitive
```

Example 7-12: UDP for an and-or gate

```
// Majority function for carry
// carryout = (a & b) | (a & carryin) | (b & carryin)
primitive carry(carryout, carryin, a, b);
    output carryout;
    input carryin, a, b;
    table
        0 00 : 0;
        0 01 : 0;
        0 10 : 0;
        0 11 : 1;
        1 00 : 0;
        1 01 : 1;
        1 10 : 1;
        1 11 : 1;
        // the following cases reduce pessimism
        0 0x : 0;
        0 x0 : 0;
        x 00 : 0;
        1 1x : 1;
        1 x1 : 1;
        x 11 : 1;
    endtable
endprimitive
```

Example 7-13: UDP for a majority function for carry

User-Defined Primitives (UDPs)

Examples

```
// Description of a 2-channel multiplexer with storage.
// The storage is level sensitive.
primitive mux_with_storage(out,clk,control,dataA,dataB);
    output out;
    reg out;
    input clk, control, dataA, dataB;

    table
    //clk control dataA dataB : current-state : next state ;
        1      0      1      ? :          ?          :      1      ;
        1      0      0      ? :          ?          :      0      ;
        1      1      ?      1 :          ?          :      1      ;
        1      1      ?      0 :          ?          :      0      ;
        1      x      0      0 :          ?          :      0      ;
        1      x      1      1 :          ?          :      1      ;
        0      ?      ?      ? :          ?          :      -      ;
        x      0      1      ? :          1          :      -      ;
        x      0      0      ? :          0          :      -      ;
        x      1      ?      1 :          1          :      -      ;
        x      1      ?      0 :          0          :      -      ;
    endtable
endprimitive
```

Example 7-14: UDP for a 2-channel multiplexor with storage

8

Figure 8-0
Example 8-0
Syntax 8-0
Table 8-0

Behavioral Modeling

The language constructs introduced so far allow hardware to be described at a relatively detailed level. Modeling a circuit with logic gates and continuous assignments reflects quite closely the logic structure of the circuit being modeled; however, these constructs do not provide the power of abstraction necessary for describing complex high level aspects of a system. The procedural constructs described in this chapter are well suited to tackling problems such as describing a microprocessor or implementing complex timing checks.

The chapter starts with a brief overview of a behavioral model to provide a context in which the reader can understand the many types of behavioral statements in Verilog. The behavioral constructs are then discussed in an order that allows us to introduce them before using them in examples.

The `+speedup` command line option can enhance the performance of behavioral code. See Section 24.4.36 for complete information on the `+speedup` command line option.

8.1 Behavioral Model Overview

Verilog behavioral models contain procedural statements that control the simulation and manipulate variables of the data types previously described. These statements are contained within procedures. Each procedure has an activity flow associated with it.

The activity starts at the control constructs `initial` and `always`. Each `initial` statement and each `always` statement starts a separate activity flow. All of the activity flows are concurrent, allowing the user to model the inherent concurrence of hardware.

Example 8-1 is a complete Verilog behavioral model.

```
module behave;
    reg [1:0]a,b;
    initial
        begin
            a = 'b1;
            b = 'b0;
        end
    always
        begin
            #50 a = ~a;
        end
    always
        begin
            #100 b = ~b;
        end
endmodule
```

Example 8-1: Simple example of behavioral modeling

During simulation of this model, all of the flows defined by the `initial` and `always` statements start together at simulation time zero. The `initial` statements execute once, and the `always` statements execute repetitively.

In this model, the register variables `a` and `b` initialize to binary 1 and 0 respectively at simulation time zero. The `initial` statement is then complete and does not execute again during this simulation run. This `initial` statement contains a `begin-end` block (also called a sequential block) of statements. In this `begin-end` block, `a` is initialized first, followed by `b`.

The `always` statements also start at time zero, but the values of the variables do not change until the times specified by the delay controls (introduced by `#`) have gone by. Thus, register `a` inverts after 50 time units, and register `b` inverts after 100 time units. Since the `always` statements repeat, this model produces two square waves. Register `a` toggles with a period of 100 time units, and register `b` toggles with a period of 200 time units. The two `always` statements proceed concurrently throughout the entire simulation run.

8.2

Procedural Assignments

As described in Chapter 5, *Assignments*, procedural assignments are for updating `reg`, `integer`, `time`, and memory variables.

There is a significant difference between procedural assignments and continuous assignments, as described below:

- Continuous assignments drive net variables and are evaluated and updated whenever an input operand changes value.
- Procedural assignments update the value of register variables under the control of the procedural flow constructs that surround them.

The right-hand side of a procedural assignment can be any expression that evaluates to a value. However, part-selects on the right-hand side must have constant indices. The left-hand side indicates the variable that receives the assignment from the right-hand side. The left-hand side of a procedural assignment can take one of the following forms:

- **register, integer, real, or time variable:**
An assignment to the name reference of one of these data types.
- **bit-select of a register, integer, real, or time variable:**
An assignment to a single bit that leaves the other bits untouched.
- **part-select of a register, integer, real, or time variable:**
A part-select of two or more contiguous bits that leaves the rest of the bits untouched. For the part-select form, only *constant* expressions are legal.
- **memory element:**
A single word of a memory. Note that bit-selects and part-selects are illegal on memory element references.
- **concatenation of any of the above:**
A concatenation of any of the previous four forms can be specified, which effectively partitions the result of the right-hand side expression and assigns the partition parts, in order, to the various parts of the concatenation.

Please note: Assignment to a register differs from assignment to a real, time, or integer variable when the right-hand side evaluates to fewer bits than the left-hand side. *Assignment to a register does not sign-extend*. Registers are unsigned; if you assign a register to an integer, real, or time variable, the variable will not sign-extend.

The Verilog HDL contains two types of procedural assignment statements:

- blocking procedural assignment statements
- non-blocking procedural assignment statements

Blocking and non-blocking procedural assignment statements specify different procedural flow in sequential blocks.

8.2.1

Blocking Procedural Assignments

A blocking procedural assignment statement must be executed before the execution of the statements that follow it in a sequential block (see Section 8.7.1). A blocking procedural assignment statement does not prevent the execution of statements that follow it in a parallel block (see Section 8.7.2).

Syntax:

The syntax for a blocking procedural assignment is as follows:

```
<lvalue> = <timing_control> <expression>
```

Where `lvalue` is a data type that is valid for a procedural assignment statement, `=` is the assignment operator, and `timing_control` is the optional intra-assignment delay. The `timing_control` delay can be either a delay control (for example, `#6`) or an event control (for example, `@(posedge clk)`). The expression is the right-hand side value the simulator assigns to the left-hand side.

The `=` assignment operator used by blocking procedural assignments is also used by procedural continuous assignments and continuous assignments.

Example 8-2 shows examples of blocking procedural assignments.

```
rega = 0;
rega[3] = 1;           // a bit-select
rega[3:5] = 7;         // a part-select
mema[address] = 8'hff; // assignment to a memory
                    // element
{carry, acc} = rega + regb; // a concatenation
```

Example 8-2: Examples of blocking procedural assignments

8.2.2

The Non-Blocking Procedural Assignment

The non-blocking procedural assignment allows you to schedule assignments without blocking the procedural flow. You can use the non-blocking procedural statement whenever you want to make several register assignments within the same time step without regard to order or dependance upon each other.

Syntax:

The syntax for a non-blocking procedural assignment is as follows:

```
<lvalue> <= <timing_control> <expression>
```

Where `lvalue` is a data type that is valid for a procedural assignment statement, `<=` is the non-blocking assignment operator, and `timing_control` is the optional intra-assignment timing control. The `timing_control` delay can be either a delay control (for example, `#6`) or an event control (for example, `@(posedge clk)`). The expression is the right-hand side value the simulator assigns to the left-hand side.

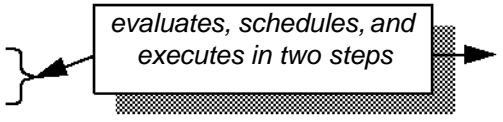
The non-blocking assignment operator is the same operator the simulator uses for the less-than-or-equal relational operator. The simulator interprets the `<=` operator to be a relational operator when you use it in an expression, and interprets the `<=` operator to be an assignment operator when you use it in a non-blocking procedural assignment construct.

How the simulator evaluates non-blocking procedural assignments

When the simulator encounters a non-blocking procedural assignment, the simulator evaluates and executes the non-blocking procedural assignment in two steps as follows:

1. The simulator evaluates the right-hand side and schedules the assignment of the new value to take place at a time specified by a procedural timing control.
2. At the end of the time step, in which the given delay has expired or the appropriate event has taken place, the simulator executes the assignment by assigning the value to the left-hand side.

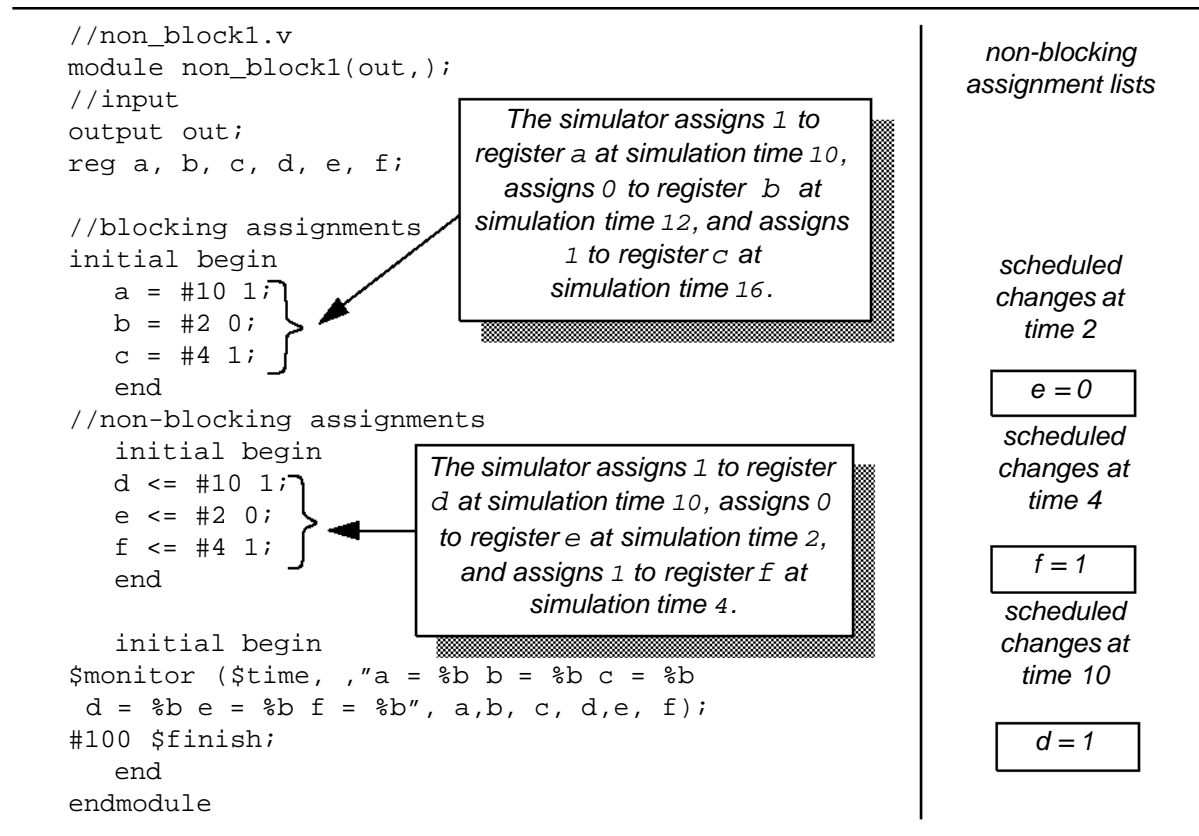
These two steps are shown in Example 8-3.

<pre> module evaluates2(out); output out; reg a, b, c; initial begin a = 0; b = 1; c = 0; end always c = #5 ~c; always @(posedge c) begin a <= b; b <= a; end endmodule </pre> 	<p><i>Step 1:</i> <i>non-blocking assignment scheduled changes at time 5</i></p> <p>The simulator evaluates the right-hand side of the non-blocking assignments and schedules the assignments of the new values at posedge c.</p> <p><i>a = 0</i> <i>b = 1</i></p> <p><i>Step 2:</i> <i>assignment values are:</i></p> <p>At posedge c, the simulator updates the left-hand side of each non-blocking assignment statement.</p> <p><i>a = 1</i> <i>b = 0</i></p>
---	---

Example 8-3: How the simulator evaluates non-blocking procedural assignments

At the end of the time step means that the non-blocking assignments are the last assignments executed in a time step—with one exception. Non-blocking assignment events can create blocking assignment events. The simulator processes these blocking assignment events after the scheduled non-blocking events.

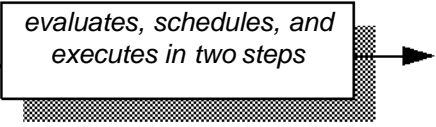
Unlike a regular event or delay control, the non-blocking assignment does not block the procedural flow. The non-blocking assignment evaluates and schedules the assignment, but does not block the execution of subsequent statements in a begin-end block, as shown in Example 8-4.



Example 8-4: Non-blocking assignments do not block execution of sequential statements

Please note: As shown in Example 8-5, the simulator evaluates and schedules assignments for the end of the current time step and can perform swapping operations with non-blocking procedural assignments.

```
//non_block1.v
module non_block1(out,);
//input
output out;
reg a, b;
initial begin
    a = 0;
    b = 1;
    a <= b;
    b <= a;
end
initial begin
    $monitor ($time, , "a = %b b = %b", a,b);
    #100 $finish;
end
endmodule
```



Step 1:

The simulator evaluates the right-hand side of the non-blocking assignments and schedules the assignments for the end of the current time step.

Step 2:

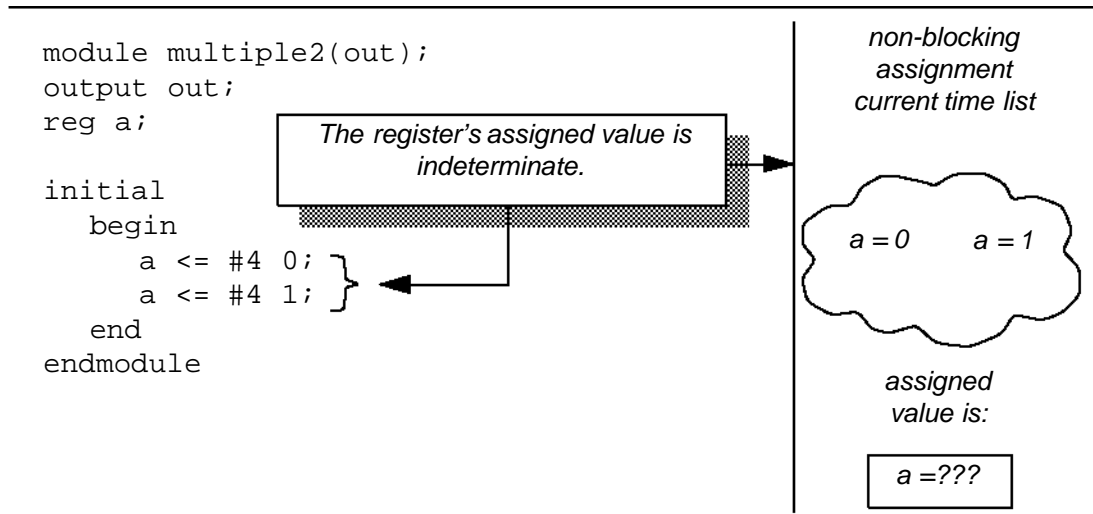
At the end of the current time step, the simulator updates the left-hand side of each non-blocking assignment statement.

assignment
values are:

$a = 1$
$b = 0$

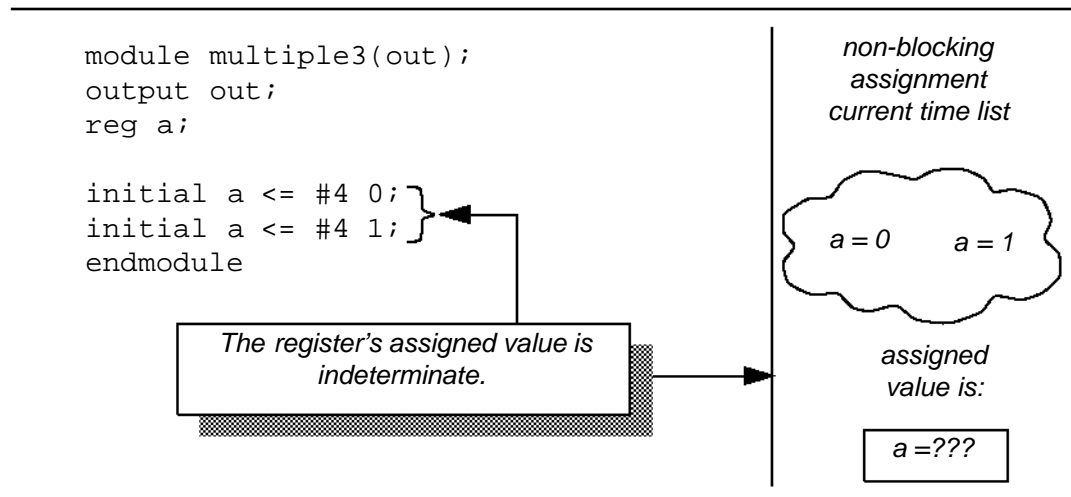
Example 8-5: Non-blocking procedural assignments used for swapping operations

When you schedule multiple non-blocking assignments to occur in the same register in a particular time slot, the simulator cannot guarantee the order in which it processes the assignments—the final value of the register is indeterminate. As shown in Example 8-6, the value of register *a* is not known until the end of time step 4.



Example 8-6: Multiple non-blocking assignments made in a single time step

If the simulator executes two procedural blocks concurrently, and these procedural blocks contain non-blocking assignment operators, the final value of the register is indeterminate. For example, in Example 8-7 the value of register *a* is indeterminate.



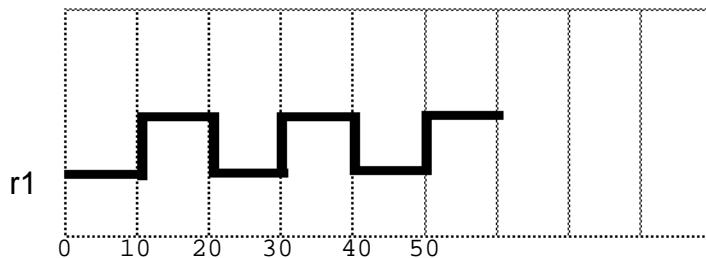
Example 8-7: Processing two procedural assignments concurrently

When multiple non-blocking assignments with *timing controls* are made to the same register, the assignments can be made without cancelling previous non-blocking assignments. In Example 8-8, the simulator evaluates the value of `i[0]` to `r1` and schedules the assignments to occur after each time delay.

```
module multiple;
reg r1;
reg [2:0] i;

initial
begin
// starts at time 0 doesn't hold the block
for (i = 0; i <= 5; i = i+1)
r1 <= # (i*10) i[0];
end
endmodule
```

Make the assignments to `r1` without
cancelling previous non-blocking
assignments.



scheduled changes at
time0

`r1 = 0`

scheduled changes at
time10

`r1 = 1`

scheduled changes at
time 20

`r1 = 0`

scheduled changes at
time 30

`r1 = 1`

scheduled changes at
time 40

`r1 = 0`

scheduled changes at
time 50

`r1 = 1`

Example 8-8: Multiple non-blocking assignments with timing controls

8.2.3

How the Simulator Processes Blocking and Non-Blocking Procedural Assignments

For each time slot during simulation, blocking and non-blocking procedural assignments are processed in the following way:

1. Evaluate the right-hand side of all assignment statements in the current time slot.
2. Execute all blocking procedural assignments and non-blocking procedural assignments that have no timing controls. At the same time, non-blocking procedural assignments with timing controls are set aside for processing.
3. Check for procedures that have timing controls and execute if timing control is set for the current time unit.
4. Advance the simulation clock.

8.3

Conditional Statement

The conditional statement (or *if-else* statement) is used to make a decision as to whether a statement is executed or not. Formally, the syntax is as follows:

```
<statement>
    ::= if ( <expression> ) <statement_or_null>
    | | = if ( <expression> ) <statement_or_null>
        else <statement_or_null>
<statement_or_null>
    ::= <statement>
    | | = ;
```

Syntax 8-1: Syntax of if statement

The <expression> is evaluated; if it is true (that is, has a non-zero known value), the first statement executes. If it is false (has a zero value or the value is x or z), the first statement does not execute. If there is an *else* statement and <expression> is false, the *else* statement executes.

Since the numeric value of the `if` expression is tested for being zero, certain shortcuts are possible. For example, the following two statements express the same logic:

```
if (expression)
```

```
if (expression != 0)
```

Because the `else` part of an `if-else` is optional, there can be confusion when an `else` is omitted from a nested `if` sequence. This is resolved by always associating the `else` with the closest previous `if` that lacks an `else`. In Example 8-9, the `else` goes with the inner `if`, as we have shown by indentation.

```
if (index > 0)
  if (rega > regb)
    result = rega;
  else      // else applies to preceding if
    result = regb;
```

Example 8-9: Association of `else` in nested `if`

If that association is not what you want, use a `begin-end` block statement to force the proper association, as shown in Example 8-10.

```
if (index > 0)
  begin
    if (rega > regb)
      result = rega;
    end
  else
    result = regb;
```

Example 8-10: Forcing correct association of `else` with `if`

Begin-end blocks left out inadvertently can change the logic behavior being expressed, as shown in Example 8-11.

```
if (index > 0)
  for (scani = 0; scani < index; scani = scani + 1)
    if (memory[scani] > 0)
      begin
        $display("...");
        memory[scani] = 0;
      end
    else /* WRONG */
      $display("error - index is zero");
```

Example 8-11: Erroneous association of else with if

The indentation in Example 8-11 shows unequivocally what you want, but the compiler does not get the message and associates the `else` with the inner `if`. This kind of bug can be very hard to find. (One way to find this kind of bug is to use the `$list` system task, which indents according to the logic of the description).

Notice that in Example 8-12, there is a semicolon after `result = rega`. This is because a `<statement>` follows the `if`, and a semicolon is an essential part of the syntax of a `<statement>`.

```
if (rega > regb)
  result = rega;
else
  result = regb;
```

Example 8-12: Use of semicolon in if statement

8.3.1

if-else-if Construct

The following construction occurs so often that it is worth a brief separate discussion.

```
if (<expression>)  
    <statement>  
else if (<expression>)  
    <statement>  
else if (<expression>)  
    <statement>  
else  
    <statement>
```

Syntax 8-2: Syntax of if-else-if construct

This sequence of if's (known as an if-else-if construct) is the most general way of writing a multi-way decision. The expressions are evaluated in order; if any expression is true, the statement associated with it is executed, and this terminates the whole chain. Each statement is either a single statement or a block of statements.

The last else part of the if-else-if construct handles the 'none of the above' or default case where none of the other conditions was satisfied. Sometimes there is no explicit action for the default; in that case, the trailing else can be omitted or it can be used for error checking to catch an impossible condition.

8.3.2

Example

The module fragment of Example 8-13 uses the if-else statement to test the variable `index` to decide whether one of three `modify_seg`n registers must be added to the memory address, and which increment is to be added to the `index` register. The first ten lines declare the registers and parameters.

```
// Declare registers and parameters
reg [31:0] instruction, segment_area[255:0];
reg [7:0] index;
reg [5:0] modify_seg1,
    modify_seg2,
    modify_seg3;
parameter
    segment1 = 0, inc_seg1 = 1,
    segment2 = 20, inc_seg2 = 2,
    segment3 = 64, inc_seg3 = 4,
    data = 128;

// Test the index variable
if (index < segment2)
    begin
        instruction = segment_area [index + modify_seg1];
        index = index + inc_seg1;
    end
else if (index < segment3)
    begin
        instruction = segment_area [index + modify_seg2];
        index = index + inc_seg2;
    end
else if (index < data)
    begin
        instruction = segment_area [index + modify_seg3];
        index = index + inc_seg3;
    end
else
    instruction = segment_area [index];
```

Example 8-13: Use of if-else-if construct

8.4 Case Statement

The `case` statement is a special multi-way decision statement that tests whether an expression matches one of a number of other expressions, and branches accordingly. The `case` statement is useful for describing, for example, the decoding of a microprocessor instruction. The `case` statement has the following syntax:

```
<statement>  
 ::= case ( <expression> ) <case_item>+ endcase  
    || = casez ( <expression> ) <case_item>+ endcase  
    || = casex ( <expression> ) <case_item>+ endcase  
<case_item>  
 ::= <expression> <,<expression>>* : <statement_or_null>  
    || = default : <statement_or_null>  
    || = default <statement_or_null>
```

Syntax 8-3: Syntax for case statement

The default statement is optional. Use of multiple default statements in one `case` statement is illegal syntax.

A simple example of the use of the case statement is the decoding of register `rega` to produce a value for `result`, as follows:

```

reg [15:0] rega;
reg [9:0] result;

    •
    •
    •

case (rega)
    16'd0: result = 10'b0111111111;
    16'd1: result = 10'b1011111111;
    16'd2: result = 10'b1101111111;
    16'd3: result = 10'b1110111111;
    16'd4: result = 10'b1111011111;
    16'd5: result = 10'b1111101111;
    16'd6: result = 10'b1111110111;
    16'd7: result = 10'b1111111011;
    16'd8: result = 10'b1111111101;
    16'd9: result = 10'b1111111110;
    default result = 'bx;
endcase

```

Example 8-14: Use of the case statement

The case expressions are evaluated and compared in the exact order in which they are given. During the linear search, if one of the case item expressions matches the expression in parentheses, then the statement associated with that case item is executed. If all comparisons fail, and the default item is given, then the default item statement is executed. If the default statement is not given, and all of the comparisons fail, then none of the case item statements is executed.

Apart from syntax, the case statement differs from the multi-way if-else-if construct in two important ways:

1. The conditional expressions in the if-else-if construct are more general than comparing one expression with several others, as in the case statement.
2. The case statement provides a definitive result when there are x and z values in an expression.

In a case comparison, the comparison only succeeds when each bit matches exactly with respect to the values 0, 1, x, and z. As a consequence, care is needed in specifying the expressions in the case statement. The bit length of all the expressions must be equal so that exact bit-wise matching can be performed. The length of all the case item expressions, as well as the controlling expression in the parentheses, will be made equal to the length of the longest <case_item> expression.

The most common mistake made here is to specify 'bx or 'bz instead of n'bx or n'bz, where n is the bit length of the expression in parentheses. The default length of x and z is the word size of the host machine, usually 32 bits.

The reason for providing a case comparison that handles the x and z values is that it provides a mechanism for detecting such values and reducing the pessimism that can be generated by their presence.

Example 8-15 illustrates the use of a case statement to properly handle x and z values.

```
case (select[1:2])
  2'b00: result = 0;
  2'b01: result = flaga;
  2'b0x,
  2'b0z: result = flaga ? 'bx : 0;
  2'b10: result = flagb;
  2'bx0,
  2'bz0: result = flagb ? 'bx : 0;
  default: result = 'bx;
endcase
```

Example 8-15: Detecting x and z values with the case statement

Example 8-15 contains a robust case statement used to trap x and z values. Notice that if select[1] is 0 and flaga is 0, then no matter what the value of select[2] is, the result is set to 0. The first, second, and third case items cause this assignment.

Example 8-16 shows another way to use a case statement to detect x and z values.

```
case(sig)
  1'bz:
    $display("signal is floating");
  1'bx:
    $display("signal is unknown");
  default:
    $display("signal is %b", sig);
endcase
```

Example 8-16: Another example of detecting x and z with case

8.4.1

Case Statement with Don't-Cares

Two other types of case statements are provided to allow handling of don't-care conditions in the case comparisons. One of these treats high-impedance values (z) as don't-cares, and the other treats both high-impedance and unknown (x) values as don't-cares.

These case statements are used in the same way as the traditional case statement, but they begin with new keywords—`casez` and `casex`, respectively.

Don't-care values (z values for `casez`, z and x values for `casex`), in any bit of either the case expression or the case items, are treated as don't-care conditions during the comparison, and that bit position is not considered.

Note that allowing don't-cares in the case items means that you can dynamically control which bits of the case expression are compared during simulation.

The syntax of literal numbers allows the use of the question mark (?) in place of z in these case statements. This provides a convenient format for specification of don't-care bits in case statements.

Example 8-17 is an example of the `casez` statement. It demonstrates an instruction decode, where values of the most significant bits select which task should be called. If the most significant bit of `ir` is a 1, then the task `instruction1` is called, regardless of the values of the other bits of `ir`.

```

reg [7:0] ir;

      •
      •
      •

casez (ir)
  8'b1??????: instruction1(ir);
  8'b01?????: instruction2(ir);
  8'b00010??? : instruction3(ir);
  8'b000001?? : instruction4(ir);
endcase

```

Example 8-17: Using the casez statement

Example 8-18 is an example of the `casex` statement. It demonstrates an extreme case of how don't-care conditions can be dynamically controlled during simulation. In this case, if `r = 8'b01100110`, then the task `stat2` is called.

```
reg [7:0] r, mask;

      •
      •
      •

mask = 8'bx0x0x0x0;
case (r ^ mask)
  8'b001100xx: stat1;
  8'b1100xx00: stat2;
  8'b00xx0011: stat3;
  8'bxx001100: stat4;
endcase
```

Example 8-18: Using the `case` statement

8.5 Looping Statements

There are four types of looping statements. They provide a means of controlling the execution of a statement zero, one, or more times.

- **forever** continuously executes a statement.
- **repeat** executes a statement a fixed number of times.
- **while** executes a statement until an expression becomes false. If the expression starts out false, the statement is not executed at all.
- **for** controls execution of its associated statement(s) by a three-step process, as follows:
 1. executes an assignment normally used to initialize a variable that controls the number of loops executed
 2. evaluates an expression—if the result is zero, the `for` loop exits, and if it is not zero, the `for` loop executes its associated statement(s) and then performs step 3
 3. executes an assignment normally used to modify the value of the loop-control variable, then repeats step 2

The following are the syntax rules for the looping statements:

```
<statement>
 ::= forever <statement>
 || =forever
      begin
          <statement>+
      end

<statement>
 ::= repeat ( <expression> ) <statement>
 || =repeat ( <expression> )
      begin
          <statement>+
      end

<statement>
 ::= while ( <expression> ) <statement>
 || =while ( <expression> )
      begin
          <statement>+
      end

<statement>
 ::= for ( <assignment> ; <expression> ; <assignment> )
          <statement>
 || =for ( <assignment> ; <expression> ; <assignment> )
      begin
          <statement>+
      end
```

Syntax 8-4: Syntax for the looping statements

The rest of this section presents examples for three of the looping statements.

8.5.1 forever Loop

The `forever` loop should only be used in conjunction with the timing controls or the `disable` statement; therefore, this example is presented in Section 8.6.3.

8.5.2

repeat Loop Example

In the following example of a repeat loop, add and shift operators implement a multiplier.

```
parameter size = 8, longsize = 16;
reg [size:1] opa, opb;
reg [longsize:1] result;
begin :mult
    reg [longsize:1] shift_opa, shift_opb;

    shift_opa = opa;
    shift_opb = opb;
    result = 0;

    repeat (size)
        begin
            if (shift_opb[1])
                result = result + shift_opa;
            shift_opa = shift_opa << 1;
            shift_opb = shift_opb >> 1;
        end
    end
end
```

Example 8-19: Use of the repeat loop to implement a multiplier

8.5.3 while Loop Example

An example of the `while` loop follows. It counts up the number of logic 1 values in `rega`.

```
begin :count1s
    reg [7:0] tempreg;
    count = 0;
    tempreg = rega;
    while(tempreg)
        begin
            if (tempreg[0]) count = count + 1;
            tempreg = tempreg >> 1;
        end
    end
end
```

Example 8-20: Use of the `while` loop to count logic values

8.5.4 for Loop Examples

The `for` loop construct accomplishes the same results as the following pseudocode that is based on the `while` loop:

```
begin
    initial_assignment;
    while (condition)
        begin
            statement
            step_assignment;
        end
    end
end
```

Example 8-21: Pseudocode equivalent of a `for` loop

The `for` loop implements the logic in the preceding 8 lines while using only two lines, as shown in the pseudocode in Example 8-22.

```
for (initial_assignment; condition; step_assignment)
    statement
```

Example 8-22: Pseudocode for a for loop

Example 8-23 uses a for loop to initialize a memory.

```
begin :init_mem
    reg [7:0] tempi;
    for (tempi = 0; tempi < memsize; tempi = tempi + 1)
        memory[tempi] = 0;
end
```

Example 8-23: Use of the for loop to initialize a memory

Here is another example of a for loop statement. It is the same multiplier that was described in Example 8-19 using the repeat loop.

```
parameter size = 8, longsize = 16;
reg [size:1] opa, opb;
reg [longsize:1] result;
begin :mult
    integer bindex;
    result = 0;
    for (bindex = 1; bindex <= size; bindex = bindex + 1)
        if (opb[bindex])
            result = result + (opa << (bindex - 1));
end
```

Example 8-24: Use of the for loop to implement a multiplier

Note that the for loop statement can be more general than the normal arithmetic progression of an index variable, as in Example 8-25. This is another way of counting the number of logic 1 values in rega (see Example 8-20).

```
begin :count1s
    reg [7:0] tempreg;
    count = 0;
    for (tempreg = rega; tempreg; tempreg = tempreg >
> 1)
        if (tempreg[0]) count = count + 1;
end
```

Example 8-25: Use of the for loop to count logic values

8.6 Procedural Timing Controls

The Verilog language provides two types of explicit timing control over when in simulation time procedural statements are to occur. The first type is a delay control in which an expression specifies the time duration between initially encountering the statement and when the statement actually executes. The delay expression can be a dynamic function of the state of the circuit, but is usually a simple number that separates statement executions in time. The delay control is an important feature when specifying stimulus waveform descriptions. It is described in Sections 8.6.1, 8.6.2, and 8.6.7.

The second type of timing control is the event expression, which allows statement execution to wait for the occurrence of some simulation event occurring in a procedure executing concurrently with this procedure. A simulation event can be a change of value on a net or register (an implicit event), or the occurrence of an explicitly named event that is triggered from other procedures (an explicit event). Most often, an event control is a positive or negative edge on a clock signal. Sections 8.6.3 through 8.6.7 discuss event control.

In Verilog, actions are scheduled in the future through the use of delay controls. A general principle of the Verilog language is that “where you do not see a timing control, simulation time does not advance”—if you specify no timing delays, the simulation completes at time zero. To schedule activity for the future, use one of the following methods of timing control:

- a delay control, which is introduced by the number symbol (#)
- an event control, which is introduced by the at symbol (@)
- the wait statement, which operates like a combination of the event control and the while loop

The next sections discuss these three methods.

8.6.1

Delay Control

The execution of a procedural statement can be delay-controlled by using the following syntax:

```
<statement>
    ::= <delay_control> <statement_or_null>
<delay_control>
    ::= # <NUMBER>
    || = # <identifier>
    || = # ( <mintypmax_expression> )
```

Syntax 8-5: Syntax for delay_control

The following example delays the execution of the assignment by 10 time units:

```
#10 rega = regb;
```

The next three examples provide an expression following the number sign (#). Execution of the assignment delays by the amount of simulation time specified by the value of the expression.

```
#d rega = regb;           // d is defined as a parameter
```

```
#((d+e)/2) rega = regb; // delay is the average of d and e
```

```
#regr regr = regr + 1; // delay is the value in regr
```

8.6.2

Zero-Delay control

A special case of the delay control is the zero-delay control, as in the following example:

```
forever
    #0 a = ~a;
```

This type of delay control has the effect of moving the assignment statement to the end of the list of statements to be evaluated at the current simulation time unit. Note that if there are several such delay controls encountered at the same simulation time, the order of evaluation of the statements which they control cannot be predicted.

8.6.3 Event Control

The execution of a procedural statement can be synchronized with a value change on a net or register, or the occurrence of a declared event, by using the following event control syntax:

```
<statement>
    ::= <event_control> <statement_or_null>
<event_control>
    ::= @ <identifier>
    | | = @ ( <event_expression> )
<event_expression>
    ::= <expression>
    | | = posedge <SCALAR_EVENT_EXPRESSION>
    | | = negedge <SCALAR_EVENT_EXPRESSION>
    | | = <event_expression> <or <event_expression>> *
<SCALAR_EVENT_EXPRESSION> is an expression that resolves
    to a one bit value.
```

Syntax 8-6: Syntax for event_control

Value changes on nets and registers can be used as events to trigger the execution of a statement. This is known as detecting an implicit event. See item 1 in Example 8-26 for a syntax example of a wait for an implicit event. Verilog syntax also allows you to detect change based on the direction of the change—that is, toward the value 1 (posedge) or toward the value 0 (negedge). The behavior of posedge and negedge for unknown expression values is as follows:

- a negedge is detected on the transition from 1 to unknown and from unknown to 0
- a posedge is detected on the transition from 0 to unknown and from unknown to 1

Items 2 and 3 in Example 8-26 show illustrations of edge-controlled statements.

- ❶ `@r rega = regb; // controlled by any value changes`
`// in the register r`
- ❷ `@(posedge clock) rega = regb; // controlled by positive`
`// edge on clock`
- ❸ `forever @(negedge clock) rega = regb; // controlled by`
`// negative edge`

Example 8-26: Event controlled statements

8.6.4 Named Events

Verilog also provides syntax to name an event and then to trigger the occurrence of that event. A model can then use an event expression to wait for the triggering of this explicit event. Named events can be made to occur from a procedure. This allows control over the enabling of multiple actions in other procedures. Named events and event control give a powerful and efficient means of describing the communication between, and synchronization of, two or more concurrently active processes. A basic example of this is a small waveform clock generator that synchronizes control of a synchronous circuit by signalling the occurrence of an explicit event periodically while the circuit waits for the event to occur.

An event name must be declared explicitly before it is used. The following is the syntax for declaring events.

```

<event_declaration>
    ::= event <name_of_event> <,<name_of_event>>* ;
<name_of_event>
    ::= <IDENTIFIER> - the name of an explicit event

```

Syntax 8-7: Syntax for event_declaration

Note that an event does not hold any data. The following are the characteristics of a Verilog event:

- it can be made to occur at any particular time
- it has no time duration
- its occurrence can be recognized by using the <event_control> syntax described in Section 8.6.3

The power of the explicit event is that it can represent any general happening. For example, it can represent a positive edge of a clock signal, or it can represent a microprocessor transferring data down a serial communications channel. A declared event is made to occur by the activation of an event-triggering statement of the following syntax:

```
-> <name_of_event> ;
```

An event-controlled statement (for example, @trig rega = regb;) causes simulation of its containing procedure to wait until some other procedure executes the appropriate event-triggering statement (for example, ->trig;).

8.6.5 Event OR Construct

The ORing of any number of events can be expressed such that the occurrence of any one will trigger the execution of the statement. The next two examples show the ORing of two and three events respectively.

```
@(trig or enable) rega = regb; // controlled by trig or enable
```

```
@(posedge clock_a or posedge clock_b or trig) rega = regb;
```

8.6.6 Level-Sensitive Event Control

The execution of a statement can also be delayed until a condition becomes true. This is accomplished using the wait statement, which is a special form of event control. The nature of the wait statement is level-sensitive, as opposed to basic event control (specified by the @ character), which is edge-sensitive. The wait statement checks a

condition, and, if it is false, causes the procedure to pause until that condition becomes true before continuing. The `wait` statement has the following form:

```
wait(condition_expression) statement
```

Example 8-27 shows the use of the `wait` statement to accomplish level-sensitive event control.

```
begin
    wait(!enable) #10 a = b;
    #10 c = d;
end
```

Example 8-27: Use of wait statement

If the value of `enable` is one when the block is entered, the `wait` statement delays the evaluation of the next statement (`#10 a = b;`) until the value of `enable` changes to zero. If `enable` is already zero when the `begin-end` block is entered, then the next statement is evaluated immediately and no delay occurs.

8.6.7 Intra-Assignment Timing Controls

The delay and event control constructs previously described precede a statement and delay its execution. The intra-assignment delay and event controls are contained within an assignment statement and modify the flow of activity in a slightly different way.

Encountering an intra-assignment delay or event control delays the assignment just as a regular delay or event control does, but the right-hand side expression is evaluated before the delay, instead of after the delay. This allows data swap and data shift operations to be described without the need for temporary variables. This section describes the purpose of intra-assignment timing controls and the repeat timing control that can be used in intra-assignment delays.

Figure 8-1 illustrates the philosophy of intra-assignment timing controls by showing the code that could accomplish the same timing effect without using intra-assignment.

Intra-assignment timing control	
with intra-assignment construct	without intra-assignment construct
<code>a = #5 b;</code>	<pre>begin temp = b; #5 a = temp; end</pre>
<code>a = @(posedge clk) b;</code>	<pre>begin temp = b; @(posedge clk) a = temp; end</pre>
<code>a = repeat(3)@(posedge clk) b;</code>	<pre>begin temp = b; @(posedge clk; @(posedge clk; @(posedge clk) a = temp; end</pre>

Figure 8-1: Equivalents to intra-assignment timing controls

The next three examples use the fork-join behavioral construct. All statements between the keywords `fork` and `join` execute concurrently. Section 8.7.2 describes this construct in more detail.

The following example shows a race condition that could be prevented by using intra-assignment timing control:

```
fork
    #5 a = b;
    #5 b = a;
join
```


The code in the previous example samples the values of both *a* and *b* at the same simulation time, thereby creating a race condition. The intra-assignment form of timing control used in the following example prevents this race condition:

```
fork                                // data swap
    a = #5 b;
    b = #5 a;
join
```

Intra-assignment timing control works because the intra-assignment delay causes the values of *a* and *b* to be evaluated *before* the delay, and the assignments to be made *after* the delay. Verilog-XL and other tools that implement intra-assignment timing control use temporary storage in evaluating each expression on the right-hand side.

Intra-assignment waiting for *events* is also effective. In the example below, the right-hand-side expressions are evaluated when the assignment statements are encountered, but the assignments are delayed until the rising edge of the clock signal.

```
fork                                // data shift
    a = @(posedge clk) b;
    b = @(posedge clk) c;
join
```

The repeat event control

The repeat event control specifies an intra-assignment delay of a specified number of occurrences of an event. This construct is convenient when events must be synchronized with counts of clock signals.

Syntax 8-8 presents the repeat event control syntax:

```
<repeat_event_controlled_assignment>  
  ::= <lvalue> = <repeat_event_control> <expression>;  
  || = <lvalue> <= <repeat_event_control> <expression>;  
  
<repeat_event_control>  
  ::= repeat(<expression>)@(<identifier>)  
  || = repeat(<expression>)@(<event_expression>)  
  
<event_expression>  
  ::= <expression>  
  || = posedge <SCALAR_EVENT_EXPRESSION>  
  || = negedge <SCALAR_EVENT_EXPRESSION>  
  || = <event_expression> or <event_expression>
```

Syntax 8-8: Syntax of the repeat event control

The event expression must resolve to a one bit value. A scalar event expression is an expression which resolves to a one bit value.

The following is an example of a repeat event control as the intra-assignment delay of a non-blocking assignment:

```
a<=repeat(5)@(posedge clk)data;
```

Figure 8-2 illustrates the activities that result from this repeat event control:

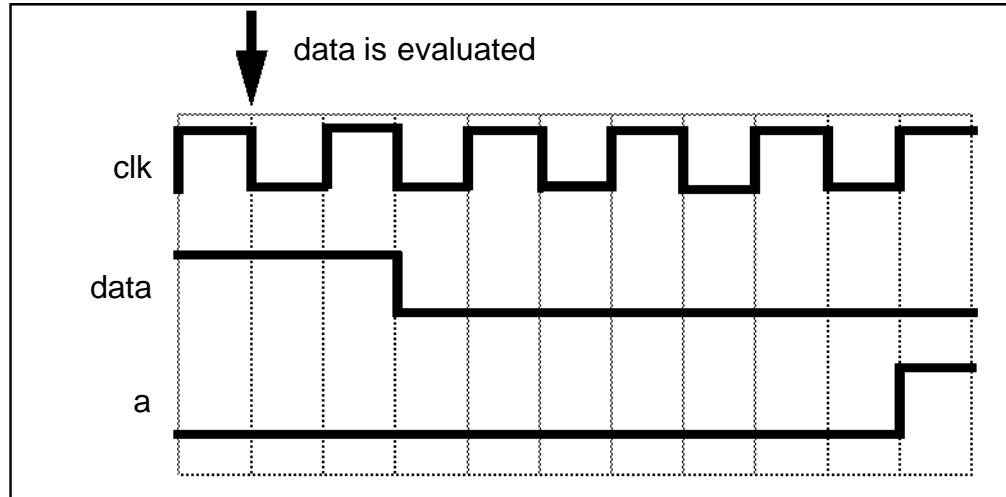


Figure 8-2: Repeat event control utilizing a clock edge

In this example, the value of **data** is evaluated when the assignment is encountered. After five occurrences of `posedge clk`, **a** is assigned the previously evaluated value of **data**.

The following is an example of a repeat event control as the intra-assignment delay of a procedural assignment:

```
a = repeat(num)@(clk)data;
```

In this example, the value of **data** is evaluated when the assignment is encountered. After the number of transitions of **clk** equals the value of **num**, **a** is assigned the previously evaluated value of **data**.

The following is an example of a repeat event control with expressions containing operations to specify both the number of event occurrences and the event that is counted:

```
a <= repeat(a+b)@(posedge phi1 or negedge phi2)data;
```

In the example above, the value of **data** is evaluated when the assignment is encountered. After the positive edges of **phi1**, the negative edges of **phi2**, or the combination of these two events occurs a total of **(a+b)** times, **a** is assigned the previously evaluated value of **data**.

8.7 Block Statements

The block statements are a means of grouping two or more statements together so that they act syntactically like a single statement. We have already introduced and used the sequential block statement which is delimited by the keywords `begin` and `end`. Section 8.7.1 discusses sequential blocks in more detail.

A second type of block, delimited by the keywords `fork` and `join`, is used for executing statements in parallel. A `fork-join` block is known as a parallel block, and enables procedures to execute concurrently through time. Section 8.7.2 discusses parallel blocks.

8.7.1 Sequential Blocks

A sequential block has the following characteristics:

- statements execute in sequence, one after another
- delay values for each statement are relative to the simulation time of the execution of the previous statement
- control passes out of the block after the last statement executes

The following is the formal syntax for a sequential block:

```

<seq_block>
    ::= begin <statement>* end
    || = begin : <name_of_block>
                <block_declaration>*
                <statement>*
    end
<name_of_block>
    ::= <IDENTIFIER>
<block_declaration>
    ::= <parameter_declaration>
    || = <reg_declaration>
    || = <integer_declaration>
    || = <real_declaration>
    || = <time_declaration>
    || = <event_declaration>

```

Syntax 8-9: Syntax for the sequential block

A sequential block enables the following two assignments to have a deterministic result:

```
begin
    areg = breg;
    creg = areg; // creg becomes the value of breg
end
```

Here the first assignment is performed and `areg` is updated before control passes to the second assignment.

Delay control can be used in a sequential block to separate the two assignments in time.

```
begin
    areg = breg;
    #10 creg = areg; // this gives a delay of 10 time
end                // units between assignments
```

Example 8-28 shows how the combination of the sequential block and delay control can be used to specify a time-sequenced waveform.

```
parameter d = 50; // d declared as a parameter
reg [7:0] r;      // and r declared as an 8-bit register
begin            // a waveform controlled by sequential
                // delay
    #d r = 'h35;
    #d r = 'hE2;
    #d r = 'h00;
    #d r = 'hF7;
    #d -> end_wave; // trigger the event called end_wave
end
```

Example 8-28: A waveform controlled by sequential delay

Example 8-29 shows three examples of sequential blocks.

```
begin
❶   @trig r = 1;
    #250 r = 0; // a 250 delay monostable
end

begin
❷   @(posedge clock) q = 0;
    @(posedge clock) q = 1;
end

begin // a waveform synchronized by the event c
❸   @c r = 'h35;
    @c r = 'hE2;
    @c r = 'h00;
    @c r = 'hF7;
    @c -> end_wave;
end
```

Example 8-29: Three examples of sequential blocks

8.7.2 Parallel Blocks

A parallel block has the following characteristics:

- statements execute concurrently
- delay values for each statement are relative to the simulation time when control enters the block
- delay control is used to provide time-ordering for assignments
- control passes out of the block when the last time-ordered statement executes or a `disable` statement executes

Syntax 8-10 gives the formal syntax for a parallel block.

```
<par_block>  
  ::= fork <statement>* join  
  || = fork : <name_of_block>  
        <block_declaration>*  
        <statement>*  
        join  
<name_of_block>  
  ::= <IDENTIFIER>  
<block_declaration>  
  ::= <parameter_declaration>  
  || = <reg_declaration>  
  || = <integer_declaration>  
  || = <real_declaration>  
  || = <time_declaration>  
  || = <event_declaration>
```

Syntax 8-10: Syntax for the parallel block

Example 8-30 codes the waveform description shown in Example 8-28 by using a parallel block instead of a sequential block. The waveform produced on the register is exactly the same for both implementations.

```
fork  
  #50 r = 'h35;  
  #100 r = 'hE2;  
  #150 r = 'h00;  
  #200 r = 'hF7;  
  #250 -> end_wave;  
join
```

Example 8-30: Use of the fork-join construct

8.7.3 Block Names

Note that blocks can be named by adding: `name_of_block` after the keywords `begin` or `fork`. The naming of blocks serves several purposes:

- It allows local variables to be declared for the block.
- It allows the block to be referenced in statements like the `disable` statement (as discussed in Chapter 10, *Disabling of Named Blocks and Tasks*).
- In the Verilog language, all variables are static—that is, a unique location exists for all variables and leaving or entering blocks does not affect the values stored in them.

Thus, block names give a means of uniquely identifying all variables at any simulation time. This is very important for debugging purposes, where it is necessary to be able to reference a local variable inside a block from outside the body of the block.

8.7.4 Start and Finish Times

Both forms of blocks have the notion of a start and finish time. For sequential blocks, the start time is when the first statement is executed, and the finish time is when the last statement has finished. For parallel blocks, the start time is the same for all the statements, and the finish time is when the last time-ordered statement has finished executing. When blocks are embedded within each other, the timing of when a block starts and finishes is important. Execution does not continue to the statement following a block until the block's finish time has been reached—that is, until the block has completely finished executing.

Moreover, the timing controls in a `fork-join` block do not have to be given sequentially in time. Example 8-31 shows the statements from Example 8-30 written in the reverse order and still producing the same waveform.

```
fork
    #250 -> end_wave;
    #200 r = 'hF7;
    #150 r = 'h00;
    #100 r = 'hE2;
    #50 r = 'h35;
join
```

Example 8-31: Timing controls in a parallel block

Sequential and parallel blocks can be embedded within each other allowing complex control structures to be expressed easily and with a high degree of structure.

One simple example of this is when an assignment is to be made after two separate events have occurred. This is known as the 'joining' of events.

```
begin
  fork
    @Aevent;
    @Bevent;
  join
  areg = breg;
end
```

Example 8-32: The joining of events

Note that the two events can occur in any order (or even at the same time), the `fork-join` block will complete, and the assignment will be made. In contrast to this, if the `fork-join` block was a `begin-end` block and the `Bevent` occurred before the `Aevent`, then the block would be deadlocked waiting for the `Bevent`.

Example 8-33 shows two sequential blocks, each of which will execute when its controlling event occurs. Because the `wait` statements are within a `fork-join` block, they execute in parallel and the sequential blocks can therefore also execute in parallel.

```
fork
  @enable_a
  begin
    #ta wa = 0;
    #ta wa = 1;
    #ta wa = 0;
  end
  @enable_b
  begin
    #tb wb = 1;
    #tb wb = 0;
    #tb wb = 1;
  end
join
```

Example 8-33: Enabling sequential blocks to execute in parallel

8.8 Structured Procedures

All procedures in Verilog are specified within one of the following four statements:

- `initial` statement
- `always` statement
- `task`
- `function`

The `initial` and `always` statements are enabled at the beginning of simulation. The `initial` statement executes only once and its activity dies when the statement has finished. In contrast, the `always` statement executes repeatedly. Its activity dies only when the simulation is terminated. There is no limit to the number of `initial` and `always` blocks that can be defined in a module.

Tasks and functions are procedures that are enabled from one or more places in other procedures. Tasks and functions are covered in detail in Chapter 9, *Tasks and Functions*.

8.8.1 initial Statement

The syntax for the initial statement is as follows:

```
<initial_statement>  
 ::= initial <statement>
```

Syntax 8-11: Syntax for <initial_statement>

Example 8-34 illustrates use of the initial statement for initialization of variables at the start of simulation.

```
initial  
  begin  
    areg = 0; // initialize a register  
    for (index = 0; index < size; index = index + 1)  
      memory[index] = 0; //initialize a memory  
      word  
  end
```

Example 8-34: Use of initial statement

Another typical usage of the initial statement is specification of waveform descriptions that execute once to provide stimulus to the main part of the circuit being simulated. Example 8-35 illustrates this usage.

```
initial  
  begin  
    inputs = 'b000000;  
    // initialize at time zero  
    #10 inputs = 'b011001; // first pattern  
    #10 inputs = 'b011011; // second pattern  
    #10 inputs = 'b011000; // third pattern  
    #10 inputs = 'b001000; // last pattern  
  end
```

Example 8-35: Another use for initial statement

8.8.2

always Statement

The `always` statement repeats continuously throughout the whole simulation run. Syntax 8-12 gives the syntax for the `always` statement.

```
<always_statement>  
 ::= always <statement>
```

Syntax 8-12: Syntax for `always_statement`

The `always` statement, because of its looping nature, is only useful when used in conjunction with some form of timing control. If an `always` statement provides no means for time to advance, the `always` statement creates a simulation deadlock condition. The following code, for example, creates an infinite zero-delay loop:

```
always areg = ~areg;
```

Providing a timing control to the above code creates a potentially useful description—as in the following example:

```
always #half_period areg = ~areg;
```

8.8.3

Examples

We have now introduced enough statement types for some complete and more practical examples to be given. These examples are given as complete descriptions enclosed in modules—such that they can be put directly through the Verilog-XL compiler, simulated and the results observed.

Example 8-36 is a simple traffic light sequencer described with its own clock generator.

```
module traffic_lights;
    reg
        clock,
        red,
        amber,
        green;
    parameter
        on = 1,
        off = 0,
        red_ticks = 350,
        amber_ticks = 30,
        green_ticks = 200;
    // the sequence to control the lights
    always
    begin
        red = on;
        amber = off;
        green = off;
        repeat (red_ticks) @(posedge clock);
        red = off;
        green = on;
        repeat (green_ticks) @(posedge clock);
        green = off;
        amber = on;
        repeat (amber_ticks) @(posedge clock);
    end
    // waveform for the clock
    always
    begin
        #100 clock = 0;
        #100 clock = 1;
    end
    // simulate for 10 changes on the red light
    initial
    begin
        repeat (10) @red;
        $finish;
    end
    // display the time and changes made to the lights
    always
    @(red or amber or green)
    $display("%d red=%b amber=%b green=%b",
        $time, red, amber, green);
endmodule
```

Example 8-36: Behavioral model of traffic light sequencer

Example 8-37 shows a use of variable delays. The module has a clock input and produces two synchronized clock outputs. Each output clock has equal mark and space times, is out of phase from the other by 45 degrees, and has a period half that of the input clock. Note that the clock generation is independent of the simulation time unit, except as it affects the accuracy of the divide operation on the input clock period.

```

module synch_clocks;
    reg
        clock,
        phase1,
        phase2;
    time clock_time;
    initial clock_time = 0;
    always @(posedge clock)
        begin :phase_gen
            time d; // a local declaration is possible
                    // because the block is named
            d = ($time - clock_time) / 8;
            clock_time = $time;
            phase1 = 0;
            #d phase2 = 1;
            #d phase1 = 1;
            #d phase2 = 0;
            #d phase1 = 0;
            #d phase2 = 1;
            #d phase1 = 1;
            #d phase2 = 0;
        end
    // set up a clock waveform, finish time,
    // and display
    always
        begin
            #100 clock = 0;
            #100 clock = 1;
        end
    initial #1000 $finish; //end simulation at time 1000
    always
        @(phase1 or phase2)
            $display($time,,
                    "clock=%b phase1=%b phase2=%b",
                    clock, phase1, phase2);
endmodule

```

Example 8-37: Behavioral model with variable delays

9

Figure 9-0
Example 9-0
Syntax 9-0
Table 9-0

Tasks and Functions

Tasks and functions provide the ability to execute common procedures from several different places in a description. They also provide a means of breaking up large procedures into smaller ones to make it easier to read and debug the source descriptions. Input, output, and inout argument values can be passed into and out of both tasks and functions. The next section discusses the differences between tasks and functions. Subsequent sections describe how to define and invoke tasks and functions and present examples of each.

9.1 Distinctions Between Tasks and Functions

The following rules distinguish tasks from functions:

- A function must execute in one simulation time unit; a task can contain time-controlling statements.
- A function cannot enable a task; a task can enable other tasks and functions.
- A function must have at least one input argument; a task can have zero or more arguments of any type.
- A function returns a single value; a task does not return a value.

The purpose of a *function* is to respond to an input value by returning a single value. A *task* can support multiple goals and can calculate multiple result values. However, only the output or inout arguments

pass result values back from the invocation of a task. A Verilog model uses a function as an operand in an expression; the value of that operand is the value returned by the function.

For example, you could define either a task or a function to switch bytes in a 16-bit word. The *task* would return the switched word in an output argument, so the source code to enable a task called `switch_bytes` could look like the following example:

```
switch_bytes (old_word, new_word);
```

The task `switch_bytes` would take the bytes in `old_word`, reverse their order, and place the reversed bytes in `new_word`. A word-switching *function* would return the switched word directly. Thus, the function call for the function `switch_bytes` might look like the following example:

```
new_word = switch_bytes (old_word);
```

9.2

Tasks and Task Enabling

A task is enabled from a statement that defines the argument values to be passed to the task and the variables that will receive the results. Control is passed back to the enabling process after the task has completed. Thus, if a task has timing controls inside it, then the time of enabling can be different from the time at which control is returned. A task can enable other tasks, which in turn can enable still other tasks—with no limit on the number of tasks enabled. Regardless of how many tasks have been enabled, control does not return until all enabled tasks have completed.

9.2.1 Defining a Task

The following is the syntax for defining tasks:

```

<task>
    ::= task <name_of_task> ;
        <tf_declaration>*
        <statement_or_null>
    endtask
<name_of_task>
    ::= <IDENTIFIER>
<tf_declaration>
    ::= <parameter_declaration>
    | = <input_declaration>
    | = <output_declaration>
    | = <inout_declaration>
    | = <reg_declaration>
    | = <time_declaration>
    | = <integer_declaration>
    | = <real_declaration>
    | = <event_declaration>

```

Syntax 9-1: Syntax for <task>

Task and function declarations specify the following:

- local variables
- I/O ports
- registers
- times
- integers
- real
- events

These declarations all have the same syntax as for the corresponding declarations in a module definition.

If there is more than one output, input, and inout port declared in a task these must be enclosed within a block.

9.2.2

Task Enabling and Argument Passing

The statement that enables a task passes the I/O arguments as a comma-separated list of expressions enclosed in parentheses. The following is the formal syntax of the task enabling statement:

```
<task_enable>  
 ::= <name_of_task> ;  
 || = <name_of_task> ( <expression> <,<expression>>* ) ;
```

Syntax 9-2: Syntax of the task enabling statement

The first form of a task enabling statement applies when there are no I/O arguments declared in the task body. In the second form, the list of <expression> items is an ordered list that must match the order of the list of I/O arguments in the task definition.

If an I/O argument is an input, then the corresponding <expression> can be any expression. If the I/O argument is an output or an inout, then Verilog restricts it to an expression that is valid on the left-hand side of a procedural assignment. The following items satisfy this requirement:

- reg, integer, real, and time variables
- memory references
- concatenations of reg, integer, real, and time variables
- concatenations of memory references
- bit-selects and part-selects of reg, integer, real, and time variables

The execution of the task enabling statement passes input values from the variables listed in the enabling statement to the variables specified within the task. Execution of the return from the task passes values from the task output and inout variables to the corresponding variables in the task enabling statement. Verilog passes all arguments by value (that is, Verilog passes the *value* rather than a *pointer* to the value).

Example 9-1 illustrates the basic structure of a task definition with five arguments.

```
module this_task;
  task my_task;
    input a, b;
    inout c;
    output d, e;
    reg foo1, foo2, foo3;
    begin
      <statements>          // the set of statements that
                          // performs the work of the task

      c = foo1;             // the assignments that initialize
      d = foo2;             // the results variables
      e = foo3;
    end
  endtask
endmodule
```

Example 9-1: Task definition with arguments

The following statement enables the task in Example 9-1:

```
my_task (v, w, x, y, z);
```

The calling arguments (v, w, x, y, z) correspond to the I/O arguments (a, b, c, d, e) defined by the task. At task enabling time, the input and inout arguments (a, b, and c) receive the values passed in v, w, and x. Thus, execution of the task enabling call effectively causes the following assignments:

```
a = v; b = w; c = x;
```

As part of the processing of the task, the task definition for my_task must place the computed results values into c, d, and e. When the task completes, the processing software performs the following assignments to return the computed values to the calling process:

```
x = c; y = d; z = e;
```

9.2.3 Task Example

Example 9-2 illustrates the use of tasks by redescribing the traffic light sequencer that was introduced in Chapter 8, *Behavioral Modeling*.

```
module traffic_lights;
    reg clock, red, amber, green;
    parameter on = 1, off = 0, red_tics = 350,
              amber_tics = 30, green_tics = 200;

    // initialize colors
    initial
        red = off;
    initial
        amber = off;
    initial
        green = off;

    // sequence to control the lights
    always begin
        red = on; // turn red light on
        light(red, red_tics); // and wait.
        green = on; // turn green light on
        light(green, green_tics); // and wait.
        amber = on; // turn amber light on
        light(amber, amber_tics); // and wait.
    end

    // task to wait for 'tics' positive edge clocks
    // before turning 'color' light off
    task light;
        output color;
        input [31:0] tics;
        begin
            repeat (tics)
                @(posedge clock);
            color = off; // turn light off
        end
    endtask

    // waveform for the clock
    always begin
        #100 clock = 0;
        #100 clock = 1;
    end
endmodule // traffic_lights
```

Example 9-2: Using tasks

9.2.4

Effect of Enabling an Already Active Task

Because Verilog supports concurrent procedures, and tasks can have non-zero time duration, you can write a model that invokes a task when that task is already executing (a special case of invoking a task that is already active is where a task recursively calls itself). Verilog-XL allows multiple copies of a task to execute concurrently, but it does not copy or otherwise preserve the task arguments or local variables. Verilog-XL uses the same storage for each invocation of the task. This means that when the simulator interrupts a task to process another instance of the same task, it overwrites the argument values from the first call with the values from the second call. The user must manage what happens to the variables of a task that is invoked while it is already active.

9.3

Functions and Function Calling

The purpose of a function is to return a value that is to be used in an expression. The rest of this chapter explains how to define and use functions.

9.3.1

Defining a Function

To define functions, use the following syntax:

```
<function>
    ::= function <range_or_type>? <name_of_function> ;
        <tf_declaration>+
        <statement_or_null>
    endfunction

<range_or_type>
    ::= <range>
    | = integer
    | = real

<name_of_function>
    ::= <IDENTIFIER>

<tf_declaration>
    ::= <parameter_declaration>
    | = <input_declaration>
    | = <reg_declaration>
    | = <time_declaration>
    | = <integer_declaration>
    | = <real_declaration>
    | = <event_declaration>
```

Syntax 9-3: Syntax for function

A function returns a value by assigning the value to the function's name. The <range_or_type> item which specifies the data type of the function's return is optional.

Example 9-3 defines a function called `getbyte`, using a `<range>` specification.

```
module fact;
  function [7:0] getbyte;
    input [15:0] address;
    reg [3:0] result_expression;
    begin
      //<statements> code to extract low-order
      // byte from addressed word
      getbyte = result_expression;
    end
  endfunction
endmodule
```

Example 9-3: A function definition using range

9.3.2 Returning a Value from a Function

The function definition implicitly declares a register, internal to the function, with the same name as the function. This register either defaults to one bit or is the type that `<range_or_type>` specifies. The `<range_or_type>` can specify that the function's return value is a `real`, an `integer`, or a value with a range of `[n:m]` bits. The function assigns its return value to the internal variable bearing the function's name. The following line from Example 9-3 illustrates this concept:

```
getbyte = result_expression;
```

9.3.3 Calling a Function

A function call is an operand within an expression. The operand has the following syntax:

<pre><function_call> ::= <name_of_function> (<expression> <,<expression>>*) <name_of_function> ::= <identifier></pre>

Syntax 9-4: Syntax for function_call

The following example creates a word by concatenating the results of two calls to the function `getbyte` (defined in Example 9-3).

```
word = control ? {getbyte(msbyte), getbyte(lsbyte)} : 0;
```

9.3.4 Function Rules

Functions are more limited than tasks. The following five rules govern their usage:

- A function definition cannot contain any time controlled statements—that is, any statements introduced with `#`, `@`, or `wait`.
- Functions cannot enable tasks.
- A function definition must contain at least one `input` argument.
- A function definition must include an assignment of the function result value to the internal variable that has the same name as the function.
- A function definition can not contain an `inout` declaration or an `output` declaration.

9.3.5 Function Example

Example 9-4 defines a function called `factorial` that returns a 32-bit register. The `factorial` function then calls itself recursively and prints some results.

```

module tryfact;
    // define function
    function [31:0] factorial;
        input [3:0] operand;
        reg [3:0] index;
        begin
            factorial = operand ? 1 : 0;

for(index = 2; index <= operand; index = index + 1)
            factorial = index * factorial;
        end
    endfunction

    // Test the function
    reg [31:0] result;
    reg [3:0] n;
    initial
        begin
            result = 1;
            for(n = 2; n <= 9; n = n+1)
                begin

$display("Partial result  n=%d result=%d",
            n, result);

result = n * factorial(n) / ((n * 2) + 1);
            end
            $display("Final result=%d", result);
        end
    endmodule // tryfact

```

Example 9-4: Defining and calling a function

Index

System Tasks, System Functions, and Timing Checks

\$ 19-15

\$async\$and\$array 22-2

\$async\$and\$plane 22-2

\$async\$nand\$array 22-2

\$async\$nand\$plane 22-2

\$async\$nor\$array 22-2

\$async\$nor\$plane 22-2

\$async\$or\$array 22-2

\$async\$or\$plane 22-2

\$bitstoreal 12-18, 21-55

\$cleartrace 21-20

\$compare 21-49
syntax 21-49

\$countdrivers 21-30 to 21-31
syntax 21-30

\$db_breakaftertime 26-22
syntax 26-22

\$db_breakatline 26-21
syntax 26-21

\$db_breakbeforetime 26-22
syntax 26-22

\$db_breakonceatline
syntax 26-21

\$db_breakonceonnegedge
syntax 26-25

\$db_breakonceonposedge 26-24

\$db_breakoncewhen
syntax 26-23

\$db_breakonnegedge 26-25
syntax 26-25

\$db_breakonposedge 26-24
syntax 26-24

\$db_breakwhen 26-23
syntax 26-23

\$db_cleartrace 26-17
syntax 26-17

\$db_deletebreak 26-26
syntax 26-26

\$db_deletefocus 26-9
syntax 26-9

\$db_disablebreak 26-27
syntax 26-27

\$db_disablefocus 26-11
syntax 26-11

\$db_enablebreak 26-26
syntax 26-26

\$db_enablefocus 26-10
syntax 26-10

\$db_help 26-6
syntax 26-6

\$db_setfocus 26-9
syntax 26-9

\$db_settrace 26-16
syntax 26-16

\$db_showbreak 26-28
syntax 26-28

\$db_showfocus 26-12
syntax 26-12

\$db_step 26-14
syntax 26-14

\$db_steptime 26-15
syntax 26-15

\$disable_warnings 21-38
syntax 21-38

\$display 20-9, 21-1 to 21-11
and mnemonic strength format 6-33
and simulation time 21-18
compared to \$monitor 21-12
compared to \$write 21-2
escape sequences 21-3
format specifications 21-4 to 21-5
size of displayed data 21-6 to 21-7
syntax 21-1

System Tasks, System Functions, and Timing Checks (continued)

\$dist_chi_square
syntax 23-5

\$dist_erlang
syntax 23-5

\$dist_exponential
syntax 23-5

\$dist_normal
syntax 23-5

\$dist_poisson
syntax 23-5

\$dist_t
syntax 23-5

\$dist_uniform
syntax 23-5

\$dumpall 19-5, 19-14, 21-58

\$dumpfile 19-3, 21-58

\$dumpflush 19-6, 21-58

\$dumplimit 19-5, 21-58

\$dumpoff 19-4 to 19-5, 19-15, 21-58

\$dumpon 19-4 to 19-5, 19-15, 21-58

\$dumpvars 19-3 to 19-4, 19-14, 20-9, 21-58

\$enable_warnings 21-40
syntax 21-40

\$fclose 21-14 to 21-17
syntax 21-14

\$fdisplay 21-14 to 21-17
syntax 21-14

\$finish 21-18 to 21-19
syntax 21-18

\$fmonitor 21-14 to 21-17
syntax 21-14

\$fopen 21-14 to 21-17
syntax 21-14

\$fstrobe 21-14 to 21-17
syntax 21-14

\$fwrite 21-14 to 21-17
syntax 21-14

\$getpattern 21-43

\$gr_regs 20-9

\$gr_waves 20-9, 27-25

\$history 21-23 to 21-24
syntax 21-23

\$hold 13-41

\$incpattern_read 21-47
syntax 21-47

\$incpattern_write 21-45
syntax 21-45

\$sincsave 21-21 to 21-23, 27-25
syntax 21-21

\$input 21-25
and asynchronous interrupts 21-25
and reading key files 21-25
and reading previous command file
21-25
syntax 21-25

\$itor 21-55

\$keepcommands 21-34

\$key 21-26 to 21-27
syntax 21-26

\$list 20-8, 21-35
and decompiling macro modules 12-12
for debugging 8-13
library definition renaming 25-27
syntax 21-35

\$listcounts 18-11, 21-35
use with +speedup 24-12

\$list_forces 21-36

\$log
syntax 21-25

\$monitor 20-9, 21-12 to 21-13
and fixed width format 21-8
and simulation time 21-18
compared to \$display 21-12
syntax 21-12
turn off 21-13

\$monitoroff 21-12 to 21-13
syntax 21-12

\$monitoron 21-12 to 21-13
syntax 21-12

System Tasks, System Functions, and Timing Checks (continued)

\$nochange 13-51 to 13-52

\$nokeepcommands 21-34

\$nokey 21-26 to 21-27
syntax 21-26

\$nolog
syntax 21-25

\$options 28-83

\$period 13-44

\$sprinttimescale 16-10 to 16-11, 21-56

\$q_add 23-2
syntax 23-1

\$q_exam 23-3
syntax 23-1

\$q_full 23-2
syntax 23-1

\$q_initialize 23-2

\$q_remove 23-2
syntax 23-1

\$readmemb 21-41 to 21-43, 21-56
and **\$getpattern** 21-43
and loading logic array personality
22-4
syntax 21-41

\$readmemh 21-41 to 21-43, 21-56
and **\$getpattern** 21-43
and loading logic array personality
22-4
syntax 21-41

\$realtime 16-7, 21-17, 21-56

\$realtobits 12-18, 21-55

\$recovery 13-47

\$reportfile 27-15

\$reportprofile 18-9, 21-59

\$reset 21-60 to 21-65, 27-27

\$reset_count 21-65 to 21-66

\$reset_value 21-67 to 21-69

\$restart 21-21 to 21-23, 27-27
and queueing tasks 23-4
syntax 21-21

\$rs_get_net 28-97

\$rs_showpaths 28-95

\$rs_trace_net 28-85

\$rs_untrace_net 28-85

\$rtoi 21-55

\$save 21-21 to 21-23, 27-27
and interactive recovery 26-5
and queueing tasks 23-4
and restarting simulator 24-8
syntax 21-21

\$scale 16-8 to 16-9, 21-56

\$scope 20-9
syntax 21-27

\$settrace 20-8, 27-12
and acceleration 27-7
and tracing statements inside macro
modules 12-12
syntax 21-20
to trace from start of simulation 24-8

\$setup 13-40

\$setuphold 13-48

\$showallinstances 27-15
and reporting resolution paths 25-26
and resolving modules 25-15
library definition renaming 25-27
syntax 21-27

\$showexpandednets 20-8
syntax 21-29

\$showmodes 17-11, 21-34

\$shownonxl 27-5
syntax 27-6

\$showportsnotcollapsed 20-8
syntax 21-29

\$showscopes
syntax 21-27

\$showvariables
syntax 21-28

System Tasks, System Functions, and Timing Checks (continued)

\$showvars 20-8
 syntax 21-28

\$skew 13-45

\$sreadmemb 20-17 to 20-19, 21-56

\$sreadmemh 20-17 to 20-19, 21-56

\$startprofile 18-2, 18-8, 21-58, 27-15

\$stime 21-17 to 21-18
 as parameters to \$display and \$monitor 21-18
 syntax 21-17

\$stop 21-18 to 21-19, 26-2
 syntax 21-18

\$stopprofile 18-10, 21-59

\$strobe 20-9, 21-12
 syntax 21-12

\$strobe_compare 21-51
 syntax 21-51

\$sync\$and\$array 22-2

\$sync\$and\$plane 22-2

\$sync\$nand\$array 22-2

\$sync\$nand\$plane 22-2

\$sync\$nor\$array 22-2

\$sync\$nor\$plane 22-2

\$sync\$or\$array 22-2

\$sync\$or\$plane 22-2

\$stopprofile 21-59

\$test\$plusargs 24-25 to 24-26
 syntax 24-25

\$time 3-16, 16-6 to 16-7, 21-17 to 21-18, 21-56
 and triggering event controls 21-18
 as parameters to \$display and \$monitor 21-18
 syntax 21-17

\$timeformat 16-11 to 16-14, 21-56

\$width 13-42
 use of threshold argument 13-43

\$write 20-9, 21-1 to 21-11
 compared to \$display 21-2
 escape sequences 21-3
 format specifications 21-4 to 21-5
 size of displayed data 21-6 to 21-7
 syntax 21-1

Compiler Directives

in compiler directives 24-26

'accelerate 24-27, 27-2 to 27-3

'autoexpand_vectornets 24-27

'celldefine 24-27

'default_decay_time 24-28

'default_nettype 24-29
 syntax 6-15

'default_rswitch_strength 24-29, 28-29

'default_switch_strength 24-30, 28-29

'default_trireg_strength 24-30, 28-29

'define 24-30
 and library search paths 25-3

'delay_mode_distributed 17-4, 24-30

'delay_mode_path 17-4, 24-30

'delay_mode_unit 17-4, 24-31

'delay_mode_zero 17-4, 24-31

'else 24-31, 24-45 to 24-54

'endcelldefine 24-27

'endif 24-31, 24-45 to 24-54

'endprotect 20-2 to 20-3, 20-7, 20-9, 24-32

'endprotected 20-3, 20-5 to 20-7, 20-11

'end_pre_16a_paths 14-16, 24-32

'expand_vectornets 24-31

'ifdef 24-31, 24-45 to 24-54

'include 24-54 to 24-59

'noaccelerate 24-27, 27-2 to 27-3

Compiler Directives, (continued)

'noexpand_vectornets 24-32
'noremove_gatenames 6-43, 24-33
'noremove_netnames 6-43, 24-33
'nounconnected_drive 24-35
'pre_16a_paths 14-15, 24-32
'protect 20-2 to 20-3, 20-7, 20-9, 24-32
'protected 20-3, 20-5 to 20-7, 20-11, 24-33, 25-25
'remove_gatenames 6-43, 24-33
'remove_netnames 6-43, 24-33
'resetall 24-33
 and library files 25-25
'rs_technology 24-34
'switch 28-8
'switch default 24-34
'switch resistive 24-34
'switch XL 24-34
'timescale 16-2 to 16-5, 24-34
 effect on performance 16-5
 usage rules 16-5
'unconnected_drive 24-35
'undef 24-35
'unprotected 24-33, 25-25
'uselib 24-37, 25-3 to 25-8

Command Line Options

-a 24-4
 compared to 'accelerate 27-2
 for accelerating entire source
 description 27-2
+autonaming 12-29, 24-11
+autoprotect 20-5 to 20-7, 20-12, 24-11
+bpi_listcounts 18-8
+bpi_profile 18-8, 24-12

-c 24-4
 and library files 25-26
 use with +protect or +autoprotect
 20-16
 use with -r 24-4
+caxl 24-12
-d 12-12, 24-4
 and source protection 20-8
+define+
 and 'define 24-13, 24-50
 and empty macros 24-49
 and library search paths 25-3
 and macro strings 24-12
+delay_mode_distributed 17-5, 24-14
+delay_mode_path 17-5, 24-14
+delay_mode_unit 17-5, 24-14
+delay_mode_zero 17-5, 24-14
+err_line_length+ 24-15
-f 24-2, 24-4
-i 24-6
 and interactive recovery 26-5
 and reading key files 21-25
 for simulation recovery 21-26
 to read command input file 21-25
+incdir+ 24-15
-k 24-7
 for changing key file name 21-26
-l 24-7
 to change log file name 21-26
+libext 25-9 to 25-11
+libext+ 24-15
 syntax 24-15
+libnonamehide 25-22 to 25-23
+liborder 24-16, 25-13 to 25-15
+librescan 24-16, 25-15 to 25-16
+libverbose 24-16
 and reporting resolution paths 25-26
 and resolving modules 25-15
+maxdelays 4-23, 6-38, 13-20, 24-17
+max_error_count 24-17

Command Line Options, (continued)

+mindelays 4-23, 6-38, 24-17
+noaccerr 24-18
+nolibcell 24-18
+notimingchecks 13-52
+no_charge_decay 24-18
+no_cond_event_error 24-18
+no_notifier 24-19
+no_pulse_msg 13-34, 24-19
+pathpulse 24-20
+pre_16a_paths 14-15, 24-20
+protect 20-3 to 20-5, 20-12, 20-17 to 20-19, 24-20
+pulse_e/n 13-33 to 13-35, 24-21
+pulse_r/m 13-33 to 13-35, 24-21
-q 24-7
-r 24-8, 26-5
 for command-line restart 21-23
 use with -c 24-4
+rswrctostr or +rsw_rc_to_str 24-21
+rsw_opt_stack 24-21
-s 24-8
+speedup 18-8, 24-21
 use with \$listcounts 24-12
+switchres 28-7
+switchres or +switch_res 24-23
+switchxl 24-23, 28-7
+switch_res 28-7
+sxl_keep_all 24-23
+sxl_keep_declared 24-24
+sxl_keep_minimum 24-24
+sxl_unidirect 24-23, 28-19
-t 24-8
+typdelays 4-23, 6-38, 13-20, 24-24
-u 24-8

-v 24-9, 25-9
-w 12-32, 24-9
-x 24-9
-y 24-10, 25-9
 syntax 24-10

Symbols

!
 compared to '==0' 4-9
 logical negation operator 4-2, 4-9
!=
 logical inequality operator 4-2, 4-4, 4-8
!==
 case inequality operator 4-2, 4-4, 4-8
%
 in format specifications 21-2, 21-6
 modulus operator 4-2, 4-4
&
 bit-wise AND operator 4-2, 4-4, 4-11
 reduction AND operator 4-2, 4-12
&&
 logical AND operator 4-2, 4-4, 4-9

 arithmetic multiplication operator 4-2
 bit-wise negation operator 4-4
 for flagging active commands 21-23
 in state table 7-15, 7-21
 logical negation operator 4-4
,
 trace-step command 26-3
-
 in state table 7-10
.
 continue command 26-3
/
 arithmetic division operator 4-2, 4-4
 arithmetic multiplication operator 4-4
:
 interactive command 26-3
;
 step command 26-3

Symbols, (continued)

<	relational less-than operator 4-2, 4-4, 4-7	^~	bit-wise equivalence operator 4-2, 4-4 bit-wise exclusive NOR operator 4-11 reduction XNOR operator 4-2
<<	left shift operator 4-2, 4-4, 4-15		bit-wise inclusive OR operator 4-2, 4-4, 4-11 reduction OR operator 4-2, 4-12
<-	and compiler error messages 24-40		logical OR operator 4-2, 4-4, 4-9
<=	relational less-than-or-equal operator 4-2, 4-4, 4-7	~	bit-wise negation operator 4-2, 4-11
=	in assignment statement 5-1	~&	reduction NAND operator 4-2
==	logical equality operator 4-2, 4-4	~^	bit-wise equivalence operator 4-2 reduction XNOR operator 4-2
===	case equality operator 4-2, 4-4	~	reduction NOR operator 4-2
>	relational greater-than operator 4-2, 4-4, 4-7	“,“,“,“	null string 4-22
>>	right shift operator 4-2, 4-4, 4-15	“,“	commas in null expressions 21-2
>=	relational greater-than-or-equal operator 4-2, 4-4, 4-7	\"	as ” character 2-7
?	equivalent to z in literal number values 2-4, 8-19 in state table 7-7, 7-10, 7-21	(??)	in state table 7-21
@	for addressing memory 21-41	(01)	in state table 7-21
\	backslash character 2-7 for escape sequences in strings 21-2	(0x)	in state table 7-21
^	bit-wise exclusive OR operator 4-2, 4-4, 4-11 for hierarchical names in macro module instances 12-27 reduction XOR operator 4-2, 4-12	(10)	in state table 7-21
		(1x)	in state table 7-21
		\\ddd	specify character as octal digits 2-7
		\\n	new line character 2-7
		\\t	tab character 2-7

Symbols, (continued)

- (vw)**
 - in state table** 7-21
- (x1)**
 - in state table** 7-21
- ?:**
 - conditional operator** 4-2, 4-4
- {}**
 - concatenation operator** 4-2, 4-16

Numbers

- 0**
 - for minimizing bit lengths of expressions** 21-6
 - in state table** 7-21
 - logic 0** 21-9
 - logic zero** 3-1
- 01 transition** 7-9
- 1**
 - in state table** 7-21
 - logic 1** 21-9
 - logic one** 3-1

A

- accelerate option** 24-4
 - compared to 'accelerate** 27-2
 - for accelerating entire source description** 27-2
- accelerated continuous assignments**
 - 5-10 to 5-33
 - compilation speed** 5-31
 - controlling** 5-22
 - different results** 5-31 to 5-33
 - effects** 5-24 to 5-33
 - memory usage** 5-31
 - restrictions** 5-10 to 5-21
 - delay expressions** 5-19
 - left-hand side** 5-11 to 5-13
 - right-hand side** 5-13 to 5-19
 - simulation speed** 5-24 to 5-30

- acceleration** 27-1 to 27-27
 - and key files containing asynchronous interrupts** 27-8
 - and module path destinations** 13-9, 13-61
 - and specify path declarations** 13-3
 - and tracing** 27-8
 - list of items that cannot be accelerated** 27-4 to 27-5
 - of events** 27-8
 - of primitives for performance** 27-12
 - potential problems** 27-8 to 27-9
 - primitives and scalar nets that can be accelerated** 27-3 to 27-4
 - processing simultaneous events** 27-7 to 27-8
 - running XL** 27-2 to 27-3
 - when pulse width equals gate delay** 27-8
 - XL option compared to normal simulation** 27-7 to 27-8
- addressing memory** 21-41 to 21-43
- alias**
 - performance** 27-20
- always**
 - and activity flow** 8-2
 - as structured procedure** 8-41
 - syntax** 8-43
- ambiguous strength** 6-20 to 6-32
- and gate** 6-6 to 6-7
- arguments**
 - for system timing checks** 13-39
 - optional** 13-38
- arithmetic operators** 4-2, 4-5 to 4-6
 - 4-5
 - + 4-5
 - % 4-5
 - * 4-5
 - / 4-5
 - and unknown logic values** 4-6

A, (continued)

arrays

- element** 3-14
- format** 22-4
- index** 3-14
- no multiple dimension** 3-14
- of integers** 3-16
- of time variables** 3-16
- word** 3-14

assign keyword 11-2 to 11-3

assignment 5-1 to 5-33

- continuous** 5-2 to 5-9, 8-3
- left hand side** 5-1
- of delays to module paths**
 - 13-15 to 13-21
- procedural** 8-3 to 8-4
- procedural versus continuous** 8-3
- right hand side** 5-1

asynchronous arrays 22-3

automatic naming 12-29 to 12-30

- +autonaming option** 12-29
- for gates** 6-6
- for user-defined primitives** 7-14

B

b

- binary number format** 2-2
- in state table** 7-15, 7-21

backslash character 2-7

base format

- binary** 2-2
- decimal** 2-2
- hexadecimal** 2-2
- octal** 2-2

begin-end block statement 8-12, 8-35

Behavior Profiler 18-1 to 18-26

- data table** 18-4 to 18-7
 - by module instance** 18-7
 - by statement** 18-4 to 18-6
- example** 18-17 to 18-26
- finding performance problems** 27-15
- system tasks** 18-8 to 18-14
 - \$listcounts** 18-11
 - \$reportprofile** 18-9
 - \$startprofile** 18-8

\$stopprofile 18-10

use with +speedup 24-12

behavioral modeling 8-1 to 8-45

+speedup 24-21

bidirectional pass gate 6-12

binary display format 2-2

- and high impedance state** 21-8
- and unknown logic value** 21-8

binary operators 4-4

- 4-4
- +** 4-4
- %** 4-4
- &** 4-4, 4-11
- *** 4-4
- <** 4-4
- >** 4-4
- ^** 4-4, 4-11
- |** 4-4, 4-11
- !=** 4-4
- &&** 4-4, 4-9
- <<** 4-4
- <=** 4-4
- ==** 4-4
- >=** 4-4
- >>** 4-4
- ^~** 4-4, 4-11
- ||** 4-4, 4-9
- !==** 4-4
- ===** 4-4
- {}** 4-16
- /** 4-4
- precedence** 4-4

bit-select

- and vector ports** 12-14
- of vector net or register** 4-17
- out of bounds** 4-17, 4-19
- performance** 27-16
- references of real numbers** 3-18

B, (continued)

bit-wise operators 4-11 to 4-12

- &** 4-11

- ^** 4-11

- |** 4-11

- ~** 4-11

- ^~** 4-11

- AND** 4-2, 4-11

- compared to logical operators** 4-12

- equivalence** 4-2

- exclusive OR** 4-2, 4-11

- exclusive NOR** 4-11

- inclusive OR** 4-2, 4-11

- negation** 4-2

- unarynegation** 4-11

blank module terminal 12-5

block statement 8-35 to 8-41

- definition** 8-35

- fork-join** 8-35

- naming of** 8-39

- parallel** 8-35

- sequential** 8-35 to 8-37

- start and finish times** 8-39 to 8-41

- timing for embedded blocks** 8-39

blocking procedural assignment 8-4

- processing assignments** 8-11

- syntax** 8-4

breakpoints 26-17 to 26-28

- continuous versus non-continuous** 26-21

- source line-based** 26-18

- time-based** 26-18

- transition-based** 26-18

- value-based** 26-18

buf gate 6-8

bufif gate 6-9 to 6-10

C

cache thrashing and performance 27-24

capacitance for switches 28-52

capacitive networks 3-10 to 3-13

capturing simulation data 27-26

case equality operator 4-2

case inequality operator 4-2

case statement 8-16 to 8-20

- compared to if-else-if statement** 8-17

- syntax** 8-16

- with don't-care** 8-19 to 8-20

casex 8-19

casez 8-19

cdiff statement 28-54

cells 12-1

cgo statement 28-53

changing default base in formatted output **system tasks** 21-17

channel delay timing model 28-16

characters

- specified as octal digits** 2-7

charge decay 6-39 to 6-42, 24-18, 24-28

- Switch-RC modes** 28-72

charge storage

- strength** 3-6 to 3-7, 6-18

charging_strength parameter 28-50

checkpoints 21-21

circular library scan order 24-16, 25-13

clearing

- trace** 26-17

clock generators and performance 27-14

cmos 6-13

cmos gate 6-13 to 6-14

code

- measuring** 27-9

- optimizing** 27-9

- optimizing for +speedup** 24-21

- reducing executed code** 27-24

collapsing ports 12-18 to 12-21

- chart of resulting net types** 12-20

- rules** 12-19 to 12-20

- that connect nets of different types** 12-20

C, (continued)

combinational UDPs 7-1, 7-6 to 7-8
 compared to level-sensitive sequential
 7-9
 input and output fields in state table
 7-6

combined signal strengths 6-18 to 6-32

combined signal values 6-18 to 6-32

command
 file option 24-4
 history 21-23 to 21-24
 input files 21-25

command line options
 +librescan 25-15 to 25-16

command-line restart 21-23

comments 2-2

compare
 string operation 4-20

compilation 24-1 to 24-59
 command line 24-1
 compile only option 24-4, 25-26
 compiling source files 24-3
 directives 24-26 to 24-35
 error messages 24-40 to 24-44
 performance 27-27
 user-defined primitives 7-14

concatenation
 and macro module instances 12-11
 and repetition multiplier 4-16
 and unsized numbers 4-16
 of library extensions to module and
 UDP names 25-10
 of names 12-22
 of operands 4-17
 of terms in synchronous and
 asynchronous system calls 22-3
 operator 4-2, 4-16
 performance 27-16
 string operation 4-20

concurrency
 of activity flow 8-2
 of procedures 9-7

condition
 deterministic 13-61
 non-deterministic 13-61

conditional compilation 24-45 to 24-54

conditional operator 4-2, 4-15 to 4-16
 and ambiguous results 4-15
 modeling tri-state output busses 4-16
 syntax 4-15

conditional statement 8-11 to 8-15
 syntax 8-11

conditioned event 13-60 to 13-61
 constraints 13-61
 versus unconditioned event 13-60

conflicts 3-8

connecting ports
 between modules 12-19
 by name 12-16 to 12-17
 by position with ordered list 12-15
 in macro modules 12-21
 rules 12-19 to 12-20

connection
 difference between full and parallel
 13-11 to 13-14
 full 13-11 to 13-14
 parallel 13-11 to 13-14

consistency
 in user-defined primitive state tables
 7-14

constant expression 4-1

continue 26-3

continuous assignment 5-2 to 5-9
 and \$getpattern 21-43
 and connecting ports 12-19
 and driving strength 6-17, 21-10
 and net variables 8-3
 and supply nets 3-13
 and wire nets 3-8
 driving strength of 5-9
 examples 5-3 to 5-4
 explicit declaration 5-3
 implicit declaration 5-3
 syntax 5-2
 versus procedural assignment 5-9

continuous assignments
 accelerated 5-10 to 5-33

continuous monitoring 21-12 to 21-13

control-C 26-2

C, (continued)

copy

string operation 4-20

counting number of drivers 21-30 to 21-31

cox statement 28-52

cWaves 26-1, 26-29

D

d

decimal number format 2-2

data structures 12-25

data types 3-1 to 3-19

data_event 13-39

deassign keyword 11-2 to 11-3

debug system tasks 26-6 to 26-28

\$db_breakaftertime 26-22

\$db_breakatline 26-21

\$db_breakbeforetime 26-22

\$db_breakonceonposedge 26-24

\$db_breakonnegedge 26-25

\$db_breakonposedge 26-24

\$db_breakwhen 26-23

\$db_cleartrace 26-17

\$db_deletebreak 26-26

\$db_deletefocus 26-9

\$db_disablebreak 26-27

\$db_disablefocus 26-11

\$db_enablebreak 26-26

\$db_enablefocus 26-10

\$db_help 26-6

\$db_setfocus 26-9

\$db_settrace 26-16

\$db_showbreak 26-28

\$db_showfocus 26-12

\$db_step 26-14

\$db_steptime 26-15

debugging 26-1 to 26-5

style and performance 27-25

decimal display format 2-2

and high impedance state 21-8

and unknown logic value 21-8

compatibility with \$monitor 21-8

decimal notation 3-17

declaration 13-30

declaring

events 8-28

multiple module paths in a single statement 13-14 to 13-15

parameters in specify blocks 13-3 to 13-4

decompilation 21-34 to 21-38

decompile option 24-4

default

base in formatted output 21-17

in case statement 8-16

in if-else-if statements 8-14

module path pulse control 13-34

word size 2-3

default delay mode 17-3

default library scan precedence 25-11 to 25-13

defparam 3-19, 12-7 to 12-8

and specify parameters 13-4

compared to module instance parameter value assignment 12-9

delay

calculating for high impedance (z) transitions 6-35

calculating for unknown logic value (x) transitions 6-35

control 8-25, 8-26, 21-18

distributed 13-5 to 13-7

fall 6-35

falling 6-37

for continuous assignment 5-5 to 5-8

gate 6-34 to 6-38

inertial 5-8

minimum:typical:maximum values 6-37 to 6-38

mixing distributed and module path 13-6

D

delay (*continued*)

- module path** 13-5 to 13-33
- net** 6-34 to 6-37
- propagation** 6-5, 6-35
- rise** 6-35, 6-37
- specify one value** 6-35
- specify three values** 6-35
- specify two values** 6-35
- syntax for delay control** 8-26
- trireg charge decay** 6-40 to 6-42, 24-18, 24-28
- turn-off** 6-37

delay mode selection 17-1 to 17-12

- and macro module expansion** 17-12
- and timescales** 17-5, 17-6 to 17-7
- command line plus options** 17-5
- compiler directives** 17-4
- decompiling with delay modes** 17-11
- default delay mode** 17-3
- distributed delay mode** 17-2
- overriding delay values** 17-8
- path delay mode** 17-3
- precedence** 17-5
- reasons to select a delay mode** 17-3
- the \$showmodes system task** 17-11
- the acc_fetch_delay_mode access routine** 17-11
- the parameter attribute mechanism** 17-8 to 17-9
- unit delay mode** 17-2
- zero delay mode** 17-2

delay specification 6-5

deleting

- breakpoint** 26-26
- foci** 26-9

deltal statement 28-53

deltaw statement 28-53

describing module paths 13-9 to 13-14

diagnostic messages

- from \$stop and \$finish** 21-19

directory

- library** 25-1 to 25-2

disable 26-3, 26-4

- and turning off monitoring tasks** 21-13
- named blocks** 10-1 to 10-5
- syntax** 10-1
- tasks** 10-1 to 10-5
- timing checks** 13-52
- use of** 10-1

disabling

- breakpoints** 26-27
- foci** 26-11

disabling warnings 21-38 to 21-40

displaying information 21-1 to 21-11

distributed delay mode 17-2

distributed delays and SDPDs 14-15

dominating net 12-20

don't-care bits

- in case statements** 8-19

don't-care condition

- in state table** 7-7

double quote character 2-7

drive strength specification 6-4

driving strength 6-17

- compared to charge storage strength** 21-10
- keywords** 5-9

driving_strength parameter 28-48

dynamic file selection 21-1

E

edge control specifiers 13-53 to 13-55

edge descriptors 3-18

edge transitions 13-53

edge-sensitive paths 13-30 to 13-33

- syntax** 13-30

edge-sensitive UDPs 7-9 to 7-10

- compared to level-sensitive UDPs** 7-9

element

- of array** 3-14

embedding modules 12-1, 12-3

enable 8-30

E, (continued)

enabling

breakpoints 26-26

foci 26-10

enabling tasks 9-2 to 9-4, 9-7

enabling warnings 21-40

endmodule keyword 12-1

endprimitive keyword 7-5

endtable keyword 7-5

end_edge_offset 13-39

equality operators 4-8 to 4-9

!= 4-8

== 4-8

!== 4-8

=== 4-8

and ambiguous results 4-9

and operands of different sizes 4-8

precedence 4-8

error messages 24-39

compiler 24-40 to 24-44

effect of source protection 20-16

syntax 24-41

escape sequences 21-2, 21-3

escaped identifiers 2-8

espresso format 22-5

establishing a metric 27-10

estimating model speed 27-9

event

accelerated 27-8

control 8-25, 8-27

declaration syntax 8-28

explicit 8-25

expression 8-25

implicit 8-25

in timing checks 13-39

level sensitive control 8-29

named 8-28 to 8-29

OR construct 8-29

syntax for event control 8-27

syntax of triggering statement 8-29

event control

performance 27-19

repeat 8-32 to 8-34

event-driven simulation 7-20

examples

\$hold timing check 13-42

\$monitor 21-10

\$nochange timing check 13-51

\$period timing check 13-45

\$sprinttimescale system task 16-11

\$realtime system function 16-7

\$recovery timing check 13-48

\$scale system function 16-9

\$setup timing check 13-40

\$setuphold 13-50

\$skew timing check 13-46

**\$sreadmemb to protect data it loads
into memory** 20-18

\$strobe 21-12

\$time system function 16-6

\$timeformat system task 16-13

\$width calls, legal and illegal 13-44

\$width timing check 13-43

'timescale compiler directive 16-4

"joining" events 8-40

%t format specification 16-13

accelerating specific modules 27-3

**accessing information in a protected
region** 20-10

adder

using zero-delay buf gates 13-63

AND-OR gate as user-defined primitive
7-22

AND-OR PLA 22-9 to 22-10

array with logic equations 22-4

asynchronous system call 22-3

begin-end block 8-36, 8-37

Behavior Profiler 18-17 to 18-26

behavioral description of D flip-flop
11-2

behavioral model 8-2

bit-select 4-17

bus select 5-4

**calculating delays for unknown logic
value transitions** 13-21

case statement 8-18

casex 8-19

casez in instruction decoder 8-19

E,

examples, (continued)

- changing default base in formatted output** 21-17
- combinational primitive** 7-7
- combinational UDP** 7-8
- command argument file** 24-5
- command history** 21-24
- command line** 24-1
- command line restart** 21-23
- command line using +liborder** 25-13
- command line using +librescan** 25-16
- command line with library directory option** 25-9
- command line with library file option** 25-9
- command line with library options** 25-10, 25-14, 25-16
- command line with multiple library directory file extensions** 25-10
- command line with null library file extensions** 25-10
- command line with options** 24-2
- compilation with +protect** 20-11
- conditioned events** 13-60
- connecting ports by name** 12-17
- declaring memory and registers in one statement** 3-15
- decompiling a macro module with \$list** 12-12 to 12-13
- defparam** 12-7
- delay control** 8-26
- delay mode selection** 17-10 to 17-11
- disable statement** 10-2 to 10-5
- disabling all timing checks** 13-52
- disabling the \$incpattern_write task** 21-46
- displaying unknown logic value in different radix formats** 21-8
- edge control specifiers** 13-54
- edge-sensitive paths** 13-31
- edge-sensitive UDP** 7-9
- escaped identifiers** 2-8, 2-10
- of establishing simulation time with display output** 21-18
- event OR construct** 8-29
- factorial function** 9-11
- for loop** 8-23
- for loop in multiplier** 8-24

examples, (continued)

- force and release used for debugging** 11-4
- fork-join block** 8-38
- function definition** 9-9
- global module path pulse control** 13-34, 13-35
- hierarchical name**
 - in macro module instance** 12-27
 - in module instance** 12-27
 - referencing** 12-25
- hierarchical path names** 12-23 to 12-24
- identifiers** 2-8
- if-else statement** 8-15
- incremental save and restart** 21-22
- infinite zero-delay loop** 8-43
- interactive command prompt** 26-2
- intra-assignment timing controls** 8-32
- invoking \$compare** 21-50
- invoking \$strobe_compare** 21-52
- J-K flip-flop** 7-16
- latch** 7-8
- latch module with tri-state outputs** 6-36 to 6-37
- level-sensitive latch** 27-7
- level-sensitive paths** 13-29
- level-sensitive sequential primitive** 7-8
- loading memories from text files** 21-42
- logic array personality declaration** 22-4
- macro module specification** 12-11
- manipulating strings** 2-6
- memory addressing** 4-18
- memory declaration** 3-14
- minimum:typical:maximum values** 4-23, 6-37, 6-38
- mixing level- and edge-sensitive user-defined primitives** 7-16
- module instance** 12-4, 12-5
- module instance parameter value assignment** 12-8
- module parameter declaration** 3-19
- module path declarations with polarity** 13-27
- multiplexer** 7-7, 7-8
- NAND plane system** 22-3
- NOR plane system** 22-3
- notifiers** 13-56 to 13-59

E,

examples, (continued)

- notifiers in edge sensitive UDP** 13-57 to 13-59
- numbers containing 'underlines'** 2-4
- overriding module parameter values** 12-7
- PAL16R4** 22-18 to 22-24
- PAL16R8** 22-11 to 22-17
- part-select** 4-18
- passing module parameters to tasks** 9-5
- PATHPULSES\$** 13-36
- PLA module** 22-5
- PLA system tasks** 22-6 to 22-7
- port declarations** 12-16
- predefined standard options** 24-10 to 24-11
- problem in string value padding** 4-21
- processing stimulus patterns with \$getpattern** 21-44
- protected region in a module** 20-2
- protected source description output** 20-12
- race condition** 8-31
- real numbers** 3-18
- real numbers in port connections** 12-18
- reducing pessimism in a user-defined J-K flip-flop primitive** 7-18
- reducing pessimism in a user-defined latch UDP** 7-17
- register and net declarations** 3-4, 3-6
- repeat loop in multiplier** 8-22
- response checking** 21-53 to 21-54
- SDPDs** 14-5 to 14-6
- sized constant numbers** 2-4
- source description containing VCD tasks** 19-6
- specify block** 13-2
- specify parameters** 13-3
- specifying a cell** 24-28
- specparams** 13-3
- strength outputs** 21-10
- string manipulations** 4-20
- string variable declaration** 2-5
- strings** 2-5
- synchronous PLA** 22-8
- synchronous system call** 22-3

- syntax error message** 24-41
- template of a data structure** 12-26
- testing plus options** 24-25
- text macro substitutions** 2-10
- timescales** 16-14 to 16-18
- time-sequenced waveform** 8-36 to 8-37, 8-38
- timing violation message** 13-52
- traffic light sequencer** 8-43 to 8-44
- traffic light sequencer using tasks** 9-6
- tri-state output bus** 4-16
- turn off monitoring** 21-13
- two sequential events working in parallel** 8-40
- two-channel multiplexer as user-defined primitive** 7-24
- unsized constant numbers** 2-3
- use of multi-channel descriptors** 21-16
- user-defined primitive instance** 7-14
- using \$realtobits and \$bitstoreal in port connections** 21-55
- value change dump file format** 19-17
- variable delays for synchronizing clock** 8-45
- vector XOR** 5-9
- waveform** 8-42
- while loop in counter** 8-23
- writing formatted output to files** 21-16
- zero-delay control** 8-26
- zero-delay oscillation** 27-7

execution 24-1 to 24-37

exit simulator 21-19

expansion

- of macro modules** 12-10, 12-21
- of vector nets** 3-5, 12-14

explicit event 8-25

expressions 4-1 to 4-26

- bit lengths** 4-23 to 4-26
- constant** 4-1
- self-determined** 4-25

extensions

- for files in library directories** 25-9 to 25-11

F

f

in state table 7-15, 7-21

fall delay 6-35, 6-37

file inclusion 24-54 to 24-59

files

extensions in library directories
25-9 to 25-11

extensions of protected source files
20-4

library 25-1 to 25-2

output to 21-14 to 21-17

finish time

in parallel block statements 8-39

in sequential block statements 8-39

floating license 24-1

foci 26-8 to 26-12

for loop

syntax 8-21

force keyword 11-3 to 11-4

precedence over assign 11-3

forever loop

syntax 8-21

fork-join block statement 8-35

format specifications 21-4 to 21-5

ASCII character 21-4

b or B 21-4

binary 21-4

c or C 21-4

d or D 21-4

decimal 21-4

h or H 21-4

hexadecimal 21-4

hierarchical name 21-4

m or M 21-4

net normalized voltage 21-4

net signal strength 21-4, 21-9 to 21-11

o or O 21-4

octal 21-4

s or S 21-4

string 21-4, 21-11

t or T 21-4, 21-6

time format 21-4

timescales 21-6

v or V 21-4

formats

array 22-4

of logic array personality 22-4 to 22-7

plane 22-5

formatted output system tasks 21-17

full connection 13-11 to 13-14

function

syntax 9-8

functions 9-8 to 9-11

and scope 12-31

as structured procedures 8-41

definition 8-41

purpose 9-1

returning a value 9-9

rules 9-10

syntax 9-3

syntax for function call 9-9

G

gate level modeling 6-1 to 6-44

logic gate syntax 6-3 to 6-6

gate type specification 6-4

gates

and 6-6 to 6-7

bidirectional pass 6-12

buf 6-8

bufif 6-9 to 6-10

cmos 6-13 to 6-14

compared to continuous assignments
6-1

connection list 6-6

delay 6-34 to 6-38

keywords for types 6-4

MOS 6-10 to 6-12

nand 6-6 to 6-7

nor 6-6 to 6-7

not 6-8

notif 6-9 to 6-10

notif0 6-9 to 6-10

notif1 6-9 to 6-10

or 6-6 to 6-7

pulldown 6-14

pullup 6-14

removal of names 6-43 to 6-44

syntax 6-3 to 6-6

terminal list 6-6

G

gates, (continued)

xnor 6-6 to 6-7

xor 6-6 to 6-7

ground 3-13

guidelines

for connection operators 13-13

H

H

logic 1 or high impedance state in strength format 21-9

h

hexadecimal number format 2-2

hardware

performance optimization 27-22

help

debugging 26-6

hexadecimal display format 2-2

and high impedance state 21-8

and unknown logic value 21-8

Hi

high impedance in strength format 21-9

hierarchy

and source protection 20-13 to 20-16

display of 21-27

effect of macro modules on path names 12-26

in libraries 25-9

level 12-22

name referencing 12-22 to 12-33, 21-4

of modules 12-1, 21-27

path names for defining abstract data structures 12-25

scope 12-22

scope rules for naming 12-31 to 12-33

structures 12-1 to 12-33

top level names 12-22

traversal of 21-27

high impedance state 6-16

and numbers 2-4

and trireg nets 3-9

and user-defined primitives 7-14

display formats 21-8

effect in different bases 2-4

strength display format 21-9

symbolic representation 3-1

highz0 6-5

highz1 6-5

history of commands 21-23 to 21-24

hold_limit 13-39

I

identifiers 2-7 to 2-8

definition 2-7

escaped 2-8

keywords 2-9

if-else statement

omitting else from nested if 8-12

purpose 8-11

if-else-if statement

compared to case statement 8-17

syntax 8-14

implicit

declarations 3-7, 6-15, 24-29

event 8-25

incremental pattern files 21-44 to 21-55

incremental restart 21-22

incremental save 21-21, 21-22

index

of array 3-14

of memory 3-16

inertial delay 5-8

initial 8-42

and activity flow 8-2

for specifying waveforms 8-42

syntax 8-42

initial statements

in UDPs 7-10 to 7-13

inout

port declaration 12-15

I, (continued)

input

port declaration 12-15

input file option 21-25, 21-26, 24-6
and interactive recovery 26-5
syntax 24-6

instantiation

macro module 12-11
of modules 12-1, 12-3 to 12-5

integers 3-16 to 3-17
division 4-6

interactive

control and debugging 26-1 to 26-5
mode 20-8, 21-19
prompt 26-2
recovery 26-5
source listing 21-34 to 21-38

interactive commands

continue 26-3
disable 26-3, 26-4
list of 26-3
re-execute 26-3, 26-4
step 26-3
syntax 26-3
trace-step 20-8, 26-3
where 26-3
: 26-3

interactive debugging environment
26-6 to 26-28

intermediary files 24-2

inter-module port connection 12-19

interrupt

accelerated simulation 27-8
with \$stop 26-2
with control-C 26-2

intra-assignment timing controls
8-30 to 8-34

invoking Verilog 24-1 to 24-2

K

key file 21-26 to 21-27, 27-8
and interactive recovery 26-5
-k option 21-26

key file option 24-7
-k 24-7

keywords 2-9

gate type list 6-4
rs_debug settings 28-84

L

L

logic 0 or high impedance state in
strength format 21-9

La

large capacitor in strength format 21-9

ldiff statement 28-53

left shift operator 4-2, 4-15

legal module paths

one output driver 13-25
multiple output drivers outside the
module 13-25

length parameter 28-45

level-sensitive

event control 8-29
paths 13-28 to 13-30
performance optimization of models
27-21
sequential UDPs 7-8 to 7-9
versus combinational UDP 7-9

level-sensitive UDPs

compared to edge-sensitive UDPs 7-9

lexical conventions 2-1 to 2-10

lexical token

comment 2-2
definition of 2-1
number 2-2
operator 2-1 to 2-2
types 2-1
white space 2-2

libraries 25-1 to 25-27

and 'resetall 24-33
circular scan order 25-13
controlling scan precedence 25-11
creating unique identifiers for
multiple modules and UDPs
with the same name 25-15

L

libraries, (continued)

- definition renaming** 25-27
- directories** 25-9 to 25-11
- directory file extensions** 25-9 to 25-11
- effect of source protection** 20-12
- files** 25-1 to 25-2
- forced scan precedence** 25-13 to 25-16
- how extensions interact with scan**
 - order** 25-12 to 25-14
- new scheme** 25-2 to 25-8
 - 'uselib definition of search paths** 25-3
 - search order with 'uselib paths** 25-7
 - unresolvable instantiations** 25-8
- null extensions** 25-10
- renaming definitions** 25-27
- reporting resolution paths** 25-26
- syntax checking in files** 25-26
- use of compiler directives with** 25-25

- library directory option** 24-10, 25-9
 - syntax** 24-10

- library file option** 24-9, 25-9
 - syntax** 24-9

- license** 1-4, 24-1
 - Switch-RC** 28-1

- limit** 13-39

- limitations**
 - saving simulation data** 21-23

- list of formatted output system tasks** 21-17

- loading memories from text files** 21-41 to 21-43

- log file** 21-25
 - option** 21-26, 24-7

- log file option**
 - syntax** 24-7

- logic array personality** 22-4 to 22-7
 - declaration** 22-4
 - formats** 22-4 to 22-7
 - loading** 22-4

logic gates

- and** 6-6 to 6-7
- bidirectional pass** 6-12
- buf** 6-8
- bufif** 6-9 to 6-10
- cmos** 6-13 to 6-14
- compared to continuous assignments** 6-1
- delay** 6-34 to 6-38
- MOS** 6-10 to 6-12
- nand** 6-6 to 6-7
- nor** 6-6 to 6-7
- not** 6-8
- notif** 6-9 to 6-10
- or** 6-6 to 6-7
- pulldown** 6-14
- pullup** 6-14
- syntax** 6-3 to 6-6
- xnor** 6-6 to 6-7
- xor** 6-6 to 6-7

- logic one** 3-1

- logic planes** 22-3

- logic strength modeling** 6-16 to 6-34

- logic zero** 3-1

logical operators 4-9

- !** 4-9
- &&** 4-9
- ||** 4-9
- AND** 4-2
- and ambiguous results** 4-9
- and unknown logic value** 4-9
- compared to bit-wise operators** 4-12
- equality** 4-2
- inequality** 4-2
- negation** 4-2
- OR** 4-2
- precedence** 4-9

- looping statement** 8-20 to 8-25

- for loop** 8-20
 - forever loop** 8-20
 - repeat loop** 8-20
 - while loop** 8-20

- lsb (least significant bit)** 3-5

M

macro module 12-9 to 12-11

and hierarchical names 12-26

and module parameters 12-11

and specify blocks 13-3

expansion 12-10, 12-21

and delay modes 17-12

instances containing part-selects or concatenations 12-11

instantiation 12-11

macromodule keyword 12-11

port connections in 12-21

syntax 12-11

macros

and 'define 24-30

and +define+ 24-12

defining for 'uselib 25-3

map_capacitance parameter 28-50

map_resistance parameter 28-48

mc_scan_plusargs 24-25 to 24-26

syntax 24-25

Me

medium capacitor in strength format 21-9

measuring code 27-9

memory 3-14 to 3-16

addressing 4-18

assigning values to 3-15

declaration syntax 3-14

index 3-16

limitations and performance 27-22

real number memories 3-18

reducing virtual storage requirements 6-43

using temporary registers for bit- and part-selects 4-19

messages 24-37 to 24-44

error 24-39

informational 24-39

levels 24-38 to 24-40

syntax 24-37 to 24-38

warning 24-39

minimum:typical:maximum values

delay 6-37 to 6-38

for module path delays 13-17, 13-19 to 13-20

format 4-22 to 4-23

minus sign(-)

arithmetic subtraction operator 4-2, 4-4

in state table 7-21

MIPDs 15-1 to 15-13

application 15-3, 29-3

definition 15-2, 29-3

hierarchical effects 15-6

how they work 15-5

on inputs and outputs only 15-5

specifying with PLI 15-10

unidirectionality 15-7

mnemonic strength notation 6-33

modeling

asynchronous clear/preset on an

edge-triggered D flip-flop 11-2

behavioral 8-1 to 8-45

logic strength 6-16 to 6-34

sequential circuits with simultaneous input changes 7-19

simplification for performance 27-24

modeling level and performance 27-10

module 12-1 to 12-5

and user-defined primitives 7-4

definition 12-1 to 12-2

hierarchy 12-1

instance parameter value assignment 12-8 to 12-9

instantiation 12-3 to 12-5

keyword 12-1

library definition renaming 25-27

macro 12-9 to 12-11

overriding parameter values 12-6 to 12-9

parameter dependencies 12-9

port 12-5

syntax 12-2

for specifying instantiations 12-3

terminal 12-5

top-level 12-3, 25-1

module input port delays, see MIPDs 15-1 to 15-13

M, (continued)

module parameter

- as delay** 3-19
- as width of variables** 3-19
- compared to specify parameter** 13-3, 13-4
- dependencies** 12-9
- overriding values** 12-6 to 12-9
- passing to tasks** 9-4 to 9-5
- syntax** 3-19
- use with macro modules** 12-11

module path

- definition** 13-9
- delay** 13-5 to 13-33
- delay assignment** 13-16
- description syntax** 13-10
- destination** 13-9, 13-15, 13-36
- in behavioral descriptions** 13-61 to 13-63
- polarity** 13-26 to 13-27
- source** 13-5, 13-9, 13-15, 13-36

module path pulse control

- for specific modules and paths** 13-35 to 13-36
- global** 13-33 to 13-35

modulus operator 4-2

- definition** 4-6

monitor flag 21-13

monitoring

- continuous** 21-12 to 21-13
- strobed** 21-12

MOS gate 6-10 to 6-12

- nmos** 6-12
- rnmos** 6-12

MOS strength handling 6-33

msb (most significant bit) 3-5

multi-channel descriptor 21-14

multiple drivers

- at same strength level** 6-30
- driving the same net** 3-8
- inside a module** 13-14, 13-24 to 13-25
- outside a module** 13-25

multiple library directory file extensions 25-10

multiple module path delays

- assigning in one statement** 13-14 to 13-15

multi-way decisions

- case statement** 8-16
- if-else-if statement** 8-14

N

n

- in state table** 7-15, 7-21

named blocks

- and hierarchical names** 12-22
- and scope** 12-31
- purpose** 8-39

named events 8-28 to 8-29

- used with event expressions** 8-28

names

- making multiple definitions of like-named modules and UDPs** 25-15
- unique** 25-15
- of hierarchical paths** 12-22 to 12-33

nand gate 6-6 to 6-7

negedge 13-55

net and register bit addressing 4-17 to 4-18

nets 3-2 to 3-13

- delay** 6-34 to 6-38, 28-78
- implicit declaration** 6-15
- initialization** 3-7
- names referenced in a hierarchical name** 6-43
- not driven by a source** 6-16
- removal of names** 6-43 to 6-44
- scalar** 12-19
- Switch-RC net behavior** 28-65
- syntax** 3-3 to 3-4
- triereg strength** 6-18
- types of** 3-8 to 3-13
- wired logic** 6-30

new line character 2-7, 21-3

nmos 6-10 to 6-12

node

- in hierarchical name tree** 12-22

non-blocking procedural assignment 8-4 to 8-11

N, (continued)

- evaluating assignments** 8-5
- multiple assignments** 8-9
- processing assignments** 8-11
- syntax** 8-5

nor gate 6-6 to 6-7

not gate 6-8

notif gate 6-9 to 6-10

- notif1** 6-10

notifier 13-39, 13-55 to 13-59

- argument in timing checks** 13-38
- in edge sensitive UDP** 13-56 to 13-59
- toggle values** 13-56

null

- expression** 21-2
- extensions** 25-10
- string** 4-22

numbers 2-2

- base format** 2-2
- size specification** 2-2
- unsized** 2-3

O

o

- octal number format** 2-2

octal display format 2-2

on/off control

- of monitoring tasks** 21-13

operands 4-17 to 4-22

- bit-select** 4-17
- concatenation** 4-17
- definition** 4-1
- function call** 4-17
- part-select** 4-17
- strings** 4-19 to 4-22

operators 4-2 to 4-17

- 4-2, 4-4
- + 4-2, 4-4
- ! 4-2, 4-9
- % 4-2, 4-3, 4-4
- & 4-2, 4-3, 4-4, 4-11, 4-12
- * 4-2, 4-4
- < 4-2, 4-4, 4-7
- = 5-1
- > 4-2, 4-4, 4-7
- ^ 4-2, 4-3, 4-4, 4-11, 4-12
- | 4-2, 4-3, 4-4, 4-11, 4-12
- ~ 4-2, 4-3
- != 4-2, 4-4, 4-8
- && 4-2, 4-4, 4-9
- *> 13-11 to 13-15
- << 4-2, 4-3, 4-4, 4-15
- <= 4-2, 4-4, 4-7, 8-5
- == 4-2, 4-4, 4-8
- => 13-11 to 13-15
- >= 4-2, 4-4, 4-7
- >> 4-2, 4-3, 4-4, 4-15
- ^~ 4-2, 4-3, 4-4, 4-11
- || 4-2, 4-4, 4-9
- ~& 4-2, 4-3
- ~^ 4-2, 4-3
- ~| 4-2, 4-3
- !== 4-2, 4-3, 4-4, 4-8
- === 4-2, 4-3, 4-4, 4-8
- ? : 4-2, 4-4
- { } 4-2, 4-3, 4-16
- and real numbers** 3-18
- arithmetic** 4-2, 4-5 to 4-6
- binary** 2-2, 4-4
- bit-wise** 4-3, 4-11 to 4-12
- bit-wise AND** 4-2
- bit-wise equivalence** 4-2
- bit-wise exclusive OR** 4-2
- bit-wise inclusive OR** 4-2
- bit-wise negation** 4-2
- case equality** 4-2, 4-3
- case inequality** 4-2
- concatenate** 4-3
- concatenation** 4-2, 4-16
- conditional** 4-2, 4-15 to 4-16
- definition** 2-1
- equality** 4-8 to 4-9
- left shift** 4-2
- left shift operator** 4-15
- logical** 4-9
- logical AND** 4-2

- operators, (continued)**
 - logical equality** 4-2
 - logical inequality** 4-2
 - logical negation** 4-2
 - logical OR** 4-2
 - modulus** 4-2
 - reduction** 4-3, 4-12 to 4-14
 - reduction AND** 4-2
 - reduction NAND** 4-2
 - reduction NOR** 4-2
 - reduction OR** 4-2
 - reduction XNOR** 4-2
 - reduction XOR** 4-2
 - relational** 4-2, 4-7
 - right shift** 4-2
 - right shift operator** 4-15
 - shift** 4-3, 4-15
 - ternary** 2-2
 - unary** 2-2
 - /** 4-2, 4-4
- optimization**
 - of processing stimulus patterns** 21-43
- optimizing code** 27-9
 - +speedup** 24-21
- optional arguments of system tasks** 13-38
- options**
 - invoking on command line** 24-2
 - predefined standard** 24-4 to 24-11
 - specifying on command line** 24-2
- or gate** 6-6 to 6-7
- order of library scan** 24-16
- output**
 - port declaration** 12-15
 - to files** 21-14 to 21-17
- overhead** 27-11
 - in simulation** 27-2
- overriding global module path pulse**
 - control** 13-35 to 13-36
- overriding module parameter values**
 - 12-6 to 12-9
 - assigning values in-line within module instances** 12-8 to 12-9
 - defparam** 12-7 to 12-9

P

P

- in state table** 7-15, 7-21
- parallel block statement**
 - finish time** 8-39
 - fork-join** 8-35
 - start time** 8-39
 - syntax** 8-38
- parallel connection** 13-11 to 13-14
- parameter**
 - keyword for module parameters** 13-3
 - module type** 3-19
 - syntax** 3-19
- parentheses**
 - and changing operator precedence** 4-5
- part-select**
 - and macro module instances** 12-11
 - and vector ports** 12-14
 - of vector net or register** 4-18
 - references of real numbers** 3-18
 - syntax** 4-18
- path delay mode** 17-3
- path delays**
 - combining types** 14-10
 - multiple** 14-9 to 14-14
 - scheduling** 14-16
- PATHPULSES** 13-35 to 13-36
- performance** 27-1 to 27-2, 27-9 to 27-27
 - accelerated primitives** 27-12
 - accelerating behavioral code** 24-21
 - aliases** 27-20
 - Behavior Profiler to find problems** 27-15
 - cache thrashing** 27-24
 - capturing simulation data** 27-26
 - clock generators** 27-14
 - coding tricks** 27-16
 - compilation** 27-27
 - debugging style** 27-25
 - establishing a metric** 27-10
 - estimating model speed** 27-9
 - event controls** 27-19
 - hardware** 27-22
 - keeping primitives accelerated** 27-12

P

performance, (continued)

- level-sensitive behavior**
 - modeling** 27-21
- measuring code** 27-9
- memory limitations** 27-22
- model simplification** 27-24
- modeling level** 27-10
- optimizing code** 27-9
- overhead due to switching algorithms**
27-11
- reducing executed code** 27-24
- UDPs** 27-19

personality

- memory** 22-3
- of logic array** 22-4 to 22-7

PLA devices 22-1 to 22-24

- array logic types** 22-3
- array types** 22-3
- list of system tasks** 22-2
- logic array personality declaration**
22-4
- logic array personality formats**
22-4 to 22-7
- logic array personality loading** 22-4
- syntax of system tasks** 22-2

plane

- format** 22-5
- in programmable logic arrays** 22-3

PLI interface routines

- mc_scan_plusargs** 24-25 to 24-26

plus options 24-11 to 24-24

- +autonaming** 12-29, 24-11
- +autoprotect** 20-5 to 20-7, 20-12, 24-11
- +bpi_profile** 24-12
- +caxl** 24-12
- +define+**
 - and 'define** 24-13, 24-50
 - and empty macros** 24-49
 - and library search paths** 25-3
 - and macro strings** 24-12
- +delay_mode_distributed** 24-14
- +delay_mode_path** 24-14
- +delay_mode_unit** 24-14

- +delay_mode_zero** 24-14
- +err_line_length+** 24-15
- +incdir+** 24-15
- +libext** 25-9 to 25-11
- +libext+** 24-15
- +libnonamehide** 25-22 to 25-23
- +liborder** 24-16, 25-13 to 25-15
- +librescan** 24-16, 25-15 to 25-16
- +libverbose** 24-16, 25-15, 25-26
- +maxdelays** 4-23, 6-38, 13-20, 24-17
- +max_error_count** 24-17
- +mindelays** 4-23, 6-38, 13-20, 24-17
- +noaccerr** 24-18
- +nolibcell** 24-18
- +notimingchecks** 13-52
- +no_charge_decay** 24-18
- +no_cond_event_error** 24-18
- +no_notifier** 24-19
- +no_pulse_msg** 13-34, 24-19
- +pre_16a_paths** 14-15, 24-20
- +protect** 20-3 to 20-5, 20-12,
20-18 to 20-19, 24-20
- +pulse_e/n** 13-33, 24-21
- +pulse_r/m** 13-33 to 13-35, 24-21
- +rswrctostr or +rsw_rc_to_str** 24-21
- +rsw_opt_stack** 24-21
- +speedup** 24-21, 27-24
- +switchres or +switch_res** 24-23, 28-27
- +switchxl** 24-23, 28-7
- +sxl_keep_all** 24-23
- +sxl_keep_declared** 24-24
- +sxl_keep_minimum** 24-24
- +sxl_unidirect** 24-23, 28-19
- +typdelays** 4-23, 6-38, 13-20, 24-24
- no error checking** 24-26
- predefined** 24-11 to 24-24
- specifying on command line** 24-2
- testing** 24-25 to 24-26
- user-defined** 24-2, 24-24 to 24-26

plus plus sign(++)

- to specify null extensions** 25-10

plus sign(+)

- arithmetic addition operator** 4-2, 4-4
- separator for +libext arguments** 25-9

pmos 6-10 to 6-12

polarity 13-26 to 13-27

- negative** 13-26
- positive** 13-26 to 13-27
- unknown** 13-26

P, (continued)

port 12-14 to 12-22

collapsing 12-18

connecting

by name 12-16 to 12-17

by position with ordered list 12-15

in macro modules 12-21

rules for 12-19 to 12-20

declaration 12-15

definition 12-14

inter-module connections 12-19

module 12-5

of user-defined primitives 7-5

rules for collapsing 12-19 to 12-20

posedge 13-55

power supplies

modeled by supply nets 3-13

precedence

binary operators 4-4

equality operators 4-8

logical operators 4-9

relational operators 4-8

predefined plus options 24-11 to 24-24

+autoprotect 24-11

+bpi_profile 24-12

+caxl 24-12

+define+

and 'define 24-13, 24-50

and empty macros 24-49

and library search paths 25-3

and macro strings 24-12

+delay_mode_distributed 24-14

+delay_mode_path 24-14

+delay_mode_unit 24-14

+delay_mode_zero 24-14

+err_line_length+ 24-15

+incdir 24-56

+incdir+ 24-15

+libext 25-9 to 25-11

+libext+ 24-15

+libnonamehide 24-16, 25-22 to 25-23

+liborder 24-16, 25-13 to 25-15

+librescan 24-16, 25-15 to 25-16

+libverbose 24-16, 25-26

+maxdelays 4-23, 6-38, 24-17

+max_error_count 24-17

+mindelays 4-23, 6-38, 24-17

+nolibcell 24-18

predefined plus options, (continued)

+no_charge_decay 24-18

+no_cond_event_error 24-18

+no_notifier 24-19

+no_pulse_msg 13-34, 24-19

+pre_16a_paths 14-15, 24-20

+protect 24-20

+pulse_e/n 24-21

+pulse_r/m 24-21

+rswrctostr or **+rsw_rc_to_str** 24-21

+rsw_opt_stack 24-21

+speedup 24-21, 27-24

+switchres or **+switch_res** 24-23, 28-27

+switchxl 24-23, 28-7

+sxl_keep_all 24-23

+sxl_keep_declared 24-24

+sxl_keep_minimum 24-24

+sxl_unidirect 24-23, 28-19

+typdelays 4-23, 6-38, 24-24

predefined standard options 24-4 to 24-11

-a 24-4

-d 24-4

-f 24-4

-i 24-6

-k 24-7

-l 24-7

-q 24-7

-r 24-8

-s 24-8

specifying on command line 24-2

-t 24-8

-u 24-8

-v 24-9

-w 24-9

-x 24-9

-y 24-10

primitive instance identifier 6-6

primitive keyword 7-4

printing command history 21-23 to 21-24

probabilistic distribution functions

 23-5 to 23-6

\$dist_chi_square 23-5

\$dist_erlang 23-5

\$dist_exponential 23-5

\$dist_normal 23-5

\$dist_poisson 23-5

\$dist_t 23-5

\$dist_uniform 23-5

P, (continued)

- procedural assignment** 8-3 to 8-4
 - and integers** 3-16
 - and time variables** 3-16
 - blocking** 8-4
 - non-blocking** 8-4 to 8-11
 - versus continuous assignment** 5-9
- procedural continuous assignments**
 - 11-1 to 11-4
 - assign** 11-2 to 11-3
 - deassign** 11-2 to 11-3
 - definition** 11-1
 - force** 11-3 to 11-4
 - precedence** 11-3
 - release** 11-3 to 11-4
 - syntax** 11-1
- procedural statements**
 - in behavioral models** 8-1
- procedural timing controls** 8-25 to 8-34
 - delay control** 8-26 to 8-27
 - event control** 8-25
 - fork-join block** 8-39
 - intra-assignment timing controls**
 - 8-30 to 8-34
 - zero-delay control** 8-26
- procedure**
 - always statement** 8-41
 - function** 8-41
 - initial statement** 8-41
 - task** 8-41
- programmable logic arrays** 22-1 to 22-24
 - list of system tasks** 22-2
 - logic types** 22-3
 - personality**
 - declaration** 22-4
 - formats** 22-4 to 22-7
 - loading** 22-4
 - syntax of system tasks** 22-2
 - types** 22-3
- propagation delay**
 - for gates and nets** 6-35
 - in logic gate syntax** 6-5
- protection**
 - of data in memory** 21-56
- Pu**
 - pull drive in strength format** 21-9

- pull0** 6-5
- pull1** 6-5
- pulldown source** 6-14
- pullup source** 6-14

Q

- qualified paths** 13-28 to 13-33
 - edge-sensitive** 13-30 to 13-33
 - level-sensitive** 13-28 to 13-30
- queue management** 23-1 to 23-4
 - \$q_add** 23-1, 23-2
 - \$q_exam** 23-1, 23-3
 - \$q_full** 23-1, 23-2
 - \$q_initialize** 23-1
 - \$q_remove** 23-1, 23-2
 - queueing tasks and \$restart** 23-4
 - queueing tasks and \$save** 23-4
 - status codes** 23-3
 - status parameters** 23-4
- queueing models** 23-1
- quiet option** 24-7

R

- r**
 - in state table** 7-15, 7-21
- race condition** 8-31, 27-8
- random access memory(RAM)**
 - modeled by register arrays** 3-14
- random number generator** 21-19, 23-5
- range**
 - syntax** 3-5
- rcmos** 6-13
- reading input commands from a file** 21-25
- read-only memory(ROM)**
 - modeled by register arrays** 3-14
- real numbers** 3-17 to 3-18, 21-55
 - and operators** 3-18
 - conversion to integers** 3-18
 - format specifications used with** 21-5
 - in port connections** 12-18

R

operators with real number operands 4-3
 specifying 3-17
 syntax 3-17

recursive -f options 24-4

recursive task calls 9-7

reducing pessimism 7-17 to 7-18, 8-18

reduction operators 4-12 to 4-14

& 4-2, 4-12

^ 4-12

| 4-12

~& 4-2

exclusive OR 4-12

inclusive OR 4-2, 4-12

syntax restrictions 4-14

unary AND 4-2, 4-12

unary NAND 4-2, 4-13

unary NOR 4-2, 4-13

XNOR 4-2

XOR 4-2

redundancy

in user-defined primitive state tables
 7-14

re-execute 26-3, 26-4

reference_event 13-39

registers 3-2 to 3-5

and level-sensitive sequential UDPs
 7-8

declaration syntax 3-14

for modeling memories 3-14

notifier 13-55

syntax 3-3 to 3-4

used in procedural assignments 5-10

relational operators 4-2, 4-7

< 4-7

> 4-7

<= 4-7

>= 4-7

and unknown bit values 4-8

precedence 4-8

release keyword 11-3 to 11-4

repeat event control 8-32 to 8-34

repeat loop

syntax 8-21

repetition multiplier 4-16

replaying a simulation run 26-5

reporting

non-xl structures 27-5 to 27-6

resistive devices

modeled with tri0 and tri1 nets 3-13

resolving modules and UDPs 25-9, 25-10,
 25-11 to 25-16, 25-26

restart file option 21-23

and -c 24-4

and compile only option 24-4

and interactive recovery 26-5

syntax 24-8

restarting

from command line 21-23

from full save 21-21

from incremental save 21-22

the simulator 21-21 to 21-23

restrictions on data types

in continuous assignments 5-1, 5-10,
 12-19

in port collapsing 12-18 to 12-19

in procedural assignments 5-1, 5-10,
 8-3

when connecting ports 12-19

restrictions on interactive commands
 26-2

right shift operator 4-2, 4-15

rise delay 6-35, 6-37

rnmos 6-10 to 6-12

rpmos 6-10 to 6-12

rs_debug keyword 28-84

rtran 6-12

rtranif0 6-12

rtranif1 6-12

rules

for describing module paths 13-14

running XL 27-2 to 27-3

S

s

- in string display format** 21-11
- saving simulation data** 21-21 to 21-23
 - limitations** 21-23
- scalared keyword** 3-5
- scalars**
 - compared to vectors** 3-5
 - scalar nets and driving strength of continuous assignment** 5-9
- scanning libraries** 24-16
- scheduling path delays** 14-16
- scientific notation** 3-17
- scope**
 - and hierarchical names** 12-22
 - rules** 12-31 to 12-33
- SDPDs** 14-1 to 14-18
 - and multiple path delays** 14-9
 - as unconditional delays** 14-15
 - combined with other delays** 14-10
 - effects of unknowns** 14-11
 - internal logic effects** 14-13
 - PLI back annotation** 14-18
 - scheduling** 14-16
 - simulating as unconditional paths** 24-32
 - Veritime considerations** 14-17
- SDPDs and distributed delays** 14-15
- seed** 23-5
- self-determined expression** 4-25
- sequential block statement** 8-35 to 8-37
 - finish time** 8-39
 - start time** 8-39
 - syntax** 8-35
- sequential UDP initialization** 7-10 to 7-13
- sequential UDPs**
 - input and output fields in state table** 7-6
- set of values (0, 1, x, z)** 3-1
- setting**
 - foci** 26-8 to 26-12
 - trace** 26-16
- setup_limit** 13-39
- shift operators** 4-15
 - <<** 4-15
 - >>** 4-15
- showing**
 - breakpoints** 26-28
 - foci** 26-12
- simulating module path delays**
 - one path output driving another** 13-22 to 13-23
 - propagating strength changes on paths** 13-23
 - when driving wired logic** 13-24 to 13-25
- simulation**
 - capturing data** 27-26
 - command line** 24-1
 - event-driven** 7-20
 - going back with incremental restart** 21-22
 - list of activities** 27-1
 - overhead** 27-2
 - response** 27-1
 - simulation time and timing controls** 8-25
 - stimulus** 27-1
 - time** 21-17 to 21-18
- size of displayed data** 21-6 to 21-7
- sized numbers** 2-2
- Sm**
 - small capacitor in strength format** 21-9
- source**
 - pulldown** 6-14
 - pullup** 6-14
- source description file** 24-1
- source protection** 19-7, 20-1 to 20-19
 - accessing protected information** 20-8 to 20-12
 - affect on libraries** 20-12
 - affect on simulation** 20-8 to 20-12
 - displaying hierarchical path names** 20-13 to 20-16
 - effect of timing checks** 20-16
 - error messages** 20-16
 - file extensions** 20-4

S

source protection, (continued)

- protecting all modules and UDPs in a source description** 20-5 to 20-7
- protecting data in memory** 20-17
- protecting selected regions** 20-1 to 20-5

specify block 13-1 to 13-63

specify block system tasks

- \$hold** 13-41
- \$nochange** 13-51 to 13-52
- \$period** 13-44
- \$recovery** 13-47
- \$setup** 13-40
- \$setuphold** 13-48
- \$skew** 13-45
- \$width** 13-42

specify parameter 13-3 to 13-4

specify parameters

- as run time constant in specify block** 13-2

specifying transition delays on module

- paths** 13-17 to 13-21
- assigning one value** 13-17
- assigning six values** 13-19
- assigning three values** 13-18
- assigning two values** 13-18
- x transitions** 13-20 to 13-21

specparam 13-3 to 13-4

- syntax** 13-3
- versus module parameter** 13-4

St

- strong drive in strength format** 21-9

stack optimization 28-79

standard output 21-14

start time

- in parallel block statements** 8-39
- in sequential block statements** 8-39

start_edge_offset 13-39

state dependent path delays 14-1 to 14-18

state dependent path delays and distributed delays 14-15

status

- of expanded nets** 21-29
- of module ports** 21-29
- of variables** 21-28 to 21-29

step 26-3

stepping 26-12 to 26-17

- in time** 26-14, 26-15

stochastic analysis 23-1 to 23-6

- probabilistic distribution functions** 23-5 to 23-6
- queue management** 23-1 to 23-4

stop option 24-8

strength 6-4 to 6-5

- ambiguous** 6-20 to 6-32
- and logic conflicts** 3-8
- and MOS gates** 6-33
- and scalar net variables** 3-1
- charge storage** 6-18
- driving** 6-17
- gates that accept specifications** 6-4
- of combined signals** 6-18 to 6-32
- on trireg nets** 3-9
- range of possible values** 6-21
- reduction by non-resistive devices** 6-33
- reduction by resistive devices** 6-33
- reduction table** 6-33
- scale of strengths** 6-18
- supply net** 6-34
- trace messages** 6-33
- tri0** 6-34
- tri1** 6-34
- trireg** 6-34

strength display format 21-9 to 21-11

- high impedance** 21-9
- large capacitor** 21-9
- logic value 0,1,H,L,X,Z** 21-9
- medium capacitor** 21-9
- pull drive** 21-9
- small capacitor** 21-9
- strong drive** 21-9
- supply drive** 21-9
- weak drive** 21-9

S, (continued)

strings 2-5 to 2-7, 4-19 to 4-22

and specifying file name arguments
 21-1

definition 2-5

display format 21-4, 21-11

in vector variables 4-20

manipulation 2-6

operations 4-20

padding 2-6

special characters 2-7

value padding 4-20 to 4-21

variable declaration 2-5

strobed monitoring 21-12

strong0 6-5

strong1 6-5

structured procedure 8-41 to 8-45

always statement 8-41

function 8-41

initial statement 8-41

task 8-41

Su

supply drive in strength format 21-9

supply net strength 6-34

supply nets 3-13

supply0 6-5

supply1 6-5

Switch XL

**conversion of channel delay to turn
 on/turn off delay** 28-16

switches

MOS 6-10 to 6-12

switch-level simulation 28-1 to 28-98

algorithms 28-1 to 28-98

algorithms' major features 28-4

networks 28-2

Switch-RC 28-35 to 28-98

algorithm 28-35, B-1 to B-23

boundaries 28-73 to 28-77

continuous assignments 28-75

fanins 28-73

fanouts 28-77

logic gates 28-73

signal transition slopes 28-98

switches 28-76

cdiff statement 28-54

cgo statement 28-53

charge decay, default 28-72

charge decay, DRM 28-72

controllable behavior 28-71

cox statement 28-52

debugging 28-81 to 28-97

dc setting for \$rs_trace_net task
 28-89

escape sequences %n and %v 28-81

**event setting for \$rs_trace_net
 task** 28-88

spike setting for \$rs_trace_net task
 28-93

tau setting for \$rs_trace_net task
 28-91

tau2 setting for \$rs_trace_net task
 28-93

the \$options task 28-83

the \$rs_get_net task 28-97

the \$rs_showpaths task 28-95

the \$rs_trace_net task 28-85

the \$rs_untrace_net task 28-85

default statement 28-43

defining technologies 28-42 to 28-59

example 28-58

deltal statement 28-53

deltaw statement 28-53

design description 28-41

differences from Verilog-XL 28-98

example netlists 28-67

force statement 28-98

highthresh statement 28-44

how the algorithm works
 28-35 to 28-40

invoking globally 28-7

ldiff statement 28-53

licensed separately 28-1

limitations 28-40

lowthresh statement 28-43

mapcap statement 28-49

charging_strength 28-50

default 28-51

examples 28-51

map_capacitance 28-50

methodology 28-51

mapres statement 28-47

default 28-49

driving_strength 28-48

examples 28-49

S

Switch-RC, *(continued)*

- map_resistance** 28-48
- methodology** 28-48
- modes** 28-71
 - and charge decay** 28-72
- name statement** 28-43
- net behavior** 28-65
- net instantiation** 28-64
 - examples** 28-66
- net model** 28-37
- network model** 28-40
- overriding delays** 28-78
 - with net delays** 28-78
 - with unit delays** 28-79
 - with zero delays** 28-79
- PLI** 28-98
- release statement** 28-98
- resistance statement** 28-44
 - context** 28-45
 - defaults** 28-47
 - examples** 28-46
 - length** 28-45
 - switch_resistance** 28-46
 - type** 28-44
 - width** 28-45
- stack optimization** 28-79
- switch capacitance**
 - and logic gates** 28-57
 - calculations** 28-55
 - parameters** 28-52
- switch instantiation** 28-59
 - examples** 28-63
 - types** 28-62
- switch model** 28-35
- technologies** 28-42 to 28-59
- technology characterization**
 - C-1 to C-48
- VCL** 28-98
- wired logic** 28-98
- xa statement** 28-54

Switch-XL 28-16 to 28-33

- default charge and drive strength**

- 28-29

- enabling** 28-7
- net removal** 28-20
- optimization** 28-20 to 28-26
- purpose** 28-5
- strength mapping** 28-31
- strength reduction** 28-30

Switch-XL algorithm 28-16 to 28-33

Switch-XL strength model 28-26 to 28-33

switch_resistance parameter 28-46

symbolic debugging 26-1 to 26-5

- and hierarchical name referencing**
12-25

synchronous arrays 22-3

syntax

- \$compare** 21-49
- \$countdrivers** 21-30
- \$db_breakaftertime** 26-22
- \$db_breakatline** 26-21
- \$db_breakbeforetime** 26-22
- \$db_breakonceatline** 26-21
- \$db_breakonceonnegedge** 26-25
- \$db_breakoncewhen** 26-23
- \$db_breakonnegedge** 26-25
- \$db_breakonposedge** 26-24
- \$db_breakwhen** 26-23
- \$db_cleartrace** 26-17
- \$db_deletebreak** 26-26
- \$db_deletefocus** 26-9
- \$db_disablebreak** 26-27
- \$db_disablefocus** 26-11
- \$db_enablebreak** 26-26
- \$db_enablefocus** 26-10
- \$db_help** 26-6
- \$db_setfocus** 26-9
- \$db_settrace** 26-16
- \$db_showbreak** 26-28
- \$db_showfocus** 26-12
- \$db_step** 26-14
- \$db_steptime** 26-15
- \$disable_warnings** 21-38
- \$display** 21-1
- \$dist_chi_square** 23-5
- \$dist_erlang** 23-5
- \$dist_exponential** 23-5
- \$dist_normal** 23-5
- \$dist_poisson** 23-5
- \$dist_t** 23-5

S

syntax, (continued)

- \$dist_uniform** 23-5
- \$enable_warnings** 21-40
- \$fclose** 21-14
- \$fdisplay** 21-14
- \$finish** 21-18
- \$fmonitor** 21-14
- \$fopen** 21-14
- \$fstrobe** 21-14
- \$fwrite** 21-14
- \$getpattern** 21-43
- \$history** 21-23
- \$incpattern_read** 21-47
- \$incpattern_write** 21-45
- \$incsave** 21-21
- \$keepcommands** 21-34
- \$key** 21-26
- \$list** 21-34, 21-35
- \$listcounts** 21-35
- \$list_forces** 21-36
- \$log** 21-25
- \$monitor** 21-12
- \$monitoroff** 21-12
- \$monitoron** 21-12
- \$nochange** 13-51
- \$nokey** 21-26
- \$nolog** 21-25
- \$options** 28-83
- \$q_add** 23-1
- \$q_exam** 23-1
- \$q_full** 23-1
- \$q_initialize** 23-1
- \$q_remove** 23-1
- \$random** 21-19
- \$readmemb** 21-41
- \$readmemh** 21-41
- \$recovery** 13-47
- \$reportprofile** 21-59
- \$reset** 21-61
- \$restart** 21-21
- \$save** 21-21
- \$scope** 21-27
- \$setup** 13-40
- \$setuphold** 13-49
- \$showallinstances** 21-27
- \$showexpandednets** 21-29
- \$showmodes** 21-34
- \$shownonxl** 27-6
- \$showportsnotcollapsed** 21-29

- \$showscopes** 21-27
- \$showvariables** 21-28
- \$showvars** 21-28
- \$startprofile** 21-58
- \$stime** 21-17
- \$stop** 21-18
- \$stopprofile** 21-59
- \$strobe** 21-12
- \$strobe_compare** 21-51
- \$test\$plusargs** 24-25
- \$time** 21-17
- \$width** 13-42
- \$write** 21-1
- 'default_nettype** 6-15
- +define+** 24-12
- +libext+** 24-15
- always** 8-43
- assign** 11-1
- behavioral statements** A-8 to A-9
- case statement** 8-16
- conditional operator** 4-15
- conditional statement** 8-11
- conditioned event** 13-60
- continuous assignment** 5-2
- deassign** 11-1
- declarations** A-5 to A-6
- declaring events** 8-28
- delay control** 8-26
- disable statement** 10-1
- edge control specifiers** 13-54
- edge-sensitive paths** 13-30
- errors and <-** 24-40
- event control** 8-27
- event triggering statement** 8-29
- expressions** A-13 to A-15
- for addressing memory** 4-18
- for enabling tasks** 9-4
- for loop** 8-21
- force** 11-1
- forever loop** 8-21
- formal definition** A-1 to A-17
- function** 9-3, 9-8
- function call** 9-9
- general** A-15
- hold** 13-41
- i option** 24-6
- if-else-if statement** 8-14
- initial statement** 8-42
- integer declaration** 3-16
- interactive commands** 26-3
- k option** 24-7

S

syntax, (continued)

- l option** 24-7
- level-sensitive paths** 13-28
- logic gates** 6-3 to 6-6
- macro module** 12-11
- mc_scan_plusargs** 24-25
- memory declaration** 3-14
- module** 12-2
- module instantiation** 12-3, A-7
- module parameter** 3-19
- module path delay assignment** 13-16
- module path description** 13-10
- net declaration** 3-3 to 3-4
- parallel block statement** 8-38
- part-select** 4-18
- PATHPULSE\$** 13-35
- period** 13-44
- port**
 - declaration** 12-15
 - definition** 12-14
- primitive instances** A-6
- procedural continuous assignments**
 - 11-1
- r option** 24-8
- range** 3-5
- real numbers** 3-17
- register declaration** 3-3 to 3-4, 3-14
- release** 11-1
- repeat loop** 8-21
- SDPD** 14-2
- sequential block statement** 8-35
- skew** 13-46
- source text** A-2 to A-4
- specify block** 13-2
- specify parameter** 13-3
- specify section** A-10 to A-12
- specparam** 13-3
- state dependent path delays** 14-2
- switch-level modeling** A-16
- Switch-RC net instantiation** 28-64
- Switch-RC switch instantiation** 28-59
- Switch-XL strength model**
 - 28-27 to 28-28
- task** 9-3
- text macro**
 - definitions** 2-9
 - usage** 2-9
- time variable declaration** 3-16
- UDPs** 7-3 to 7-4

user-defined primitives 7-3 to 7-4

-v option 24-9

wait statement 8-29

while loop 8-21

-y option 24-10

system tasks 21-1 to 21-69

effect of source protection

20-8 to 20-12

file name arguments 21-1

for changing base in formatted output
21-17

for continuous monitoring 21-12 to
21-13

for displaying information 21-1 to
21-11

for displaying the delay mode 21-34

for fetching simulation time

21-17 to 21-18

for generating key files 21-26 to 21-27

for generating random numbers 21-19

for interrupting the simulator

21-18 to 21-19

for loading memories from text files

21-41 to 21-43

for printing command history

21-23 to 21-24

for processing stimulus patterns faster

21-43

for producing an interactive source

listing 21-34 to 21-38

for reading input commands from a file

21-25

for resetting Verilog-XL 21-59 to 21-69

for restarting the simulator

21-21 to 21-23

for running the behavior profiler

21-58 to 21-59

for saving simulation data

21-21 to 21-23

for showing hierarchy 21-27

for showing module port status 21-29

for showing number of drivers

21-30 to 21-31

for showing status of expanded nets

21-29

for showing variable status

21-28 to 21-29

for storing interactive commands

21-34

S

syntax, (continued)

- for writing formatted output to files**
21-14 to 21-17
- generating a checkpoint in the value change dump file** 19-5
- limiting the size of the value change dump file** 19-5
- list of formatted output system tasks**
21-17
- reading the value change dump file during a simulation** 19-6
- resuming the dump into the value change dump file** 19-4 to 19-5
- showing the timescale of a module**
16-10 to 16-11
- specifying how %t reports time information** 16-11 to 16-14
- specifying the name of the value change dump file** 19-3
- specifying the time unit of delays entered interactively**
16-11 to 16-14
- specifying the variables to be dumped in the value change dump file**
19-3 to 19-4
- stopping the dump into the value change dump file** 19-4 to 19-5
- system tasks' optional arguments** 13-38

T

t

- timescale format** 16-10, 21-6, 21-56
- tab character** 2-7
- table keyword** 7-5
- tasks** 9-1 to 9-11
 - and hierarchical names** 12-22
 - and scope** 12-31
 - as structured procedures** 8-41
 - definition** 8-41
 - disabling within a nested chain** 10-1
 - enabling** 9-2 to 9-4, 9-7
 - passing parameters** 9-4 to 9-5
 - purpose** 9-2
 - syntax** 9-3
 - for enabling** 9-4

technology characterization for Switch-RC C-1 to C-48

terminal

- in logic gate syntax** 6-6
- module** 12-5

ternary operators

- ?:** 4-4

text macro substitutions 2-9 to 2-10

- and 'define** 24-30
- and 'undef** 24-35
- and +define+** 24-12
- definition syntax** 2-9
- in interactive mode** 2-9
- redefinition** 2-10
- usage syntax** 2-9

threshold 13-39

time 21-17 to 21-18

- and incremental restart** 21-22
- arithmetic operations performed on time variables** 3-17
- simulation** 8-25
- variables** 3-16

time precision 16-2

time unit 16-2

timescales 16-1 to 16-18, 21-55

timing checks 13-37 to 13-61

- \$hold** 13-41
- \$nochange** 13-51 to 13-52
- \$period** 13-44
- \$recovery** 13-47
- \$setup** 13-40
- \$setuphold** 13-48
- \$skew** 13-45
- \$width** 13-42
- and detecting simultaneous input transitions** 7-20
- arguments** 13-39
- data_event** 13-38, 13-39
- disabling** 13-52
- end_edge_offset** 13-38, 13-39
- hold_limit** 13-38, 13-39

T

timing checks, (continued)
 in behavioral descriptions 13-61 to 13-63
 limit 13-38, 13-39
 list of system tasks 13-38
 notifier 13-38, 13-39
 reference_event 13-38, 13-39
 setup_limit 13-38, 13-39
 start_edge_offset 13-38, 13-39
 threshold 13-38, 13-39

timing violation messages 13-52

top-level module 12-3, 25-1

trace
 \$cleartrace 21-20
 \$settrace 12-12, 21-20
 and acceleration 27-8
 option 24-8
 single step 12-12

trace-step 26-3

tracing 26-14

tran 6-12

tranif0 6-12

tranif1 6-12

transistors 6-12

transitions
 01 7-9
 order for module path delay assignment 13-19
 unspecified 7-10

tree structure
 of hierarchical names 12-22

tri nets 3-8 to 3-13

trireg
 and charge storage strength 6-18
 vectored keyword inapplicable 3-6, 3-9

turn on/turn off delay timing model 28-16

turn-off delay 6-37

types of nets

supply nets 3-13
tri nets 3-8, 6-34
tri0 3-13, 6-34
tri1 3-13, 6-34
triand 3-8
trior 3-8
trireg 3-9, 6-34, 21-10
wire 3-8
wired AND 3-8
wired logic 6-30
wired OR 3-8

U

UDPs 7-1 to 7-24
 - in state table 7-21
 *** in state table** 7-21
 ? in state table 7-21
 *** symbol** 7-15
 (??) in state table 7-21
 (01) in state table 7-21
 (0x) in state table 7-21
 (10) in state table 7-21
 (1x) in state table 7-21
 (vw) in state table 7-21
 (x1) in state table 7-21
 0 in state table 7-21
 1 in state table 7-21
 and memory considerations 7-2
 and performance 7-1
 b in state table 7-21
 b symbol 7-15
 combinational UDPs 7-6 to 7-8
 compilation 7-14
 definition 7-4 to 7-6
 edge-sensitive UDPs 7-9 to 7-10
 f in state table 7-21
 f symbol 7-15
 instances 7-14
 level-sensitive dominance 7-19, 7-19
 level-sensitive sequential UDPs
 7-8 to 7-9
 mixing level- and edge-sensitive descriptions 7-16 to 7-17
 n in state table 7-21
 n symbol 7-15
 p in state table 7-21
 p symbol 7-15
 performance 27-19
 ports 7-5

U

UDPs, (continued)

- processing simultaneous input changes** 7-19
- r in state table** 7-21
- r symbol** 7-15
- reducing pessimism** 7-17 to 7-18
- state table** 7-5 to 7-6
- summary of symbols in state table** 7-21
- syntax** 7-3 to 7-4
- table of memory requirements** 7-2
- x in state table** 7-21

ULM 24-1

unary operators

- !** 4-9
- &** 4-12
- ^** 4-12
- |** 4-12
- ~** 4-11
- <<** 4-15
- >>** 4-15

unconnected port 12-5

underline character 2-4

unit delay mode 17-2, 28-79

unknown logic value

- and numbers** 2-4
- display formats** 21-8
- effect in different bases** 2-4
- in state table** 7-6, 7-10, 7-21
- symbolic representation** 3-1

unsized numbers 2-3

unspecified transitions 7-10

upper case option 24-8

upwards name referencing 12-27 to 12-33

user-defined options 24-24 to 24-26

- no error checking** 24-26
- specifying on command line** 24-2
- testing** 24-25 to 24-26

user-defined primitives 7-1 to 7-24

- in state table** 7-21
- * in state table** 7-21
- ? in state table** 7-21
- * symbol** 7-15
- (??) in state table** 7-21
- (01) in state table** 7-21
- (0x) in state table** 7-21
- (10) in state table** 7-21
- (1x) in state table** 7-21
- (vw) in state table** 7-21
- (x1) in state table** 7-21
- 0 in state table** 7-21
- 1 in state table** 7-21
- and memory considerations** 7-2
- and performance** 7-1
- b in state table** 7-21
- b symbol** 7-15
- combinational** 7-6 to 7-8
- compilation** 7-14
- definition** 7-4 to 7-6
- edge-sensitive** 7-9 to 7-10
- f in state table** 7-21
- f symbol** 7-15
- instances** 7-14
- level-sensitive dominance** 7-19
- level-sensitive sequential** 7-8 to 7-9
- mixing level- and edge-sensitive descriptions** 7-16 to 7-17
- n in state table** 7-21
- n symbol** 7-15
- p in state table** 7-21
- p symbol** 7-15
- ports** 7-5
- processing simultaneous input changes** 7-19
- r in state table** 7-21
- r symbol** 7-15
- reducing pessimism** 7-17 to 7-18
- state table** 7-5 to 7-6
- summary of symbols in state table** 7-21
- syntax** 7-3 to 7-4
- table of memory requirements** 7-2
- x in state table** 7-21

V

-v

syntax 24-9

value change dump file 19-1 to 19-20

contents 19-8

creating 19-2 to 19-7

effect of source protection 19-7

format 19-8 to 19-20

example 19-17 to 19-18

formats of variable values 19-9 to 19-10

generating a checkpoint 19-5

keyword commands 19-10 to 19-15

\$comment 19-11

\$date 19-11

\$dumpall 19-14

\$dumpoff 19-15

\$dumpon 19-15

\$dumpvars 19-14

\$enddefinitions 19-14

\$scope 19-13

\$timescale 19-15

\$upscope 19-14

\$var 19-12

\$version 19-12

limiting the size 19-5

reading the value change dump file

during a simulation 19-6

resuming the dump 19-4 to 19-5

specifying the name 19-3

specifying the variables to be dumped
19-3 to 19-4

stopping the dump 19-4 to 19-5

structure 19-8

syntax of VCD file 19-16

value set (0, 1, x, z) 3-1

values

of combined signals 6-18 to 6-32

Vcc 3-13

VCD file

syntax 19-16

Vdd 3-13

vectored keyword 3-5

vectors 3-5

and timing violations 13-53

and vector net expansion 3-5, 24-9

Vss 3-13

W

wait statement

as level-sensitive event control 8-29

syntax 8-29

to advance simulation time 8-25

warning messages 24-39

and -w 24-9

enabling and disabling 21-38 to 21-40

warning suppression option 12-32, 24-9

warnings

disabling 21-38 to 21-40

enabling 21-40

We

weak drive in strength format 21-9

weak0 6-5

weak1 6-5

where 26-3

while loop

syntax 8-21

white space 2-2

width parameter 28-45

wired logic nets

wand 6-30

wired-AND configurations 3-8

wired-OR configurations 3-8

wor 6-30

wires 3-8

word

of array 3-14

writing formatted output to files

21-14 to 21-17

X

X

as display format for unknown logic value 21-8
unknown logic value in strength format 21-9

x

as display format for unknown logic value 21-8
in state table 7-6, 7-21
unknown logic value 3-1

xa statement 28-54

XL option 27-1 to 27-27

and 'accelerate 24-27

and key files containing asynchronous interrupts 27-8

and specify blocks 13-3

and tracing 27-8

compared to normal simulation
27-7 to 27-8

list of items that cannot be accelerated 27-4 to 27-5

potential problems 27-8 to 27-9

primitives and scalar nets that can be accelerated 27-3 to 27-4

processing simultaneous events
27-7 to 27-8

running XL 27-2 to 27-3

when pulse width equals gate delay
27-8

xnor gate 6-6 to 6-7

xor gate 6-6 to 6-7

Z

Z

as display format for high impedance state 21-8
high impedance state in strength format 21-9

z

as display format for high impedance state 21-8
high impedance state 3-1

zero delay mode 17-2, 28-79

zero-delay

control 8-26

oscillation 27-7 to 27-8