

FPGA Compiler II /
FPGA *Express*
Verilog HDL
Reference Manual

Version 1999.05, May 1999

Comments?

E-mail your comments about Synopsys
documentation to doc@synopsys.com

SYNOPSYS®

Copyright Notice and Proprietary Information

Copyright © 1999 Synopsys, Inc. All rights reserved. This software and documentation are owned by Synopsys, Inc., and furnished under a license agreement. The software and documentation may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

“This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____.”

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks

Synopsys, the Synopsys logo, BiNMOS-CBA, CMOS-CBA, COSSAP, DESIGN (ARROWS), DesignPower, DesignWare, dont_use, Eagle Design Automation, ExpressModel, in-Sync, LM-1000, LM-1200, Logic Modeling, Logic Modeling (logo), Memory Architect, ModelAccess, ModelTools, PathMill, PLdebug, Powerview, Retargeter, SmartLicense, SmartLogic, SmartModel, SmartModels, SNUG, SOLV-IT!, SourceModel Library, Stream Driven Simulator_, Synopsys, Synopsys (logo), Synopsys VHDL Compiler, Synthetic Designs, Synthetic Libraries, TestBench Manager, TimeMill, ViewBase, ViewData, ViewDoc, ViewDraw, ViewFault, ViewFlow, VIEWFPGA, ViewGen, Viewlogic, ViewPlace, ViewPLD, ViewScript, ViewSim, ViewState, ViewSynthesis, ViewText, Workview, Workview Office, and Workview Plus are registered trademarks of Synopsys, Inc.

Trademarks

3-D Debugging, AC/Grade, AMPS, Arcadia, Arkos, Aurora, BCView, BOA, BRT, CBA Design System, CBA-Frame, characterize, Chip Architect, Chronologic, Compiler Designs, Core Network, Core Store, Cyclone, Data Path Express, DataPath Architect, DC Expert, DC Expert *Plus*, DC Professional, Delay Mill, Design Advisor, Design Analyzer_proposed, Design Exchange, Design Source, DesignTime, DesignWare Developer, Direct RTL, Direct Silicon Access, dont_touch, dont_touch_network, DW 8051, DWPCI, DxDataBook, DxDataManager, Eagle, Eagle*i*, Eagle V, Embedded System Prototype, Floorplan Manager, Formality, FoundryModel, FPGA Compiler II, FPGA Express, Fusion, FusionHDL, General Purpose Post-Processor, GPP, HDL Advisor, HTX, Integrator, IntelliFlow, Interactive Waveform Viewer, ISIS, ISIS PreVUE, LM-1400, LM-700, LM-family, Logic Model, ModelSource, ModelWare, MOTIVE, MS-3200, MS-3400, PathBlazer, PDQ, POET, PowerArc, PowerCODE, PowerGate, PowerMill, PreVUE, PrimeTime, Protocol Compiler, QUIET, QUIET Expert, RailMill, RTL Analyzer, Shadow Debugger, Silicon Architects, SimuBus, SmartCircuit, SmartModel Windows, Source-Level Design, SourceModel, SpeedWave, SWIFT, SWIFT interface, Synopsys Behavioral Compiler, Synopsys Design Compiler, Synopsys ECL Compiler, Synopsys ECO Compiler, Synopsys FPGA Compiler, Synopsys Frame Compiler, Synopsys Graphical Environment, Synopsys HDL Compiler, Synopsys Library Compiler, Synopsys ModelFactory, Synopsys Module Compiler, Synopsys Power Compiler, Synopsys Test Compiler, Synopsys Test Compiler Plus, TAP-in, Test Manager, TestGen, TestGen Expert Plus, TestSim, Timing Annotator, TLC, Trace-On-Demand, VCS, DCS Express, VCSi, VHDL System Simulator, ViewAnalog, ViewDatabook, ViewDRC, ViewLibrarian, ViewLibrary, ViewProject, ViewSymbol, ViewTrace, Visualyze, Vivace, VMD, VSS Expert, VSS Professional VWaves, XFX, XNS, and XTK are trademarks of Synopsys, Inc.

Service Marks

SolvNET is a service mark of Synopsys, Inc.

All other product or company names may be trademarks of their respective owners.

Printed in the U.S.A.

FPGA Compiler II / FPGA Express Verilog HDL Reference Manual, Version 1999.05

About This Manual

This manual describes the Verilog portion of Synopsys FPGA Compiler II / *FPGA Express* application, part of the Synopsys suite of synthesis tools. *FPGA Compiler II / FPGA Express* reads an RTL Verilog HDL model of a discrete electronic system and synthesizes this description into a gate-level netlist.

FPGA Compiler II / FPGA Express supports v1.6 of the Verilog language. Deviations from the definition of the Verilog language are explicitly noted. Constructs added in versions subsequent to Verilog 1.6 might not be supported. Aspects of the Verilog language that are not supported are listed in Appendix B.

Audience

This manual is written for logic designers and electronic engineers who are familiar with Synopsys synthesis products. Knowledge of the Verilog language is required, and knowledge of a high-level programming language is helpful.

Other Sources of Information

The resources in the following sections provide additional information:

- Related Publications
- SolvNET Online Help
- Customer Support

Related Publications

These Synopsys documents supply additional information:

- *FPGA Compiler II / FPGA Express Getting Started Manual*
- *Design Compiler Command-Line Interface Guide*
- *Design Compiler Reference Manual: Constraints and Timing*
- *Design Compiler Reference Manual: Optimization and Timing Analysis*
- *Design Compiler Tutorial*
- *Design Compiler User Guide*
- *DesignWare Developer Guide*
- *VSS User Guide*

Man Pages

You can view man pages from `fc2_shell` / `fe_shell` environment. From the shell prompt, enter:

```
fc2_shell> help command_name
```

or

```
fe_shell> help command_name
```

SolvNET Online Help

SOLV-IT! is the Synopsys electronic knowledge base. It contains information about Synopsys and its tools and is updated daily.

Access SOLV-IT! through e-mail or through the World Wide Web (WWW). For more information about SOLV-IT!, send e-mail to

`solvitfb@synopsys.com`

or view the Synopsys Web page at

`http://www.synopsys.com`

Customer Support

If you have problems, questions, or suggestions, contact the Synopsys Technical Support Center in one of the following ways:

- Send e-mail to

`support_center@synopsys.com`

- Call (650) 584-4200 outside the continental United States or call (800) 245-8005 inside the continental United States, from 7 a.m. to 5:30 p.m. Pacific time, Monday through Friday.
- Send a fax to (650) 584-2539.

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
<code>courier</code>	Indicates command syntax.
	In command syntax and examples, shows system prompts, text from files, error messages, and reports printed by the system.
<code><i>courier italic</i></code>	Indicates a user specification, such as <code><i>object_name</i></code>
<code>courier bold</code>	In command syntax and examples, indicates user input (text the user types verbatim).
<code>[]</code>	Denotes optional parameters, such as <code><i>pin1</i></code> [<code><i>pin2</i></code> , . . . <code><i>pinN</i></code>]
<code> </code>	Indicates a choice among alternatives, such as <code>low medium high</code> This example indicates that you can enter one of three possible values for an option: low, medium, or high.
<code>—</code>	Connects two terms that are read as a single term by the system. For example, <code>design_space</code> .
<code>(Ctrl-c)</code>	Indicates a keyboard combination, such as holding down the Ctrl key and pressing c.
<code>\</code>	Indicates a continuation of a command line.
<code>/</code>	Indicates levels of directory structure.
<code>Edit > Copy</code>	Shows a menu selection. <i>Edit</i> is the menu name and <i>Copy</i> is the item on the menu.

Table of Contents

About This Manual

1. FPGA Compiler II / FPGA <i>Express</i> with Verilog HDL	
Hardware Description Languages	1-2
FPGA Compiler II / FPGA <i>Express</i> and the Design Process	1-4
Using FPGA Compiler II / FPGA <i>Express</i> to Compile a Verilog HDL Design 1-5	
Design Methodology	1-6
2. Description Styles	
Design Hierarchy	2-2
Structural Descriptions	2-3
Functional Descriptions	2-3
Mixing Structural and Functional Descriptions	2-4
Design Methodology	2-6
Description Style	2-6
Language Constructs	2-6

Register Selection	2-7
Asynchronous Designs	2-8
3. Structural Descriptions	
Modules	3-2
Macromodules	3-3
Port Definitions	3-4
Port Names	3-5
Renaming Ports	3-6
Module Statements and Constructs	3-7
Structural Data Types	3-8
parameter	3-8
wire	3-9
wand	3-10
wor	3-11
tri	3-12
supply0 and supply1	3-13
reg	3-13
Port Declarations	3-14
input	3-14
output	3-14
inout	3-15
Continuous Assignment	3-15
Module Instantiations	3-17
Named and Positional Notation	3-18
Parameterized Designs	3-19

Gate-Level Modeling	3-20
Three-State Buffer Instantiation	3-22
4. Expressions	
Constant-Valued Expressions	4-2
Operators	4-3
Arithmetic Operators	4-4
Relational Operators	4-5
Equality Operators	4-6
Handling Comparisons to X or Z	4-7
Logical Operators.	4-8
Bitwise Operators.	4-9
Reduction Operators	4-10
Shift Operators.	4-11
Conditional Operator	4-12
Concatenation Operators.	4-13
Operator Precedence.	4-15
Operands.	4-16
Numbers	4-16
Wires and Registers.	4-17
Bit-Selects	4-17
Part-Selects	4-18
Function Calls	4-18
Concatenation of Operands.	4-19
Expression Bit-Widths	4-19

5. Functional Descriptions

Sequential Constructs	5-2
Function Declarations	5-3
Input Declarations	5-5
Output From a Function	5-5
Register Declarations.	5-6
Memory Declarations.	5-7
Parameter Declarations	5-8
Integer Declarations.	5-9
Function Statements	5-9
Procedural Assignments	5-10
RTL Assignments.	5-11
begin...end Block Statements	5-14
if...else Statements	5-15
Conditional Assignments	5-18
case Statements	5-18
Full Case and Parallel Case.	5-20
casex Statements	5-22
casez Statements	5-25
for Loops	5-27
while Loops	5-29
forever Loops	5-30
disable Statements	5-31
task Statements.	5-32
always Blocks	5-34

Event Expression	5-34
Incomplete Event Specification	5-37

6. Register and Three-State Inference

Register Inference	6-1
The Inference Report	6-2
Latch Inference Warnings	6-4
Controlling Register Inference	6-4
Attributes That Control Register Inference	6-4
Inferring Latches	6-7
Inferring SR Latches	6-7
Inferring D Latches	6-9
Understanding the Limitations of D Latch Inference	6-19
Inferring Master-Slave Latches	6-19
Inferring Flip-Flops	6-21
Inferring D Flip-Flops	6-21
Understanding the Limitations of D Flip-Flop Inference	6-35
Inferring JK Flip-Flops	6-37
Inferring Toggle Flip-Flops	6-41
Getting the Best Results	6-46
Understanding Limitations of Register Inference	6-50
Three-State Inference	6-51
Reporting Three-State Inference	6-51
Controlling Three-State Inference	6-51
Inferring Three-State Drivers	6-52
Simple Three-State Driver	6-52
Registered Three-State Drivers	6-57

Understanding the Limitations of Three-State Inference 6-60

7. Writing Circuit Descriptions

How Statements Are Mapped to Logic	7-2
Design Structure	7-3
Using Design Knowledge.	7-7
Optimizing Arithmetic Expressions	7-7
Arranging Expression Trees for Minimum Delay.	7-7
Sharing Common Subexpressions.	7-12
Using Operator Bit-Width Efficiently.	7-15
Using State Information	7-16
Describing State Machines	7-19
Minimizing Registers	7-24
Separating Sequential and Combinational Assignments.	7-27
Don't Care Inference	7-28
Limitations of Using Don't Care Values	7-29
Differences Between Simulation and Synthesis.	7-29
Propagating Constants	7-31
Synthesis Issues	7-31
Feedback Paths and Latches.	7-32
Synthesizing Asynchronous Designs.	7-32
Designing for Overall Efficiency.	7-34
Describing Random Logic	7-35
Sharing Complex Operators	7-35

8. FPGA Compiler II / FPGA <i>Express</i> Directives	
Notation for FPGA Compiler II / FPGA <i>Express</i> Directives	8-2
translate_off and translate_on Directives	8-2
parallel_case Directive	8-4
full_case Directive	8-5
state_vector Directive	8-8
enum Directive.	8-10
Component Implication	8-16
A. Examples	
Count Zeros—Combinational Version	A-2
Count Zeros—Sequential Version	A-5
Drink Machine—State Machine Version	A-7
Drink Machine—Count Nickels Version.	A-10
Carry-Lookahead Adder	A-12
B. Verilog Syntax	
Syntax	B-1
BNF Syntax Formalism	B-2
BNF Syntax	B-3
Lexical Conventions.	B-12
White Space	B-13
Comments	B-13

Numbers	B-13
Identifiers	B-15
Operators	B-15
Macro Substitution	B-16
include Construct	B-17
Simulation Directives	B-18
Verilog System Functions	B-18
Verilog Keywords	B-19
Unsupported Verilog Language Constructs.	B-20

List of Figures

Figure 1-1	FPGA Compiler II / FPGA <i>Express</i> Design Process. . . .	1-4
Figure 1-2	Design Flow	1-6
Figure 3-1	Structural Parts of a Module	3-2
Figure 5-1	Schematic of RTL Nonblocking Assignments	5-13
Figure 5-2	Schematic of Blocking Assignment.	5-14
Figure 6-1	SR Latch	6-9
Figure 6-2	D Latch	6-13
Figure 6-3	D Latch With Asynchronous Set	6-15
Figure 6-4	D Latch With Asynchronous Reset	6-16
Figure 6-5	D Latch With Asynchronous Set and Reset	6-18
Figure 6-6	Two-Phase Clocks	6-20
Figure 6-7	Positive Edge-Triggered D Flip-Flop	6-23
Figure 6-8	Negative Edge-Triggered D Flip-Flop	6-24
Figure 6-9	D Flip-Flop With Asynchronous Set	6-25
Figure 6-10	D Flip-Flop With Asynchronous Reset	6-26
Figure 6-11	D Flip-Flop With Asynchronous Set and Reset	6-28

Figure 6-12	D Flip-Flop With Synchronous Set	6-30
Figure 6-13	D Flip-Flop With Synchronous Reset	6-31
Figure 6-14	D Flip-Flop With Synchronous and Asynchronous Load	6-33
Figure 6-15	MultipleFlip-FlopsWithAsynchronousandSynchronousControls	6-35
Figure 6-16	JK Flip-Flop.	6-39
Figure 6-17	JK Flip-Flop With Asynchronous Set and Reset.	6-41
Figure 6-18	Toggle Flip-Flop With Asynchronous Set	6-43
Figure 6-19	Toggle Flip-Flop With Asynchronous Reset	6-44
Figure 6-20	Toggle Flip-Flop With Enable and Asynchronous Reset.	6-46
Figure 6-21	Schematic of Simple Three-State Driver	6-53
Figure 6-22	One Three-State Driver Inferred From a Single Block . .	6-55
Figure 6-23	Two Three-State Drivers Inferred From Separate Blocks	6-57
Figure 6-24	Three-State Driver With Registered Enable	6-58
Figure 6-25	Three-State Driver Without Registered Enable.	6-60
Figure 7-1	Ripple Carry Chain Implementation	7-4
Figure 7-2	Carry-Lookahead Chain Implementation	7-5
Figure 7-3	Default Expression Tree	7-8
Figure 7-4	Balanced Adder Tree (Same Arrival Times for All Signals)	7-9
Figure 7-5	Expression Tree With Minimum Delay (Signal A Arrives Last)	7-9
Figure 7-6	Expression Tree With Subexpressions Dictated by Parentheses	7-10
Figure 7-7	Default Expression Tree With 4-Bit Temporary Variable .	7-11
Figure 7-8	Expression Tree With 5-Bit Intermediate Result	7-12

Figure 7-9	Synthesized Circuit With Six Implied Registers	7-25
Figure 7-10	Synthesized Circuit With Three Implied Registers	7-26
Figure 7-11	Mealy Machine Schematic	7-28
Figure 7-12	Circuit Schematic With Two Array Indexes	7-37
Figure 7-13	Circuit Schematic With One Array Index.	7-39
Figure A-1	Count Zeros—Combinational Version Block Diagram . .	A-4
Figure A-2	Count Zeros—Sequential Version Block Diagram	A-7
Figure A-3	Drink Machine—State Machine Version Block Diagram .	A-10
Figure A-4	Drink Machine—Count Nickels Version Block Diagram .	A-12
Figure A-5	Carry-Lookahead Adder Block Diagram	A-14

List of Tables

Table 4-1	Verilog Operators Supported by FPGA Compiler II/FPGA Express 4-3	
Table 4-2	Operator Precedence	4-15
Table 4-3	Expression Bit-Widths	4-20
Table 6-1	SR Latch Truth Table (Nand Type)	6-8
Table 6-2	Truth Table for JK Flip-Flop	6-38
Table B-1	Verilog Radices	B-14
Table B-2	Verilog Keywords	B-19

List of Examples

Example 2-1	Mixed Structural and Functional Descriptions.	2-5
Example 3-1	Module Definition	3-3
Example 3-2	Macromodule Construct	3-3
Example 3-3	Module Port Lists	3-5
Example 3-4	Renaming Ports in Modules	3-6
Example 3-5	parameter Declaration Syntax Error	3-9
Example 3-6	parameter Declarations.	3-9
Example 3-7	wire Declarations.	3-10
Example 3-8	wand (wired-AND).	3-11
Example 3-9	wor (wired-OR)	3-11
Example 3-10	tri (Three-State).	3-12
Example 3-11	supply0 and supply1 Constructs.	3-13
Example 3-12	reg Declarations	3-13
Example 3-13	Two Equivalent Continuous Assignments	3-15
Example 3-14	Module Instantiations	3-18
Example 3-15	parameter Declaration in a Module.	3-20

Example 3-16	Gate-Level Instantiations	3-21
Example 3-17	Three-State Gate Instantiation	3-22
Example 4-1	Valid Expressions	4-2
Example 4-2	Addition Operator	4-5
Example 4-3	Relational Operator	4-6
Example 4-4	Equality Operator	4-6
Example 4-5	Comparison to X Ignored	4-7
Example 4-6	Logical Operators	4-8
Example 4-7	Bitwise Operators	4-9
Example 4-8	Reduction Operators	4-10
Example 4-9	Shift Operator	4-11
Example 4-10	Conditional Operator	4-12
Example 4-11	Nested Conditional Operator	4-13
Example 4-12	Concatenation Operator	4-14
Example 4-13	Concatenation Equivalent	4-14
Example 4-14	Wire Operands	4-17
Example 4-15	Bit-Select Operands	4-17
Example 4-16	Part-Select Operands	4-18
Example 4-17	Function Call Used as an Operand	4-18
Example 4-18	Concatenation of Operands	4-19
Example 4-19	Self-Determined Expression	4-21
Example 4-20	Context-Determined Expressions	4-21
Example 5-1	Sequential Statements	5-2
Example 5-2	Equivalent Combinational Description	5-2

Example 5-3	Combinational Ripple Carry Adder	5-3
Example 5-4	Simple Function Declaration	5-4
Example 5-5	Many Outputs From a Function.	5-6
Example 5-6	Register Declarations	5-7
Example 5-7	Memory Declarations	5-7
Example 5-8	Parameter Declaration in a Function.	5-8
Example 5-9	Integer Declarations	5-9
Example 5-10	Procedural Assignments	5-11
Example 5-11	RTL Nonblocking Assignments	5-12
Example 5-12	Blocking Assignment.	5-13
Example 5-13	Block Statement With a Named Block	5-14
Example 5-14	if Statement That Synthesizes Multiplexer Logic.	5-16
Example 5-15	if...else if...else Structure.	5-17
Example 5-16	Nested if and else Statements	5-17
Example 5-17	Synthesizing a Latch for a Conditionally Driven Variable 5-18	
Example 5-18	case Statement	5-20
Example 5-19	A case Statement That Is Both Full and Parallel.	5-21
Example 5-20	A case Statement That Is Parallel but Not Full	5-22
Example 5-21	A case Statement That Is Not Full or Parallel	5-22
Example 5-22	casex Statement With x	5-23
Example 5-23	Before Using casex With ?	5-24
Example 5-24	After Using casex With ?	5-24
Example 5-25	Invalid casex Expression.	5-24

Example 5-26	casez Statement With z	5-26
Example 5-27	Invalid casez Expression.	5-26
Example 5-28	A Simple for Loop	5-27
Example 5-29	Nested for Loops.	5-28
Example 5-30	Example for Loop	5-28
Example 5-31	Expanded for Loop	5-28
Example 5-32	Unsupported while Loop	5-29
Example 5-33	Supported while Loop	5-30
Example 5-34	Supported forever Loop	5-30
Example 5-35	Comparator Using disable.	5-31
Example 5-36	Synchronous Reset of State Register Using disable in a forever Loop	5-32
Example 5-37	Using the task Statement	5-33
Example 5-38	A Simple always Block	5-34
Example 5-39	Incomplete Event List	5-37
Example 5-40	Complete Event List	5-37
Example 5-41	Incomplete Event List for Asynchronous Preload	5-37
Example 6-1	Inference Report for a JK Flip-Flop.	6-2
Example 6-2	SR Latch	6-8
Example 6-3	Inference Report for an SR Latch	6-8
Example 6-4	Latch Inference Using an if Statement	6-10
Example 6-5	Latch Inference Using a case Statement	6-10
Example 6-6	Avoiding Latch Inference.	6-11
Example 6-7	Another Way to Avoid Latch Inference	6-11

Example 6-8	Function: No Latch Inference	6-11
Example 6-9	D Latch	6-12
Example 6-10	Inference Report for a D Latch	6-13
Example 6-11	D Latch With Asynchronous Set	6-14
Example 6-12	Inference Report for D Latch With Asynchronous Set	6-14
Example 6-13	D Latch With Asynchronous Reset	6-16
Example 6-14	Inference Report for D Latch With Asynchronous Reset	6-16
Example 6-15	D Latch With Asynchronous Set and Reset	6-17
Example 6-16	Inference Report for D Latch With Asynchronous Set and Reset	6-18
Example 6-17	Invalid Use of a Conditionally Assigned Variable	6-19
Example 6-18	Two-Phase Clocks.	6-20
Example 6-19	Using an always Block to Infer a Flip-Flop	6-21
Example 6-20	Positive Edge-Triggered D Flip-Flop	6-22
Example 6-21	Inference Report for a Positive Edge-Triggered D Flip-Flop	6-22
Example 6-22	Negative Edge-Triggered D Flip-Flop	6-23
Example 6-23	Inference Report for a Negative Edge-Triggered D Flip-Flop	6-23
Example 6-24	D Flip-Flop With Asynchronous Set	6-24
Example 6-25	Inference Report for a D Flip-Flop With Asynchronous Set	6-25
Example 6-26	D Flip-Flop With Asynchronous Reset	6-26
Example 6-27	Inference Report for a D Flip-Flop With Asynchronous Reset	6-26

Example 6-28	D Flip-Flop With Asynchronous Set and Reset	6-27
Example 6-29	Inference Report for a D Flip-Flop With Asynchronous Set and Reset	6-28
Example 6-30	D Flip-Flop With Synchronous Set	6-29
Example 6-31	Inference Report for a D Flip-Flop With Synchronous Set	6-30
Example 6-32	D Flip-Flop With Synchronous Reset	6-31
Example 6-33	Inference Report for a D Flip-Flop With Synchronous Reset	6-31
Example 6-34	D Flip-Flop With Synchronous and Asynchronous Load	6-32
Example 6-35	Inference Report for a D Flip-Flop With Synchronous and Asynchronous Load	6-32
Example 6-36	Multiple Flip-Flops With Asynchronous and Synchronous Controls	6-34
Example 6-37	Inference Reports for Multiple Flip-Flops With Asynchronous and Synchronous Controls	6-34
Example 6-38	JK Flip-Flop	6-38
Example 6-39	Inference Report for JK Flip-Flop	6-38
Example 6-40	JK Flip-Flop With Asynchronous Set and Reset	6-40
Example 6-41	Inference Report for JK Flip-Flop With Asynchronous Set and Reset	6-41
Example 6-42	Toggle Flip-Flop With Asynchronous Set	6-42
Example 6-43	Inference Report for a Toggle Flip-Flop With Asynchronous Set	6-42
Example 6-44	Toggle Flip-Flop With Asynchronous Reset	6-43

Example 6-45	Inference Report: Toggle Flip-Flop With Asynchronous Reset 6-44	
Example 6-46	Toggle Flip-Flop With Enable and Asynchronous Reset	6-45
Example 6-47	Inference Report: Toggle Flip-Flop With Enable and Asynchronous Reset	6-45
Example 6-48	Circuit With Six Implied Registers	6-47
Example 6-49	Circuit With Three Implied Registers	6-48
Example 6-50	Delays in Registers	6-49
Example 6-51	Three-State Inference Report	6-51
Example 6-52	Simple Three-State Driver	6-52
Example 6-53	Inference Report for Simple Three-State Driver	6-53
Example 6-54	Inferring One Three-State Driver From a Single Block	6-54
Example 6-55	Single Block Inference Report	6-54
Example 6-56	Inferring Three-State Drivers From Separate Blocks .	6-56
Example 6-57	Inference Report for Two Three-State Drivers	6-56
Example 6-58	Three-State Driver With Registered Enable	6-57
Example 6-59	Inference Report for Three-State Driver With Registered Enable 6-58	
Example 6-60	Three-State Driver Without Registered Enable	6-59
Example 6-61	Inference Report for Three-State Driver Without Registered Enable 6-59	
Example 7-1	Four Logic Blocks	7-3
Example 7-2	Ripple Carry Chain	7-4
Example 7-3	Carry-Lookahead Chain	7-4
Example 7-4	4-Input Adder	7-6

Example 7-5	4-Input Adder With Parentheses	7-6
Example 7-6	Simple Arithmetic Expression	7-8
Example 7-7	Parentheses in an Arithmetic Expression	7-10
Example 7-8	Adding Numbers of Different Bit-Widths	7-11
Example 7-9	Simple Additions With a Common Subexpression . . .	7-13
Example 7-10	Sharing Common Subexpressions	7-13
Example 7-11	Unidentified Common Subexpressions	7-14
Example 7-12	More Efficient Use of Operators	7-15
Example 7-13	A Simple Finite State Machine	7-16
Example 7-14	Better Implementation of a Finite State Machine	7-18
Example 7-15	Summing Three Cycles of Data in the Implicit State Style (Preferred) 7-20	
Example 7-16	Summing Three Cycles of Data in the Explicit State Style (Not Advisable)	7-21
Example 7-17	Synchronous Reset—Explicit State Style (Preferred). .	7-22
Example 7-18	Synchronous Reset—Implicit State Style (Not Advisable) 7-23	
Example 7-19	Inefficient Circuit Description With Six Implied Registers 7-24	
Example 7-20	Circuit With Three Implied Registers.	7-26
Example 7-21	Mealy Machine	7-27
Example 7-22	Fully Synchronous Counter Design.	7-33
Example 7-23	Asynchronous Counter Design	7-33
Example 7-24	Equivalent Statements	7-35
Example 7-25	Inefficient Circuit Description With Two Array Indexes	7-36

Example 7-26	Efficient Circuit Description With One Array Index . . .	7-38
Example 8-1	//synopsystranslate_onand//synopsystranslate_offDirectives 8-3	
Example 8-2	// synopsys parallel_case Directives	8-4
Example 8-3	// synopsys full_case Directives	8-6
Example 8-4	Latches and // synopsys full_case	8-7
Example 8-5	// synopsys state_vector Example.	8-9
Example 8-6	Enumeration of Type Colors	8-10
Example 8-7	Invalid enum Declaration.	8-10
Example 8-8	More enum Type Declarations	8-11
Example 8-9	Invalid Bit Value Encoding for Colors	8-11
Example 8-10	Enumeration Literals Used as Constants	8-11
Example 8-11	Finite State Machine With // synopsys enum and // synopsys state_vector.	8-12
Example 8-12	Unsupported Bit-Select From Enumerated Type.	8-13
Example 8-13	Unsupported Bit-Select (With Component Instantiation) From Enumerated Type.	8-13
Example 8-14	Using Inference With Enumerated Types	8-14
Example 8-15	Changing the Enumeration Encoding	8-14
Example 8-16	Supported Bit-Select From Enumerated Type.	8-15
Example 8-17	Enumerated Type Declaration for a Port	8-15
Example 8-18	Incorrect Enumerated Type Declaration for a Port . . .	8-16
Example 8-19	Component Implication	8-17
Example A-1	Count Zeros—Combinational Version.	A-3
Example A-2	Count Zeros—Sequential Version.	A-5

Example A-3	Drink Machine—State Machine Version	A-8
Example A-4	Drink Machine—Count Nickels Version.	A-10
Example A-5	Carry-Lookahead Adder	A-15
Example B-1	Valid Verilog Number Declarations	B-14
Example B-2	Sample Escaped Identifiers	B-15
Example B-3	Macro Variable Declarations	B-16
Example B-4	Macro With Sized Constants.	B-17
Example B-5	Including a File Within a File	B-17

1

FPGA Compiler II / FPGA *Express* with Verilog HDL

FPGA Compiler II / FPGA *Express* translates and optimizes Verilog HDL descriptions into an internal gate-level equivalent, and then compiles this representation to produce optimized gate-level designs in a given FPGA technology.

This chapter introduces the main concepts and capabilities of FPGA Compiler II / FPGA *Express* in the following sections:

- Hardware Description Languages
- FPGA Compiler II / FPGA *Express* and the Design Process
- Using FPGA Compiler II / FPGA *Express* to Compile a Verilog HDL Design
- Design Methodology

Hardware Description Languages

Hardware description languages (HDLs) describe the architecture and behavior of discrete electronic systems. Modern HDLs and their associated simulators are very powerful tools for integrated circuit designers.

A typical HDL supports a mixed-level description in which gate and netlist constructs are used with functional descriptions. This mixed-level capability enables you to describe system architectures at a very high level of abstraction and then incrementally refine a design's detailed gate-level implementation.

HDL descriptions play an important role in modern design methodology, for three main reasons:

- Design functionality can be verified early in the design process. A design written as an HDL description can be simulated immediately. Design simulation at this higher level, before implementation at the gate level, allows you to evaluate architectural and design decisions.
- FPGA Compiler II / FPGA *Express* provides Verilog compilation and logic synthesis, allowing you to automatically convert an HDL description to a gate-level implementation in a target FPGA technology. This step eliminates the former technology-specific design bottleneck, the majority of circuit design time, and the errors that occur when you hand-translate an HDL specification to gates.

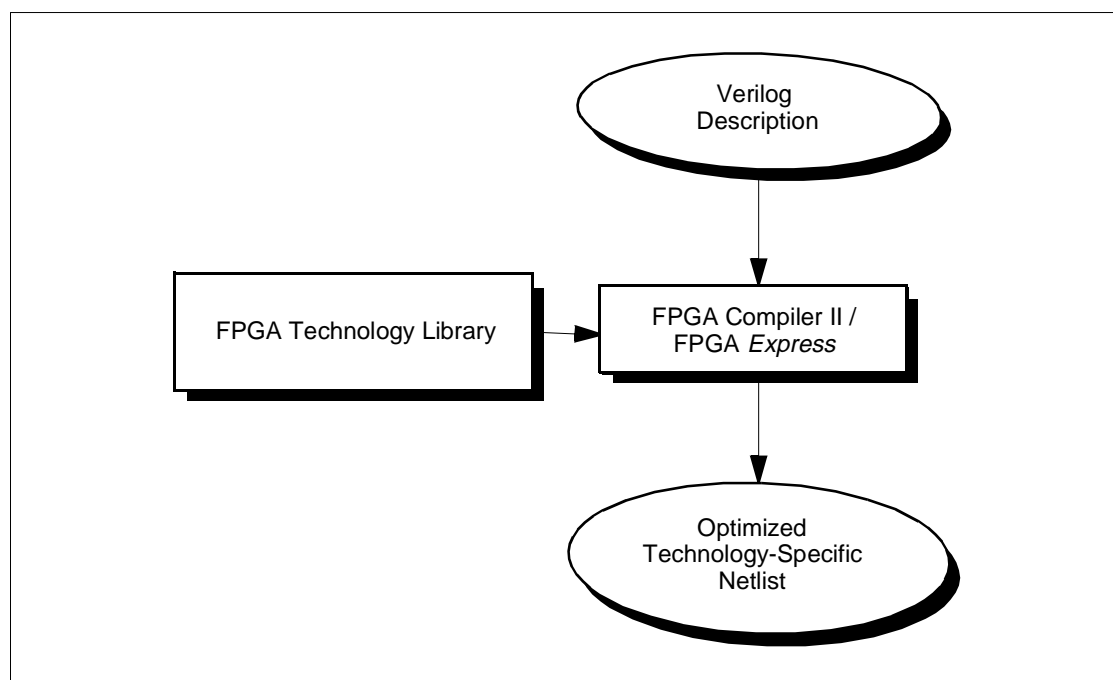
With FPGA Compiler II / FPGA *Express* logic optimization, you can automatically transform a synthesized design into a smaller or faster circuit. FPGA Compiler II / FPGA *Express* provides both logic synthesis and optimization. For further information, refer to FPGA Compiler II / FPGA *Express* online help.

- HDL descriptions provide technology-independent documentation of a design and its functionality. An HDL description is easier to read and understand than a netlist or a schematic description. Because the initial HDL design description is technology-independent, you can reuse it to generate the design in a different technology, without having to translate from the original technology.

FPGA Compiler II / FPGA *Express* and the Design Process

FPGA Compiler II / FPGA *Express* translates hardware descriptions in Verilog to a Synopsys internal design format. The design can then be optimized and mapped to a specific FPGA technology library by FPGA Compiler II / FPGA *Express*, as Figure 1-1 shows.

Figure 1-1 *FPGA Compiler II / FPGA Express Design Process*



FPGA Compiler II / FPGA *Express* supports a majority of the Verilog constructs. (For exceptions, see “Unsupported Verilog Language Constructs” on page B-20.)

Using FPGA Compiler II / FPGA *Express* to Compile a Verilog HDL Design

When a Verilog design is read into FPGA Compiler II / FPGA *Express*, it is converted to an internal database format so FPGA Compiler II / FPGA *Express* can synthesize and optimize the design. When FPGA Compiler II / FPGA *Express* optimizes a design, it may restructure part or all of the design. You control the degree of restructuring. Options include:

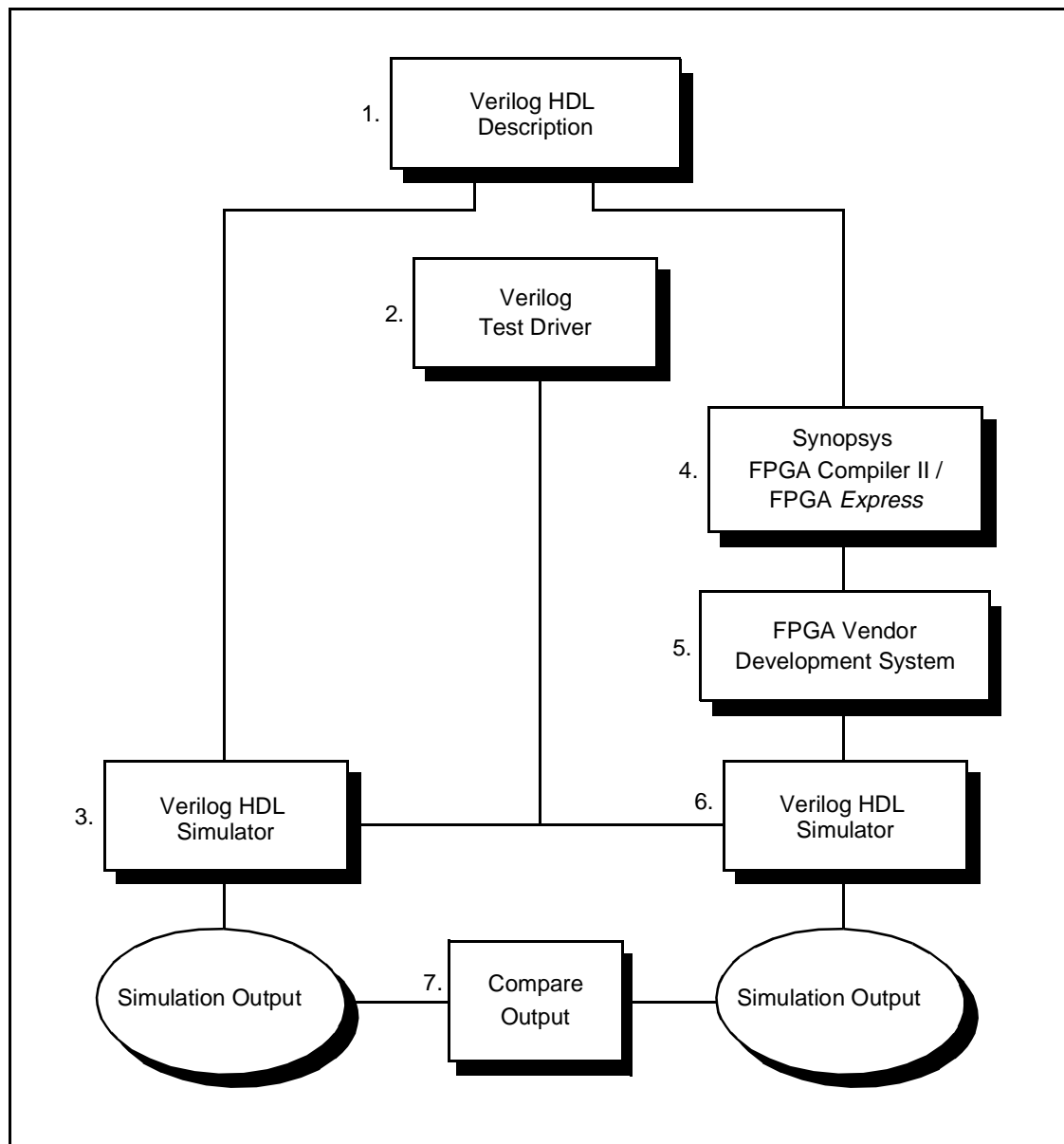
- Fully preserving a design's hierarchy
- Allowing certain modules to be combined with others
- Compressing the entire design into one module (called flattening the design), if that is beneficial

The following section describes the design process that uses FPGA Compiler II / FPGA *Express* with a Verilog HDL Simulator.

Design Methodology

Figure 1-2 shows a typical design process that uses FPGA Compiler II / FPGA Express and a Verilog HDL Simulator.

Figure 1-2 Design Flow



The steps in the design flow shown in Figure 1-2 are:

1. Write a design description in the Verilog language. This description can be a combination of structural and functional elements (as shown in Chapter 2, "Description Styles"). This description is for use with both Synopsys FPGA Compiler II / *FPGA Express* and a Verilog simulator.
2. Provide Verilog-language test drivers for the Verilog HDL simulator. For information on writing these drivers, see the appropriate simulator manual. The drivers supply test vectors for simulation and gather output data.
3. Simulate the design by using a Verilog HDL simulator. Verify that the description is correct.
4. Use *FPGA Compiler II / FPGA Express* to synthesize and optimize the Verilog description into a gate-level design. *FPGA Compiler II / FPGA Express* generates optimized netlists to satisfy timing constraints for a targeted FPGA architecture.
5. Use your FPGA development system to place and route the FPGA netlist. Then generate a Verilog netlist for post-place-and-route simulation. The development system includes simulation models and interfaces required for the design flow.
6. Simulate the technology-specific version of the design with the Verilog simulator. You can use the original Verilog simulation drivers from step 3 because module and port definitions are preserved through the translation and optimization processes.
7. Compare the output of the gate-level simulation (step 6) with the output of the original Verilog description simulation (step 3) to verify that the implementation is correct.

2

Description Styles

A Verilog circuit description can be one of two types: structural or functional. A structural description explains the physical makeup of the circuit, detailing gates and the connections between them. A functional description, also referred to as an RTL (Register Transfer Level) description, describes what the circuit does.

This chapter covers the following topics:

- Design Hierarchy
- Structural Descriptions
- Functional Descriptions
- Mixing Structural and Functional Descriptions
- Register Selection
- Asynchronous Designs

Design Hierarchy

Synopsys FPGA Compiler II / FPGA *Express* maintains the hierarchical boundaries you define when you use structural Verilog. These boundaries have two major effects:

- Constraints can be specified on a per-module basis. For example, this allows some modules to be optimized for area while others are optimized for speed.
- Module instantiations within HDL descriptions are maintained during input. The instance name you assign to user-defined components is carried through to the gate-level implementation.

Chapter 3, "Structural Descriptions", discusses modules and module instantiations.

Note:

FPGA Compiler II / FPGA *Express* does not automatically maintain (create) the hierarchy of other, nonstructural Verilog constructs such as blocks, loops, functions, and tasks. These elements of an HDL description are translated in the context of their design.

The choice of hierarchical boundaries has a significant effect on the quality of the synthesized design. Using FPGA Compiler II / FPGA *Express*, you can optimize a design while preserving these hierarchical boundaries. However, FPGA Compiler II / FPGA *Express* only partially optimizes logic across hierarchical modules. Full optimization is possible across those parts of the design hierarchy that are collapsed in FPGA Compiler II / FPGA *Express*.

Structural Descriptions

The structural elements of a Verilog structural description are generic logic gates, library-specific components, and user-defined components connected by wires. In one way, a structural description can be viewed as a simple netlist composed of nets that connect instantiations of gates. However, unlike in a netlist, nets in the structural description can be driven by an arbitrary expression that describes the value assigned to the net. A statement that drives an arbitrary expression onto a net is called a continuous assignment. Continuous assignments are convenient links between pure netlist descriptions and functional descriptions.

A Verilog structural description can define a range of hierarchical and gate-level constructs, including module definitions, module instantiations, and netlist connections. Refer to Chapter 3, "Structural Descriptions", for more information.

Functional Descriptions

The functional elements of a Verilog description are function declarations, task statements, and always blocks. These elements describe the function of the circuit but do not describe its physical makeup or layout. The choice of gates and components is left entirely to FPGA Compiler II / FPGA *Express*.

You can construct functional descriptions with the Verilog functional constructs described in Chapter 5, "Functional Descriptions". These constructs can appear within functions or always blocks. Functions imply only combinational logic; always blocks can imply either combinational or sequential logic.

Although many Verilog functional constructs (for example, for loops and multiple assignments to the same variable) appear sequential in nature, they describe combinational-logic networks. Other functional constructs imply sequential-logic networks. Latches and registers are inferred from these constructs. Refer to Chapter 6, "Register and Three-State Inference" for details.

Mixing Structural and Functional Descriptions

When you use a functional description style in a design, you typically describe the combinational portions of the design in Verilog functions, always blocks, and assignments. The complexity of the logic determines whether you use one or many functions.

Example 2-1 shows how structural and functional description styles are mixed in a design specification. In Example 2-1, the function `detect_logic` determines whether the input bit is a 0 or a 1. After making this determination, `detect_logic` sets `ns` to the next state of the machine. An always block infers flip-flops to hold the state information between clock cycles.

You can specify elements of a design directly as module instantiations at the structural level. For example, see the three-state buffer `t1` in Example 2-1. (Note that three-states can be inferred. For more information, refer to "Three-State Inference" on page 6-51.) You can also use this description style to identify the wires and ports that carry information from one part of the design to another.

Example 2-1 Mixed Structural and Functional Descriptions

```
// This finite-state machine (Mealy type) reads one bit per
// clock cycle and detects three or more consecutive 1s.
module three_ones( signal, clock, detect, output_enable );
input signal, clock, output_enable;
output detect;

// Declare current state and next state variables.
reg [1:0] cs;
reg [1:0] ns;
wire ungated_detect;

// declare the symbolic names for states
parameter NO_ONES = 0, ONE_ONE = 1,
          TWO_ONES = 2, AT_LEAST_THREE_ONES = 3;

// ***** STRUCTURAL DESCRIPTION *****
// Instance of a three-state gate that enables output
three_state t1 (ungated_detect, output_enable, detect);

// ***** ALWAYS BLOCK *****
// always block infers flip-flops to hold the state of
// the FSM.
always @ ( posedge clock ) begin
    cs <= ns;
end

// ***** FUNCTIONAL DESCRIPTION *****
function detect_logic;
input [1:0] cs;
input signal;
begin
    detect_logic = 0; //default value
    if ( signal == 0 ) //bit is zero
        ns = NO_ONES;
    else //bit is one, increment state
        case (cs)
            NO_ONES: ns = ONE_ONE;
            ONE_ONE: ns = TWO_ONES;
            TWO_ONES, AT_LEAST_THREE_ONES:
                begin
                    ns = AT_LEAST_THREE_ONES;
                    detect_logic = 1;
                end
        endcase
    end
endfunction
```

```
// ***** assign STATEMENT *****  
assign ungated_detect = detect_logic( cs, signal );  
endmodule
```

For a structural or functional HDL description to be synthesized, it must follow the Synopsys synthesis policy, which has three parts:

- Design Methodology
- Description Style
- Language Constructs

Design Methodology

Design methodology refers to the synthesis design process that uses FPGA Compiler II / FPGA *Express* and Verilog HDL Simulator. This process is described in Chapter 1, "FPGA Compiler II / FPGA Express with Verilog HDL".

Description Style

Use the HDL design and coding style that makes the best use of the synthesis process to obtain high-quality results from FPGA Compiler II / FPGA *Express*. See Chapter 7, "Writing Circuit Descriptions", for guidelines.

Language Constructs

The third component of the Verilog synthesis policy is the set of Verilog constructs that describe your design, determine its architecture, and give consistently good results.

Synopsys uses HDL constructs that maximize coding flexibility while producing consistently good results. Although FPGA Compiler II / *FPGA Express* can read the entire Verilog language, a few HDL constructs cannot be synthesized. These constructs are unsupported because they cannot be realized in logic. For example, you cannot use simulation time as a trigger because time is an element of the simulation process and cannot be realized. “Unsupported Verilog Language Constructs” on page B-20 lists these constructs.

Register Selection

The clocking scheme and the placement of registers are important architectural factors. There are two ways to define registers in your Verilog description. Each method has specific advantages.

- You can directly instantiate registers into a Verilog description, selecting from any element in your FPGA library.

Clocking schemes can be arbitrarily complex. You can choose between a flip-flop and a latch-based architecture. The main disadvantages to this approach are that

- The Verilog description is specific to a given technology, because you choose structural elements from that technology library. However, you can isolate the portion of your design with directly instantiated registers as a separate component (module), and then connect it to the rest of the design.
- The description is more difficult to write.

- You can use some Verilog constructs to direct FPGA Compiler II / *FPGA Express* to infer registers from the description.

The advantages to this approach directly counter the disadvantages of the previous approach. With register inference, the Verilog description is much easier to write and is technology-independent. This method allows FPGA Compiler II / *FPGA Express* to select the type of component inferred, based on constraints. Therefore, if a specific component is necessary, use instantiation. Some types of registers and latches cannot be inferred.

See “Register Inference” on page 6-1 for a discussion of latch and register inference.

Asynchronous Designs

You can use FPGA Compiler II / *FPGA Express* to construct asynchronous designs that use multiple or gated clocks. However, although these designs are logically and statically correct, they may not simulate or operate correctly because of race conditions.

“Synthesis Issues” on page 7-31 describes how to write Verilog descriptions of asynchronous designs.

3

Structural Descriptions

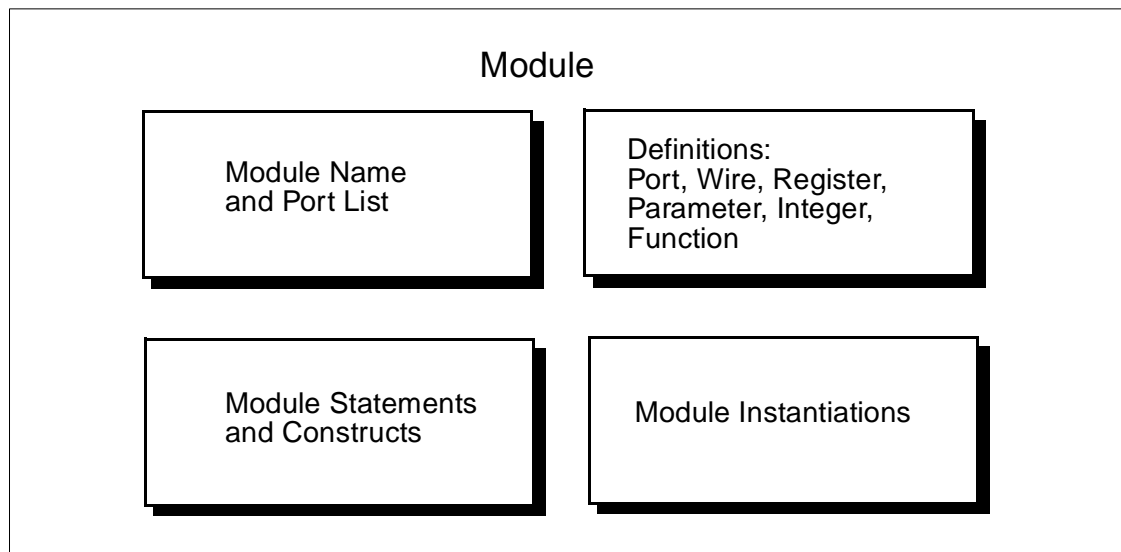
A Verilog structural description defines a connection of components that form a physical circuit. This chapter details the construction of structural descriptions, in the following major sections:

- Modules
- Macromodules
- Port Definitions
- Module Statements and Constructs
- Module Instantiations

Modules

The principal design entity in the Verilog language is the module. A module consists of the module name, its input and output description (port definition), a description of the functionality or implementation for the module (module statements and constructs), and named instantiations. Figure 3-1 illustrates the basic structural parts of a module.

Figure 3-1 Structural Parts of a Module



Example 3-1 shows a simple module that implements a 2-input NAND gate by instantiating an AND gate and an INV gate. The first line of the module definition gives the name of the module and a list of ports. The second and third lines give the direction for all ports. (Ports are either inputs, outputs, or bidirectionals.)

The fourth line of the description creates a wire variable. The next two lines instantiate the two components, creating copies named instance1 and instance2 of the components AND and INV. These components connect to the ports of the module and are finally connected by use of the variable and_out.

Example 3-1 Module Definition

```
module NAND(a,b,z);
    input a,b;      //Inputs to NAND gate
    output z;       //Outputs from NAND gate
    wire and_out;  //Output from AND gate

    AND instance1(a,b,and_out);
    INV instance2(and_out, z);
endmodule
```

Macromodules

The macromodule construct makes simulation more efficient, by merging the macromodule definition with the definition of the calling (parent) module. However, FPGA Compiler II / *FPGA Express* treats the macromodule construct as a module construct. Whether you use module or macromodule, the synthesis process, the hierarchy synthesis creates, and its result are the same. Example 3-2 shows how to use the macromodule construct.

Example 3-2 Macromodule Construct

```
macromodule adder (in1,in2,out1);
    input [3:0] in1,in2;
    output [4:0] out1;

    assign out1 = in1 + in2;
endmodule
```

Note:

When FPGA Compiler II / FPGA *Express* instantiates a macromodule, a new level of hierarchy is created.

Port Definitions

A port list consists of port expressions that describe the input and output interfaces for a module. Define the port list in parentheses after the module name, as shown here:

```
module name ( port_list );
```

A port expression in a port list can be any of the following:

- An identifier
- A single bit selected from a bit vector declared within the module
- A group of bits selected from a bit vector declared within the module
- A concatenation of any of the above

Concatenation is the process of combining several single-bit or multiple-bit operands into one large bit vector. For more information, see “Concatenation Operators” on page 4-13.

Declare each port in a port list as input, output, or bidirectional in the module by use of an input, output, or inout statement. (See “Port Declarations” on page 3-14.) For example, the module definition in Example 3-1 on page 3-3 shows that module NAND has three ports: a, b, and z, connected to 1-bit nets a, b, and z. Declare these connections in the input and output statements.

Port Names

Some port expressions are identifiers. If the port expression is an identifier, the port name is the same as the identifier. A port expression is not an identifier if the expression is a single bit, a group of bits selected from a vector of bits, or a concatenation of signals. In these cases, the port is unnamed unless you explicitly name it.

Example 3-3 shows some module definition fragments that illustrate the use of port names. The ports for module ex1, named a, b, and z, are connected to nets a, b, and z, respectively. The first two ports of module ex2 are unnamed; the third port is named z. The ports are connected to nets a[1], a[0], and z, respectively. Module ex3 has two ports: the first port, unnamed, is connected to a concatenation of nets a and b; the second port, named z, is connected to net z.

Example 3-3 Module Port Lists

```
module ex1( a, b, z );
    input a, b;
    output z;
endmodule

module ex2( a[1], a[0], z );
    input [1:0] a;
    output z;
endmodule

module ex3( {a,b}, z );
    input a,b;
    output z;
endmodule
```

Renaming Ports

You can rename a port by explicitly assigning a name to a port expression by using the dot (.) operator. The module definition fragments in Example 3-4 show how to rename ports. The ports for module `ex4` are explicitly named `in_a`, `in_b`, and `out` and are connected to nets `a`, `b`, and `z`. Module `ex5` shows ports named `i1`, `i0`, and `z` connected to nets `a[1]`, `a[0]`, and `z`, respectively. The first port for module `ex6` (the concatenation of nets `a` and `b`) is named `i`.

Example 3-4 Renaming Ports in Modules

```
module ex4( .in_a(a), .in_b(b), .out(z) );
    input a, b;
    output z;
endmodule

module ex5( .i1(a[1]), .i0(a[0]), z );
    input [1:0] a;
    output z;
endmodule

module ex6( .i({a,b}), z );
    input a,b;
    output z;
endmodule
```

Module Statements and Constructs

FPGA Compiler II / *FPGA Express* recognizes the following Verilog statements and constructs when they are used in a Verilog module:

- parameter declarations
- wire, wand, wor, tri, supply0, and supply1 declarations
- reg declarations
- input declarations
- output declarations
- inout declarations
- Continuous assignments
- Module instantiations
- Gate instantiations
- Function definitions
- always blocks
- task statements

Data declarations and assignments are described in this section. Module and gate instantiations are described in “Module Instantiations” on page 3-17. Function definitions, always blocks, and task statements are described in Chapter 5, “Functional Descriptions”.

Structural Data Types

Verilog structural data types include `wire`, `wand`, `wor`, `tri`, `supply0`, and `supply1`. Although `parameter` does not fall into the category of structural data types, it is presented here because it is used with structural data types.

You can define an optional range for all the data types presented in this section. The range provides a means for creating a bit vector. The syntax for a range specification is

```
[msb : lsb]
```

Expressions for most significant bit (`msb`) and least significant bit (`lsb`) must be nonnegative constant-valued expressions. Constant-valued expressions are composed only of constants, Verilog parameters, and operators.

parameter

Verilog parameters allow you to customize each instantiation of a module. By setting different values for the parameter when you instantiate the module, you can cause constructions of different logic. For more information, see “Parameterized Designs” on page 3-19.

A parameter represents constant values symbolically. The definition for a parameter consists of the parameter name and the value assigned to it. The value can be any constant-valued integer or Boolean expression. If you do not set the size of the parameter with a range definition or a sized constant, the parameter is unsized and defaults to a 32-bit quantity. Refer to “Constant-Valued Expressions” on page 4-2 for a discussion of constant formats.

You can use a parameter wherever a number is allowed, except when declaring the number of bits in an assignment statement, which will generate a syntax error as shown in Example 3-5.

Example 3-5 parameter Declaration Syntax Error

```
parameter size = 4;  
assign out = in ? 4'b0000 : size'b0101; // syntax error
```

You can define a parameter anywhere within a module definition. However, the Verilog language requires that you define the parameter before you use it.

Example 3-6 shows two parameter declarations. Parameters `true` and `false` are unsized and have values of 1 and 0, respectively. Parameters `S0`, `S1`, `S2`, and `S3` have values of 3, 1, 0, and 2, respectively, and are stored as 2-bit quantities.

Example 3-6 parameter Declarations

```
parameter TRUE=1, FALSE=0;  
parameter [1:0] S0=3, S1=1, S2=0, S3=2;
```

wire

A wire data type in a Verilog description represents the physical wires in a circuit. A wire connects gate-level instantiations and module instantiations. The Verilog language allows you to *read* a value from a wire from within a function or a `begin...end` block, but you cannot *assign* a value to a wire within a function or a `begin...end` block. (An `always` block is a specific type of `begin...end` block.)

A wire does not store its value. It must be driven in one of two ways:

- By connecting the wire to the output of a gate or module
- By assigning a value to the wire in a *continuous assignment*

In the Verilog language, an undriven wire defaults to a value of Z (high impedance). However, FPGA Compiler II / FPGA *Express* leaves undriven wires unconnected. Multiple connections or assignments to a wire simply short the wires together.

In Example 3-7, two wires are declared: a is a single-bit wire, and b is a 3-bit vector of wires. Its most significant bit (msb) has an index of 2, and its least significant bit (lsb) has an index of 0.

Example 3-7 wire Declarations

```
wire a;  
wire [2:0] b;
```

You can assign a delay value in a wire declaration, and you can use the Verilog keywords *scalared* and *vectored* for simulation. FPGA Compiler II / FPGA *Express* accepts the syntax of these constructs, but they are ignored when the circuit is synthesized.

Note:

You can use delay information for modeling, but FPGA Compiler II / FPGA *Express* ignores delay information. If the functionality of your circuit depends on the delay information, FPGA Compiler II / FPGA *Express* might create logic whose behavior does not agree with the behavior of the simulated circuit.

wand

The wand (wired-AND) data type is a specific type of wire.

In Example 3-8, two variables drive the variable c. The value of c is determined by the logical AND of a and b.

Example 3-8 wand (wired-AND)

```
module wand_test(a, b, c);
    input a, b;
    output c;

    wand c;

    assign c = a;
    assign c = b;
endmodule
```

You can assign a delay value in a wand declaration, and you can use the Verilog keywords *scalared* and *vectored* for simulation. FPGA Compiler II / FPGA *Express* accepts the syntax of these constructs, but ignores the constructs during synthesis of the circuit.

wor

The *wor* (wired-OR) data type is a specific type of wire.

In Example 3-9, two variables drive the variable *c*. The value of *c* is determined by the logical OR of *a* and *b*.

Example 3-9 wor (wired-OR)

```
module wor_test(a, b, c);
    input a, b;
    output c;

    wor c;

    assign c = a;
    assign c = b;
endmodule
```

tri

The tri (three-state) data type is a specific type of wire. All variables that drive the tri must have a value of Z (high-impedance), except one. This single variable determines the value of the tri.

Note:

FPGA Compiler II / *FPGA Express* does not enforce the previous condition. You must ensure that no more than one variable driving a tri has a value other than Z.

In Example 3-10, three variables drive the variable out.

Example 3-10 tri (Three-State)

```
module tri_test (out, condition);
    input [1:0] condition;
    output out;

    reg a, b, c;
    tri out;

    always @ ( condition ) begin
        a = 1'bz;           //set all variables to Z
        b = 1'bz;
        c = 1'bz;
        case ( condition ) //set only one variable to non-Z
            2'b00 : a = 1'b1;
            2'b01 : b = 1'b0;
            2'b10 : c = 1'b1;
        endcase
    end

    assign out = a;        //make the tri connection
    assign out = b;
    assign out = c;
endmodule
```

supply0 and supply1

The supply0 and supply1 data types define wires tied to logic 0 (ground) and logic 1 (power). Using supply0 and supply1 is the same as declaring a wire and assigning a 0 or a 1 to it. In Example 3-11, power is tied to logic 1 and gnd (ground) is tied to logic 0.

Example 3-11 supply0 and supply1 Constructs

```
supply0 gnd;  
supply1 power;
```

reg

A reg represents a variable in Verilog. A reg can be a 1-bit quantity or a vector of bits. For a vector of bits, the range indicates the most significant bit and least significant bit of the vector. Both must be nonnegative constants, parameters, or constant-valued expressions. Example 3-12 shows some reg declarations.

Example 3-12 reg Declarations

```
reg x;           //single bit  
reg a,b,c;      //3 1-bit quantities  
reg [7:0] q;    //an 8-bit vector
```

Port Declarations

You must explicitly declare the direction (input, output, or bidirectional) of each port that appears in the port list of a port definition. Use the input, output, and inout statements, as described in the following sections.

input

You declare all input ports of a module with an input statement. An input is a type of wire and is governed by the syntax of wire. You can use a range specification to declare an input that is a vector of signals, as in the case of input b in the following example. The input statements can appear in any order in the description, but you must declare them before using them. For example,

```
input a;  
input [2:0] b;
```

output

You declare all output ports of a module with an output statement. Unless otherwise defined by a reg, wand, wor, or tri declaration, an output is a type of wire and is governed by the syntax of wire. An output statement can appear in any order in the description, but you must declare the statement before you use it.

You can use a range specification to declare an output that is a vector of signals. If you use a reg declaration for an output, the reg must have the same range as the vector of signals. For example,

```
output a;  
output [2:0] b;  
reg [2:0] b;
```

inout

You can declare bidirectional ports with the inout statement. An inout is a type of wire and is governed by the syntax of wire. FPGA Compiler II / FPGA *Express* allows you to connect only inout ports to module or gate instantiations. You must declare an inout before you use it. For example,

```
inout a;  
inout [2:0] b;
```

Continuous Assignment

If you want to drive a value onto a wire, wand, wor, or tri, use a continuous assignment to specify an expression for the wire value. You can specify a continuous assignment in two ways:

- Use an explicit continuous assignment statement after the wire, wand, wor, or tri declaration.
- Specify the continuous assignment in the same line as the declaration for a wire.

Example 3-13 shows two equivalent methods for specifying a continuous assignment for wire a.

Example 3-13 Two Equivalent Continuous Assignments

```
wire a;                //declare  
assign a = b & c;     //assign  
wire a = b & c;       //declare and assign
```

The left side of a continuous assignment can be

- A wire, wand, wor, or tri
- One or more bits selected from a vector
- A concatenation of any of these

The right side of the continuous assignment statement can be any supported Verilog operator or any arbitrary expression that uses previously declared variables and functions. You cannot assign a value to a reg in a continuous assignment.

Verilog allows you to assign drive strength for each continuous assignment statement. FPGA Compiler II / FPGA *Express* accepts drive strength, but it does not affect the synthesis of the circuit. Keep this in mind when you use drive strength in your Verilog source.

Assignments are done bitwise, with the low bit on the right side assigned to the low bit on the left side. If the number of bits on the right side is greater than the number on the left side, the high-order bits on the right side are discarded. If the number of bits on the left side is greater than the number on the right side, operands on the right side are zero-extended.

Module Instantiations

Module instantiations are copies of the logic in a module that define component interconnections.

```
module_name instance_name1 (terminal, terminal, ...),  
                instance_name2 (terminal, terminal, ...);
```

A module instantiation consists of the name of the module (*module_name*) followed by one or more instantiations. An instantiation consists of an instantiation name (*instance_name*) and a connection list. A connection list is a list of expressions called terminals, separated by commas. These terminals are connected to the ports of the instantiated module. Module instantiations have this syntax:

```
(terminal1, terminal2, ...),  
(terminal1, terminal2, ...);
```

Terminals connected to input ports can be any arbitrary expression. Terminals connected to output and inout ports can be identifiers, single- or multiple-bit slices of an array, or a concatenation of these. The bit-widths for a terminal and its module port must be the same.

If you use an undeclared variable as a terminal, the terminal is implicitly declared as a scalar (1-bit) wire. After the variable is implicitly declared as a wire, it can appear wherever a wire is allowed.

Example 3-14 shows the declaration for the module SEQ with two instantiations (SEQ_1 and SEQ_2).

Example 3-14 Module Instantiations

```
module SEQ(BUS0,BUS1,OUT); //description of module SEQ
    input BUS0, BUS1;
    output OUT;
    ...
endmodule

module top( D0, D1, D2, D3, OUT0, OUT1 );
    input D0, D1, D2, D3;
    output OUT0, OUT1;

    SEQ SEQ_1(D0,D1,OUT0), //instantiations of module SEQ
        SEQ_2(.OUT(OUT1),.BUS1(D3),.BUS0(D2));
endmodule
```

Named and Positional Notation

Module instantiations can use either named or positional notation to specify the terminal connections.

In name-based module instantiation, you explicitly designate which port is connected to each terminal in the list. Undesignated ports in the module are unconnected.

In position-based module instantiation, you list the terminals and specify connections to the module according to each terminal's position in the list. The first terminal in the connection list is connected to the first module port, the second terminal to the second module port, and so on. Omitted terminals indicate that the corresponding port on the module is unconnected.

In Example 3-14, SEQ_2 is instantiated by the use of named notation, as follows:

- Signal OUT1 is connected to port OUT of the module SEQ.
- Signal D3 is connected to port BUS1.
- Signal D2 is connected to port BUS0.

SEQ_1 is instantiated by the use of positional notation, as follows:

- Signal D0 is connected to port BUS0 of module SEQ.
- Signal D1 is connected to port BUS1.
- Signal OUT0 is connected to port OUT.

Parameterized Designs

The Verilog language allows you to create parameterized designs by overriding parameter values in a module during instantiation. You can do this with the defparam statement or with the following syntax:

```
module_name #( parameter_value, parameter_value, ... )  
    instance_name ( terminal_list )
```

FPGA Compiler II / FPGA *Express* does not support the defparam statement but does support the previous syntax.

The module in Example 3-15 contains a parameter declaration.

Example 3-15 *parameter Declaration in a Module*

```
module foo (a,b,c);  
  
    parameter width = 8;  
  
    input [width-1:0] a,b;  
    output [width-1:0] c;  
  
    assign c = a & b;  
  
endmodule
```

In Example 3-15, the default value of the parameter width is 8, unless you override the value when the module is instantiated. When you change the value, you build a different version of your design. This type of design is called a parameterized design.

FPGA Compiler II / FPGA *Express* automatically manages templates and parameters. Some errors due to parameter or port size mismatch are detected when an implementation is created, not when the Verilog is read.

Gate-Level Modeling

Verilog provides several basic logic gates that enable modeling at the gate level. Gate-level modeling is a special case of positional notation for module instantiation that uses a set of predefined module names. FPGA Compiler II / FPGA *Express* supports the following gate types:

- and
- nand
- or
- nor

- xor
- xnor
- buf
- not
- tran

Connection lists for instantiations of a gate-level model use positional notation. In the connection lists for and, nand, or, nor, xor, and xnor gates, the first terminal connects to the output of the gate and the remaining terminals connect to the inputs of the gate. You can build arbitrarily wide logic gates with as many inputs as you want.

Connection lists for buf, not, and tran gates also use positional notation. You can have as many outputs as you want, followed by only one input. Each terminal in a gate-level instantiation can be a 1-bit expression or signal.

In gate-level modeling, instance names are optional. Drive strengths and delays are allowed, but they FPGA Compiler II / FPGA *Express* ignores them. Example 3-16 shows two gate-level instantiations.

Example 3-16 Gate-Level Instantiations

```
buf (buf_out,e);  
and and4(and_out,a,b,c,d);
```

Note:

FPGA Compiler II / FPGA *Express* parses but ignores delay options for gate primitives. Because FPGA Compiler II / FPGA *Express* ignores the delay information, it can create logic whose behavior does not agree with the simulated behavior of the circuit. See “D Flip-Flop With Asynchronous Set or Reset” on page 6-24.

Three-State Buffer Instantiation

FPGA Compiler II / FPGA *Express* supports the following gate types for instantiation of three-state gates:

- buif0 (active-low enable line)
- buif1 (active-high enable line)
- notif0 (active-low enable line, output inverted)
- notif1 (active-high enable line, output inverted)

Connection lists for buif and notif gates use positional notation. Specify the order of the terminals as follows:

- The first terminal connects to the output of the gate.
- The second terminal connects to the input of the gate.
- The third terminal connects to the control line.

Example 3-17 shows a three-state gate instantiation with an active-high enable and no inverted output.

Example 3-17 Three-State Gate Instantiation

```
module three_state (in1,out1,cntrl1);
    input in1,cntrl1;
    output out1;

    buif1 (out1,in1,cntrl1);

endmodule
```

4

Expressions

In Verilog, expressions consist of a single operand or multiple operands separated by operators. Use expressions where a value is required in Verilog.

This chapter explains how to build and use expressions using

- Constant-Valued Expressions
- Operators
- Operands
- Expression Bit-Widths

Constant-Valued Expressions

A constant-valued expression is an expression whose operands are either constants or parameters. FPGA Compiler II / *FPGA Express* determines the value of these expressions.

In Example 4-1, `size-1` is a constant-valued expression. The expression `(op == ADD) ? a + b : a - b` is not a constant-valued expression, because the value depends on the variable `op`. If the value of `op` is 1, `b` is added to `a`; otherwise, `b` is subtracted from `a`.

Example 4-1 Valid Expressions

```
// all expressions are constant-valued,
// except in the assign statement.
module add_or_subtract( a, b, op, s );
// performs s = a+b if op is ADD
// performs s = a-b if op is not ADD
    parameter size=8;
    parameter ADD=1'b1;

    input  op;
    input  [size-1:0] a, b;
    output [size-1:0] s;
    assign s = (op == ADD) ? a+b : a-b;//not a constant-
                                                //valued expression
endmodule
```

The operators and operands in an expression influence the way a design is synthesized. FPGA Compiler II / *FPGA Express* evaluates constant-valued expressions and does not synthesize circuitry to compute their value. If an expression contains constants, they are propagated to reduce the amount of circuitry required. FPGA Compiler II / *FPGA Express* does synthesize circuitry for an expression that contains variables, however.

Operators

Operators identify the operation to be performed on their operands to produce a new value. Most operators are either unary operators, which apply to only one operand, or binary operators, which apply to two operands. Two exceptions are conditional operators, which take three operands, and concatenation operators, which take any number of operands.

The Verilog language operators supported by FPGA Compiler II / FPGA Express are listed in Table 4-1. A description of the operators and their order of precedence appears in the following sections.

Table 4-1 Verilog Operators Supported by FPGA Compiler II / FPGA Express

Operator Type	Operator	Description
Arithmetic Operators	+ - * / %	Arithmetic Modules
Relational Operators	> >= < <=	Relational
Equality Operators	== !=	Logical equality Logical inequality
Logical Operators	! && 	Logical NOT Logical AND Logical OR
Bitwise Operators	~ & ^ ^~ ~^	Bitwise NOT Bitwise AND Bitwise OR Bitwise XOR Bitwise XNOR
Reduction Operators	& ~& ~ ^ ~^ ^~	Reduction AND Reduction OR Reduction NAND Reduction NOR Reduction XOR Reduction XNOR

Table 4-1 Verilog Operators Supported by FPGA Compiler II / FPGA Express(Continued)

Operator Type	Operator	Description
Shift Operators	<<	Shift left
	>>	Shift right
Conditional Operator	? :	Conditions
Concatenation Operator	{ }	Concatenation

In the following descriptions, the terms *variable* and *variable operand* refer to operands or expressions that are not constant-valued expressions. This group includes wires and registers, bit-selects and part-selects of wires and registers, function calls, and expressions that contain any of these elements.

Arithmetic Operators

Arithmetic operators perform simple arithmetic on operands. The Verilog arithmetic operators are

- Addition (+)
- Subtraction (–)
- Multiplication (*)
- Division (/)
- Modules (%)

You can use the +, –, and * operators with any operand form (constants or variables). The + and – operators can be used as either unary or binary operators. FPGA Compiler II / FPGA Express requires that the / and % operators have constant-valued operands.

Example 4-2 shows three forms of the addition operator. The circuitry built for each addition operation is different, because of the different operand types. The first addition requires no logic, the second synthesizes an incrementer, and the third synthesizes an adder.

Example 4-2 Addition Operator

```
parameter size=8;
wire [3:0] a,b,c,d,e;

assign c = size + 2; //constant + constant
assign d = a + 1;   //variable + constant
assign e = a + b;   //variable + variable
```

Relational Operators

Relational operators compare two quantities and yield a 0 or 1 value. A true comparison evaluates to 1; a false comparison evaluates to 0. All comparisons assume unsigned quantities. The circuitry synthesized for relational operators is a bitwise comparator whose size is based on the sizes of the two operands.

The Verilog relational operators are

- Less than (<)
- Less than or equal to (<=)
- Greater than (>)
- Greater than or equal to (>=)

Example 4-3 shows the use of a relational operator.

Example 4-3 Relational Operator

```
function [7:0] max( a, b );
input  [7:0] a,b;
    if ( a >= b )    max = a;
    else             max = b;
endfunction
```

Equality Operators

Equality operators generate a 0 if the expressions being compared are not equal and a 1 if the expressions are equal. Equality and inequality comparisons are performed by bit.

The Verilog equality operators are

- Equality (==)
- Inequality (!=)

Example 4-4 shows the equality operator testing for a JMP instruction. The output signal jump is set to 1 if the two high-order bits of instruction are equal to the value of parameter JMP; otherwise, jump is set to 0.

Example 4-4 Equality Operator

```
module is_jump_instruction (instruction, jump);
    parameter JMP = 2'h3;

    input  [7:0] instruction;
    output jump;
    assign jump = (instruction[7:6] == JMP);

endmodule
```

Handling Comparisons to X or Z

FPGA Compiler II / FPGA *Express* always ignores comparisons to an X or a Z. If your code contains a comparison to an X or a Z, a warning message displays, indicating that the comparison is always evaluated to false, which might cause simulation to disagree with synthesis.

Example 4-5 shows code from a file called test2.v. FPGA Compiler II / FPGA *Express* always assigns the variable B to the value 1, because the comparison to X is ignored.

Example 4-5 Comparison to X Ignored

```
always begin
    if (A == 1'bx)    //this is line 10
        B = 0;
    else
        B = 1;
end
```

When FPGA Compiler II / FPGA *Express* reads this code, it generates the following warning message:

```
Warning: Comparisons to a "don't care" are treated as always
being false in routine test2 line 10 in file 'test2.v'. This
may cause simulation to disagree with synthesis. (HDL-170)
```

For an alternative method of handling comparisons to X or Z, use the `translate_off` and `translate_on` directives to comment out the condition and its first branch (the true clause) so that only the else branch goes through synthesis.

Logical Operators

Logical operators generate a 1 or a 0, according to whether an expression evaluates to true (1) or false (0). The Verilog logical operators are

- Logical NOT (!)
- Logical AND (&&)
- Logical OR (||)

The logical NOT operator produces a value of 1 if its operand is zero and a value of 0 if its operand is nonzero. The logical AND operator produces a value of 1 if both operands are nonzero. The logical OR operator produces a value of 1 if either operand is nonzero.

Example 4-6 shows some logical operators.

Example 4-6 Logical Operators

```
module is_valid_sub_inst(inst,mode,valid,unimp);

    parameter IMMEDIATE=2'b00, DIRECT=2'b01;
    parameter SUBA_imm=8'h80, SUBA_dir=8'h90,
              SUBB_imm=8'hc0, SUBB_dir=8'hd0;
    input [7:0] inst;
    input [1:0] mode;
    output valid, unimp;

    assign valid = (((mode == IMMEDIATE) && (
                    (inst == SUBA_imm) ||
                    (inst == SUBB_imm))) ||
                  ((mode == DIRECT) && (
                    (inst == SUBA_dir) ||
                    (inst == SUBB_dir))));

    assign unimp = !valid;
endmodule
```

Bitwise Operators

Bitwise operators act on the operand bit by bit. The Verilog bitwise operators are

- Unary negation (\sim)
- Binary AND ($\&$)
- Binary OR (\mid)
- Binary XOR (\wedge)
- Binary XNOR ($\wedge\sim$ or $\sim\wedge$)

Example 4-7 shows some bitwise operators.

Example 4-7 Bitwise Operators

```
module full_adder( a, b, cin, s, cout );
    input  a, b, cin;
    output s, cout;

    assign s      = a ^ b ^ cin;
    assign cout = (a&b) | (cin & (a|b));
endmodule
```

Reduction Operators

Reduction operators take one operand and return a single bit. For example, the reduction AND operator takes the AND value of all the bits of the operand and returns a 1-bit result. The Verilog reduction operators are

- Reduction AND (&)
- Reduction OR (|)
- Reduction NAND(~&)
- Reduction NOR (~|)
- Reduction XOR (^)
- Reduction NXOR (^~ or ~^)

Example 4-8 shows the use of some reduction operators.

Example 4-8 Reduction Operators

```
module check_input ( in, parity, all_ones );
    input  [7:0] in;
    output parity, all_ones;

    assign parity    = ^ in;
    assign all_ones = & in;
endmodule
```

Shift Operators

A shift operator takes two operands and shifts the value of the first operand right or left by the number of bits given by the second operand.

The Verilog shift operators are

- Shift left (<<)
- Shift right (>>)

After the shift, vacated bits fill with zeros. Shifting by a constant results in minor circuitry modification (because only rewiring is required). Shifting by a variable causes a general shifter to be synthesized. Example 4-9 shows use of a shift-right operator to perform division by 4.

Example 4-9 Shift Operator

```
module divide_by_4( dividend, quotient );
    input  [7:0] dividend;
    output [7:0] quotient;

    assign quotient = dividend >> 2; //shift right 2 bits
endmodule
```

Conditional Operator

The conditional operator (? :) evaluates an expression and returns a value that is based on the truth of the expression.

Example 4-10 shows how to use the conditional operator. If the expression (op == ADD) evaluates to true, the value a + b is assigned to result; otherwise, the value a – b is assigned to result.

Example 4-10 Conditional Operator

```
module add_or_subtract( a, b, op, result );

    parameter ADD=1'b0;
    input  [7:0] a, b;
    input  op;
    output [7:0] result;

    assign result = (op == ADD) ? a+b : a-b;
endmodule
```

You can nest conditional operators to produce an if...then construct. Example 4-11 shows the conditional operators used to evaluate the value of op successively and perform the correct operation.

Example 4-11 Nested Conditional Operator

```
module arithmetic( a, b, op, result );

    parameter ADD=3'h0, SUB=3'h1, AND=3'h2,
              OR=3'h3, XOR=3'h4;

    input  [7:0] a,b;
    input  [2:0] op;
    output [7:0] result;

    assign result = ((op == ADD) ? a+b : (
                    (op == SUB) ? a-b : (
                    (op == AND) ? a&b : (
                    (op == OR) ? a|b : (
                    (op == XOR) ? a^b : (a))))));

endmodule
```

Concatenation Operators

Concatenation combines one or more expressions to form a larger vector. In the Verilog language, you indicate concatenation by listing all expressions to be concatenated, separated by commas, in curly braces (`{}`). Any expression, except an unsized constant, is allowed in a concatenation. For example, the concatenation `{1'b1,1'b0,1'b0}` yields the value `3'b100`.

You can also use a constant-valued repetition multiplier to repeat the concatenation of an expression. The concatenation `{1'b1,1'b0,1'b0}` can also be written as `{1'b1,{2{1'b0}}}` to yield `3'b100`. The expression `{2{expr}}` within the concatenation repeats *expr* two times.

Example 4-12 shows a concatenation that forms the value of a condition-code register.

Example 4-12 Concatenation Operator

```
output [7:0] ccr;  
wire half_carry, interrupt, negative, zero, overflow, carry;  
...  
assign ccr = { 2'b00, half_carry, interrupt,  
              negative, zero, overflow, carry };
```

Example 4-13 shows an equivalent description for the concatenation.

Example 4-13 Concatenation Equivalent

```
output [7:0] ccr;  
...  
assign ccr[7] = 1'b0;  
assign ccr[6] = 1'b0;  
assign ccr[5] = half_carry;  
assign ccr[4] = interrupt;  
assign ccr[3] = negative;  
assign ccr[2] = zero;  
assign ccr[1] = overflow;  
assign ccr[0] = carry;
```

Operator Precedence

Table 4-2 lists the precedence of all operators, from highest to lowest. All operators at the same level in the table are evaluated from left to right, except the conditional operator (?:), which is evaluated from right to left.

Table 4-2 Operator Precedence

Operator	Description
[]	Bit-select or part-select
()	Parentheses
! ~	Logical and bitwise negation
& ~& ~ ^ ~^ ~	Reduction operators
+ -	Unary arithmetic
{ }	Concatenation
* / %	Arithmetic
+ -	Arithmetic
<< >>	Shift
> >= < <=	Relational
== !=	Logical equality and inequality
&	Bitwise AND
^ ^~ ~^	Bitwise XOR and XNOR
	Bitwise OR
&&	Logical AND
	Logical OR
? :	Conditional

Operands

You can use the following kinds of operands in an expression:

- Numbers
- Wires and registers
 - Bit-selects
 - Part-selects
- Function calls

The following sections explain each of these operands.

Numbers

A number is either a constant value or a value specified as a parameter. The expression `size-1` in Example 4-1 on page 4-2 illustrates how you can use both a parameter and a constant in an expression.

You can define constants as sized or unsized, in binary, octal, decimal, or hexadecimal bases. The default size of an unsized constant is 32 bits. Refer to “Numbers” on page B-13 for a discussion of the number format.

Wires and Registers

Variables that represent wires as well as registers are allowed in an expression. If the variable is a multiple-bit vector and you use only the name of the variable, the entire vector is used in the expression. Bit-selects and part-selects allow you to select single or multiple bits, respectively, from a vector. These are described in the next two sections.

Wires are described in “Module Statements and Constructs” on page 3-7, and registers are described in “Function Declarations” on page 5-3.

In the Verilog fragment shown in Example 4-14, a, b, and c are 8-bit vectors of wires. Because only the variable names appear in the expression, the entire vector of each wire is used in evaluation of the expression.

Example 4-14 Wire Operands

```
wire [7:0] a,b,c;  
assign c = a & b;
```

Bit-Selects

A bit-select is the selection of a single bit from a wire, register, or parameter vector. The value of the expression in brackets ([]) selects the bit you want from the vector. The selected bit must be within the declared range of the vector. Example 4-15 shows a simple example of a bit-select with an expression.

Example 4-15 Bit-Select Operands

```
wire [7:0] a,b,c;  
assign c[0] = a[0] & b[0];
```

Part-Selects

A part-select is the selection of a group of bits from a wire, register, or parameter vector. The part-select expression must be constant-valued in the Verilog language, unlike the bit-select operator. If a variable is declared with ascending or descending indexes, the part-select (when applied to that variable) must be in the same order.

You can also write the expression in Example 4-14 with part-select, as shown in Example 4-16.

Example 4-16 Part-Select Operands

```
assign c[7:0] = a[7:0] & b[7:0]
```

Function Calls

Verilog allows you to call one function from inside an expression and use the return value from the called function as an operand. Functions in Verilog return a value consisting of 1 or more bits. The syntax of a function call is the function name followed by a comma-separated list of function inputs enclosed in parentheses. Example 4-17 uses the function call legal in an expression.

Example 4-17 Function Call Used as an Operand

```
assign error = ! legal(in1, in2);
```

Functions are described in “Function Declarations” on page 5-3.

Concatenation of Operands

Concatenation is the process of combining several single- or multiple-bit operands into one large bit vector. The use of the concatenation operator, a pair of braces (`{}`), is described in “Concatenation Operators” on page 4-13.

Example 4-18 shows two 4-bit vectors (`nibble1` and `nibble2`) that are joined to form an 8-bit vector that is assigned to an 8-bit wire vector (`byte`).

Example 4-18 Concatenation of Operands

```
wire [7:0] byte;  
wire [3:0] nibble1, nibble2;  
assign byte = {nibble1,nibble2};
```

Expression Bit-Widths

The bit-width of an expression depends on the widths of the operands and the types of operators in the expression.

Table 4-3 shows the bit-width for each operand and operator. In the table, i , j , and k are expressions; $L(i)$ is the bit-width of expression i .

To preserve significant bits within an expression, Verilog fills in zeros for smaller-width operands. The rules for this zero extension depend on the operand type. These rules appear in Table 4-3.

Verilog classifies expressions (and operands) as either self-determined or context-determined. A self-determined expression is

one in which the width of the operands is determined solely by the expression itself. These operand widths are never extended.

Table 4-3 Expression Bit-Widths

Expression	Bit Length	Comments
unsized constant	32 bits	Self-determined
sized constant	as specified	Self-determined
$i + j$	$\max(L(i), L(j))$	Context-determined
$i - j$	$\max(L(i), L(j))$	Context-determined
$i * j$	$\max(L(i), L(j))$	Context-determined
i / j	$\max(L(i), L(j))$	Context-determined
$i \% j$	$\max(L(i), L(j))$	Context-determined
$i \& j$	$\max(L(i), L(j))$	Context-determined
$i j$	$\max(L(i), L(j))$	Context-determined
$i \wedge j$	$\max(L(i), L(j))$	Context-determined
$i \wedge \sim j$	$\max(L(i), L(j))$	Context-determined
$\sim i$	$L(i)$	Context-determined
$i == j$	1 bit	Self-determined
$i != j$	1 bit	Self-determined
$i \&\& j$	1 bit	Self-determined
$i j$	1 bit	Self-determined
$i > j$	1 bit	Self-determined
$i >= j$	1 bit	Self-determined
$i < j$	1 bit	Self-determined
$i <= j$	1 bit	Self-determined
$\&i$	1 bit	Self-determined
$ i$	1 bit	Self-determined
$\wedge i$	1 bit	Self-determined
$\sim \&i$	1 bit	Self-determined
$\sim i$	1 bit	Self-determined
$\sim \wedge i$	1 bit	Self-determined
$i \gg j$	$L(i)$	j is self-determined
$\{i\{j\}\}$	$i * L(j)$	j is self-determined
$i \ll j$	$L(i)$	j is self-determined
$\{i, \dots, j\}$	$L(i) + \dots + L(j)$	Self-determined
$\{i \{j, \dots, k\}\}$	$i * (L(j) + \dots + L(k))$	Self-determined

Table 4-3 Expression Bit-Widths(Continued)

Expression	Bit Length	Comments
$i?j:k$	$\text{Max}(L(j),L(k))$	i is self-determined

Example 4-19 shows a self-determined expression that is a concatenation of variables with known widths.

Example 4-19 Self-Determined Expression

```
output [7:0] result;  
wire [3:0] temp;  
  
assign temp = 4'b1111;  
assign result = {temp,temp};
```

The concatenation has two operands. Each operand has a width of 4 bits and a value of 4'b1111. The resulting width of the concatenation is 8 bits, which is the sum of the width of the operands. The value of the concatenation is 8'b11111111.

A context-determined expression is one in which the width of the expression depends on all the operand widths in the expression. For example, Verilog defines the resulting width of an addition as the greater of the widths of its two operands. The addition of two 8-bit quantities produces an 8-bit value; however, if the result of the addition is assigned to a 9-bit quantity, the addition produces a 9-bit result. Because the addition operands are context-determined, they are zero-extended to the width of the largest quantity in the entire expression.

Example 4-20 shows some context-determined expressions.

Example 4-20 Context-Determined Expressions

```
if ( ((1'b1 << 15) >> 15) == 1'b0 )  
    //This expression is ALWAYS true.
```

```
if ( (((1'b1 << 15) >> 15) | 20'b0) == 1'b0 )  
    //This expression is NEVER true.
```

The expression $((1'b1 \ll 15) \gg 15)$ produces a 1-bit 0 value ($1'b0$). The 1 is shifted off the left end of the vector, producing a value of 0. The right shift has no additional effect. For a shift operator, the first operand ($1'b1$) is context-dependent; the second operand (15) is self-determined.

The expression $((((1'b1 \ll 15) \gg 15) | 20'b0)$ produces a 20-bit 1 value ($20'b1$). $20'b1$ has a 1 in the least significant bit position and 0s in the other 19 bit positions. Because the largest operand in the expression has a width of 20, the first operand of the shift is zero-extended to a 20-bit value. The left shift of 15 does not drop the 1 value off the left end; the right shift brings the 1 value back to the right end, resulting in a 20-bit 1 value ($20'b1$).

5

Functional Descriptions

A Verilog functional description defines a circuit in terms of what it does.

This chapter describes the construction and use of functional descriptions in the following major sections:

- Sequential Constructs
- Function Declarations
- Function Statements
- task Statements
- always Blocks

Sequential Constructs

Although many Verilog constructs appear sequential in nature, they describe combinational circuitry. A simple description that appears to be sequential is shown in Example 5-1.

Example 5-1 Sequential Statements

```
x = b;  
if (y)  
    x = x + a;
```

FPGA Compiler II / *FPGA Express* determines the combinational equivalent of this description. In fact, it treats the statements in Example 5-1 exactly as it treats the statements in Example 5-2.

Example 5-2 Equivalent Combinational Description

```
if (y)  
    x = b + a;  
else  
    x = b;
```

To describe combinational logic, you write a sequence of statements and operators to generate the outputs you want. For example, suppose the addition operator (+) is not supported and you want to create a combinational ripple carry adder. The easiest way to describe this circuit is as a cascade of full adders, as in Example 5-3. The example has eight full adders, with each adder following the one before. From this description, *FPGA Compiler II / FPGA Express* generates a fully combinational adder.

Example 5-3 Combinational Ripple Carry Adder

```
function [7:0] adder;
input [7:0] a, b;
    reg c;
    integer i;
    begin
        c = 0;
        for (i = 0; i <= 7; i = i + 1) begin
            adder[i] = a[i] ^ b[i] ^ c;
            c = a[i] & b[i] | a[i] & c | b[i] & c;
        end
    end
endfunction
```

Function Declarations

Using a function declaration is one of three methods for describing combinational logic. The other two methods are the always block, described in “always Blocks” on page 5-34, and the continuous assignment, described in “Continuous Assignment” on page 3-15. You must declare and use Verilog functions within a module. You can call functions from the structural part of a Verilog description by using them in a continuous assignment statement or as a terminal in a module instantiation. You can also call functions from other functions or from always blocks.

FPGA Compiler II / FPGA *Express* supports the following Verilog function declarations:

- Input declarations
- Output from a function
- Register declarations

- Memory declarations
- Parameter declarations
- Integer declarations

Functions begin with the keyword `function` and end with the keyword `endfunction`. The width of the function's return value (if any) and the name of the function follow the function keyword, as the following syntax shows.

```
function [range] name_of_function ;
           [func_declaration]
           statement_or_null
endfunction
```

Defining the bit range of the return value is optional. Specify the range inside square brackets ([]). If you do not define the range, a function returns a 1-bit quantity by default. You set the function's output by assigning it to the function name. A function can contain one or more statements. If you use multiple statements, enclose the statements inside a `begin...end` pair.

A simple function declaration is shown in Example 5-4.

Example 5-4 Simple Function Declaration

```
function [7:0] scramble;
input [7:0] a;
input [2:0] control;
integer i;
    begin
        for (i = 0; i <= 7; i = i + 1)
            scramble[i] = a[ i ^ control ];
        end
    endfunction
```


Function statements FPGA Compiler II / FPGA *Express* supports are discussed in “Function Statements” on page 5-9.

Input Declarations

The input declarations specify the input signals for a function. You must declare the inputs to a Verilog function immediately after you declare the function name. The syntax of input declarations for a function is the same as the syntax of input declarations for a module:

```
input [range] list_of_variables ;
```

The optional range specification declares an input as a vector of signals. Specify range inside square brackets ([]).

Note:

The order in which you declare the inputs must match the order of the inputs in the function call.

Output From a Function

The output from a function is assigned to the function name. A Verilog function has only one output, which can be a vector. For multiple outputs from a function, use the concatenation operation to bundle several values into one return value. This single return value can then be unbundled by the caller. Example 5-5 shows how unbundling is done.

Example 5-5 Many Outputs From a Function

```
function [9:0] signed_add;
input [7:0] a, b;
    reg [7:0] sum;
    reg carry, overflow;

    begin
        ...
        signed_add = {carry, overflow, sum};
    end
endfunction
...
assign {C, V, result_bus} = signed_add(busA, busB);
```

The `signed_add` function bundles the values of `carry`, `overflow`, and `sum` into one value. This new value is returned in the `assign` statement following the function. The original values are then unbundled by the function that called the `signed_add` function.

Register Declarations

A register represents a variable in Verilog. The syntax for a register declaration is

```
reg [range] list_of_register_variables ;
```

A `reg` can be a single-bit quantity or a vector of bits. The range specifies the most significant bit (msb) and the least significant bit (lsb) of the vector enclosed in square brackets ([]). Both bits must be nonnegative constants, parameters, or constant-valued expressions. Example 5-6 shows some `reg` declarations.

Example 5-6 Register Declarations

```
reg x;                //single bit
reg a, b, c;         //3 single-bit quantities
reg [7:0] q;         //an 8-bit vector
```

The Verilog language allows you to assign a value to a reg variable only within a function or an always block.

In the Verilog simulator, reg variables can hold state information. A reg can hold its value across separate calls to a function. In some cases, FPGA Compiler II / FPGA *Express* emulates this behavior by inserting flow-through latches. In other cases, it emulates this behavior without a latch. The concept of holding state is elaborated on in “Inferring Latches” on page 6-7.

Memory Declarations

The memory declaration models a bank of registers or memory. In Verilog, the memory declaration is a two-dimensional array of reg variables. Sample memory declarations are shown in Example 5-7.

Example 5-7 Memory Declarations

```
reg [7:0] byte_reg;
reg [7:0] mem_block [255:0];
```

In Example 5-7, `byte_reg` is an 8-bit register and `mem_block` is an array of 256 registers, each of which is 8 bits wide. You can index the array of registers to access individual registers, but you cannot access individual bits of a register directly. Instead, you must copy the appropriate register into a temporary one-dimensional register. For example, to access the fourth bit of the eighth register in `mem_block`, enter

```
byte_reg = mem_block [7];  
individual_bit = byte_reg [3];
```

Parameter Declarations

Parameter variables are local or global variables that hold values. The syntax for a parameter declaration is

```
parameter [range] identifier = expression,  
        identifier = expression;
```

The range specification is optional.

You can declare parameter variables as being local to a function. However, you cannot use a local variable outside that function. Parameter declarations in a function are identical to parameter declarations in a module. The function in Example 5-8 contains a parameter declaration.

Example 5-8 Parameter Declaration in a Function

```
function gte;  
    parameter width = 8;  
    input [width-1:0] a,b;  
    gte = (a >= b);  
endfunction
```

Integer Declarations

Integer variables are local or global variables that hold numeric values. The syntax for an integer declaration is

```
integer identifier_list;
```

You can declare integer variables locally at the function level or globally at the module level. The default size for integers is 32 bits. FPGA Compiler II / FPGA *Express* determines bit-widths, except in the case of a don't care condition resulting during compile.

Example 5-9 illustrates integer declarations.

Example 5-9 Integer Declarations

```
integer a;           //single 32-bit integer
integer b, c;       //two integers
```

Function Statements

The function statements FPGA Compiler II / FPGA *Express* supports are

- Procedural assignments
- RTL assignments
- begin...end block statements
- if...else statements
- case, casex, and casez statements

- for loops
- while loops
- forever loops
- disable statements

Procedural Assignments

Procedural assignments are assignment statements used inside a function. They are similar to the continuous assignment statements described in “Continuous Assignment” on page 3-15, except that the left side of a procedural assignment can contain only reg variables and integers. Assignment statements set the value of the left side to the current value of the right side. The right side of the assignment can contain any arbitrary expression of the data types described in “Structural Data Types” on page 3-8, including simple constants and variables.

The left side of the procedural assignment statement can contain only the following data types:

- reg variables
- Bit-selects of reg variables
- Part-selects of reg variables (must be constant-valued)
- Integers
- Concatenations of the previous data types

FPGA Compiler II / *FPGA Express* assigns the low bit on the right side to the low bit on the left side. If the number of bits on the right side is greater than the number on the left side, the high-order bits on the right side are discarded. If the number of bits on the left side is greater than the number on the right side, the right-side bits are zero-extended. FPGA Compiler II / *FPGA Express* allows multiple procedural assignments.

Some examples of procedural assignments are shown in Example 5-10.

Example 5-10 Procedural Assignments

```
sum = a + b;  
control[5] = (instruction == 8'h2e);  
{carry_in, a[7:0]} = 9'h 120;
```

RTL Assignments

FPGA Compiler II / *FPGA Express* handles variables driven by an RTL (nonblocking) assignment differently than those driven by a procedural (blocking) assignment.

In procedural assignments, a value passed along from variable A to variable B to variable C results in all three variables having the same value in every clock cycle. In the netlist, procedural assignments are indicated when the input net of one flip-flop is connected to the input net of another flip-flop. Both flip-flops input the same value in the same clock cycle.

In RTL assignments, however, values are passed on in the next clock cycle. Assignment from variable A to variable B occurs after one clock cycle, if variable A has been a previous target of an RTL assignment. Assignment from variable B to variable C always takes place after one clock cycle, because B is the target when RTL assigns variable A's value to B. In the netlist, an RTL assignment shows flip-flop B receiving its input from the output net of flip-flop A. It takes one clock cycle for the value held by flip-flop A to propagate to flip-flop B.

A variable can follow only one assignment method and therefore cannot be the target of RTL as well as procedural assignments.

Example 5-11 is a description of a serial register implemented with RTL assignments. Figure 5-1 shows the resulting schematic for Example 5-11.

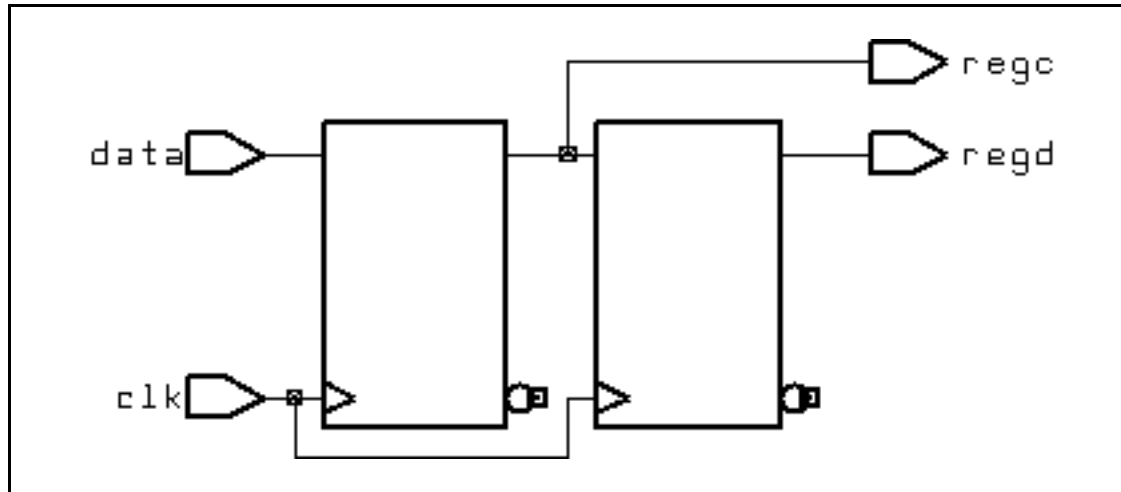
Example 5-11 RTL Nonblocking Assignments

```
module rtl (clk, data, regc, regd);
input data, clk;
output regc, regd;

reg regc, regd;

always @(posedge clk)
begin
    regc <= data;
    regd <= regc;
end
endmodule
```


Figure 5-1 Schematic of RTL Nonblocking Assignments



If you use a procedural assignment, as in Example 5-12, FPGA Compiler II / *FPGA Express* does not synthesize a serial register. Therefore, the recently assigned value of `rega`, which is `data`, is assigned to `regb`, as the schematic in Figure 5-2 indicates.

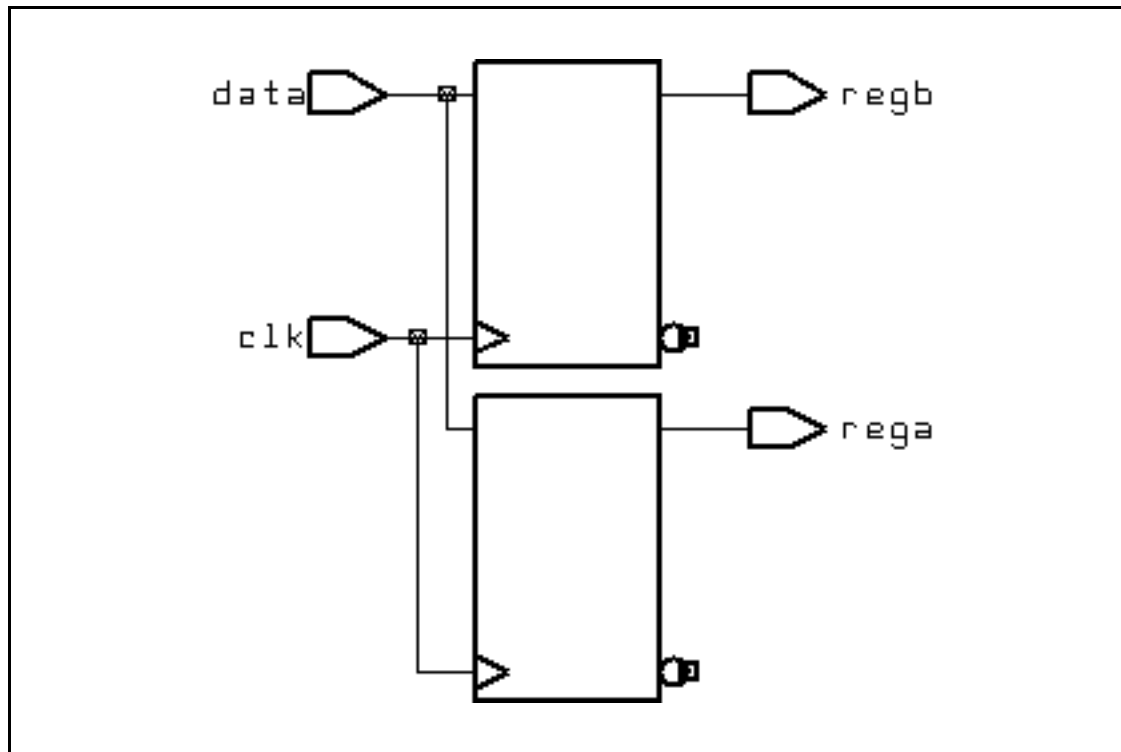
Example 5-12 Blocking Assignment

```
module rtl (clk, data, rega, regb);
input data, clk;
output rega, regb;

reg rega, regb;

always @(posedge clk)
begin
    rega = data;
    regb = rega;
end
endmodule
```

Figure 5-2 Schematic of Blocking Assignment



begin...end Block Statements

Using block statements is a way of syntactically grouping several statements into a single statement.

In Verilog, sequential blocks are delimited by the keywords `begin` and `end`. These `begin...end` pairs are commonly used in conjunction with `if`, `case`, and `for` statements to group several statements. Functions and `always` blocks that contain more than one statement require a `begin...end` pair to group the statements. Verilog also provides a construct called a named block, as in Example 5-13.

Example 5-13 Block Statement With a Named Block

```
begin : block_name
```

```
    reg local_variable_1;
      integer local_variable_2;
      parameter local_variable_3;
      ... statements ...
end
```

In Verilog, no semicolon (;) follows the begin or end keywords. You identify named blocks by following the begin with a colon (:), and a `block_name`, as shown. Verilog syntax allows you to declare variables locally in a named block. You can include reg, integer, and parameter declarations within a named block but not in an unnamed block. Named blocks allow you to use the disable statement.

if...else Statements

The if ... else statements execute a block of statements according to the value of one or more expressions.

The syntax of if...else statements is

```
if ( expr )
  begin
    ... statements ...
  end
else
  begin
    ... statements ...
  end
```

The if statement consists of the keyword if followed by an expression in parentheses. The if statement is followed by a statement or block of statements enclosed by begin and end. If the value of the expression is nonzero, the expression is true and the statement block that follows is executed. If the value of the expression is zero, the expression is false and the statement block that follows is *not* executed.

An optional else statement can follow an if statement. If the expression following if is false, the statement or block of statements following else is executed.

The if...else statements can cause synthesis of registers. Registers are synthesized when you do not assign a value to the same reg in all branches of a conditional construct. Information on registers is in “Register Inference” on page 6-1.

FPGA Compiler II / FPGA *Express* synthesizes multiplexer logic (or similar select logic) from a single if statement. The conditional expression in an if statement is synthesized as a control signal to a multiplexer, which determines the appropriate path through the multiplexer. For example, the statements in Example 5-14 create multiplexer logic controlled by c and place either a or b in the variable x.

Example 5-14 if Statement That Synthesizes Multiplexer Logic

```
if (c)
    x = a;
else
    x = b;
```

Example 5-15 illustrates how if and else can be used to create an arbitrarily long if...else if...else structure.

Example 5-15 if...else if...else Structure

```
if (instruction == ADD)
  begin
    carry_in = 0;
    complement_arg = 0;
  end
else if (instruction == SUB)
  begin
    carry_in = 1;
    complement_arg = 1;
  end
else
  illegal_instruction = 1;
```

Example 5-16 shows how to use nested if and else statements.

Example 5-16 Nested if and else Statements

```
if (select[1])
  begin
    if (select[0]) out = in[3];
    else out = in[2];
  end
else
  begin
    if (select[0]) out = in[1];
    else out = in[0];
  end
```

Conditional Assignments

FPGA Compiler II / FPGA *Express* can synthesize a latch for a conditionally assigned variable. A variable is conditionally assigned if there is a path that does not explicitly assign a value to that variable. See “Understanding the Limitations of D Latch Inference” on page 6-19 for more information.

In Example 5-17, the variable value is conditionally driven. If c is not true, value is not assigned and retains its previous value.

Example 5-17 Synthesizing a Latch for a Conditionally Driven Variable

```
always begin
  if ( c ) begin
    value = x;
  end
  y = value; //causes a latch to be synthesized for value
end
```

case Statements

The case statement is similar in function to the if...else conditional statement. The case statement allows a multipath branch in logic that is based on the value of an expression. One way to describe a multicycle circuit is with a case statement (see Example 5-18). Another way is with multiple @ (clock edge) statements, which are discussed in the subsequent sections on loops.

The syntax for a case statement is

```
case ( expr )
  case_item1: begin
    ... statements ...
  end
  case_item2: begin
    ... statements ...
  end
  default: begin
    ... statements ...
  end
endcase
```

The case statement consists of the keyword `case`, followed by an expression in parentheses, followed by one or more case items (and associated statements to be executed), followed by the keyword `endcase`. A case item consists of an expression (usually a simple constant) or a list of expressions separated by commas, followed by a colon (:).

The expression following the case keyword is compared with each case item expression, one by one. When the expressions are equal, the condition evaluates to true. Multiple expressions separated by commas can be used in each case item. When multiple expressions are used, the condition is said to be true if any of the expressions in the case item match the expression following the case keyword.

The first case item that evaluates to true determines the path. All subsequent case items are ignored, even if they are true. If no case item is true, no action is taken. You can define a default case item with the expression default, which is used when no other case item is true.

An example of a case statement is shown in Example 5-18.

Example 5-18 case Statement

```
case (state)
  IDLE: begin
    if (start)
      next_state = STEP1;
    else
      next_state = IDLE;
    end
  STEP1: begin
    //do first state processing here
    next_state = STEP2;
  end
  STEP2: begin
    //do second state processing here
    next_state = IDLE;
  end
endcase
```

Full Case and Parallel Case

FPGA Compiler II / FPGA *Express* automatically determines whether a case statement is full or parallel. A case statement is full if all possible branches are specified. If you do not specify all possible branches but you know that one or more branches can never occur, you can declare a case statement as full-case with the // synopsis

full_case directive. Otherwise, FPGA Compiler II / FPGA *Express* synthesizes a latch. See “parallel_case Directive” on page 8-4 and “full_case Directive” on page 8-5 for more information.

FPGA Compiler II / FPGA *Express* synthesizes optimal logic for the control signals of a case statement. If FPGA Compiler II / FPGA *Express* cannot statically determine that branches are parallel, it synthesizes hardware that includes a priority encoder. If FPGA Compiler II / FPGA *Express* can determine that no cases overlap (parallel case), it synthesizes a multiplexer, because a priority encoder is not necessary. You can also declare a case statement as parallel case with the //synopsys parallel_case directive. Refer to “parallel_case Directive” on page 8-4. Example 5-19 does not result in either a latch or a priority encoder.

Example 5-19 A case Statement That Is Both Full and Parallel

```
input [1:0] a;
always @(a or w or x or y or z) begin
    case (a)
        2'b11:
            b = w ;
        2'b10:
            b = x ;
        2'b01:
            b = y ;
        2'b00:
            b = z ;
    endcase
end
```

Example 5-20 shows a case statement that is missing branches for the cases 2'b01 and 2'b10. Example 5-20 infers a latch for b.

Example 5-20 A case Statement That Is Parallel but Not Full

```
input [1:0] a;
always @(a or w or z) begin
    case (a)
        2'b11:
            b = w ;
        2'00:
            b = z ;
    endcase
end
```

The case statement in Example 5-21 is not parallel or full because the values of inputs w and x cannot be determined. However, if you know that only one of the inputs equals 2'b11 at a given time, you can use the // synopsys parallel_case directive to avoid synthesizing a priority encoder. If you know that either w or x always equals 2'b11 (a situation known as a one-branch tree), you can use the // synopsys full_case directive to avoid synthesizing a latch.

Example 5-21 A case Statement That Is Not Full or Parallel

```
always @(w or x) begin
    case (2'b11)
        w:
            b = 10 ;
        x:
            b = 01 ;
    endcase
end
```

casex Statements

The casex statement allows a multipath branch in logic, according to the value of an expression, just as the case statement does. The differences between the case statement and the casex statement are the keyword and the processing of the expressions.

The syntax for a casex statement is

```
casex ( expr )
    case_item1: begin
        ... statements ...
    end
    case_item2: begin
        ... statements ...
    end
    default: begin
        ... statements ...
    end
endcase
```

A case item can have expressions consisting of

- A simple constant
- A list of identifiers or expressions separated by commas, followed by a colon (:)
- Concatenated, bit-selected, or part-selected expressions
- A constant containing z, x, or ?

When a z, x, or ? appears in a case item, it means that the corresponding bit of the casex expression is not compared. Example 5-22 shows a case item that includes an x.

Example 5-22 *casex Statement With x*

```
reg [3:0] cond;
casex (cond)
    4'b100x: out = 1;
    default: out = 0;
endcase
```

In Example 5-22, out is set to 1 if cond is equal to 4'b1000 or 4'b1001, because the last bit of cond is defined as x.

Example 5-23 shows a complicated section of code that can be simplified with a casex statement that uses the ? value.

Example 5-23 Before Using casex With ?

```
if (cond[3]) out = 0;
else if (!cond[3] & cond[2] ) out = 1;
else if (!cond[3] & !cond[2] & cond[1] ) out = 2;
else if (!cond[3] & !cond[2] & !cond[1] & cond[0] ) out = 3;
else if (!cond[3] & !cond[2] & !cond[1] & !cond[0] ) out = 4;
```

Example 5-24 shows the simplified version of the same code.

Example 5-24 After Using casex With ?

```
casex (cond)
    4'b1???: out = 0;
    4'b01???: out = 1;
    4'b001?: out = 2;
    4'b0001: out = 3;
    4'b0000: out = 4;
endcase
```

FPGA Compiler II / FPGA *Express* allows ?, z, and x bits in case items but not in casex expressions. Example 5-25 shows an invalid casex expression.

Example 5-25 Invalid casex Expression

```
express = 3'bxz?;
...
casex (express) //illegal testing of an expression
...
endcase
```

casez Statements

The casez statement allows a multipath branch in logic according to the value of an expression, just like the case statement. The differences between the case statement and the casez statement are the keyword and the way the expressions are processed. The casez statement acts exactly the same as casex, except that x is not allowed in case items; only z and ? are accepted as special characters.

The syntax for a casez statement is

```
casez ( expr )
    case_item1: begin
        ... statements ...
    end
    case_item2: begin
        ... statements ...
    end
    default: begin
        ... statements ...
    end
endcase
```

A case item can have expressions consisting of

- A simple constant
- A list of identifiers or expressions separated by commas, followed by a colon (:)
- Concatenated, bit-selected, or part-selected expressions
- A constant containing z or ?

When a casez statement is evaluated, the value z in the case item is ignored. An example of a casez statement with z in the case item is shown in Example 5-26.

Example 5-26 casez Statement With z

```
casez (what_is_it)
  2'bz0: begin
    //accept anything with least significant bit zero
    it_is = even;
  end
  2'bz1: begin
    //accept anything with least significant bit one
    it_is = odd;
  end
endcase
```

FPGA Compiler II / FPGA *Express* allows ? and z bits in case items but not in casez expressions. Example 5-27 shows an invalid expression in a casez statement.

Example 5-27 Invalid casez Expression

```
express = 1'bz;
...
casez (express) //illegal testing of an expression
...
endcase
```

for Loops

The for loop repeatedly executes a single statement or block of statements. The repetitions are performed over a range determined by the range expressions assigned to an index. Two range expressions appear in each for loop: *low_range* and *high_range*. In the syntax lines that follow, *high_range* is greater than or equal to *low_range*. FPGA Compiler II / FPGA *Express* recognizes incrementing as well as decrementing loops. The statement to be duplicated is surrounded by begin and end statements.

Note:

FPGA Compiler II / FPGA *Express* allows four syntax forms for a for loop. They are

```
for (index = low_range; index < high_range; index = index + step)
for (index = high_range; index > low_range; index = index - step)
for (index = low_range; index <= high_range; index = index + step)
for (index = high_range; index >= low_range; index = index - step)
```

Example 5-28 shows a simple for loop.

Example 5-28 A Simple for Loop

```
for (i = 0; i <= 31; i = i + 1) begin
    s[i] = a[i] ^ b[i] ^ carry;
    carry = a[i] & b[i] | a[i] & carry |
           b[i] & carry;
end
```

The for loops can be nested, as shown in Example 5-29.

Example 5-29 Nested for Loops

```
for (i = 6; i >= 0; i = i - 1)
  for (j = 0; j <= i; j = j + 1)
    if (value[j] > value[j+1]) begin
      temp = value[j+1];
      value[j+1] = value[j];
      value[j] = temp;
    end
```

You can use for loops as duplicating statements. Example 5-30 shows a for loop that is expanded into its longhand equivalent in Example 5-31.

Example 5-30 Example for Loop

```
for ( i=0; i < 8; i=i+1 )
  example[i] = a[i] & b[7-i];
```

Example 5-31 Expanded for Loop

```
example[0] = a[0] & b[7];
example[1] = a[1] & b[6];
example[2] = a[2] & b[5];
example[3] = a[3] & b[4];
example[4] = a[4] & b[3];
example[5] = a[5] & b[2];
example[6] = a[6] & b[1];
example[7] = a[7] & b[0];
```

while Loops

The while loop executes a statement until the controlling expression evaluates to false. A while loop creates a conditional branch that must be broken by one of the following statements to prevent combinational feedback.

```
@ (posedge clock)
```

or

```
@ (negedge clock)
```

FPGA Compiler II / FPGA *Express* supports while loops if you insert one of these expressions in every path through the loop:

```
@ (posedge clock)
```

or

```
@ (negedge clock)
```

Example 5-32 shows an unsupported while loop that has no event expression.

Example 5-32 Unsupported while Loop

```
always
  while (x < y)
    x = x + z;
```

If you add @ (posedge clock) expressions after the while loop in Example 5-32, you get the supported version shown in Example 5-33.

Example 5-33 Supported while Loop

```
always
  begin @ (posedge clock)
    while (x < y)
      begin
        @ (posedge clock);
        x = x + z;
      end
    end
end;
```

forever Loops

Infinite loops in Verilog use the keyword *forever*. You must break up an infinite loop with an @ (posedge clock) or @ (negedge clock) expression to prevent combinational feedback, as shown in Example 5-34.

Example 5-34 Supported forever Loop

```
always
  forever
  begin
    @ (posedge clock);
    x = x + z;
  end
```

You can use forever loops with a disable statement to implement synchronous resets for flip-flops. The disable statement is described in the next section. See “Register Inference” on page 6-1 for more information on synchronous resets.

Using the style illustrated in Example 5-34 is not a good idea, because you cannot test it. The synthesized state machine does not reset to a known state; therefore, it is impossible to create a test program for it. Example 5-36 illustrates how a synchronous reset for the state machine can be synthesized.

disable Statements

FPGA Compiler II / FPGA *Express* supports the disable statement when you use it in named blocks. When a disable statement is executed, it causes the named block to terminate. A comparator description that uses disable is shown in Example 5-35.

Example 5-35 Comparator Using disable

```
begin : compare
    for (i = 7; i >= 0; i = i - 1) begin
        if (a[i] != b[i]) begin
            greater_than = a[i];
            less_than = ~a[i];
            equal_to = 0;
            //comparison is done so stop looping
            disable compare;
        end
    end
end

// If we get here a == b
// If the disable statement is executed, the next three
// lines will not be executed
greater_than = 0;
less_than = 0;
equal_to = 1;
end
```

Example 5-35 describes a combinational comparator. Although the description appears sequential, the generated logic runs in a single clock cycle.

You can also use a disable statement to implement a synchronous reset, as shown in Example 5-36.

Example 5-36 Synchronous Reset of State Register Using disable in a forever Loop

```
always
  forever
  begin: Block
    @ (posedge clk)
    if (Reset)
      begin
        z <= 1'b0;
        disable Block;
      end
    z <= a;
  end
```

The disable statement in Example 5-36 causes the block reset_label to terminate immediately and return to the beginning of the block.

task Statements

In Verilog, task statements are similar to functions, but task statements can have output and inout ports. You can use the task statement to structure your Verilog code so that a portion of code is reusable.

In Verilog, tasks can have timing controls and can take a nonzero time to return. However, FPGA Compiler II / *FPGA Express* ignores all timing controls, so synthesis might disagree with simulation if timing controls are critical to the function of the circuit.

Example 5-37 shows how a task statement is used to define an adder function.

Example 5-37 Using the task Statement

```
module task_example (a,b,c);
    input [7:0] a,b;
    output [7:0] c;
    reg [7:0] c;

    task adder;
        input [7:0] a,b;
        output [7:0] adder;
        reg c;
        integer i;

        begin
            c = 0;
            for (i = 0; i <= 7; i = i+1) begin
                adder[i] = a[i] ^ b[i] ^ c;
                c = (a[i] & b[i]) | (a[i] & c) | (b[i] & c);
            end
        end
    endtask
    always
        adder (a,b,c); //c is a reg

endmodule
```

Note:

Only reg variables can receive output values from a task; wire variables cannot.

always Blocks

An always block can imply latches or flip-flops, or it can specify purely combinational logic. An always block can contain logic triggered in response to a change in a level or the rising or falling edge of a signal. The syntax of an always block is

```
always @ ( event-expression [or event-expression*] ) begin
    ... statements ...
end
```

Event Expression

The event expression declares the triggers or timing controls. The word *or* groups several triggers. The Verilog language specifies that if triggers in the event expression occur, the block is executed. Only one trigger in a group of triggers needs to occur for the block to be executed. However, FPGA Compiler II / FPGA *Express* ignores the event expression unless it is a synchronous trigger that infers a register. Refer to Chapter 6, "Register and Three-State Inference", for details.

Example 5-38 shows a simple example of an always block with triggers.

Example 5-38 A Simple always Block

```
always @ ( a or b or c ) begin
    f = a & b & c
end
```

In Example 5-38, a, b, and c are asynchronous triggers. If any triggers change, the simulator resimulates the always block and recalculates the value of f. FPGA Compiler II / FPGA *Express* ignores the triggers in this example because they are not synchronous. However, you must indicate all variables that are read in the always block as triggers. If you do not indicate all the variables as triggers, FPGA Compiler II / FPGA *Express* gives a warning message similar to the following:

```
Warning: Variable 'foo' is being read in block 'bar' declared
on line 88 but does not occur in the timing control of the
block.
```

For a synchronous always block, FPGA Compiler II / FPGA *Express* does not require listing of all variables.

Any of the following types of event expressions can trigger an always block:

- A change in a specified value. For example,

```
always @ ( identifier ) begin
    ... statements ...
end
```

In the previous example, FPGA Compiler II / FPGA *Express* ignores the trigger.

- The rising edge of a clock. For example,

```
always @ ( posedge event ) begin
    ... statements ...
end
```

- The falling edge of a clock. For example,

```
always @ ( negedge event ) begin
    ... statements ...
end
```

- A clock or an asynchronous preload condition. For example,

```
always @ ( posedge CLOCK or negedge reset ) begin
    if !reset begin
        ... statements ...
    end
    else begin
        ... statements ...
    end
end
```

- An asynchronous preload that is based on two events joined by the word *or*. For example,

```
always @ ( posedge CLOCK or posedge event1 or
           negedge event2 ) begin
    if ( event1 ) begin
        ... statements ...
    end
    else if ( !event2 ) begin
        ... statements ...
    end
    else begin
        ... statements ...
    end
end
```

When the event expression does not contain *posedge* or *negedge*, combinational logic (no registers) is usually generated, although flow-through latches can be generated.

Note:

The statements *@ (posedge clock)* and *@ (negedge clock)* are not supported in functions or tasks.

Incomplete Event Specification

You risk misinterpretation of an always block if you do not list all the signals entering an always block in the event specification. Example 5-39 shows an incomplete event list.

Example 5-39 Incomplete Event List

```
always @(a or b) begin
    f = a & b & c;
end
```

FPGA Compiler II / FPGA *Express* builds a 3-input AND gate for the description in Example 5-39, but in simulation of this description, *f* is not recalculated when *c* changes, because *c* is not listed in the event expression. The simulated behavior is not that of a 3-input AND gate.

The simulated behavior of the description in Example 5-40 is correct, because it includes all the signals in the event expression.

Example 5-40 Complete Event List

```
always @(a or b or c) begin
    f = a & b & c;
end
```

In some cases, you cannot list all the signals in the event specification. Example 5-41 illustrates this problem.

Example 5-41 Incomplete Event List for Asynchronous Preload

```
always @ (posedge c or posedge p)
    if (p)
        z = d;
    else
        z = a;
```

In the logic synthesized for Example 5-41, if *d* changes while *p* is high, the change is reflected immediately in the output, *z*. However, when this description is simulated, *z* is not recalculated when *d* changes, because *d* is not listed in the event specification. As a result, synthesis might not match simulation.

Asynchronous preloads can be correctly modeled in FPGA Compiler II / *FPGA Express* only when you want changes in the load data to be reflected immediately in the output. In Example 5-41, data *d* must change to the preload value before preload condition *p* transits from low to high. If you attempt to read a value in an asynchronous preload, FPGA Compiler II / *FPGA Express* prints a warning similar to the following:

```
Warning:Variable 'd' is being read asynchronously in routine
reset line 21 in file '/usr/tests/hdl/asyn.v'. This may cause
simulation-synthesis mismatches.
```

6

Register and Three-State Inference

FPGA Compiler II / FPGA *Express* can infer registers (latches and flip-flops) and three-state cells. This chapter explains inference behavior and results, in the following sections:

- Register Inference
- Three-State Inference

Register Inference

Register inference allows you to use sequential logic in your designs and keep your designs technology-independent. A register is a simple, 1-bit memory device, either a latch or a flip-flop. A latch is a level-sensitive memory device. A flip-flop is an edge-triggered memory device.

The register inference capability can support coding styles other than those described in this chapter. However, for best results,

- Restrict each always block to a single type of memory-element inferencing: latch, latch with asynchronous set or reset, flip-flop, flip-flop with asynchronous reset, or flip-flop with synchronous reset.
- Use the templates provided in “Inferring Latches” on page 6-7 and “Inferring Flip-Flops” on page 6-21

The Inference Report

FPGA Compiler II / *FPGA Express* generates a general inference report when building a design. It provides the asynchronous set or reset, synchronous set or reset, and synchronous toggle conditions of each latch or flip-flop, expressed as Boolean formulas. Example 6-1 shows an inference report for a JK flip-flop.

Example 6-1 Inference Report for a JK Flip-Flop

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	Y	Y	N

Q_reg

Sync-reset: J' K

Sync-set: J K'

Sync-toggle: J K

Sync-set and Sync-reset ==> Q: X

In the inference reports in Example 6-1:

- Y indicates that the flip-flop has a synchronous reset (SR) and a synchronous set (SS)
- N indicates that the flip-flop does not have an asynchronous reset (AR), an asynchronous set (AS), or a synchronous toggle (ST)

In the inference report (Example 6-1), the last part of the report lists the objects that control the synchronous reset and set conditions. In this example, a synchronous reset occurs when J is low (logic 0) and K is high (logic 1). The last line of the report indicates the register output value when both set and reset are active:

zero (0)

Indicates that the reset has priority and that the output goes to logic 0.

one (1)

Indicates that the set has priority and that the output goes to logic 1.

X

Indicates that there is no priority and that the output is unstable.

“Inferring Latches” on page 6-7 and “Inferring Flip-Flops” on page 6-21 provide inference reports for each register template. After you input a Verilog description, check the inference report to verify the information.

Latch Inference Warnings

FPGA Compiler II / *FPGA Express* generates a warning message when it infers a latch. This is useful for verifying that a combinational design does not contain memory components.

Controlling Register Inference

Use directives to direct the type of sequential device you want inferred. The default is to implement the type of latch described in the HDL code. These attributes override this behavior.

Attributes That Control Register Inference

FPGA Compiler II / *FPGA Express* provides the following directives for controlling register inference:

`async_set_reset`

When a signal has this directive set to true, FPGA Compiler II / *FPGA Express* searches for a branch that uses the signal as a condition. *FPGA Compiler II / FPGA Express* then checks whether the branch contains an assignment to a constant value. If the branch does, the signal becomes an asynchronous reset or set.

Attach this directive to single-bit signals, using the following syntax:

```
// synopsys async_set_reset "signal_name_list"
```

`async_set_reset_local`

FPGA Compiler II / *FPGA Express* treats listed signals in the specified block as if they have the `async_set_reset` directive set to true.

Attach this directive to a block label, using the following syntax:

```
/* synopsys async_set_reset_local block_label
   "signal_name_list" */
```

async_set_reset_local_all

FPGA Compiler II / FPGA *Express* treats all signals in the specified blocks as if they have the `async_set_reset` directive set to true.

Attach this directive to block labels, using the following syntax:

```
/* synopsys async_set_reset_local_all
   "block_label_list" */
```

sync_set_reset

When a signal has this directive set to true, FPGA Compiler II / FPGA *Express* checks the signal to determine whether it synchronously sets or resets a register in the design.

Attach this directive to single-bit signals, using the following syntax:

```
//synopsys sync_set_reset "signal_name_list"
```

sync_set_reset_local

FPGA Compiler II / FPGA *Express* treats listed signals, in the specified block as if they have the `sync_set_reset` directive set to true.

Attach this directive to a block label, using the following syntax:

```
/* synopsys sync_set_reset_local block_label
   "signal_name_list" */
```

sync_set_reset_local_all

FPGA Compiler II / FPGA *Express* treats all signals in the specified blocks as if they have the `sync_set_reset` directive set to true.

Attach this directive to block labels, using the following syntax:

```
/* synopsys sync_set_reset_local_all  
   "block_label_list" */
```

one_cold

A one-cold implementation means that all signals in a group are active low and that only one signal can be active at a given time. The `one_cold` directive prevents FPGA Compiler II / FPGA *Express* from implementing priority encoding logic for the set and reset signals.

Add a check to the Verilog code to ensure that the group of signals has a one-cold implementation. FPGA Compiler II / FPGA *Express* does not produce any logic to check this assertion.

Attach this directive to set or reset signals on sequential devices, using the following syntax:

```
// synopsys one_cold "signal_name_list"
```

one_hot

A one-hot implementation means that all signals in a group are active-high and that only one signal can be active at a given time. The `one_hot` directive prevents FPGA Compiler II / FPGA *Express* from implementing priority encoding logic for the set and reset signals.

Add a check to the Verilog code to ensure that the group of signals has a one-hot implementation. FPGA Compiler II / FPGA *Express* does not produce any logic to check this assertion.

Attach this directive to set or reset signals on sequential devices, using the following syntax:

```
// synopsys one_hot "signal_name_list"
```

Inferring Latches

In simulation, a signal or variable holds its value until that output is reassigned. In hardware, a latch implements this holding-of-state capability. FPGA Compiler II / FPGA *Express* supports inference of the following types of latches:

- SR latch
- D latch
- Master-slave latch

The following sections provide details about each of these latch types.

Inferring SR Latches

Use SR latches with caution, because they are difficult to test. If you decide to use SR latches, verify that the inputs are hazard-free (that they do not glitch). During synthesis, FPGA Compiler II / FPGA *Express* does not ensure that the logic driving the inputs is hazard-free.

Example 6-2 shows the Verilog code that implements the inferred SR latch shown in Figure 6-1 on page 6-9 and described in Table 6-1. Because the output *y* is unstable when both inputs have a logic 0 value, you might want to include a check in the Verilog code to detect this condition during simulation.

Synthesis does not support such checks, so you must put the `translate_off` and `translate_on` directives around the check. See “`translate_off` and `translate_on` Directives” on page 8-2 for more information about special comments in the Verilog source code.

Example 6-3 shows the inference report FPGA Compiler II / FPGA Express generates.

Table 6-1 SR Latch Truth Table (Nand Type)

set	reset	y
0	0	Not stable
0	1	1
1	0	0
1	1	y

Example 6-2 SR Latch

```

module sr_latch (SET, RESET, Q);
  input SET, RESET;
  output Q;
  reg Q;

  //synopsys async_set_reset "SET, RESET"
  always @(RESET or SET)
    if (~RESET)
      Q = 0;
    else if (~SET)
      Q = 1;
endmodule

```

Example 6-3 Inference Report for an SR Latch

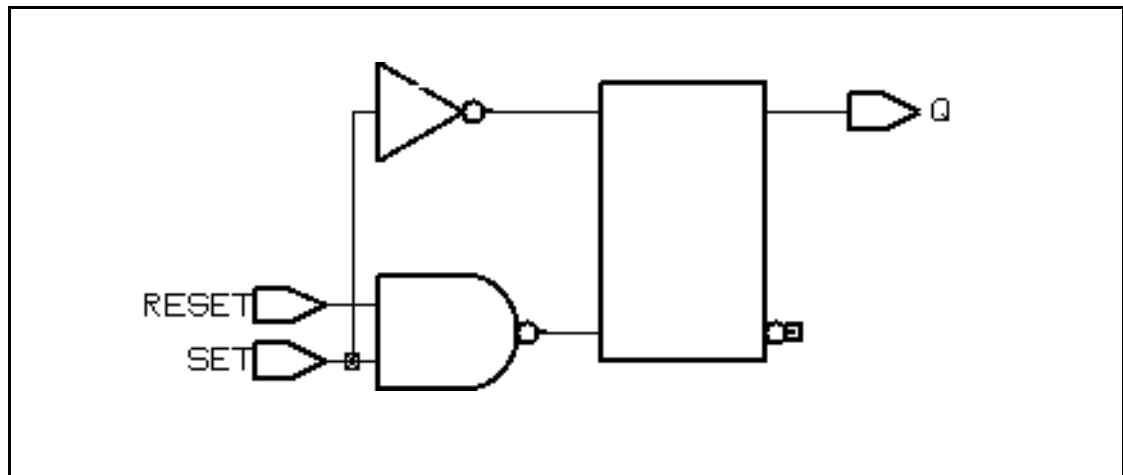
Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Latch	1	-	-	Y	Y	-	-	-

```

y_reg
  Async-reset: RESET'
  Async-set: SET'
  Async-set and Async-reset ==> Q: 1

```

Figure 6-1 SR Latch



Inferring D Latches

When you do not specify the resulting value for an output under all conditions, as in an incompletely specified if or case statement, FPGA Compiler II / FPGA Express infers a D latch.

For example, the if statement in Example 6-4 infers a D latch because there is no else clause. The Verilog code specifies a value for output Q only when input enable has a logic 1 value. As a result, output Q becomes a latched value.

Example 6-4 Latch Inference Using an if Statement

```
always @ (DATA or GATE) begin
    if (GATE) begin
        Q = DATA;
    end
end
```

The case statement in Example 6-5 infers D latches because the case statement does not provide assignments to decimal for values of I between 10 and 15.

Example 6-5 Latch Inference Using a case Statement

```
always @(I) begin
    case(I)
        4'h0: decimal= 10'b0000000001;
        4'h1: decimal= 10'b0000000010;
        4'h2: decimal= 10'b0000000100;
        4'h3: decimal= 10'b0000001000;
        4'h4: decimal= 10'b0000010000;
        4'h5: decimal= 10'b0000100000;
        4'h6: decimal= 10'b0001000000;
        4'h7: decimal= 10'b0010000000;
        4'h8: decimal= 10'b0100000000;
        4'h9: decimal= 10'b1000000000;
    endcase
end
```

To avoid latch inference, assign a value to the signal under all conditions. To avoid latch inference by the if statement in Example 6-4, modify the block as shown in Example 6-6 or Example 6-7. To avoid latch inference by the case statement in Example 6-5, add the following statement before the endcase statement:

```
default: decimal= 10'b0000000000;
```

Example 6-6 *Avoiding Latch Inference*

```
always @ (DATA, GATE) begin
    Q = 0;
    if (GATE)
        Q = DATA;
end
```

Example 6-7 *Another Way to Avoid Latch Inference*

```
always @ (DATA, GATE) begin
    if (GATE)
        Q = DATA;
    else
        Q = 0;
end
```

Variables declared locally within a subprogram do not hold their value over time because every time a subprogram is called, its variables are reinitialized. Therefore, FPGA Compiler II / *FPGA Express* does not infer latches for variables declared in subprograms. In Example 6-8, FPGA Compiler II / *FPGA Express* does not infer a latch for output Q.

Example 6-8 *Function: No Latch Inference*

```
function MY_FUNC
    input DATA, GATE;
    reg STATE;

    begin
        if (GATE) begin
            STATE = DATA;
        end
        MY_FUNC = STATE;
    end
end function
. . .
Q = MY_FUNC(DATA, GATE);
```

The following sections provide truth tables, code examples, and figures for these types of D latches:

- Simple D Latch
- D Latch With Asynchronous Set or Reset
- D Latch With Asynchronous Set and Reset

Simple D Latch

When you infer a D latch, make sure you can control the gate and data signals from the top-level design ports or through combinational logic. Controllable gate and data signals ensure that simulation can initialize the design.

Example 6-9 provides the Verilog template for a D latch. FPGA Compiler II / *FPGA Express* generates the inference report shown in Example 6-10. Figure 6-2 shows the inferred latch.

Example 6-9 D Latch

```
module d_latch (GATE, DATA, Q);
    input GATE, DATA;
    output Q;
    reg Q;

    always @(GATE or DATA)
        if (GATE)
            Q = DATA;

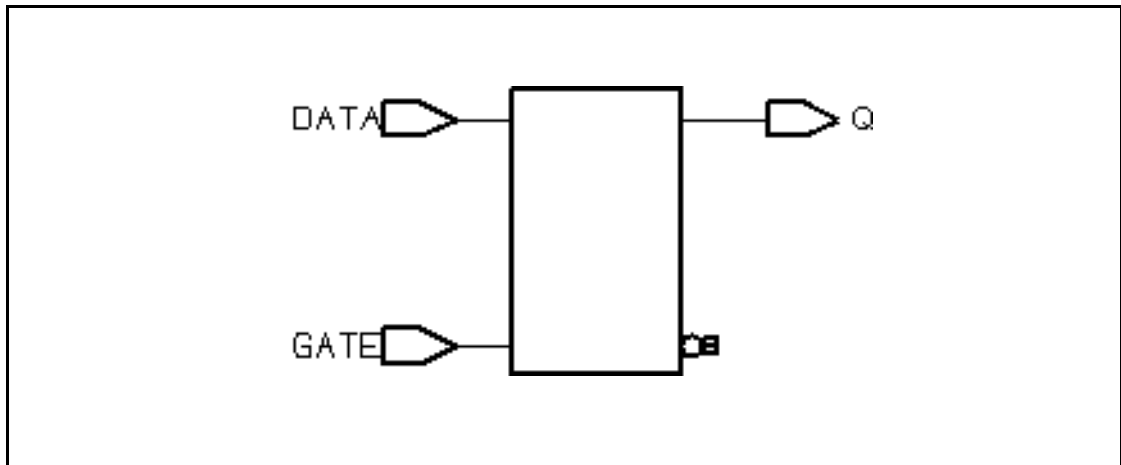
endmodule
```

Example 6-10 Inference Report for a D Latch

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Latch	1	-	-	N	N	-	-	-

Q_reg
reset/set: none

Figure 6-2 D Latch



D Latch With Asynchronous Set or Reset

The templates in this section use the `async_set_reset` directive to direct FPGA Compiler II / *FPGA Express* to the asynchronous set or reset pins of the inferred latch.

Example 6-11 provides the Verilog template for a D latch with an asynchronous set. FPGA Compiler II / *FPGA Express* generates the inference report shown in Example 6-12. Figure 6-3 shows the inferred latch.

Example 6-11 D Latch With Asynchronous Set

```
module d_latch_async_set (GATE, DATA, SET, Q);
    input GATE, DATA, SET;
    output Q;
    reg Q;

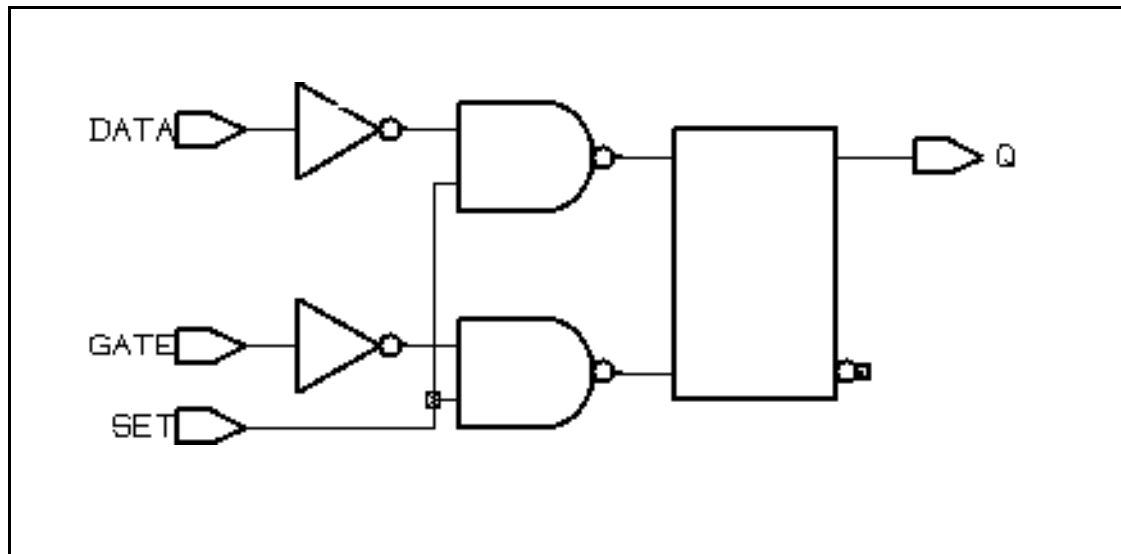
    //synopsys async_set_reset "SET"
    always @(GATE or DATA or SET)
        if (~SET)
            Q = 1'b1;
        else if (GATE)
            Q = DATA;
endmodule
```

Example 6-12 Inference Report for D Latch With Asynchronous Set

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Latch	1	-	-	N	Y	-	-	-

Q_reg
Async-set: SET'

Figure 6-3 D Latch With Asynchronous Set



Note:

Because the target technology library does not contain a latch with an asynchronous set, FPGA Compiler II / *FPGA Express* synthesizes the set logic by using combinational logic.

Example 6-13 provides the Verilog template for a D latch with an asynchronous reset. FPGA Compiler II / *FPGA Express* generates the inference report shown in Example 6-14. Figure 6-4 shows the inferred latch.

Example 6-13 D Latch With Asynchronous Reset

```
module d_latch_async_reset (RESET, GATE, DATA, Q);
    input RESET, GATE, DATA;
    output Q;
    reg Q;

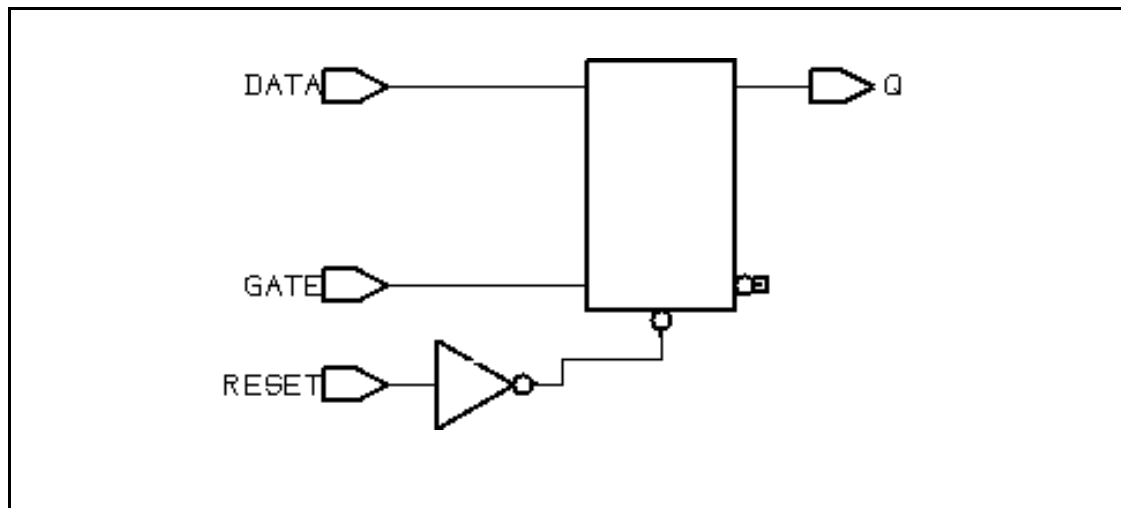
    //synopsys async_set_reset "RESET"
    always @ (RESET or GATE or DATA)
        if (~RESET)
            Q = 1'b0;
        else if (GATE)
            Q = DATA;
endmodule
```

Example 6-14 Inference Report for D Latch With Asynchronous Reset

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Latch	1	-	-	Y	N	-	-	-

Q_reg
Async-reset: RESET'

Figure 6-4 D Latch With Asynchronous Reset



D Latch With Asynchronous Set and Reset

Example 6-15 provides the Verilog template for a D latch with an active-low asynchronous set and reset. This template uses the `async_set_reset_local` directive to direct FPGA Compiler II / *FPGA Express* to the asynchronous signals in block infer. This template uses the `one_cold` directive to prevent priority encoding of the set and reset signals.

For this template, if you do not specify the `one_cold` directive, the set signal has priority, because it serves as the condition for the if clause. *FPGA Compiler II / FPGA Express* generates the inference report shown in Example 6-16. Figure 6-5 shows the inferred latch.

Example 6-15 D Latch With Asynchronous Set and Reset

```
module d_latch_async (GATE, DATA, RESET, SET, Q);
    input GATE, DATA, RESET, SET;
    output Q;
    reg Q;

    // synopsys async_set_reset_local infer "RESET, SET"
    // synopsys one_cold "RESET, SET"
    always @ (GATE or DATA or RESET or SET)
    begin : infer
        if (!SET)
            Q = 1'b1;
        else if (!RESET)
            Q = 1'b0;
        else if (GATE)
            Q = DATA;
    end

    // synopsys translate_off
    always @ (RESET or SET)
        if (RESET == 1'b0 & SET == 1'b0)
            $write ("ONE-COLD violation for RESET and SET.");
    // synopsys translate_on
endmodule
```

Example 6-16 Inference Report for D Latch With Asynchronous Set and Reset

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Latch	1	-	-	Y	Y	-	-	-

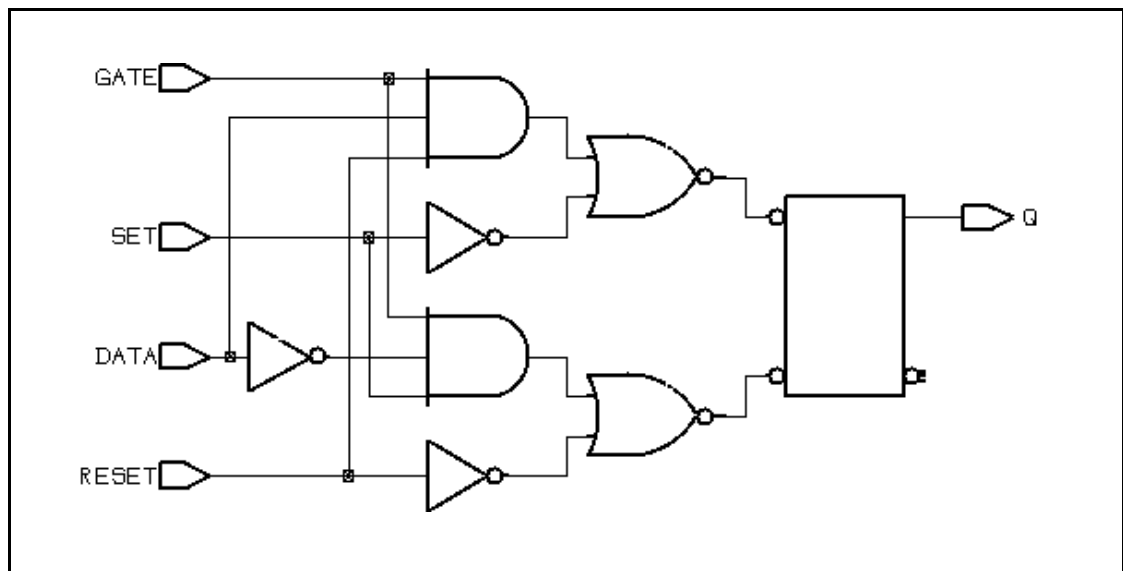
Q_reg

Async-reset: RESET'

Async-set: SET'

Async-set and Async-reset ==> Q: X

Figure 6-5 D Latch With Asynchronous Set and Reset



Understanding the Limitations of D Latch Inference

A variable must always have a value before it is read. As a result, a conditionally assigned variable cannot be read after the if statement in which it is assigned. A conditionally assigned variable is assigned a new value under some, but not all, conditions. Example 6-17 shows an invalid use of the conditionally assigned variable VALUE.

Example 6-17 Invalid Use of a Conditionally Assigned Variable

```
begin
    if (condition) then
        VALUE <= X;
    Y <= VALUE; // Invalid read of variable VALUE
end
```

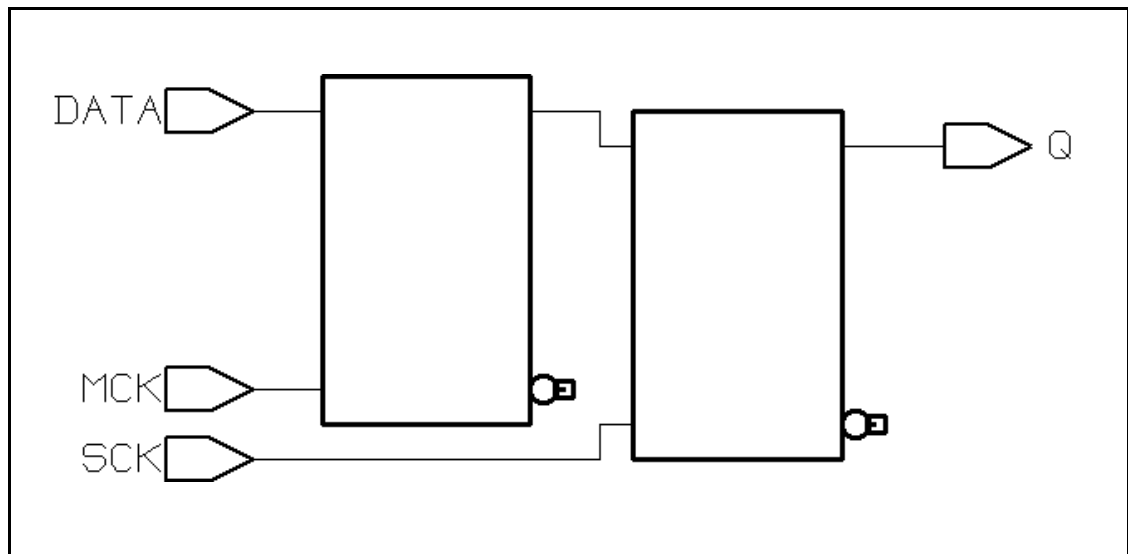
Inferring Master-Slave Latches

You can infer two-phase systems by using D latches. Example 6-18 shows a simple two-phase system with clocks MCK and SCK. Figure 6-6 shows the inferred latches.

Example 6-18 Two-Phase Clocks

```
module latch_verilog (DATA, MCK, SCK, Q);  
  input DATA, MCK, SCK;  
  output Q;  
  reg Q;  
  
  reg TEMP;  
  
  always @(DATA or MCK)  
    if (MCK)  
      TEMP = DATA;  
  
  always @(TEMP or SCK)  
    if (SCK)  
      Q = TEMP;  
endmodule
```

Figure 6-6 Two-Phase Clocks



Inferring Flip-Flops

FPGA Compiler II / *FPGA Express* can infer D flip-flops, JK flip-flops, and toggle flip-flops. The following sections provide details about each of these flip-flop types.

Many FPGA devices have a dedicated set/reset hardware resource that should be used. For this reason, you should infer asynchronous set/reset signals for all flip-flops in the design. FPGA Compiler II / *FPGA Express* will then use the global set/reset lines.

Inferring D Flip-Flops

FPGA Compiler II / *FPGA Express* infers a D flip-flop whenever the sensitivity list of an always block includes an edge expression (a test for the rising or falling edge of a signal). Use the following syntax to describe a rising edge:

```
posedge SIGNAL
```

Use the following syntax to describe a falling edge:

```
negedge SIGNAL
```

When the sensitivity list of an always block contains an edge expression, FPGA Compiler II / *FPGA Express* creates flip-flops for all the variables that are assigned values in the block. Example 6-19 shows the most common use of an always block to infer a flip-flop.

Example 6-19 Using an always Block to Infer a Flip-Flop

```
always @(edge)
begin
.
end
```

Simple D Flip-Flop

When you infer a D flip-flop, make sure you can control the clock and data signals from the top-level design ports or through combinational logic. Controllable clock and data signals ensure that simulation can initialize the design. If you cannot control the clock and data signals, infer a D flip-flop with an asynchronous reset or set or with a synchronous reset or set.

When you are inferring a simple D flip-flop, the always block can contain only one edge expression.

Example 6-20 provides the Verilog template for a positive edge-triggered D flip-flop. FPGA Compiler II / *FPGA Express* generates the inference report shown in Example 6-21. Figure 6-7 shows the inferred flip-flop.

Example 6-20 Positive Edge-Triggered D Flip-Flop

```
module dff_pos (DATA, CLK, Q);
    input DATA, CLK;
    output Q;
    reg Q;

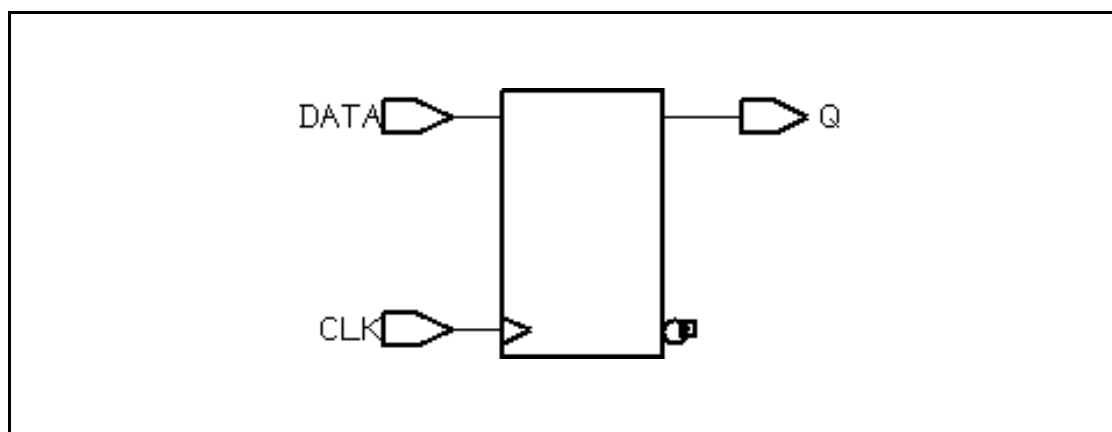
    always @(posedge CLK)
        Q = DATA;
endmodule
```

Example 6-21 Inference Report for a Positive Edge-Triggered D Flip-Flop

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	N	N	N

```
Q_reg
    set/reset/toggle: none
```


Figure 6-7 Positive Edge-Triggered D Flip-Flop



Example 6-22 provides the Verilog template for a negative edge-triggered D flip-flop. FPGA Compiler II / *FPGA Express* generates the inference report shown in Example 6-23. Figure 6-8 shows the inferred flip-flop.

Example 6-22 Negative Edge-Triggered D Flip-Flop

```

module dff_neg (DATA, CLK, Q);
  input DATA, CLK;
  output Q;
  reg Q;

  always @(negedge CLK)
    Q = DATA;
endmodule

```

Example 6-23 Inference Report for a Negative Edge-Triggered D Flip-Flop

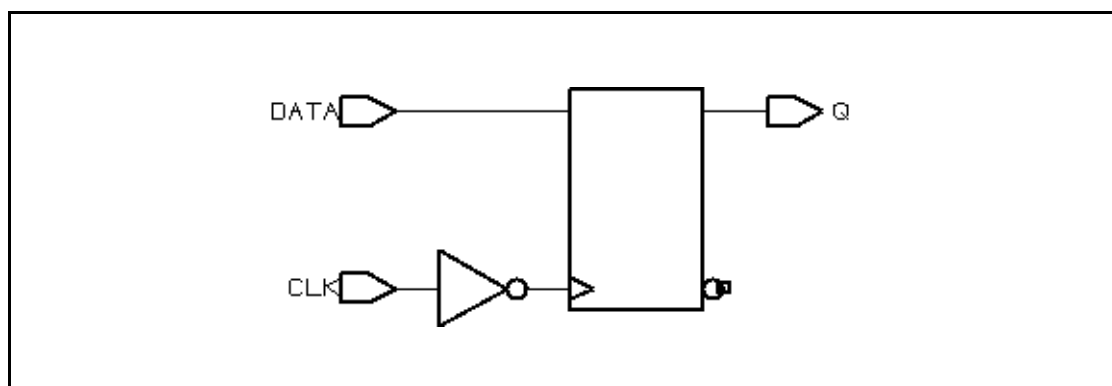
Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	N	N	N

```

Q_reg
  set/reset/toggle: none

```

Figure 6-8 Negative Edge-Triggered D Flip-Flop



D Flip-Flop With Asynchronous Set or Reset

When inferring a D flip-flop with an asynchronous set or reset, include edge expressions for the clock and the asynchronous signals in the sensitivity list of the always block. Specify the asynchronous conditions by using if statements. Specify the branches for the asynchronous conditions before the branches for the synchronous conditions. Example 6-24 provides the Verilog template for a D flip-flop with an asynchronous set. FPGA Compiler II / *FPGA Express* generates the inference report shown in Example 6-25. Figure 6-9 shows the inferred flip-flop.

Example 6-24 D Flip-Flop With Asynchronous Set

```
module dff_async_set (DATA, CLK, SET, Q);
  input DATA, CLK, SET;
  output Q;
  reg Q;

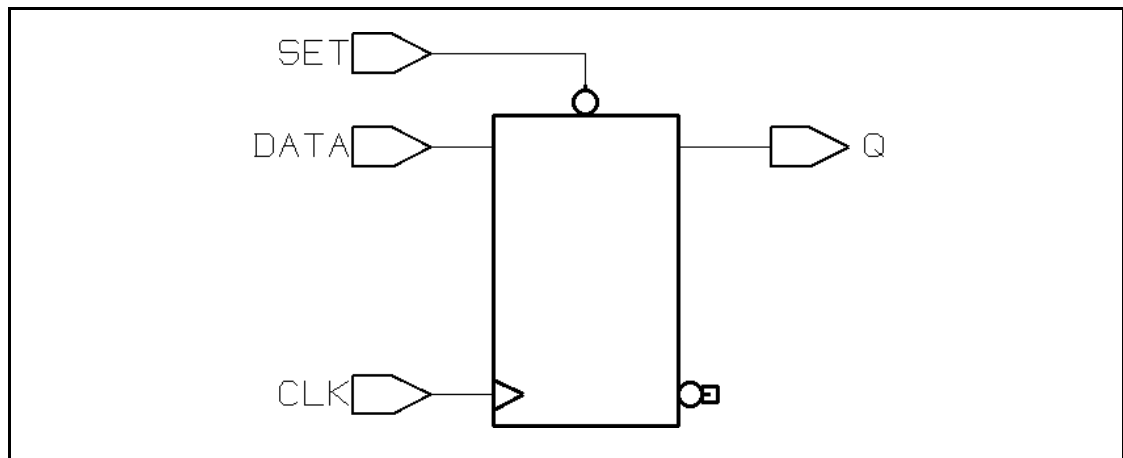
  always @(posedge CLK or negedge SET)
    if (~SET)
      Q = 1'b1;
    else
      Q = DATA;
endmodule
```

Example 6-25 Inference Report for a D Flip-Flop With Asynchronous Set

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	Y	N	N	N

Q_reg
 Async-set: SET'

Figure 6-9 D Flip-Flop With Asynchronous Set



Example 6-26 provides the Verilog template for a D flip-flop with an asynchronous reset. FPGA Compiler II / FPGA *Express* generates the inference report shown in Example 6-27. Figure 6-10 shows the inferred flip-flop.

Example 6-26 D Flip-Flop With Asynchronous Reset

```
module dff_async_reset (DATA, CLK, RESET, Q);
  input DATA, CLK, RESET;
  output Q;
  reg Q;

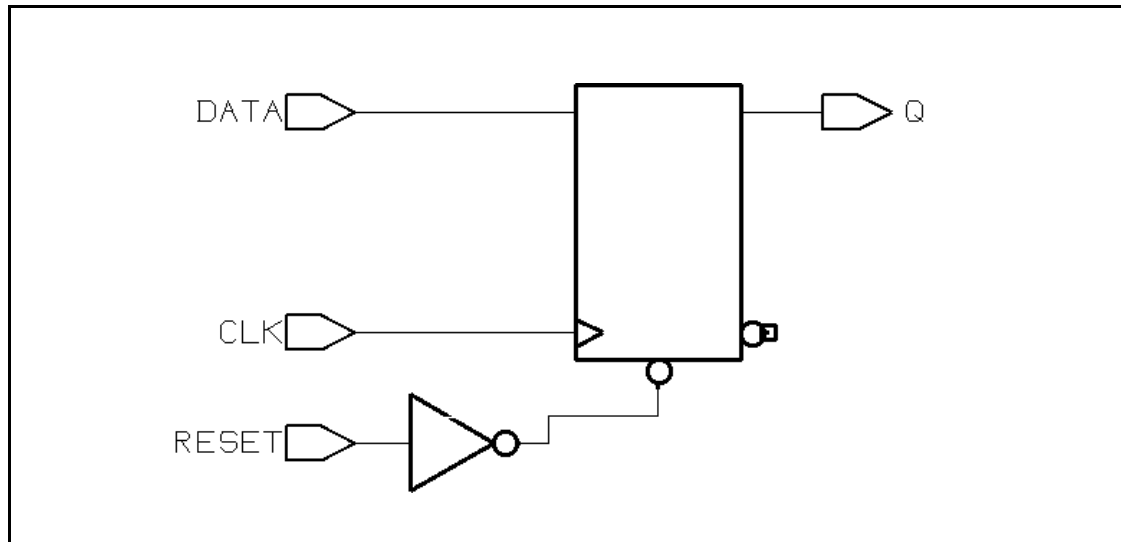
  always @(posedge CLK or posedge RESET)
    if (RESET)
      Q = 1'b0;
    else
      Q = DATA;
endmodule
```

Example 6-27 Inference Report for a D Flip-Flop With Asynchronous Reset

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	Y	N	N	N	N

Q_reg
 Async-reset: RESET

Figure 6-10 D Flip-Flop With Asynchronous Reset



D Flip-Flop With Asynchronous Set and Reset

Example 6-28 provides the Verilog template for a D flip-flop with active high asynchronous set and reset pins. The template uses the `one_hot` directive to prevent priority encoding of the set and reset signals. For this template, if you do not specify the `one_hot` directive, the reset signal has priority, because it is used as the condition for the if clause. FPGA Compiler II / *FPGA Express* generates the inference report shown in Example 6-29. Figure 6-11 shows the inferred flip-flop.

Note:

Most FPGA architectures donot have a register with an asynchronous set and asynchronous reset cell available. For this reason, you should avoid this construct.

Example 6-28 D Flip-Flop With Asynchronous Set and Reset

```
module dff_async (RESET, SET, DATA, Q, CLK);
    input CLK;
    input RESET, SET, DATA;
    output Q;
    reg Q;

    // synopsys one_hot "RESET, SET"
    always @(posedge CLK or posedge RESET or
           posedge SET)
        if (RESET)
            Q= 1'b0;
        else if (SET)
            Q= 1'b1;
        else Q= DATA;

    // synopsys translate_off
    always @ (RESET or SET)
        if (RESET + SET > 1)
            $write ("ONE-HOT violation for RESET and SET.");
    // synopsys translate_on
endmodule
```

Example 6-29 Inference Report for a D Flip-Flop With Asynchronous Set and Reset

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	Y	Y	N	N	N

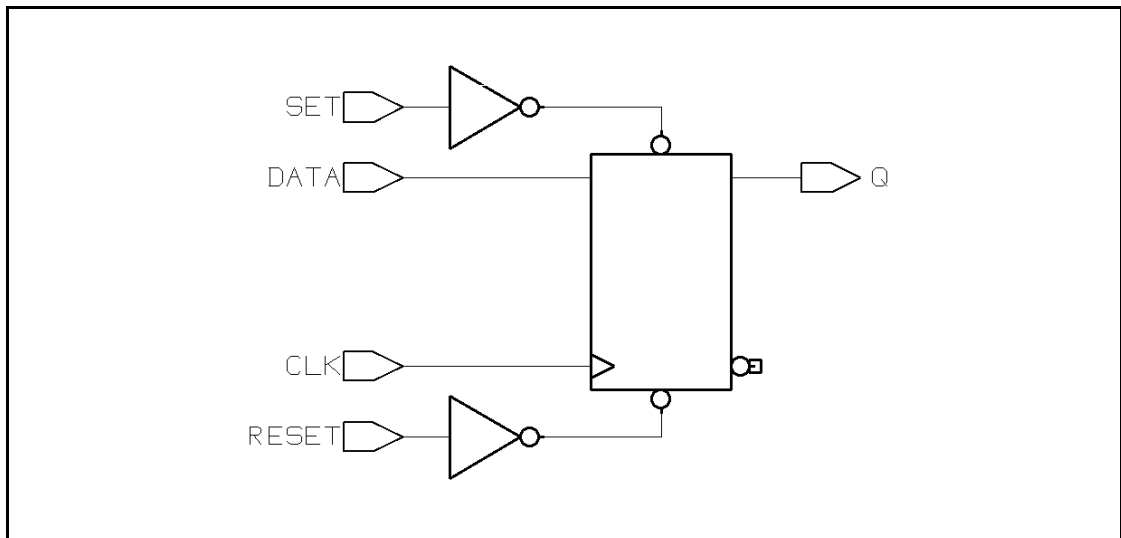
Q_reg

 Async-reset: RESET

 Async-set: SET

 Async-set and Async-reset ==> Q: X

Figure 6-11 D Flip-Flop With Asynchronous Set and Reset



D Flip-Flop With Synchronous Set or Reset

The previous examples illustrate how to infer a D flip-flop with asynchronous controls—one way to initialize or control the state of a sequential device. You can also synchronously reset or set a flip-flop (see Example 6-30 and Example 6-32). The `sync_set_reset` directive directs FPGA Compiler II / *FPGA Express* to the synchronous controls of the sequential device.

When the target technology library does not have a D flip-flop with synchronous reset, FPGA Compiler II / *FPGA Express* infers a D flip-flop with synchronous reset logic as the input to the D pin of the flip-flop. If the reset (or set) logic is not directly in front of the D pin of the flip-flop, initialization problems can occur during gate-level simulation of the design.

Example 6-30 provides the Verilog template for a D flip-flop with synchronous set. FPGA Compiler II / *FPGA Express* generates the inference report shown in Example 6-31. Figure 6-12 shows the inferred flip-flop.

Example 6-30 D Flip-Flop With Synchronous Set

```
module dff_sync_set (DATA, CLK, SET, Q);
    input DATA, CLK, SET;
    output Q;
    reg Q;

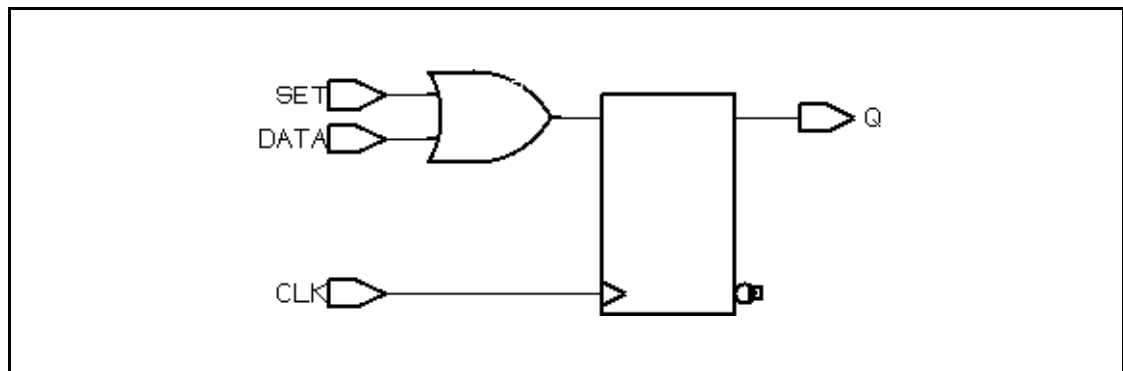
    //synopsys sync_set_reset "SET"
    always @(posedge CLK)
        if (SET)
            Q = 1'b1;
        else
            Q = DATA;
endmodule
```

Example 6-31 Inference Report for a D Flip-Flop With Synchronous Set

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	N	Y	N

Q_reg
Sync-set: SET

Figure 6-12 D Flip-Flop With Synchronous Set



Example 6-32 provides the Verilog template for a D flip-flop with synchronous reset. FPGA Compiler II / *FPGA Express* generates the inference report shown in Example 6-33. Figure 6-13 shows the inferred flip-flop.

Example 6-32 D Flip-Flop With Synchronous Reset

```

module dff_sync_reset (DATA, CLK, RESET, Q);
  input DATA, CLK, RESET;
  output Q;
  reg Q;

  //synopsys sync_set_reset "RESET"
  always @(posedge CLK)
    if (~RESET)
      Q = 1'b0;
    else
      Q = DATA;
endmodule

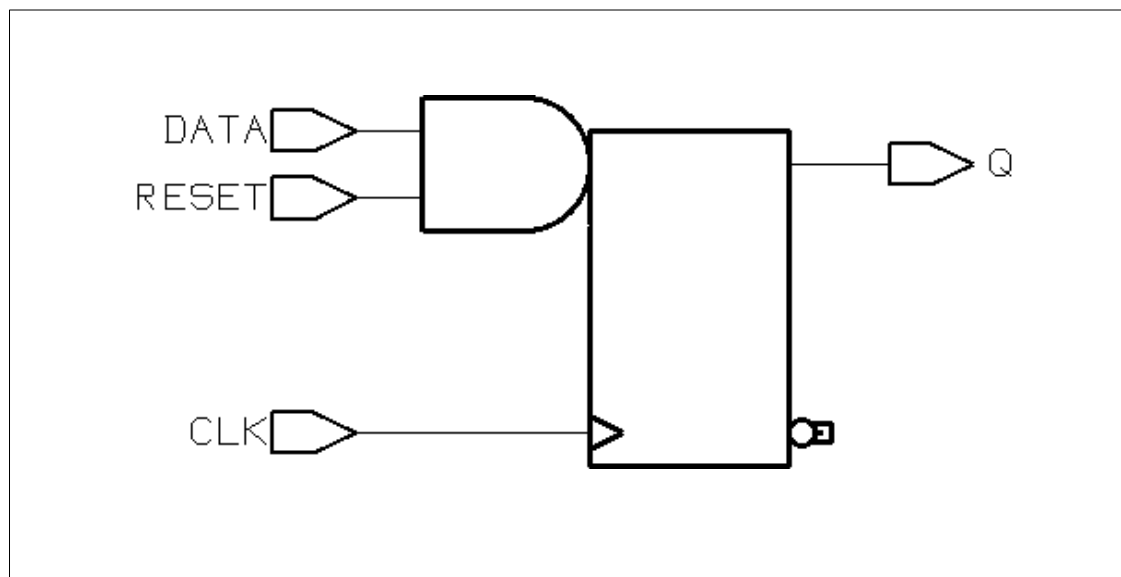
```

Example 6-33 Inference Report for a D Flip-Flop With Synchronous Reset

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	Y	N	N

Q_reg
Sync-reset: RESET'

Figure 6-13 D Flip-Flop With Synchronous Reset



D Flip-Flop With Synchronous and Asynchronous Load

D flip-flops can have asynchronous or synchronous controls. To infer a component with synchronous as well as asynchronous controls, you must check the asynchronous conditions before you check the synchronous conditions.

Example 6-34 provides the Verilog template for a D flip-flop with a synchronous load (called SLOAD) and an asynchronous load (called ALOAD). FPGA Compiler II / FPGA *Express* generates the inference report shown in Example 6-35. Figure 6-14 shows the inferred flip-flop.

Example 6-34 D Flip-Flop With Synchronous and Asynchronous Load

```
module dff_a_s_load (ALOAD, SLOAD, ADATA, SDATA, CLK, Q);
  input ALOAD, ADATA, SLOAD, SDATA, CLK;
  output Q;
  reg Q;

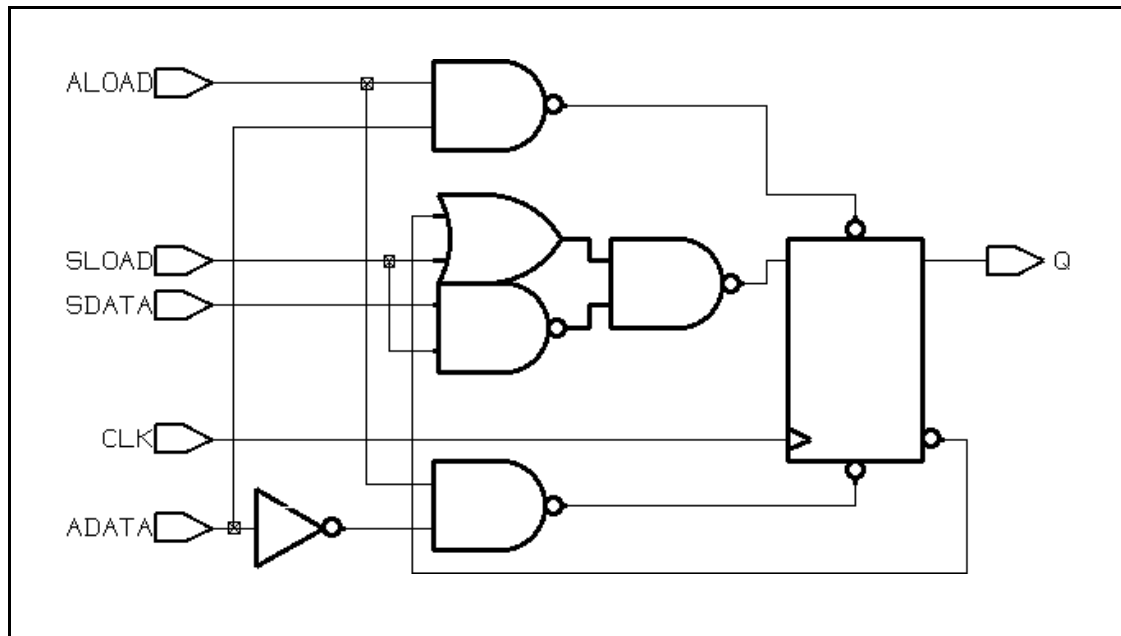
  always @ (posedge CLK or posedge ALOAD)
    if (ALOAD)
      Q= ADATA;
    else if (SLOAD)
      Q = SDATA;
endmodule
```

Example 6-35 Inference Report for a D Flip-Flop With Synchronous and Asynchronous Load

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	N	N	N

Q_reg
set/reset/toggle: none

Figure 6-14 D Flip-Flop With Synchronous and Asynchronous Load



Multiple Flip-Flops With Asynchronous and Synchronous Controls

If a signal is synchronous in one block but asynchronous in another block, use the `sync_set_reset_local` and `async_set_reset_local` directives to direct FPGA Compiler II / *FPGA Express* to the correct implementation.

In Example 6-36, block `infer_sync` uses the reset signal as a synchronous reset and block `infer_async` uses the reset signal as an asynchronous reset. FPGA Compiler II / *FPGA Express* generates the inference reports shown in Example 6-37. Figure 6-15 shows the resulting design.

Example 6-36 Multiple Flip-Flops With Asynchronous and Synchronous Controls

```

module multi_attr (DATA1, DATA2, CLK, RESET, SLOAD,
                  Q1, Q2);
  input DATA1, DATA2, CLK, RESET, SLOAD;
  output Q1, Q2;
  reg Q1, Q2;

  //synopsys sync_set_reset_local infer_sync "RESET"
  always @(posedge CLK)
  begin : infer_sync
    if (~RESET)
      Q1 = 1'b0;
    else if (SLOAD)
      Q1 = DATA1;
  end

  //synopsys async_set_reset_local infer_async "RESET"
  always @(posedge CLK or negedge RESET)
  begin: infer_async
    if (~RESET)
      Q2 = 1'b0;
    else if (SLOAD)
      Q2 = DATA2;
  end
endmodule

```

Example 6-37 Inference Reports for Multiple Flip-Flops With Asynchronous and Synchronous Controls

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q1_reg	Flip-flop	1	-	-	N	N	Y	N	N

Q1_reg

Sync-reset: RESET'

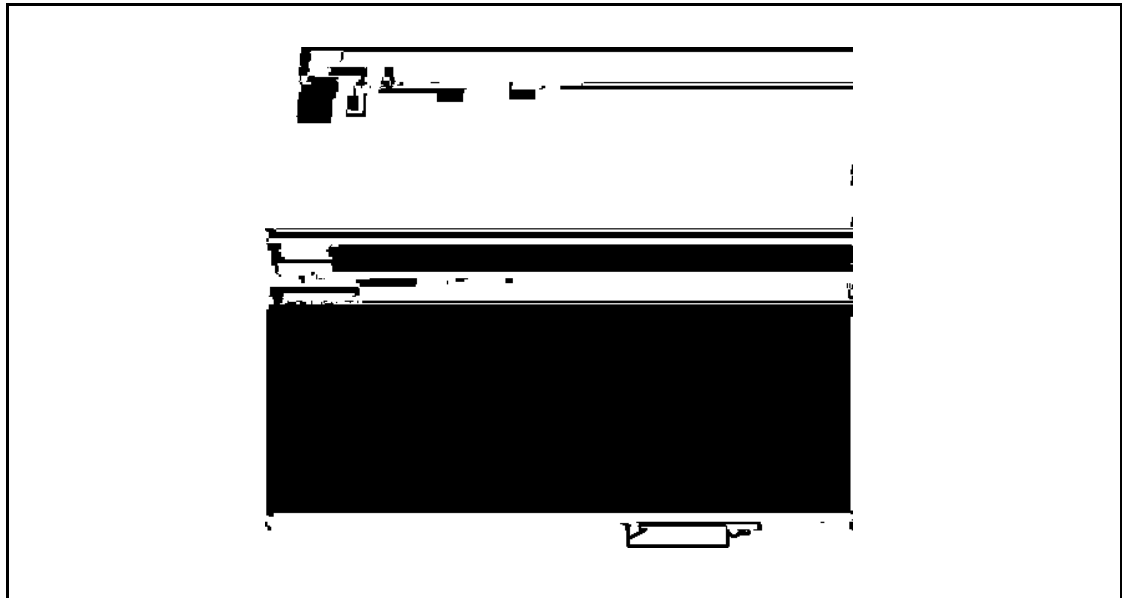
Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q2_reg	Flip-flop	1	-	-	Y	N	N	N	N

Q2_reg

Async-reset: RESET'

Register and Three-State Inference

Figure 6-15 Multiple Flip-Flops With Asynchronous and Synchronous Controls



Understanding the Limitations of D Flip-Flop Inference

If you use an if statement to infer D flip-flops, your design must meet the following requirements:

- The signal in an edge expression cannot be an indexed expression.

The following always block is invalid, because it uses an indexed expression:

```
always @(posedge clk[1])
```

FPGA Compiler II / FPGA *Express* generates the following message when you use an indexed expression in the always block:

```
Error: In an event expression with 'posedge' and 'negedge' qualifiers, only simple identifiers are allowed %s.
```

(VE-91)

- Set and reset conditions must be single-bit variables.

The following reset condition is invalid, because it uses a bused variable:

```
always @(posedge clk and negedge reset_bus)
  if (!reset_bus[1])
    .
end
```

FPGA Compiler II / FPGA *Express* generates the following message when you use a bused variable in a set or reset condition:

```
Error: The expression for the reset condition of the 'if'
statement in this 'always' block can only be a simple
identifier or its negation (%s). (VE-92)
```

- Set and reset conditions cannot use complex expressions.

The following reset condition is invalid, because it uses a complex expression:

```
always @(posedge clk and negedge reset)
  if (reset == (1-1))
    .
end
```

FPGA Compiler II / FPGA *Express* generates the VE-92 message when you use a complex expression in a set or reset condition.

- An if statement must occur at the top level of the always block.

The following example is invalid, because the if statement does not occur at the top level:

```
always @(posedge clk or posedge reset) begin
    #1;
    if (reset)
        .
end
```

FPGA Compiler II / FPGA *Express* generates the following message when the if statement does not occur at the top level:

```
Error: The statements in this 'always' block are outside
the scope of the synthesis policy (%s). Only an 'if'
statement is allowed at the top level in this 'always'
block. Please refer to the HDL Compiler reference manual
for ways to infer flip-flops and latches from 'always'
blocks. (VE-93)
```

Inferring JK Flip-Flops

When you infer a JK flip-flop, make sure you can control the J, K, and clock signals from the top-level design ports to ensure that simulation can initialize the design. The following sections provide code examples, inference reports, and figures for these types of JK flip-flops:

- JK flip-flop
- JK flip-flop with asynchronous set and reset

JK Flip-Flop

Example 6-38 provides the Verilog code that implements the JK flip-flop described in Table 6-2.

In the JK flip-flop, the J and K signals act as active-high synchronous set and reset. Use the `sync_set_reset` directive to indicate that the J and K signals are the synchronous set and reset for the design.

Example 6-39 shows the inference report generated by FPGA Compiler II / *FPGA Express*. Figure 6-16 shows the inferred flip-flop.

Table 6-2 Truth Table for JK Flip-Flop

J	K	CLK	Q_{n+1}
0	0	Rising	Q_n
0	1	Rising	0
1	0	Rising	1
1	1	Rising	Q_nB
X	X	Falling	Q_n

Example 6-38 JK Flip-Flop

```

module JK(J, K, CLK, Q);
  input J, K;
  input CLK;
  output Q;
  reg Q;

  // synopsys sync_set_reset "J, K"
  always @ (posedge CLK)
    case ({J, K})
      2'b01 : Q = 0;
      2'b10 : Q = 1;
      2'b11 : Q = ~Q;
    endcase
endmodule

```

Example 6-39 Inference Report for JK Flip-Flop

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	Y	Y	Y

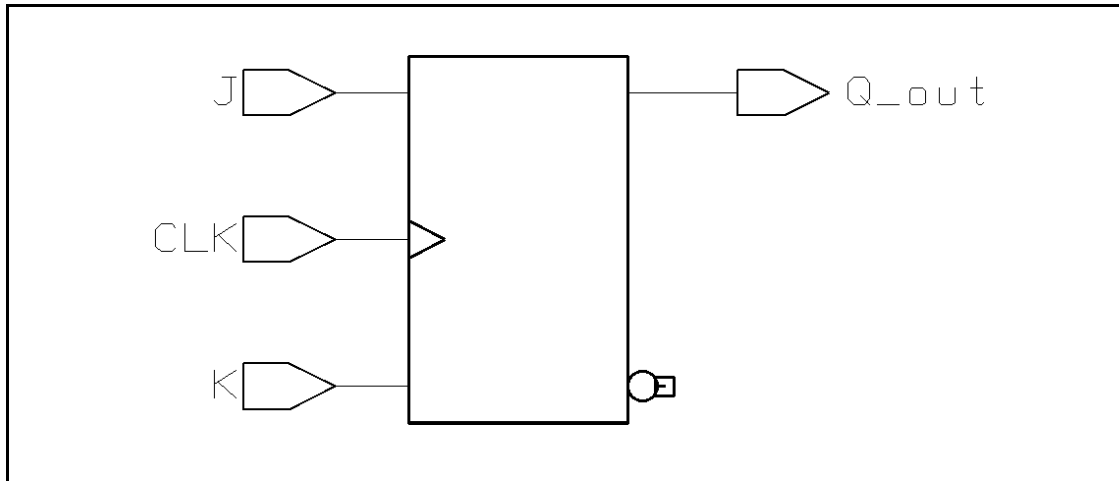
```

Q_reg
  Sync-reset: J' K
  Sync-set: J K'
  Sync-toggle: J K
  Sync-set and Sync-reset ==> Q: X

```

Register and Three-State Inference

Figure 6-16 JK Flip-Flop



JK Flip-Flop With Asynchronous Set and Reset

Example 6-40 provides the Verilog template for a JK flip-flop with asynchronous set and reset. Use the `sync_set_reset` directive to indicate the JK function. Use the `one_hot` directive to prevent priority encoding of the J and K signals.

FPGA Compiler II / *FPGA Express* generates the inference report shown in Example 6-41. Figure 6-17 shows the inferred flip-flop.

Example 6-40 JK Flip-Flop With Asynchronous Set and Reset

```
module jk_async_sr (RESET, SET, J, K, CLK, Q);
    input RESET, SET, J, K, CLK;
    output Q;
    reg Q;

    // synopsys sync_set_reset "J, K"
    // synopsys one_hot "RESET, SET"
    always @ (posedge CLK or posedge RESET or
              posedge SET)
        if (RESET)
            Q=1'b0;
        else if (SET)
            Q=1'b1;
        else
            case ({J, K})
                2'b01 : Q = 0;
                2'b10 : Q = 1;
                2'b11 : Q = ~Q;
            endcase

    //synopsys translate_off
    always @(RESET or SET)
        if (RESET + SET > 1)
            $write ("ONE-HOT violation for RESET and SET.");
    // synopsys translate_on
endmodule
```

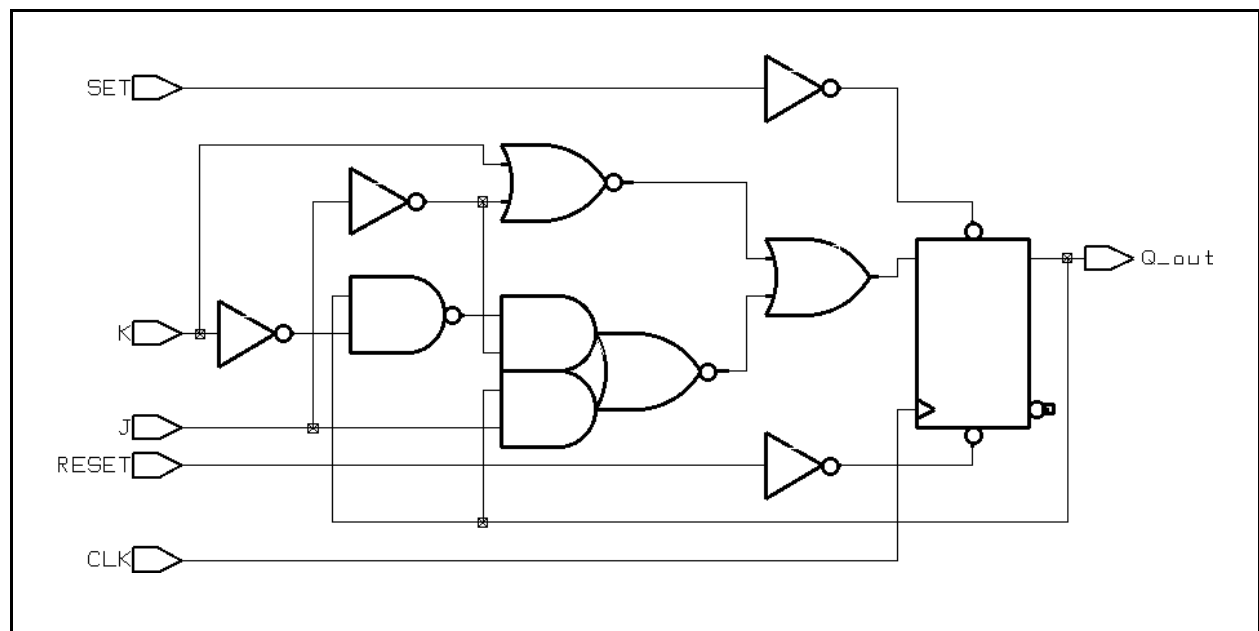
Example 6-41 Inference Report for JK Flip-Flop With Asynchronous Set and Reset

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	Y	Y	Y	Y	Y

Q_reg

- Async-reset: RESET
- Async-set: SET
- Sync-reset: J' K
- Sync-set: J K'
- Sync-toggle: J K
- Async-set and Async-reset ==> Q: X
- Sync-set and Sync-reset ==> Q: X

Figure 6-17 JK Flip-Flop With Asynchronous Set and Reset



Inferring Toggle Flip-Flops

To infer toggle flip-flops, follow the coding style in the following examples.

You must include asynchronous controls in the toggle flip-flop description. Without them, you cannot initialize toggle flip-flops to a known state.

This section describes toggle flip-flops with an asynchronous set or reset and toggle flip-flops with an enable and an asynchronous reset.

Toggle Flip-Flop With Asynchronous Set or Reset

Example 6-42 shows the template for a toggle flip-flop with asynchronous set. FPGA Compiler II / FPGA *Express* generates the inference report shown in Example 6-43. Figure 6-18 shows the flip-flop.

Example 6-42 Toggle Flip-Flop With Asynchronous Set

```

module t_async_set (SET, CLK, Q);
  input SET, CLK;
  output Q;
  reg Q;

  always @ (posedge CLK or posedge SET)
    if (SET)
      Q = 1;
    else
      Q = ~Q;
endmodule

```

Example 6-43 Inference Report for a Toggle Flip-Flop With Asynchronous Set

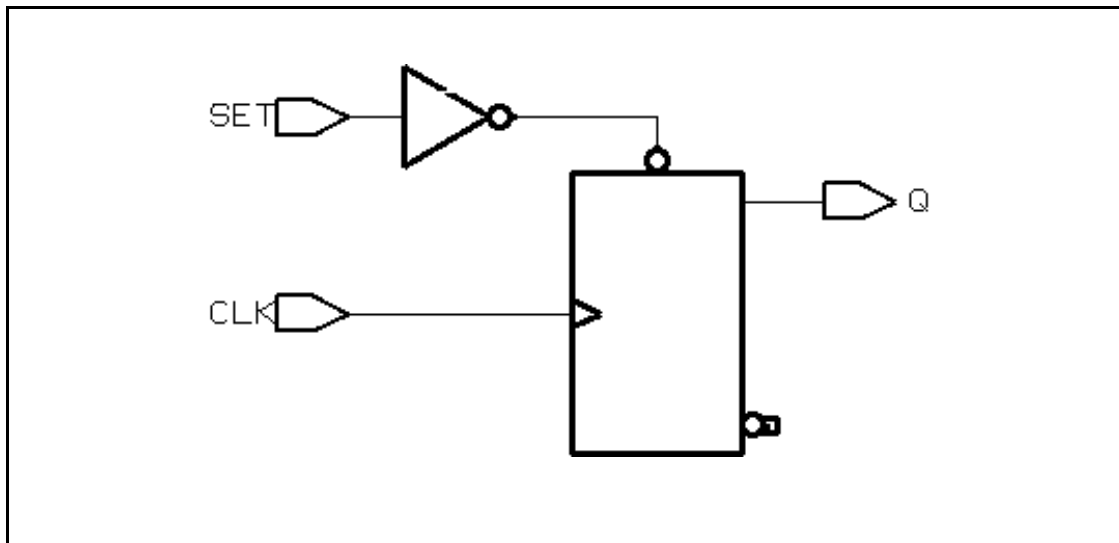
Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
TMP_Q_reg	Flip-flop	1	-	-	N	Y	N	N	Y

```

TMP_Q_reg
  Async-set: SET
  Sync-toggle: true

```

Figure 6-18 Toggle Flip-Flop With Asynchronous Set



Example 6-44 provides the Verilog template for a toggle flip-flop with asynchronous reset. Example 6-45 shows the inference report. Figure 6-19 shows the inferred flip-flop.

Example 6-44 Toggle Flip-Flop With Asynchronous Reset

```
module t_async_reset (RESET, CLK, Q);
    input RESET, CLK;
    output Q;
    reg Q;

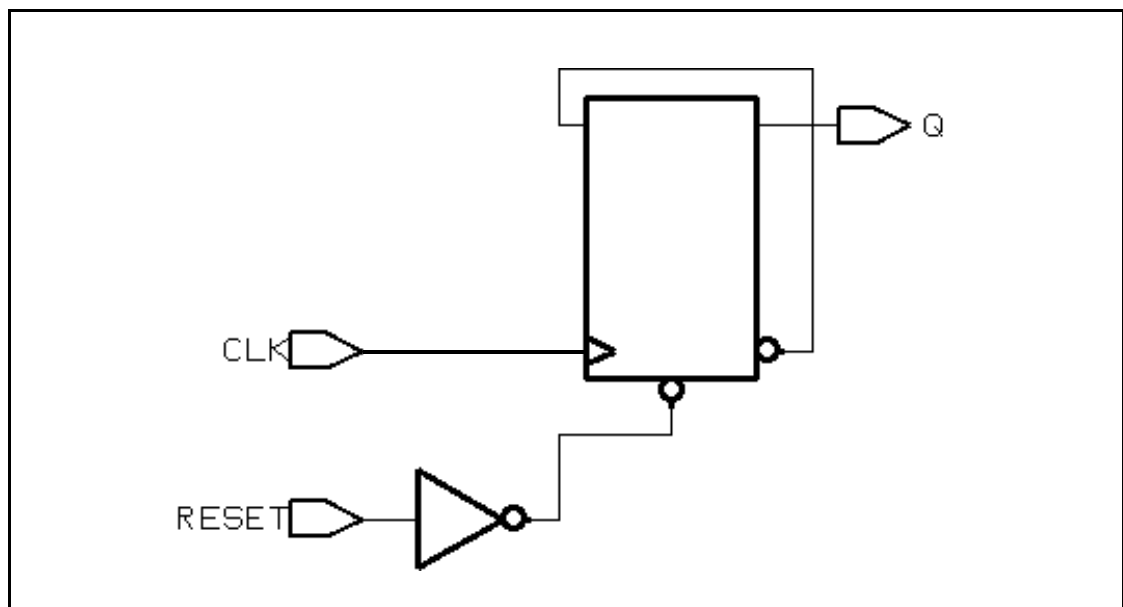
    always @ (posedge CLK or posedge RESET)
        if (RESET)
            Q = 0;
        else
            Q = ~Q;
endmodule
```

Example 6-45 Inference Report: Toggle Flip-Flop With Asynchronous Reset

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
TMP_Q_reg	Flip-flop	1	-	-	Y	N	N	N	Y

TMP_Q_reg
 Async-reset: RESET
 Sync-toggle: true

Figure 6-19 Toggle Flip-Flop With Asynchronous Reset



Toggle Flip-Flops With Enable and Asynchronous Reset

Example 6-46 provides the Verilog template for a toggle flip-flop with an enable and an asynchronous reset. The flip-flop toggles only when the enable (TOGGLE signal) has a logic 1 value. FPGA Compiler II / FPGA Express generates the inference report shown in Example 6-47. Figure 6-20 shows the inferred flip-flop.

Example 6-46 Toggle Flip-Flop With Enable and Asynchronous Reset

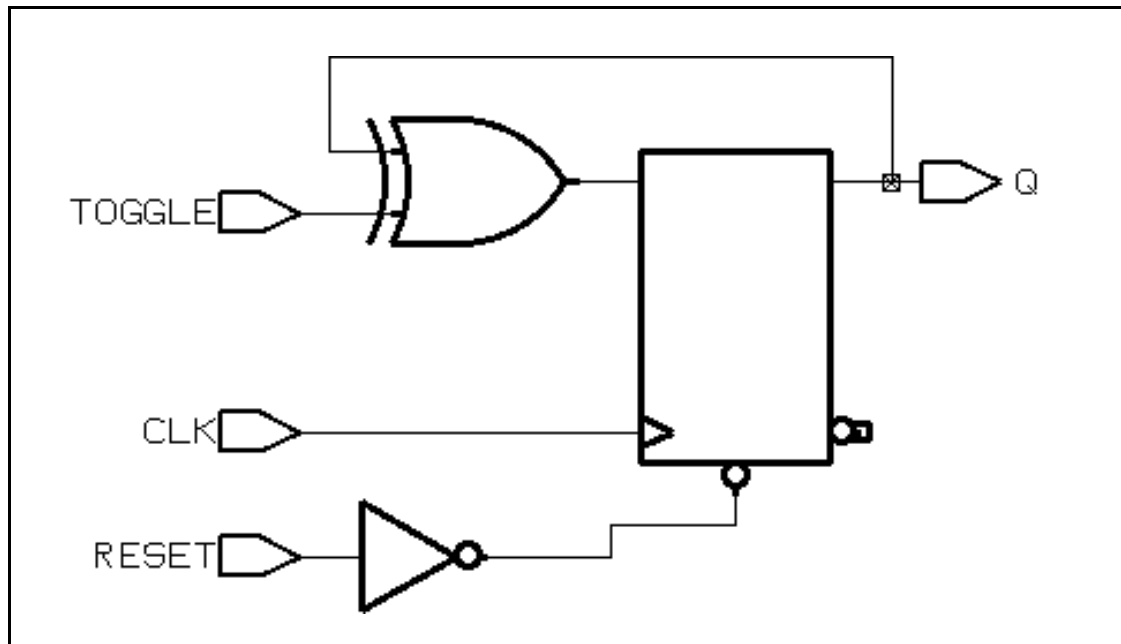
```
module t_async_en_r (RESET, TOGGLE, CLK, Q);
  input RESET, TOGGLE, CLK;
  output Q;
  reg Q;
  always @ (posedge CLK or posedge RESET)
  begin : infer
    if (RESET)
      Q = 0;
    else if (TOGGLE)
      Q = ~Q;
  end
end
endmodule
```

Example 6-47 Inference Report: Toggle Flip-Flop With Enable and Asynchronous Reset

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
TMP_Q_reg	Flip-flop	1	-	-	Y	N	N	N	Y

TMP_Q_reg
 Async-reset: RESET
 Sync-toggle: TOGGLE

Figure 6-20 Toggle Flip-Flop With Enable and Asynchronous Reset



Getting the Best Results

This section provides tips for improving the results you achieve during flip-flop inference. Topics include

- Minimizing flip-flop count
- Correlating synthesis results with simulation results

Minimizing Flip-Flop Count

An always block that contains a clock edge in the sensitivity list causes FPGA Compiler II / FPGA Express to infer a flip-flop for each variable assigned a value in that block. It might not be necessary to register all variables in the block. Make sure your HDL description builds only as many flip-flops as the design requires.

The description in Example 6-48 builds six flip-flops, one for each variable assigned a value in the block (COUNT(2:0), AND_BITS, OR_BITS, and XOR_BITS).

Example 6-48 Circuit With Six Implied Registers

```
module count (CLK, RESET, AND_BITS, OR_BITS, XOR_BITS);
  input CLK, RESET;
  output AND_BITS, OR_BITS, XOR_BITS;
  reg AND_BITS, OR_BITS, XOR_BITS;

  reg [2:0] COUNT;

  always @(posedge CLK) begin
    if (RESET)
      COUNT = 0;
    else
      COUNT = COUNT + 1;

    AND_BITS = & COUNT;
    OR_BITS = | COUNT;
    XOR_BITS = ^ COUNT;
  end
endmodule
```

In this design, the outputs—AND_BITS, OR_BITS, and XOR_BITS—depend solely on the value of the variable COUNT. If the variable COUNT is registered, these three outputs do not need to be registered.

To compute values synchronously and store them in flip-flops, set up an always block with a signal edge trigger. To let other values change asynchronously, make a separate always block with no signal edge trigger. Put the assignments you want clocked in the always block with the signal edge trigger, and put the other assignments in the other always block. You use this technique for creating Mealy machines.

To avoid inferring extra registers, assign the outputs in an always block that does not have a clock edge in its condition expression. Example 6-49 shows a description with two always blocks, one with a clock edge condition and one without. Put the registered (synchronous) assignments into the block with the clock edge condition. Put the other (asynchronous) assignments in the other block. This description style lets you choose the variables that are registered and those that are not.

Example 6-49 Circuit With Three Implied Registers

```
module count (CLK, RESET,
             AND_BITS, OR_BITS, XOR_BITS);
    input CLK, RESET;
    output AND_BITS, OR_BITS, XOR_BITS;
    reg AND_BITS, OR_BITS, XOR_BITS;

    reg [2:0] COUNT;

    //synchronous block
    always @(posedge CLK) begin
        if (RESET)
            COUNT = 0;
        else
            COUNT = COUNT + 1;
    end
    //asynchronous block
    always @(COUNT) begin
        AND_BITS = & COUNT;
        OR_BITS = | COUNT;
        XOR_BITS = ^ COUNT;
    end
end
endmodule
```

The technique of separating combinational logic from registered or sequential logic is useful for describing state machines. See the following examples in Appendix A:

- “Count Zeros—Combinational Version” on page A-2
- “Count Zeros—Sequential Version” on page A-5
- “Drink Machine—State Machine Version” on page A-7
- “Drink Machine—Count Nickels Version” on page A-10
- “Carry-Lookahead Adder” on page A-12

Correlating With Simulation Results

Using delay specifications with registered values can cause the simulation to behave differently from the logic FPGA Compiler II / *FPGA Express* synthesizes. For example, the description in Example 6-50 contains delay information that causes FPGA Compiler II / *FPGA Express* to synthesize a circuit that behaves unexpectedly (the post-synthesis simulation results do not match the pre-synthesis simulation results).

Example 6-50 Delays in Registers

```

module flip_flop (D, CLK, Q);
    input D, CLK;
    output Q;
    .
endmodule

module top (A, C, D, CLK);
    .
    reg B;

    always @ (A or C or D or CLK)
    begin
        B <= #100 A;
        flip_flop F1(A, CLK, C);
        flip_flop F2(B, CLK, D);
    end
endmodule

```

In Example 6-50, B changes 100 nanoseconds after A changes. If the clock period is less than 100 nanoseconds, output D is one or more clock cycles behind output C during simulation of the design. However, because FPGA Compiler II / FPGA *Express* ignores the delay information, A and B change values at the same time, and so do C and D. This behavior is *not* the same as in the post-synthesis simulation.

When using delay information in your designs, make sure that the delays do not affect registered values. In general, you can safely include delay information in your description if it does not change the value that gets clocked into a flip-flop.

Understanding Limitations of Register Inference

FPGA Compiler II / FPGA *Express* cannot infer the following components. You must instantiate these components in your Verilog description.

- Flip-flops and latches with three-state outputs
- Flip-flops with bidirectional pins
- Flip-flops with multiple clock inputs
- Multiport latches
- Register banks

Note:

Although you can instantiate flip-flops with bidirectional pins, FPGA Compiler II / FPGA *Express* interprets these cells as black boxes.

Three-State Inference

FPGA Compiler II / *FPGA Express* infers a three-state driver when you assign the value of *z* to a variable. The *z* value represents the high-impedance state. FPGA Compiler II / *FPGA Express* infers one three-state driver per block. You can assign high-impedance values to single-bit or bused variables.

Reporting Three-State Inference

FPGA Compiler II / *FPGA Express* generates an inference report that shows information about the inferred devices. Example 6-51 shows a three-state inference report.

Example 6-51 Three-State Inference Report

Three-State Device Name	Type	MB
OUT1_tri	Three-State Buffer	N

The first column of the report indicates the name of the inferred three-state device. The second column indicates the type of three-state device FPGA Compiler II / *FPGA Express* inferred. The third column indicates whether the three-state device has multiple bits.

Controlling Three-State Inference

FPGA Compiler II / *FPGA Express* always infers a three-state driver when you assign the value of *z* to a variable. FPGA Compiler II / *FPGA Express* does not provide any means of controlling the inference.

Inferring Three-State Drivers

This section contains Verilog examples that infer the following types of three-state drivers:

- Simple three-state drivers
- Registered three-state drivers

Simple Three-State Driver

This section provides a template for a simple three-state driver. In addition, it provides examples of how allocating high-impedance assignments to different blocks affects three-state inference.

Example 6-52 provides the Verilog template for a simple three-state driver. FPGA Compiler II / FPGA *Express* generates the inference report shown in Example 6-53. Figure 6-21 shows the inferred three-state driver.

Example 6-52 Simple Three-State Driver

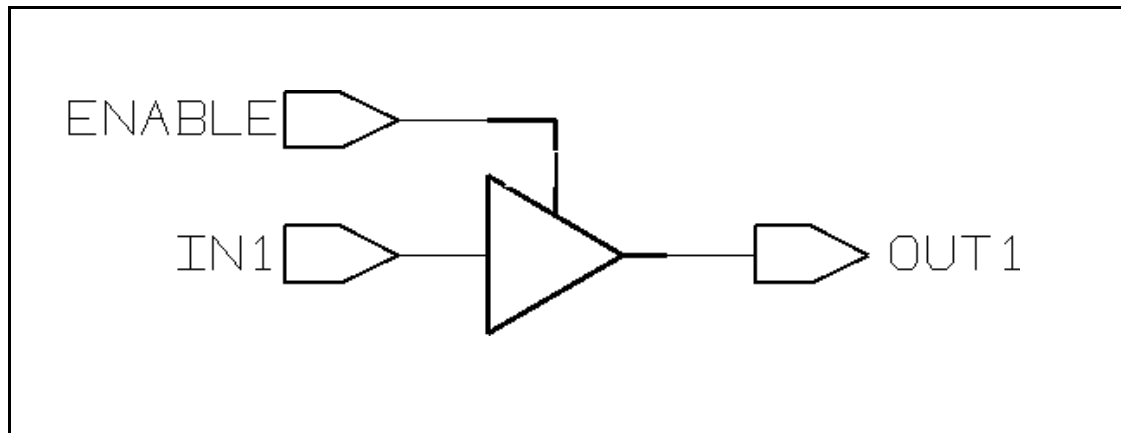
```
module three_state (ENABLE, IN1, OUT1);
    input IN1, ENABLE;
    output OUT1;
    reg OUT1;

    always @(ENABLE or IN1) begin
        if (ENABLE)
            OUT1 = IN1;
        else
            OUT1 = 1'bz; //assigns high-impedance state
        end
    end
endmodule
```

Example 6-53 Inference Report for Simple Three-State Driver

Three-State Device Name	Type	MB
OUT1_tri	Three-State Buffer	N

Figure 6-21 Schematic of Simple Three-State Driver



Example 6-54 provides an example of placing all high-impedance assignments in a single block. In this case, the data is gated and FPGA Compiler II / FPGA Express infers a single three-state driver. Example 6-55 shows the inference report. Figure 6-22 shows the schematic the code generates.

Example 6-54 Inferring One Three-State Driver From a Single Block

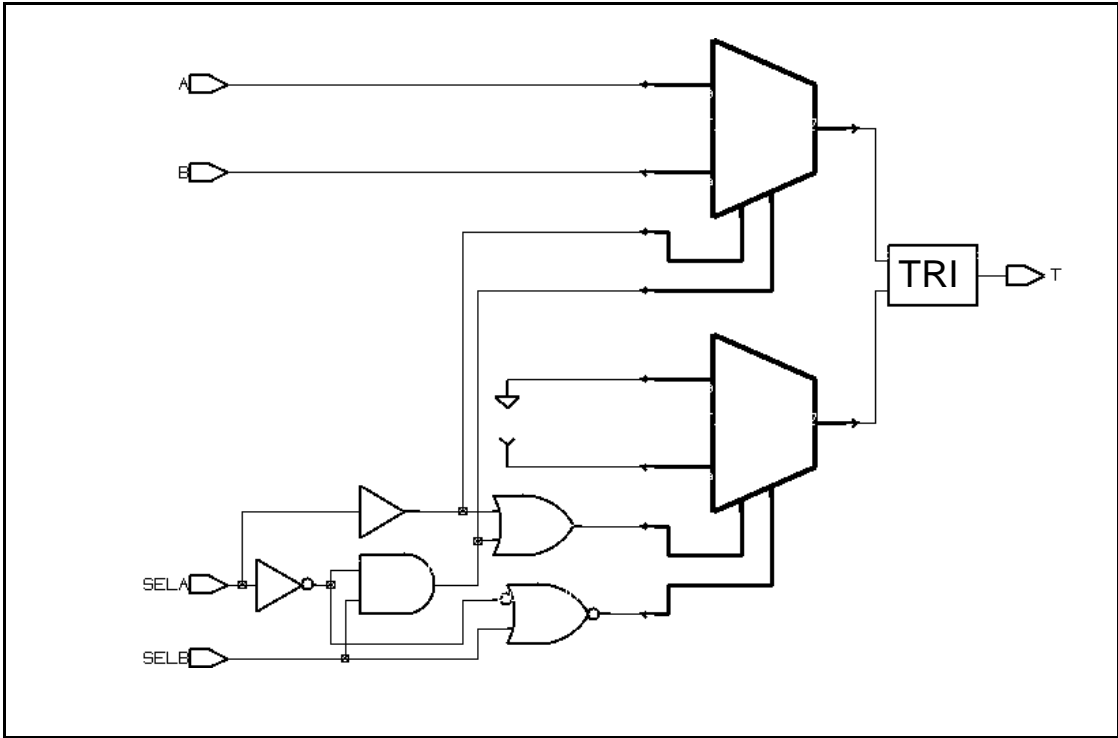
```
module three_state (A, B, SELA, SELB, T);
  input  A, B, SELA, SELB;
  output T;
  reg T;

  always @(SELA or SELB or A or B) begin
    T = 1'bz;
    if (SELA)
      T = A;
    if (SELB)
      T = B;
  end
end
endmodule
```

Example 6-55 Single Block Inference Report

Three-State Device Name	Type	MB
T_tri	Three-State Buffer	N

Figure 6-22 One Three-State Driver Inferred From a Single Block



Example 6-56 provides an example of placing each high-impedance assignment in a separate block. In this case, FPGA Compiler II / *FPGA Express* infers multiple three-state drivers. Example 6-57 shows the inference report. Figure 6-23 shows the schematic the code generates.

Example 6-56 *Inferring Three-State Drivers From Separate Blocks*

```
module three_state (A, B, SELA, SELB, T);
  input  A, B, SELA, SELB;
  output T;
  reg T;

  always @(SELA or A)
    if (SELA)
      T = A;
    else
      T = 1'bz;

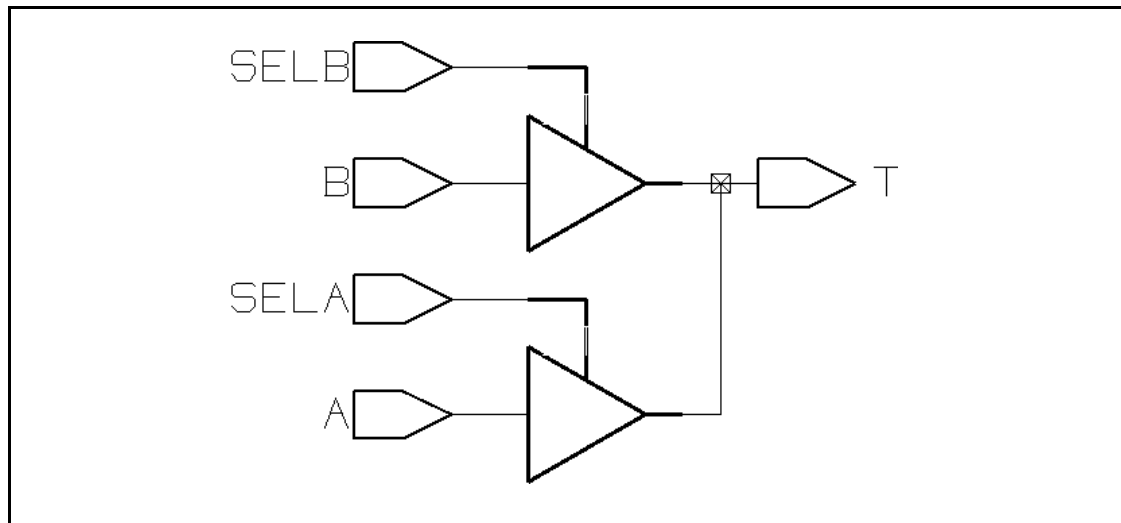
  always @(SELB or B)
    if (SELB)
      T = B;
    else
      T = 1'bz;
endmodule
```

Example 6-57 *Inference Report for Two Three-State Drivers*

Three-State Device Name	Type	MB
T_tri	Three-State Buffer	N

Three-State Device Name	Type	MB
T_tri2	Three-State Buffer	N

Figure 6-23 Two Three-State Drivers Inferred From Separate Blocks



Registered Three-State Drivers

When a variable is registered in the same block in which it is three-stated, FPGA Compiler II / *FPGA Express* also registers the enable pin of the three-state gate. Example 6-58 shows an example of this type of code. Example 6-59 shows the inference report. Figure 6-24 shows the schematic generated by the code.

Example 6-58 Three-State Driver With Registered Enable

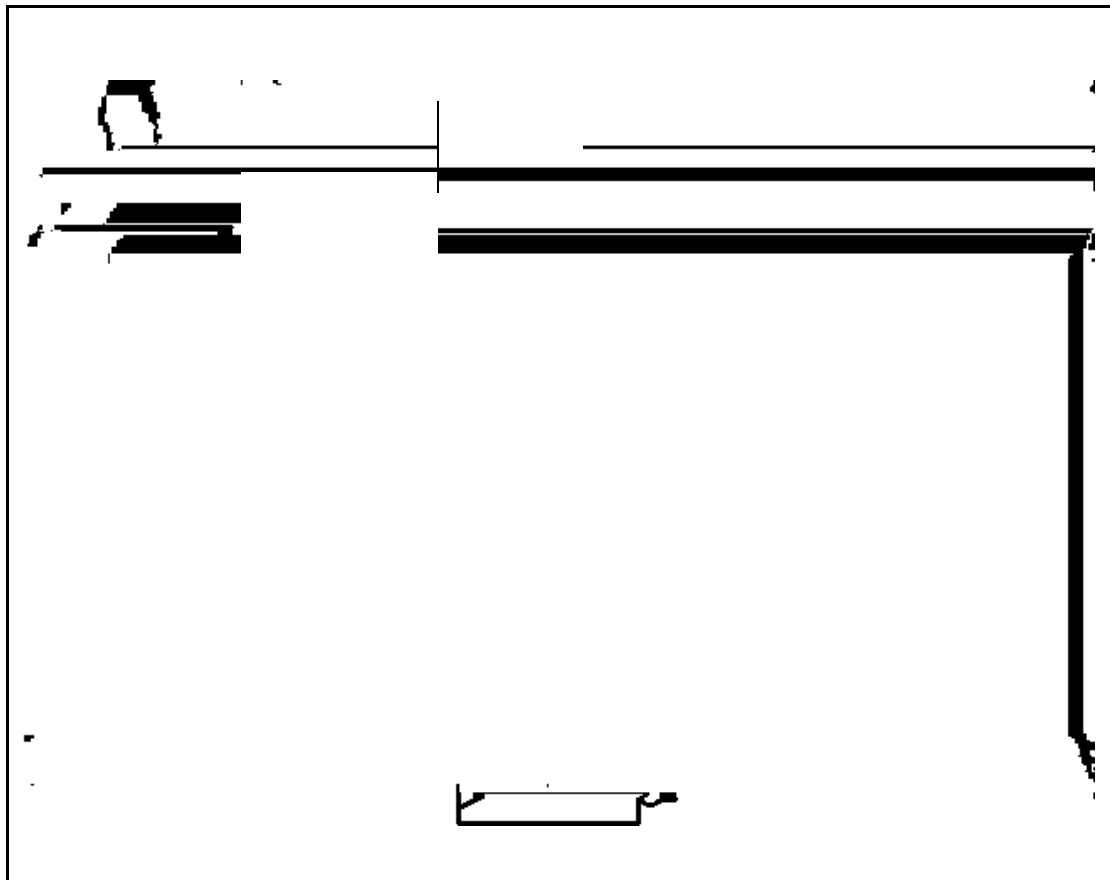
```
module ff_3state (DATA, CLK, THREE_STATE, OUT1);
  input DATA, CLK, THREE_STATE;
  output OUT1;
  reg OUT1;

  always @ (posedge CLK) begin
    if (THREE_STATE)
      OUT1 = 1'bz;
    else
      OUT1 = DATA;
  end
endmodule
```

Example 6-59 Inference Report for Three-State Driver With Registered Enable

Three-state Device Name	Type	MB
OUT1_tri	Three-State Buffer	N
OUT1_tr_enable_reg	Flip-flop (width 1)	N

Figure 6-24 Three-State Driver With Registered Enable



In Figure 6-24, the three-state gate has a register on its enable pin. Example 6-60 uses two blocks to instantiate a three-state gate, with a flip-flop only on the input. Example 6-61 shows the inference report. Figure 6-25 shows the schematic the code generates.

Example 6-60 Three-State Driver Without Registered Enable

```
module ff_3state (DATA, CLK, THREE_STATE, OUT1);
  input DATA, CLK, THREE_STATE;
  output OUT1;
  reg OUT1;

  reg TEMP;

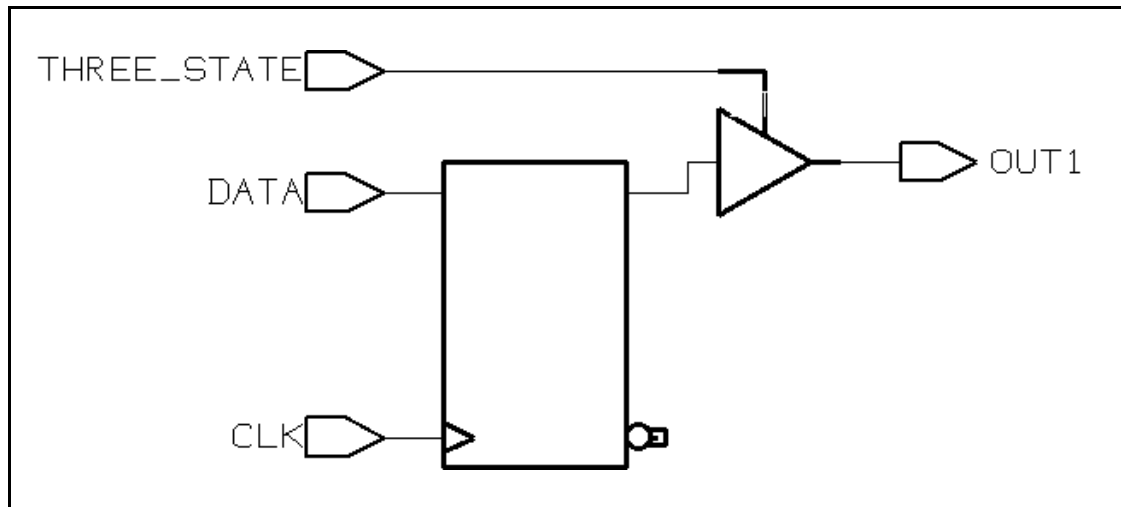
  always @(posedge CLK)
    TEMP = DATA;

  always @(THREE_STATE or TEMP)
    if (THREE_STATE)
      OUT1 = TEMP;
    else
      OUT1 = 1'bz;
endmodule
```

Example 6-61 Inference Report for Three-State Driver Without Registered Enable

Three-State Device Name	Type	MB
OUT1_tri	Three-State Buffer	N

Figure 6-25 Three-State Driver Without Registered Enable



Understanding the Limitations of Three-State Inference

You can use the z value in the following ways:

- Variable assignment
- Function call argument
- Return value

You cannot use the z value in an expression, except for comparison with z. Be careful when using expressions that compare with the z value. FPGA Compiler II / FPGA *Express* always evaluates these expressions to false, and the pre-synthesis and post-synthesis simulation results might differ. For this reason, FPGA Compiler II / FPGA *Express* issues a warning when it synthesizes such comparisons.

This is an example of incorrect use of the z value in an expression:

```
OUT_VAL = (1'bz && IN_VAL);
```

This is an example of correct use of the z value in an expression:

```
if (IN_VAL == 1'bz) then
```


7

Writing Circuit Descriptions

You can write many logically equivalent descriptions in Verilog to describe a circuit design. However, some descriptions are more efficient than others in terms of the synthesized circuit's area and speed. The way you write your Verilog source code can affect synthesis.

This chapter describes how to write a Verilog description to ensure an efficient implementation. Topics include

- How Statements Are Mapped to Logic
- Don't Care Inference
- Propagating Constants
- Synthesis Issues
- Designing for Overall Efficiency

Here are some general guidelines for writing efficient circuit descriptions:

- Restructure a design that makes repeated use of several large components, to minimize the number of instantiations.
- In a design that needs some, but not all, of its variables or signals stored during operation, minimize the number of latches or flip-flops required.
- Consider collapsing hierarchy for more-efficient synthesis.

How Statements Are Mapped to Logic

Verilog descriptions are mapped to logic by the creation of blocks of combinational circuits and storage elements. A statement or an operator in a Verilog function can represent a block of combinational logic or, in some cases, a latch or register.

The description fragment shown in Example 7-1 represents four logic blocks:

- A comparator that compares the value of b with 10
- An adder that has a and b as inputs
- An adder that has a and 10 as inputs
- A multiplexer (implied by the if statement) that controls the final value of y

Example 7-1 Four Logic Blocks

```
if (b < 10)
    y = a + b;
else
    y = a + 10;
```

The logic blocks created by FPGA Compiler II / *FPGA Express* are custom-built for their environment. That is, if a and b are 4-bit quantities, a 4-bit adder is built. If a and b are 9-bit quantities, a 9-bit adder is built. Because FPGA Compiler II / *FPGA Express* incorporates a large set of these customized logic blocks, it can translate most Verilog statements and operators.

Design Structure

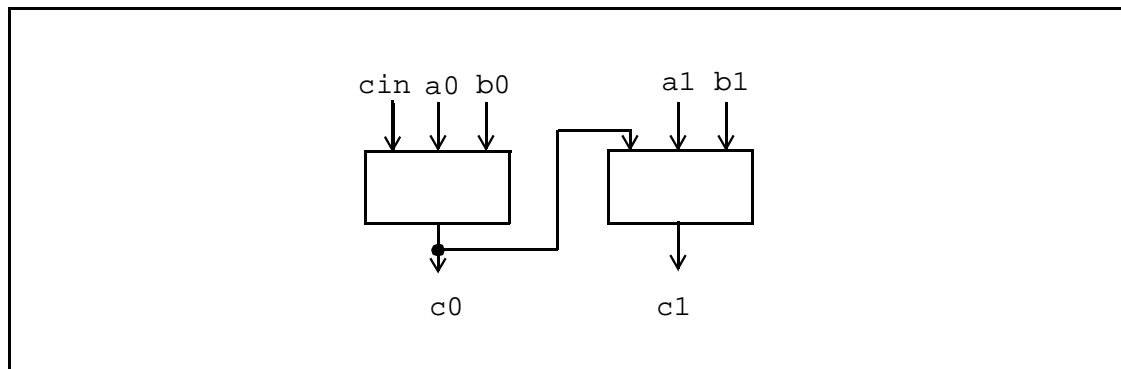
FPGA Compiler II / *FPGA Express* provides significant control over the preoptimization structure, or organization of components, in your design. Whether or not your design structure is preserved after optimization depends on the options you select. FPGA Compiler II / *FPGA Express* automatically chooses the best structure for your design. You can view the preoptimized structure in the schematic window and then correlate it back to the original HDL source code.

You control structure by the way you order assignment statements and the way you use variables. Each Verilog assignment statement implies a piece of logic. The following examples illustrate two possible descriptions of an adder's carry chain. Example 7-2 results in a ripple carry implementation, as in Figure 7-1. Example 7-3 has more structure (gates), because the HDL source includes temporary registers, and it results in a carry-lookahead implementation, as in Figure 7-2.

Example 7-2 Ripple Carry Chain

```
// a is the addend
// b is the augend
// c is the carry
// cin is the carry in
c0 = (a0 & b0) |
      (a0 | b0) & cin;
c1 = (a1 & b1) |
      (a1 | b1) & c0;
```

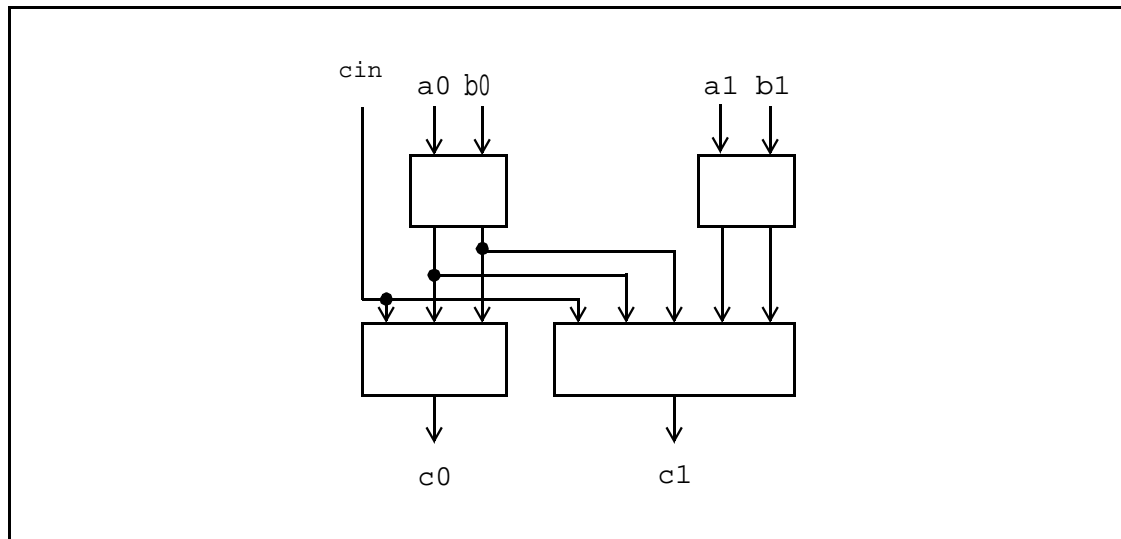
Figure 7-1 Ripple Carry Chain Implementation



Example 7-3 Carry-Lookahead Chain

```
// p's are propagate
// g's are generate
p0 = a0 | b0;
g0 = a0 & b0;
p1 = a1 | b1;
g1 = a1 & b1;
c0 = g0 | p0 & cin;
c1 = g1 | p1 & g0 |
      p1 & p0 & cin;
```

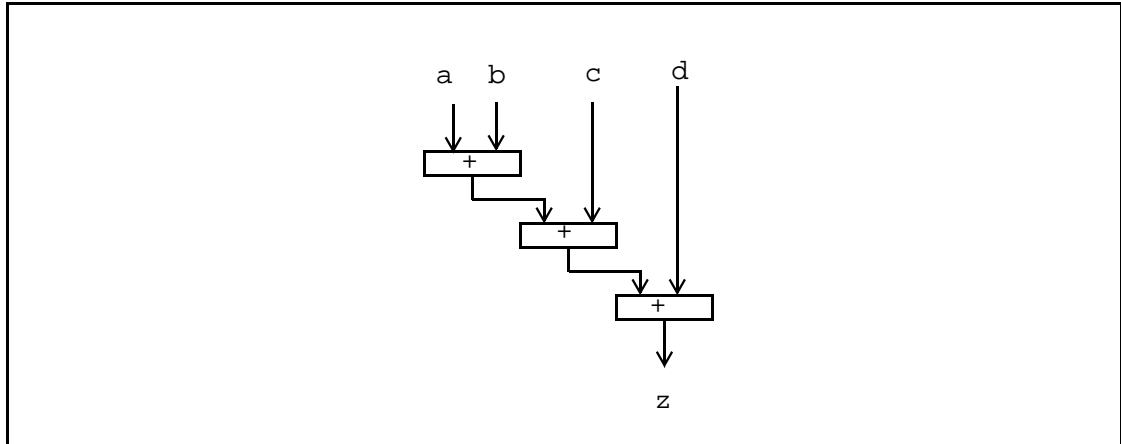
Figure 7-2 Carry-Lookahead Chain Implementation



You can also use parentheses to control the structure of complex components in a design. FPGA Compiler II / *FPGA Express* uses parentheses to define logic groupings. Example 7-4 and Example 7-5 illustrate two groupings of adders. The circuit diagrams show how grouping the logic affects the way the circuit is synthesized. When Example 7-4 is parsed, $(a + b)$ is grouped together by default, then c and d are added one at a time.

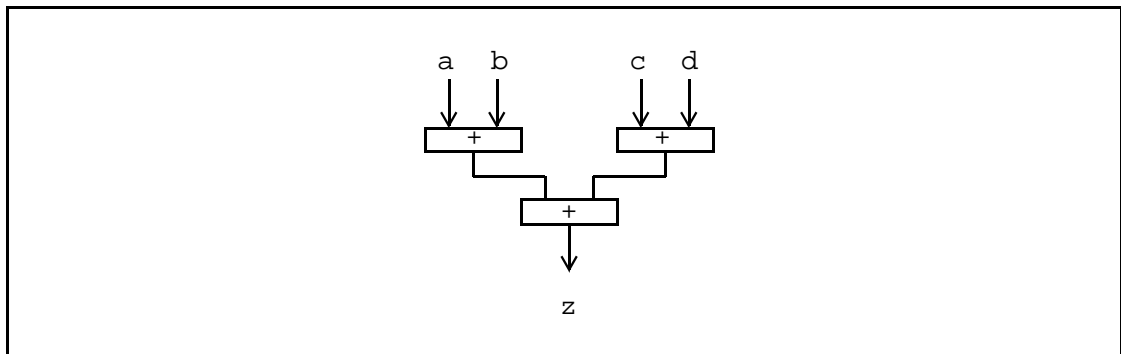
Example 7-4 4-Input Adder

$$z = a + b + c + d;$$



Example 7-5 4-Input Adder With Parentheses

$$z = (a + b) + (c + d);$$



Note:

Manual or automatic resource sharing can also affect the structure of a design.

Using Design Knowledge

In many circumstances, you can improve the quality of synthesized circuits by better describing your high-level knowledge of a circuit. FPGA Compiler II / *FPGA Express* cannot always derive details of a circuit architecture. Any additional architectural information you can provide to FPGA Compiler II / *FPGA Express* can result in a more efficient circuit.

Optimizing Arithmetic Expressions

FPGA Compiler II / *FPGA Express* uses the properties of arithmetic operators (such as the associative and commutative properties of addition) to rearrange an expression so that it results in an optimized implementation. You can also use arithmetic properties to control the choice of implementation for an expression. Three forms of arithmetic optimization are discussed in this section:

- Arranging Expression Trees for Minimum Delay
- Sharing Common Subexpressions

Arranging Expression Trees for Minimum Delay

If your goal is to speed up your design, arithmetic optimization can minimize the delay through an expression tree by rearranging the sequence of the operations. Consider the statement in Example 7-6.

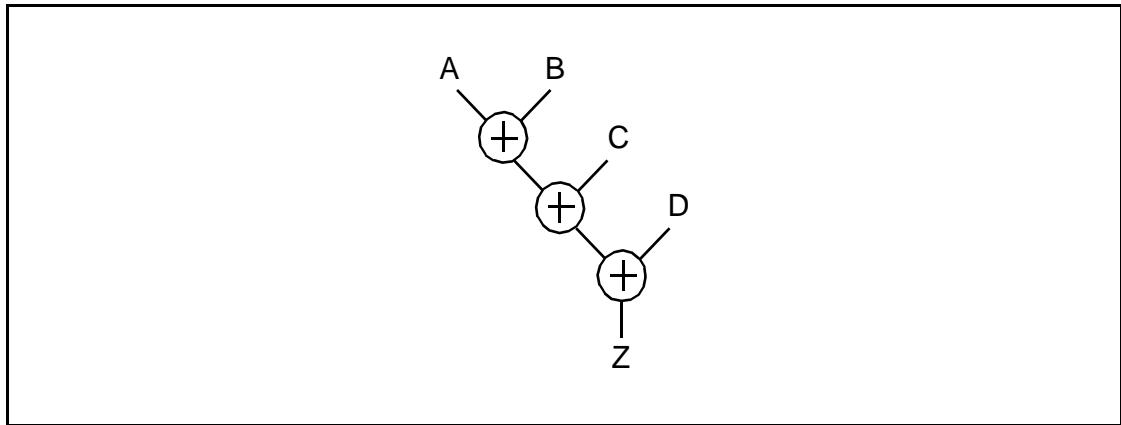
Example 7-6 Simple Arithmetic Expression

$Z \leq A + B + C + D;$

The parser performs each addition in order, as though parentheses were placed as shown, and constructs the expression tree shown in Figure 7-3:

$Z \leq ((A + B) + C) + D;$

Figure 7-3 Default Expression Tree



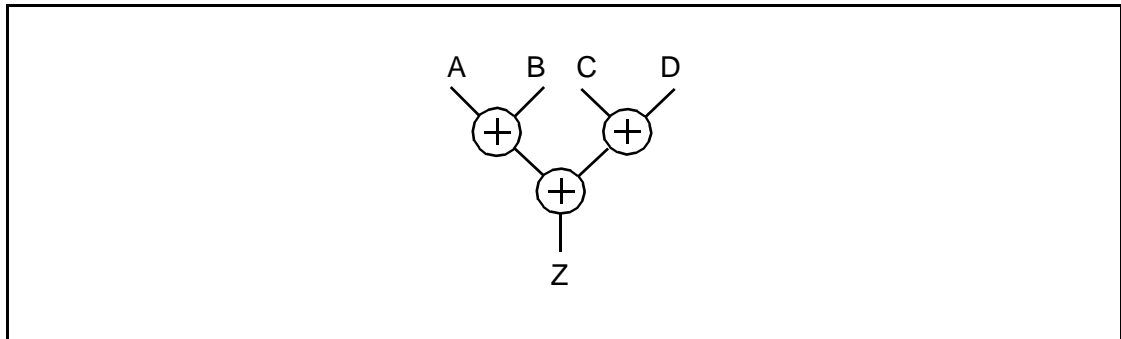
Considering Signal Arrival Times

If all signals arrive at the same time, the critical path can be reduced to two adders.

$Z \leq (A + B) + (C + D);$

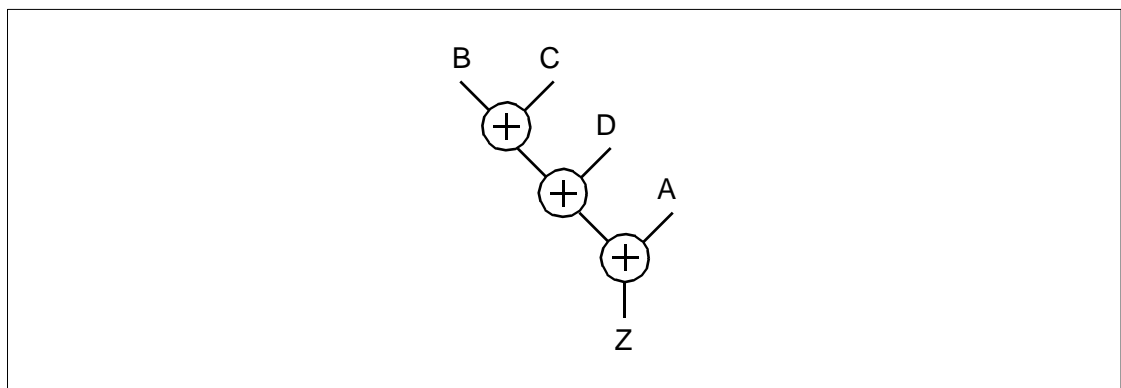
The parser evaluates the expressions in parentheses first and constructs a balanced adder tree, as shown in Figure 7-4.

Figure 7-4 *Balanced Adder Tree (Same Arrival Times for All Signals)*



Suppose signals B, C, and D arrive at the same time and signal A arrives last. The expression tree that produces the minimum delay is shown in Figure 7-5.

Figure 7-5 *Expression Tree With Minimum Delay (Signal A Arrives Last)*



Using Parentheses

You can use parentheses in expressions to exercise more control over the way expression trees are constructed. Parentheses are regarded as user directives that force an expression tree to use the groupings inside the parentheses. The expression tree cannot be rearranged to violate these groupings.

To illustrate the effect of parentheses on the construction of an expression tree, consider Example 7-7.

Example 7-7 Parentheses in an Arithmetic Expression

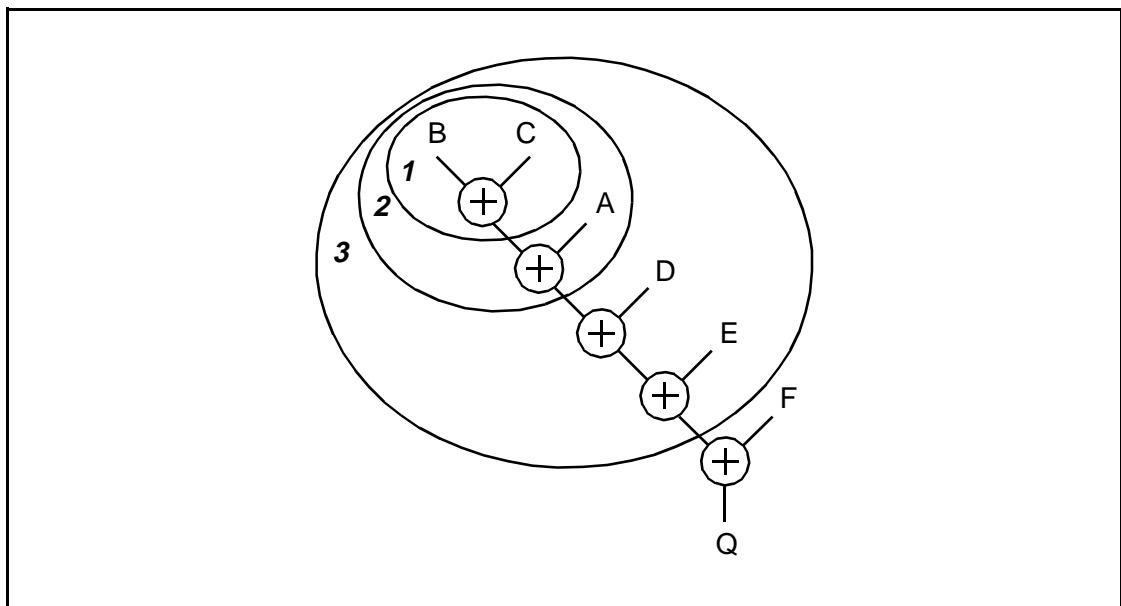
$$Q \leftarrow ((A + (B + C)) + D + E) + F;$$

The parentheses in the expression in Example 7-7 define the following subexpressions, whose numbers correspond to those in Figure 7-6:

- 1 (B + C)
- 2 (A + (B + C))
- 3 ((A + (B + C)) + D + E)

These subexpressions must be preserved in the expression tree. The default expression tree for Example 7-7 is shown in Figure 7-6.

Figure 7-6 Expression Tree With Subexpressions Dictated by Parentheses



Considering Overflow Characteristics

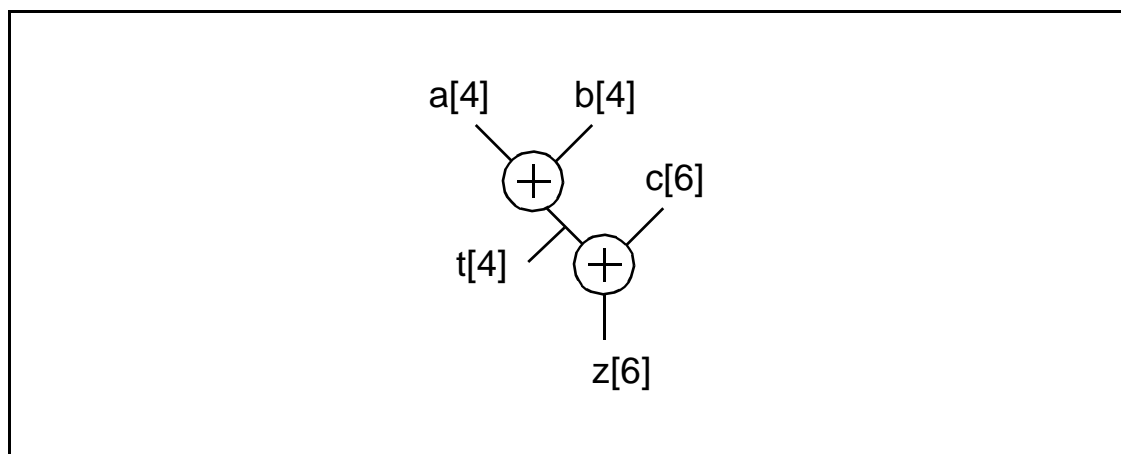
When FPGA Compiler II / FPGA *Express* performs arithmetic optimization, it considers how to handle the overflow from carry bits during addition. The optimized structure of an expression tree is affected by the bit-widths you declare for storing intermediate results. For example, suppose you write an expression that adds two 4-bit numbers and stores the result in a 4-bit register. If the result of the addition overflows the 4-bit output, the most significant bits are truncated. Example 7-8 shows how FPGA Compiler II / FPGA *Express* handles overflow characteristics.

Example 7-8 Adding Numbers of Different Bit-Widths

```
t <= a + b; // a and b are 4-bit numbers
z <= t + c; // c is a 6-bit number
```

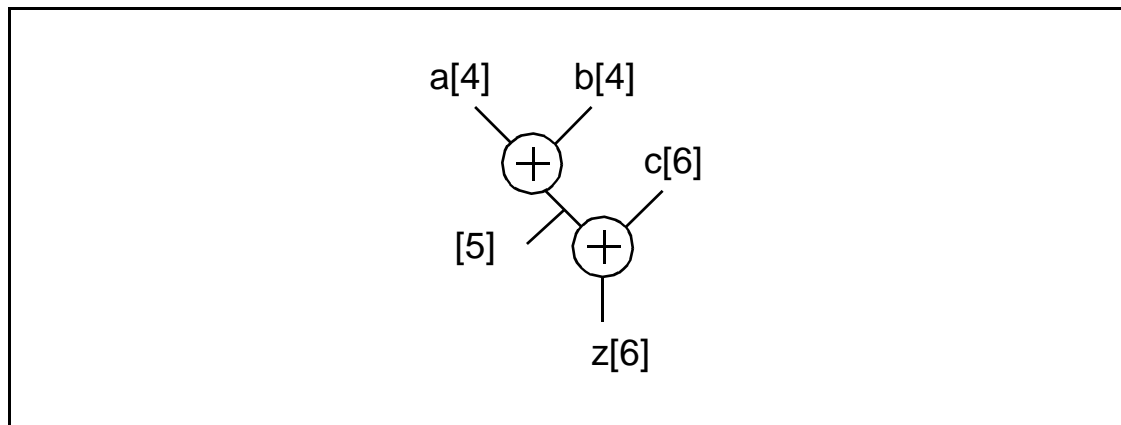
In Example 7-8, three variables are added ($a + b + c$). A temporary variable, t , holds the intermediate result of $a + b$. Suppose t is declared as a 4-bit variable so the overflow bits from the addition of $a + b$ are truncated. The parser determines the default structure of the expression tree, which is shown in Figure 7-7.

Figure 7-7 Default Expression Tree With 4-Bit Temporary Variable



Now suppose the addition is performed without a temporary variable ($z = a + b + c$). FPGA Compiler II / FPGA Express determines that five bits are needed to store the intermediate result of the addition, so no overflow condition exists. The results of the final addition might be different from the first case, where a 4-bit temporary variable is declared that truncates the result of the intermediate addition. Therefore, these two expression trees do not always yield the same result. The expression tree for the second case is shown in Figure 7-8.

Figure 7-8 Expression Tree With 5-Bit Intermediate Result



Sharing Common Subexpressions

Subexpressions consist of two or more variables in an expression. If the same subexpression appears in more than one equation, you might want to share these operations to reduce the area of your circuit. You can force common subexpressions to be shared by declaring a temporary variable to store the subexpression, then use the temporary variable wherever you want to repeat the subexpression. Example 7-9 shows a group of simple additions that use the common subexpression $(a + b)$.

Example 7-9 Simple Additions With a Common Subexpression

```
temp <= a + b;  
x <= temp;  
y <= temp + c;
```

Instead of manually forcing common subexpressions to be shared, you can let FPGA Compiler II / *FPGA Express* automatically determine whether sharing common subexpressions improves your circuit. You do not need to declare a temporary variable to hold the common subexpression in this case.

In some cases, sharing common subexpressions results in more adders being built. Consider Example 7-10, where $A + B$ is a common subexpression.

Example 7-10 Sharing Common Subexpressions

```
if cond1  
    Y <= A + B;  
else  
    Y <= C + D;  
end;  
if cond2  
    Z <= E + F;  
else  
    Z <= A + B;  
end;
```

If the common subexpression $A + B$ is shared, three adders are needed to implement this section of code:

```
(A + B)  
(C + D)  
(E + F)
```

If the common subexpression is not shared, only two adders are needed: one to implement the additions $A + B$ and $C + D$ and one to implement the additions $E + F$ and $A + B$.

FPGA Compiler II / *FPGA Express* analyzes common subexpressions during the resource sharing phase of the compile command and considers area costs and timing characteristics. To turn off the sharing of common subexpressions for the current design, use the constraint manager.

The FPGA Compiler II / *FPGA Express* parser does not identify common subexpressions unless you use parentheses or write them in the same order. For example, the two equations in Example 7-11 use the common subexpression $A + B$.

Example 7-11 Unidentified Common Subexpressions

```
Y <= A + B + C;  
Z <= D + A + B;
```

The parser does not recognize $A + B$ as a common subexpression, because it parses the second equation as $(D + A) + B$. You can force the parser to recognize the common subexpression by rewriting the second assignment statement as

```
Z <= A + B + D;
```

or

```
Z <= D + (A + B);
```

Note:

You do not have to rewrite the assignment statement, because FPGA Compiler II / *FPGA Express* recognizes common subexpressions automatically.

Using Operator Bit-Width Efficiently

You can improve circuits by using operators more carefully. In Example 7-12, the adder sums the 8-bit value of *a* with the lower 4 bits of *temp*. Although *temp* is declared as an 8-bit value, the upper 4 bits of *temp* are always 0, so only the lower 4 bits of *temp* are needed for the addition.

You can simplify the addition by changing *temp* to *temp* [3:0], as shown in Example 7-12. Now, instead of using eight full adders to perform the addition, four full adders are used for the lower 4 bits and four half adders are used for the upper 4 bits. This yields a significant savings in circuit area.

Example 7-12 More Efficient Use of Operators

```
module all (a,b,y);
input  [7:0] a,b;
output [8:0] y;
function [8:0] add_lt_10;
input  [7:0] a,b;
reg [7:0] temp;
begin
    if (b < 10)
        temp = b;
    else
        temp = 10;
    add_lt_10 = a + temp [3:0]; // use [3:0] for temp
end
endfunction
assign y = add_lt_10(a,b);
endmodule
```

Using State Information

When you build finite state machines, you can often specify a constant value of a signal in a particular state. You can write your Verilog description so that FPGA Compiler II / *FPGA Express* produces a more efficient circuit.

Example 7-13, shows the Verilog description of a simple finite state machine.

Example 7-13 A Simple Finite State Machine

```
module machine (x, clock, current_state, z);

input  x, clock;
output [1:0] current_state;
output  z;

reg [1:0]  current_state;
reg       z;
/* Redeclared as reg so they can be assigned to in always
statements. By default, ports are wires and cannot be
assigned to in 'always'
*/
reg [1:0] next_state;
reg previous_z;

parameter [1:0] set0  = 0,
              hold0  = 1,
              set1   = 2;

always @ (x or current_state) begin
    case (current_state)          //synopsys full_case
        /* declared full_case to avoid extraneous latches */
        set0:
            begin
                z = 0 ;           //set z to 0
                next_state = hold0;
            end
    endcase
end
```



```

    hold0:
        begin
            z = previous_z;           //hold value of z
            if (x == 0)
                next_state = hold0;
            else
                next_state = set1;
            end
        set1:
            begin
                z = 1;               //set z to 1
                next_state = set0;
            end
        endcase
    end
always @ (posedge clock) begin
    current_state = next_state;
    previous_z    = z;
end
endmodule

```

In the state hold0, the output z retains its value from the previous state. To synthesize this circuit, a flip-flop is inserted to hold the state previous_z. However, you can make some assertions about the value of z. In the state hold0, the value of z is always 0. This can be deduced from the fact that the state hold0 is entered only from the state set0, where z is always assigned the value 0.

Example 7-14 shows how the Verilog description can be changed to use this assertion, resulting in a simpler circuit (because the flip-flop for previous_z is not required). The changed line is shown in bold.

Example 7-14 Better Implementation of a Finite State Machine

```
module machine (x, clock, current_state, z);

    input  x, clock;
    output [1:0]current_state;
    output  z;

    reg [1:0] current_state;
    reg      z;
    /* Redeclared as reg so they can be assigned to in always
    statements. By default, ports are wires and cannot be
    assigned to in 'always'
    */
    reg [1:0] next_state;

    parameter [1:0] set0  = 0,
                  hold0  = 1,
                  set1   = 2;

    always @ (x or current_state) begin
        case (current_state) //synopsys full_case
            /* declared full_case to avoid extraneous latches */
            set0:
                begin
                    z = 0 ;           //set z to 0
                    next_state = hold0;
                end
            hold0:
                begin
                    z = 0;           //hold z at 0
                    if (x == 0)
                        next_state = hold0;
                    else
                        next_state = set1;
                end
            set1:
                begin
                    z = 1;           //set z to 1
                    next_state = set0;
                end
        endcase
    end
```

```
end
always @ (posedge clock) begin
    current_state = next_state;
end
endmodule
```

Describing State Machines

You can use an implicit state style or an explicit state style to describe a state machine. In the implicit state style, a clock edge (negedge or posedge) signals a transition in the circuit from one state to another. In the explicit state style, you use a constant declaration to assign a value to all states. Each state and its transition to the next state are defined under the case statement. Use the implicit state style to describe a single flow of control through a circuit (where each state in the state machine can be reached only from one other state). Use the explicit state style to describe operations such as synchronous resets.

Example 7-15 shows a description of a circuit that sums data over three clock cycles. The circuit has a single flow of control, so the implicit style is preferable.

Example 7-15 Summing Three Cycles of Data in the Implicit State Style (Preferred)

```
module sum3 ( data, clk, total );
input [7:0] data;
input clk;
output [7:0] total;

reg total;

always
begin
    @ (posedge clk)
        total = data;
    @ (posedge clk)
        total = total + data;
    @ (posedge clk)
        total = total + data;
end
endmodule
```

Note:

With the implicit state style, you must use the same clock phase (either `posedge` or `negedge`) for each event expression. Implicit states can be updated only if they are controlled by a single clock phase.

Example 7-16 shows a description of the same circuit in the explicit state style. This circuit description requires more lines of code than the previous example, although FPGA Compiler II / *FPGA Express* synthesizes the same circuit for both descriptions.

Example 7-16 *Summing Three Cycles of Data in the Explicit State Style (Not Advisable)*

```
module sum3 ( data, clk, total );
input [7:0] data;
input clk;
output [7:0] total;

reg total;
reg [1:0] state;

parameter S0 = 0, S1 = 1, S2 = 2;

always @ (posedge clk)
begin
    case (state)
    S0: begin
        total = data;
        state = S1;
    end
    S1: begin
        total = total + data;
        state = S2;
    end
    default : begin
        total = total + data;
        state = S0;
    end
    endcase
end
endmodule
```

Example 7-17 shows a description of the same circuit with a synchronous reset added. This example is coded in the explicit state style. Notice that the reset operation is addressed once before the case statement.

Example 7-17 Synchronous Reset—Explicit State Style (Preferred)

```
module SUM3 ( data, clk, total, reset );
input [7:0] data;
input clk, reset;
output [7:0] total;

reg total;
reg [1:0] state;

parameter S0 = 0, S1 = 1, S2 = 2;

always @ (posedge clk)
begin
    if (reset)
        state = S0;
    else
        case (state)
        S0: begin
            total = data;
            state = S1;
        end
        S1: begin
            total = total + data;
            state = S2;
        end
        default : begin
            total = total + data;
            state = S0;
        end
        endcase;
end
endmodule
```

Example 7-18 shows how to describe the same function in the implicit state style. This style is not as efficient for describing synchronous resets. In this case, the reset operation has to be addressed for every `always @` statement.

Example 7-18 Synchronous Reset—Implicit State Style (Not Advisable)

```
module SUM3 ( data, clk, total, reset );
input [7:0] data;
input clk, reset;
output [7:0] total;

reg total;

always
    begin: reset_label

        @ (posedge clk)
        if (reset)
            begin
                total = 8'b0;
                disable reset_label;
            end
        else
            total = data;

        @ (posedge clk)
        if (reset)
            begin
                total = 8'b0;
                disable reset_label;
            end
        else
            total = total + data;

        @ (posedge clk)
        if (reset)
            begin
                total = 8'b0;
                disable reset_label;
            end
        else
            total = total + data;
    end
endmodule
```

Minimizing Registers

In an always block that is triggered by a clock edge, every variable that has a value assigned has its value held in a flip-flop.

Organize your Verilog description so you build only as many registers as you need. Example 7-19 shows a description where extra registers are implied.

Example 7-19 Inefficient Circuit Description With Six Implied Registers

```
module count (clock, reset, and_bits, or_bits, xor_bits);
input clock, reset;
output and_bits, or_bits, xor_bits;
reg and_bits, or_bits, xor_bits;

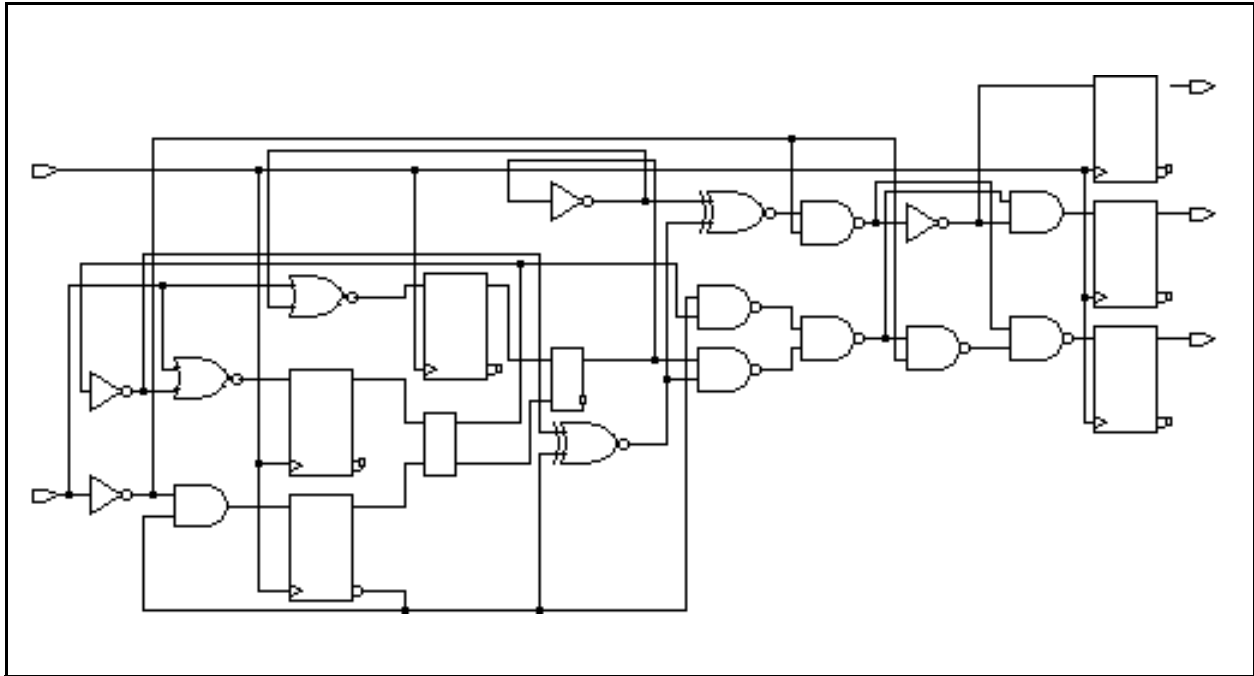
reg [2:0] count;

    always @(posedge clock) begin
        if (reset)
            count = 3'60;
        else
            count = count + 1;

        and_bits = & count;
        or_bits  = | count;
        xor_bits = ^ count;
    end
endmodule
```

This description implies the use of six flip-flops: three to hold the values of count and one each to hold and_bits, or_bits, and xor_bits. However, the values of the outputs and_bits, or_bits, and xor_bits depend solely on the values of count. Because count is registered, there is no reason to register the three outputs. The synthesized circuit is shown in Figure 7-9.

Figure 7-9 Synthesized Circuit With Six Implied Registers



To avoid implying extra registers, you can assign the outputs from within an asynchronous always block. Example 7-20 shows the same logic described with two always blocks, one synchronous and one asynchronous, which separate registered or sequential logic from combinational logic. This technique is useful for describing finite state machines. Signal assignments in the synchronous always block are registered. Signal assignments in the asynchronous always block are not. Therefore, this version of the design uses three fewer flip-flops than the version in Example 7-19.

Example 7-20 Circuit With Three Implied Registers

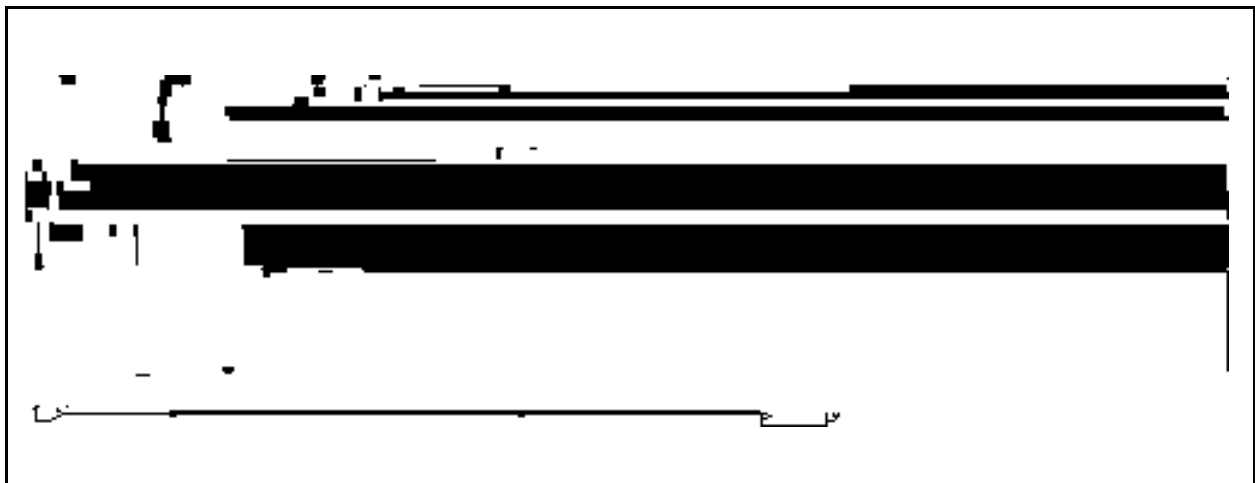
```
module count (clock, reset, and_bits, or_bits, xor_bits);
input clock, reset;
output and_bits, or_bits, xor_bits;
reg and_bits, or_bits, xor_bits;

reg [2:0] count;

always @(posedge clock) begin//synchronous
    if (reset)
        count = 3'b0;
    else
        count = count + 1;
end
always @(count) begin//asynchronous
    and_bits = & count;
    or_bits = | count;
    xor_bits = ^ count;
end
endmodule
```

The more efficient version of the circuit is shown in Figure 7-10.

Figure 7-10 Synthesized Circuit With Three Implied Registers



Separating Sequential and Combinational Assignments

To compute values synchronously and store them in flip-flops, set up an always block with a signal edge trigger. To let other values change asynchronously, make a separate always block with no signal edge trigger. Put the assignments you want clocked in the always block with the signal edge trigger and the other assignments in the other always block. This technique is used for creating Mealy machines, such as the one in Example 7-21. Note that out changes asynchronously with in1 or in2.

Example 7-21 Mealy Machine

```
module mealy (in1, in2, clk, reset, out);
    input in1, in2, clk, reset;
    output out;
    reg current_state, next_state, out;

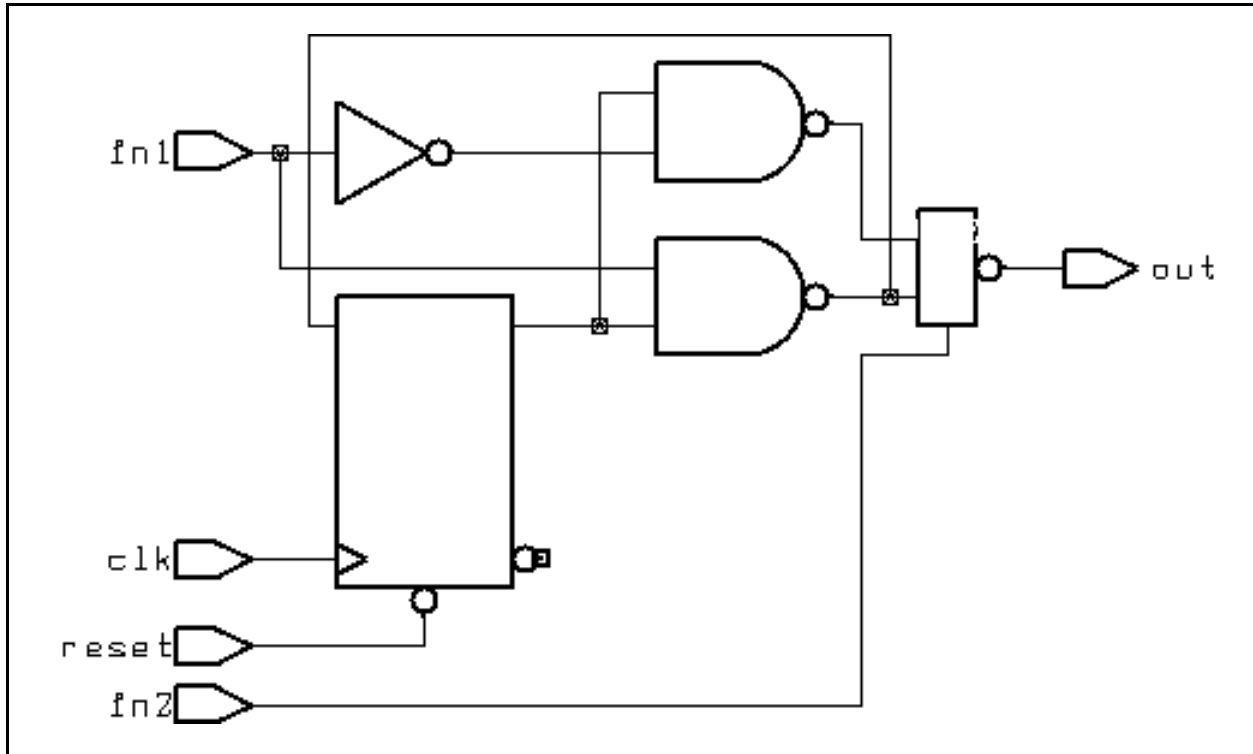
    always @(posedge clk or negedge reset)
    // state vector flip-flops (sequential)
        if (!reset)
            current_state = 1'b0;
        else
            current_state = next_state;

    always @(in1 or in2 or current_state)
    // output and state vector decode (combinational)
        case (current_state)
            0: begin
                    next_state = 1;
                    out = 1'b0;
                end
            1: if (in1) begin
                    next_state = 1'b0;
                    out = in2;
                end
                else begin
                    next_state = 1'b1;
                    out = !in2;
                end
        endcase

endmodule
```

The schematic for this circuit is shown in Figure 7-11.

Figure 7-11 Mealy Machine Schematic



Don't Care Inference

You can greatly reduce circuit area by using don't care values. To use a don't care value in your design, create an enumerated type for the don't care value.

Don't care values are best used as default assignments to variables. You can assign a don't care value to a variable at the beginning of a module, in the default section of a case statement, or in the else section of an if statement.

Limitations of Using Don't Care Values

In some cases, using don't care values as default assignments can cause these problems:

- Don't care values create a greater potential for mismatches between simulation and synthesis.
- Defaults for variables can hide mistakes in the Verilog code.

For example, you might assign a default don't care value to VAR. If you later assign a value to VAR, expecting VAR to be a don't care value, you might have overlooked an intervening condition under which VAR is assigned.

Therefore, when you assign a value to a variable (or signal) that contains a don't care value, make sure that the variable (or signal) is really a don't care value under those conditions. Note that assignment to an x is interpreted as a don't care value.

Differences Between Simulation and Synthesis

Don't care values are treated differently in simulation and in synthesis, and there can be a mismatch between the two. To a simulator, a don't care is a distinct value, different from a one or a zero. In synthesis, however, a don't care becomes a zero or a one (and hardware is built that treats the don't care value as either a zero or a one).

Whenever a comparison is made with a variable whose value is don't care, simulation and synthesis can differ. Therefore, the safest way to use don't care values is to

- Assign don't care values only to output ports
- Make sure that the design never reads output ports

These guidelines guarantee that when you simulate within the scope of the design, the only difference between simulation and synthesis occurs when the simulator indicates that an output is a don't care value.

If you use don't care values internally to a design, expressions FPGA Compiler II / FPGA *Express* compares to don't care values (X) are synthesized as though values are not equal to X.

For example,

```
if A = 'X' then
...
```

is synthesized as

```
if FALSE then
...
```

If you use expressions comparing values with X, pre-synthesis and post-synthesis simulation results might not agree. For this reason, FPGA Compiler II / FPGA *Express* issues the following warning:

```
Warning: A partial don't-care value was read in routine test
line 24 in file 'test.v' This may cause simulation to
disagree with synthesis. (HDL-171)
```

Propagating Constants

Constant propagation is the compile-time evaluation of expressions that contain constants. FPGA Compiler II / *FPGA Express* uses constant propagation to reduce the amount of hardware required to implement complex operators. Therefore, when you know that a variable is a constant, specify it as a constant. For example, a + operator with a constant of 1 as one of its arguments causes an incrementer, rather than a general adder, to be built. If both arguments of an operator are constants, no hardware is constructed, because FPGA Compiler II / *FPGA Express* can calculate the expression's value and insert it directly into the circuit.

Comparators and shifters also benefit from constant propagation. When you shift a vector by a constant, the implementation requires only a reordering (rewiring) of bits, so no logic is needed.

Synthesis Issues

The next two sections describe feedback paths and latches that result from ambiguities in signal or variable assignments, and asynchronous behavior.

Feedback Paths and Latches

Sometimes your Verilog source can imply combinational feedback paths or latches in synthesized logic. This happens when a signal or a variable in a combinational logic block (an always block without a posedge or negedge clock statement) is not fully specified. A variable or signal is fully specified when it is assigned under all possible conditions.

Synthesizing Asynchronous Designs

In a synchronous design, all registers use the same clock signal. That clock signal must be a primary input to the design. A synchronous design has no combinational feedback paths, one-shots, or delay lines. Synchronous designs perform the same function regardless of the clock rate, as long as the rate is slow enough to allow signals to propagate all the way through the combinational logic between registers.

Synopsys synthesis tools offer limited support for asynchronous designs. The most common way to produce asynchronous logic in Verilog is to use gated clocks on registers. If you use asynchronous design techniques, synthesis and simulation results might not agree. Because FPGA Compiler II / FPGA *Express* does not issue warning messages for asynchronous designs, you are responsible for verifying the correctness of your circuit.

The following examples show two approaches to the same counter design: Example 7-22 is synchronous, and Example 7-23 is asynchronous.

Example 7-22 Fully Synchronous Counter Design

```
module COUNT (RESET, ENABLE, CLK, Z);

    input RESET, ENABLE, CLK;
    output [2:0] Z;
    reg [2:0] Z;

    always @ (posedge CLK) begin
        if (RESET) begin
            Z = 3'b0;
        end else if (ENABLE == 1'b1) begin
            if (Z == 3'd7) begin
                Z = 3'b0;
            end else begin
                Z = Z + 3'b1;
            end
        end
    end
end

endmodule
```

Example 7-23 Asynchronous Counter Design

```
module COUNT (RESET, ENABLE, CLK, Z);

    input RESET, ENABLE, CLK;
    output [2:0] Z;
    reg [2:0] Z;
    wire GATED_CLK = CLK & ENABLE;

    always @ (posedge GATED_CLK or posedge RESET) begin
        if (RESET) begin
            Z = 3'b0;
        end else begin
            if (Z == 3'd7) begin
                Z = 3'b0;
            end else begin
                Z = Z + 3'b1;
            end
        end
    end
end

endmodule
```

The asynchronous version of the design uses two asynchronous design techniques. The first technique is to enable the counter by ANDing the clock with the enable line. The second technique is to use an asynchronous reset. These techniques work if the proper timing relationships exist between the asynchronous control lines (ENABLE and RESET) and the clock (CLK) and if the control lines are glitch-free.

Some forms of asynchronous behavior are not supported. For example, you might expect the following circuit description of a one-shot signal generator to generate three inverters (an inverting delay line) and a NAND gate.

```
X = A ~& ( ~( ~( ~ A) ) );
```

However, this circuit description is optimized to

```
X = A ~& ( ~ A ); then X = 1;
```

Designing for Overall Efficiency

The efficiency of a synthesized design depends primarily on how you describe its component structure. The next two sections explain how to describe random logic and how to share complex operators.

Describing Random Logic

You can describe random logic with many different shorthand Verilog expressions. FPGA Compiler II / FPGA *Express* often generates the same optimized logic for equivalent expressions, so your description style for random logic does not affect the efficiency of the circuit.

Example 7-24 shows four groups of statements that are equivalent. (Assume that a, b, and c are 4-bit variables.) FPGA Compiler II / FPGA *Express* creates the same optimized logic in all four cases.

Example 7-24 Equivalent Statements

```
c = a & b;

c[3:0] = a[3:0] & b[3:0];

c[3] = a[3] & b[3];
c[2] = a[2] & b[2];
c[1] = a[1] & b[1];
c[0] = a[0] & b[0];

for (i = 0; i <= 3; i = i + 1)
    c[i] = a[i] & b[i];
```

Sharing Complex Operators

You can use automatic resource sharing to share most operators. However, some complex operators can be shared only if you rewrite your source description more efficiently. These operators are

- Noncomputable array index
- Function call
- Shifter

Example 7-25 shows a circuit description that creates more functional units than necessary when automatic resource sharing is turned off.

Example 7-25 Inefficient Circuit Description With Two Array Indexes

```
module rs(a, i, j, c, y, z);

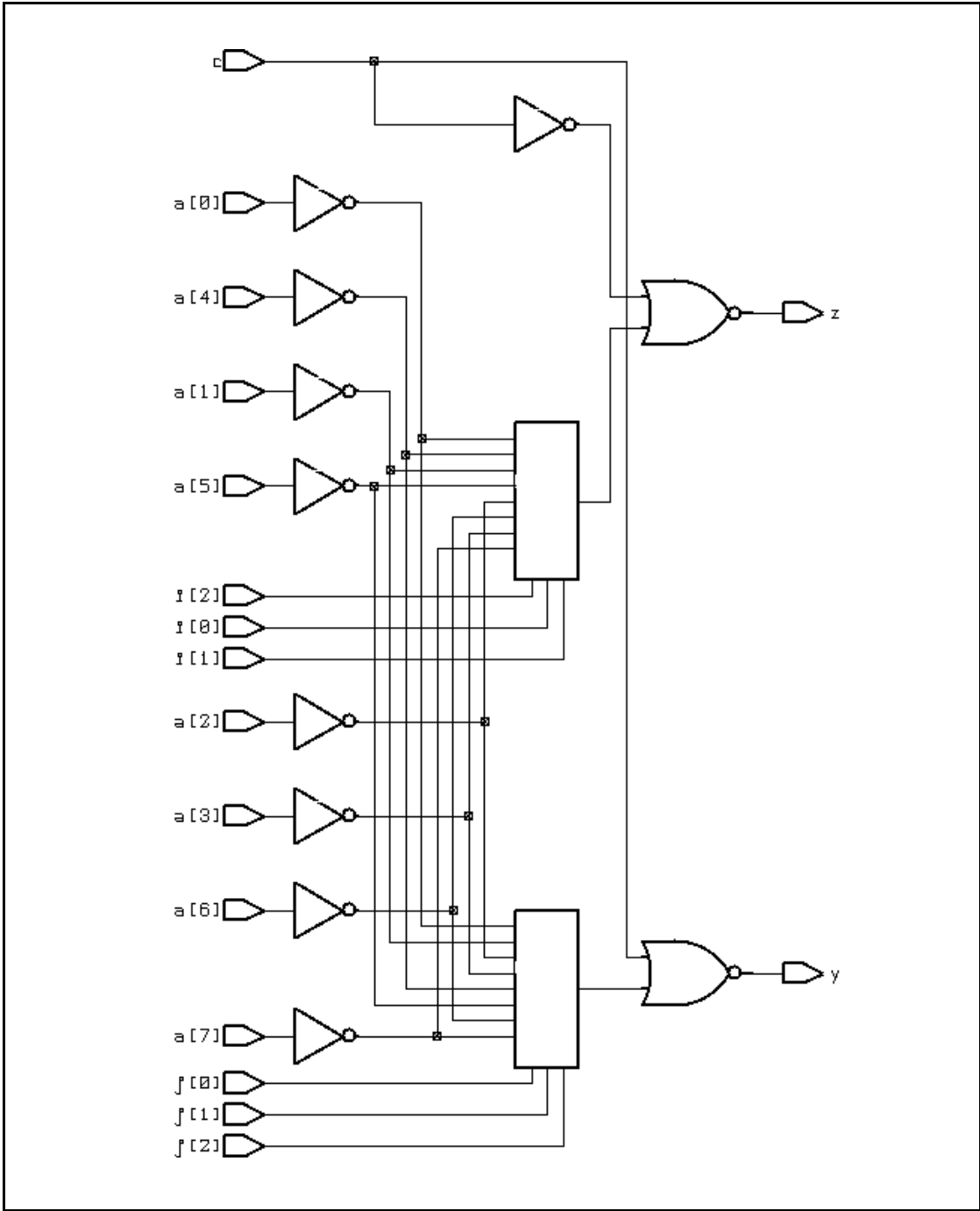
    input [7:0] a;
    input [2:0] i,j;
    input c;

    output y, z;
    reg y, z;

    always @(a or i or j or c)
        begin
            z=0;
            y=0;
            if(c)
                begin
                    z = a[i];
                end
            else
                begin
                    y = a[j];
                end
            end
        end
    endmodule
```

The schematic for this code description is shown in Figure 7-12.

Figure 7-12 Circuit Schematic With Two Array Indexes



You can rewrite the circuit description in Example 7-25 so that it contains only one array index, as shown in Example 7-26.

Example 7-26 Efficient Circuit Description With One Array Index

```
module rs1(a, i, j, c, y, z);

    input [7:0] a;
    input [2:0] i,j;
    input c;

    output y, z;
    reg y, z;

    reg [3:0] index;
    reg temp;

    always @(a or i or j or c) begin
        if(c)
            begin
                index = i;
            end
        else
            begin
                index = j;
            end

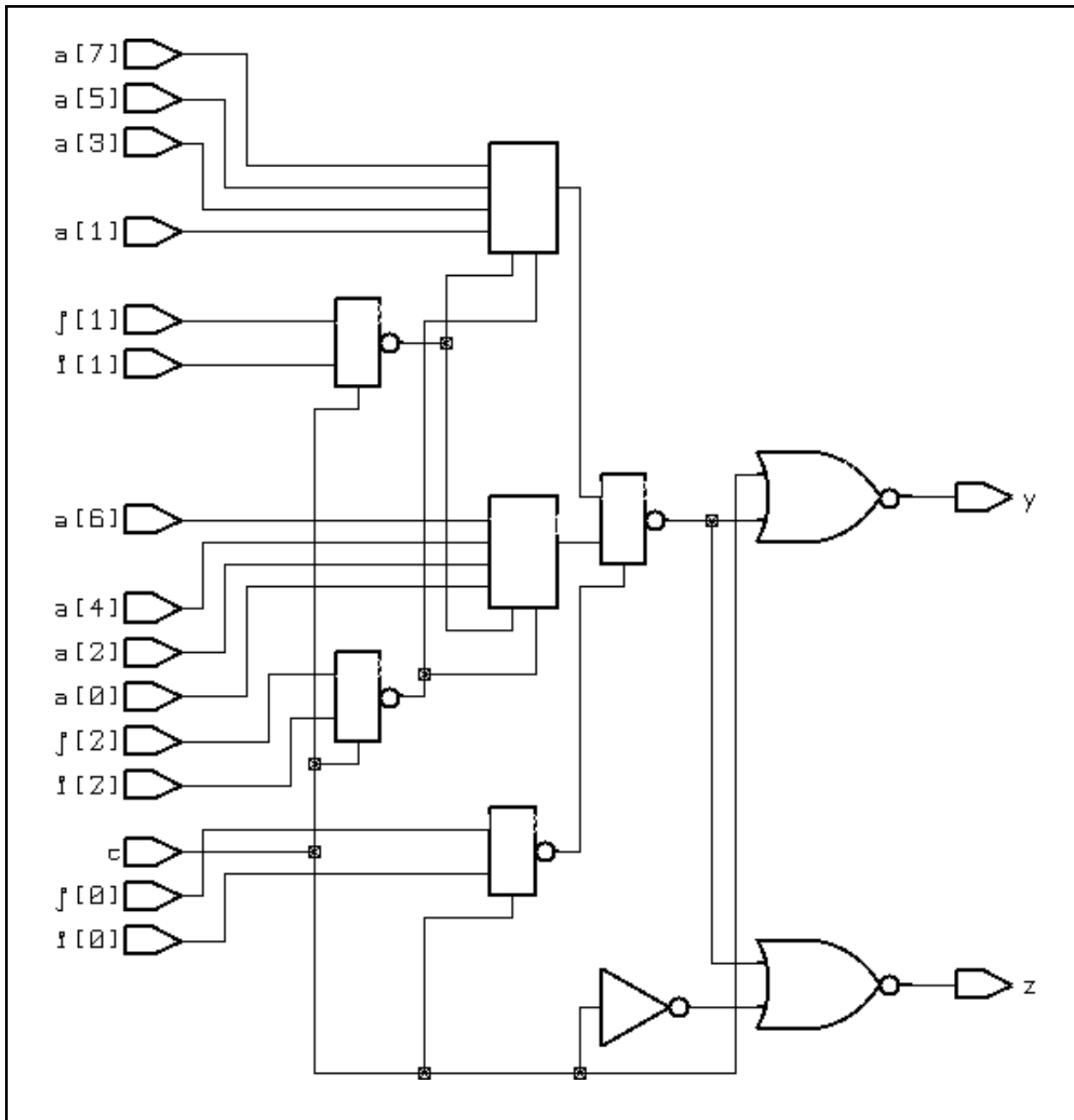
        temp = a[index];

        z=0;
        y=0;
        if(c)
            begin
                z = temp;
            end
        else
            begin
                y = temp;
            end
    end

endmodule
```

The circuit in Example 7-26 is more efficient than that in Example 7-25, since it uses a temporary register, temp, to store the value evaluated in the if statement. The schematic is shown in Figure 7-13.

Figure 7-13 Circuit Schematic With One Array Index



Consider resource sharing whenever you use a complex operation more than once. Complex operations include adders, multipliers, shifters (only when shifting by a variable amount), comparators, and most user-defined functions.

8

FPGA Compiler II / FPGA *Express* Directives

Specific aspects of the synthesis process can be controlled by special comments in the Verilog source code called FPGA Compiler II / FPGA *Express* directives. Because these directives are just a special case of regular comments, they are ignored by the Verilog HDL Simulator and do not affect simulation. This chapter describes FPGA Compiler II / FPGA *Express* directives and their effect on translation, in the following sections:

- Notation for FPGA Compiler II / FPGA Express Directives
- `translate_off` and `translate_on` Directives
- `parallel_case` Directive
- `full_case` Directive
- `state_vector` Directive
- `enum` Directive

- Component Implication

Notation for FPGA Compiler II / FPGA *Express* Directives

The special comments that make up FPGA Compiler II / FPGA *Express* directives begin, like all other Verilog comments, with the characters `//` or `/*`. The `//` characters begin a comment that fits on one line (most FPGA Compiler II / FPGA *Express* directives do). If you use the `/*` characters to begin a multiline comment, you must end the comment with `*/`. You do not need to use the `/*` characters at the beginning of each line but only at the beginning of the first line.

Note:

You cannot use `// synopsys` in a regular comment. Also, the compiler displays a syntax error if Verilog code is in a `// synopsys` directive.

`translate_off` and `translate_on` Directives

When the `// synopsys translate_off` and `// synopsys translate_on` directives are present, FPGA Compiler II / FPGA *Express* suspends translation of the source code and restarts translation at a later point. Use these directives when your Verilog source code contains commands specific to simulation that FPGA Compiler II / FPGA *Express* does not accept.

You turn translation off by using either

```
// synopsys translate_off  
  
/* synopsys translate_off */
```

You turn translation back on by using either

```
// synopsys translate_on  
  
/* synopsys translate_on */
```

At the beginning of each Verilog file, translation is enabled. After that, you can use the `translate_off` and `translate_on` directives anywhere in the text. These directives must be used in pairs. Each `translate_off` must appear before its corresponding `translate_on`. Example 8-1 shows a simulation driver protected by a `translate_off` directive.

Example 8-1 // synopsys translate_on and // synopsys translate_off Directives

```
module trivial (a, b, f);  
input a,b;  
output f;  
    assign f = a & b;  
    // synopsys translate_off  
    initial $monitor (a, b, f);  
    // synopsys translate_on  
endmodule  
  
/* synopsys translate_off */  
module driver;  
    reg [1:0] value_in;  
    integer i;  
    trivial triv1(value_in[1], value_in[0]);  
    initial begin  
        for (i = 0; i < 4; i = i + 1)  
            #10 value_in = i;  
    end  
endmodule  
/* synopsys translate_on */
```

parallel_case Directive

The `// synopsys parallel_case` directive affects the way logic is generated for the case statement. As presented in “Full Case and Parallel Case” on page 5-20, a case statement generates the logic for a priority encoder. Under certain circumstances, you might not want to build a priority encoder to handle a case statement. You can use the `parallel_case` directive to force FPGA Compiler II / FPGA *Express* to generate multiplexer logic instead.

The syntax for the `parallel_case` directive is

```
// synopsys parallel_case

or

/* synopsys parallel_case */
```

In Example 8-2, the states of a state machine are encoded as one hot signal. If the case statement were implemented as a priority encoder, the generated logic would be unnecessarily complex.

Example 8-2 // synopsys parallel_case Directives

```
reg [3:0] current_state, next_state;
parameter state1 = 4'b0001, state2 = 4'b0010,
           state3 = 4'b0100, state4 = 4'b1000;

case (1)//synopsys parallel_case

    current_state[0] : next_state = state2;
    current_state[1] : next_state = state3;
    current_state[2] : next_state = state4;
    current_state[3] : next_state = state1;

endcase
```

Use the `parallel_case` directive immediately after the case expression, as shown. This directive makes all case-item evaluations in parallel. All case items that evaluate to true are executed, not just the first, which could give you unexpected results.

In general, use `parallel_case` when you know that only one case item is executed. If only one case item is executed, the logic generated from a `parallel_case` directive performs the same function as the circuit when it is simulated. If two case items are executed and you have used the `parallel_case` directive, the generated logic is not the same as the simulated description.

full_case Directive

The `// synopsys full_case` directive asserts that all possible clauses of a case statement have been covered and that no default clause is necessary. This directive has two uses: It avoids the need for default logic, and it can avoid latch inference from a case statement by asserting that all necessary conditions are covered by the given branches of the case statement. As shown in “Full Case and Parallel Case” on page 5-20, a latch can be inferred whenever a variable is not assigned a value under all conditions.

The syntax for the `full_case` directive is either

```
// synopsys full_case  
  
/* synopsys full_case */
```

If the case statement contains a default clause, FPGA Compiler II / FPGA *Express* assumes that all conditions are covered. If there is no default clause and you do not want latches to be created, use the `full_case` directive to indicate that all necessary conditions are described in the case statement.

Example 8-3 shows two uses of `full_case`. The `parallel_case` and `full_case` directives can be combined in one comment.

Example 8-3 // synopsys full_case Directives

```
reg [1:0] in, out;
reg [3:0] current_state, next_state;
parameter state1 = 4'b0001, state2 = 4'b0010,
           state3 = 4'b0100, state4 = 4'b1000;

case (in) // synopsys full_case
  0: out = 2;
  1: out = 3;
  2: out = 0;
endcase

case (1) // synopsys parallel_case full_case
  current_state[0] : next_state = state2;
  current_state[1] : next_state = state3;
  current_state[2] : next_state = state4;
  current_state[3] : next_state = state1;
endcase
```

In the first case statement, the condition `in == 3` is not covered. You can either use a default clause to cover all other conditions or use the `full_case` directive (as in Example 8-3) to indicate that other branch conditions do not occur. If you cover all possible conditions explicitly, FPGA Compiler II / FPGA *Express* recognizes the case statement as full-case, so the `full_case` directive is not necessary.

The second case statement in Example 8-3 does not cover all 16 possible branch conditions. For example, `current_state == 4'b0101` is not covered. The `parallel_case` directive is used in this example because only one of the four case items can evaluate to true and be executed.

Although you can use the `full_case` directive to avoid creating latches, using this directive does not guarantee that latches will not be built. You still must assign a value to each variable used in the case statement in all branches of the case statement. Example 8-4 illustrates a situation in which the `full_case` directive prevents a latch from being inferred for variable `b` but not for variable `a`.

Example 8-4 Latches and // synopsys full_case

```
reg a, b;
reg [1:0] c;
case (c) // synopsys full_case
    0: begin a = 1; b = 0; end
    1: begin a = 0; b = 0; end
    2: begin a = 1; b = 1; end
    3: b = 1; // a is not assigned here
endcase
```

In general, use `full_case` when you know that all possible branches of the case statement have been enumerated, or at least all branches that can occur. If all branches that can occur are enumerated, the logic generated from the case statement performs the same function as the simulated circuit. If a case condition is not fully enumerated, the generated logic and the simulation are not the same.

Note:

You do not need the `full_case` directive if you have a default branch or you enumerate all possible branches in a case statement, because FPGA Compiler II / *FPGA Express* assumes the case statement is `full_case`.

state_vector Directive

The `// synopsys state_vector` directive labels a variable in a Verilog description as the state vector of an equivalent finite state machine.

The syntax for the `state_vector` directive is

```
// synopsys state_vector vector_name
```

or

```
/* synopsys state_vector vector_name */
```

The `vector_name` variable is the name chosen as a state vector. This declaration allows Synopsys FPGA Compiler II / *FPGA Express* to extract the labeled state vector from the Verilog description. Used with the `enum` directive, described in the next section, the `state_vector` directive allows you to define the state vector of a finite state machine (and its encodings) from a Verilog description. Example 8-5 shows one way to use the `state_vector` directive.

Caution!

Do not define two `state_vector` directives in one module. Although FPGA Compiler II / *FPGA Express* does not issue an error message, it recognizes only the first `state_vector` directive and ignores the second.

Example 8-5 //synopsys state_vector Example

```
reg [1:0] state, next_state;
// synopsys state_vector state

always @ (state or in) begin
    case (state) // synopsys full_case
        0: begin
            out = 3;
            next_state = 1;
        end
        1: begin
            out = 2;
            next_state = 2;
        end
        2: begin
            out = 1;
            next_state = 3;
        end
        3: begin
            out = 0;
            if (in)
                next_state = 0;
            else
                next_state = 3;
        end
    endcase
end

always @ (posedge clock)
    state = next_state;
```

enum Directive

The `// synopsys enum` directive is designed for use with the Verilog parameter definition statement to specify state machine encodings. When a variable is marked as a `state_vector` (see “`state_vector Directive`” on page 8-8) and it is declared as an enum, the Synopsys FPGA Compiler II / FPGA Express uses the enum values and names for the states of an extracted state machine.

The syntax of the enum directive is either

```
// synopsys enum enum_name
/* synopsys enum enum_name */
```

Example 8-6 shows the declaration of an enumeration of type colors that is 3 bits wide and has the enumeration literals red, green, blue, and cyan with the values shown.

Example 8-6 Enumeration of Type Colors

```
parameter [2:0] // synopsys enum colors
red = 3'b000, green = 3'b001, blue = 3'b010, cyan = 3'b011;
```

The enumeration must include a size (bit-width) specification. Example 8-7 shows an invalid enum declaration.

Example 8-7 Invalid enum Declaration

```
parameter /* synopsys enum colors */
red = 3'b000, green = 1;
// [2:0] required
```

Example 8-8 shows a register, a wire, and an input port with the declared type of colors. In each of the following declarations, the array bounds must match those of the enumeration declaration. If you use different bounds, synthesis might not agree with simulation behavior.

Example 8-8 More enum Type Declarations

```
reg    [2:0] /* synopsys enum colors */ counter;
wire   [2:0] /* synopsys enum colors */ peri_bus;
input  [2:0] /* synopsys enum colors */ input_port;
```

Even though you declare a variable to be of type enum, it can still be assigned a bit value that is not one of the enumeration values in the definition. Example 8-9 relates to Example 8-8 and shows an invalid encoding for colors.

Example 8-9 Invalid Bit Value Encoding for Colors

```
counter = 3'b111;
```

Because 111 is not in the definition for colors, it is not a valid encoding. FPGA Compiler II / FPGA Express accepts this encoding, because it is valid Verilog code, but FPGA Compiler II / FPGA Express recognizes this assignment as an invalid encoding and ignores it.

You can use enumeration literals just like constants, as shown in Example 8-10.

Example 8-10 Enumeration Literals Used as Constants

```
if (input_port == blue)
    counter = red;
```

You can also use enumeration with the `state_vector` directive. Example 8-11 shows how the `state_vector` variable is tagged by use of enumeration.

Example 8-11 *Finite State Machine With // synopsys enum and // synopsys state_vector*

```
// This finite-state machine (Mealy type) reads 1 bit
// per cycle and detects 3 or more consecutive 1s.

module enum2_V(signal, clock, detect);
input signal, clock;
output detect;
reg detect;

// Declare the symbolic names for states
parameter [1:0]//synopsys enum state_info
    NO_ONES = 2'h0,
    ONE_ONE = 2'h1,
    TWO_ONES = 2'h2,
    AT_LEAST_THREE_ONES = 2'h3;

// Declare current state and next state variables.
reg [1:0] /* synopsys enum state_info */    cs;
reg [1:0] /* synopsys enum state_info */    ns;

// synopsys state_vector cs

always @ (cs or signal)

begin
    detect = 0;// default values
    if (signal == 0)
        ns = NO_ONES;
    else
        case (cs)          // synopsys full_case
            NO_ONES: ns = ONE_ONE;
            ONE_ONE: ns = TWO_ONES;
            TWO_ONES, ns = AT_LEAST_THREE_ONES;
            AT_LEAST_THREE_ONES:
                begin
                    ns = AT_LEAST_THREE_ONES;
                    detect = 1;
                end
        endcase
end
```

```

always @ (posedge clock) begin
    cs = ns;
end
endmodule

```

Enumerated types are designed to be used as whole entities. This design allows FPGA Compiler II / FPGA *Express* to rebind the encodings of an enumerated type more easily. You cannot select a bit or a part from a variable that has been given an enumerated type. If you do, the overall behavior of your design changes when FPGA Compiler II / FPGA *Express* changes the original encoding. Example 8-12 shows an unsupported bit-select.

Example 8-12 Unsupported Bit-Select From Enumerated Type

```

parameter [2:0] /* synopsys enum states */
    s0 = 3'd0, s1 = 3'd1, s2 = 3'd2, s3 = 3'd3,
    s4 = 3'd4, s5 = 3'd5, s6 = 3'd6, s7 = 3'd7;
reg [2:0] /* synopsys enum states */ state, next_state;

assign high_bit = state[2]; // not supported

```

Because you cannot access individual bits of an enumerated type, you cannot use component instantiation to hook up single-bit flip-flops or three-states. Example 8-13 shows an example of this type of unsupported bit-select.

Example 8-13 Unsupported Bit-Select (With Component Instantiation) From Enumerated Type

```

DFF ff0 ( next_state[0], clk, state[0] );
DFF ff1 ( next_state[1], clk, state[1] );
DFF ff2 ( next_state[2], clk, state[2] );

```

To create flip-flops and three-states for enum values, you must imply them with the posedge construct or the literal z, as shown in Example 8-14.

Example 8-14 Using Inference With Enumerated Types

```
parameter [2:0] /* synopsys enum states */
    s0 = 3'd0, s1 = 3'd1, s2 = 3'd2, s3 = 3'd3,
    s4 = 3'd4, s5 = 3'd5, s6 = 3'd6, s7 = 3'd7;
reg [2:0] /* synopsys enum states */ state, next_state;

parameter [1:0] /* synopsys enum outputs */
    DONE = 2'd0, PROCESSING = 2'd1, IDLE = 2'd2;
reg [1:0] /* synopsys enum outputs */ out, triout;

always @ (posedge clk) state = next_state;
assign triout = trienable ? out : 'bz;
```

If you use the constructs shown in Example 8-14, you can change the enumeration encodings by changing the parameter and reg declarations, as shown in Example 8-15. You can also allow FPGA Compiler II / *FPGA Express* to change the encodings.

Example 8-15 Changing the Enumeration Encoding

```
parameter [3:0] /* synopsys enum states */
    s0 = 4'd0, s1 = 4'd10, s2 = 4'd15, s3 = 4'd5,
    s4 = 4'd2, s5 = 4'd4, s6 = 4'd6, s7 = 4'd8;
reg [3:0] /* synopsys enum states */ state, next_state;

parameter [1:0] /* synopsys enum outputs */
    DONE = 2'd3, PROCESSING = 2'd1, IDLE = 2'd0;
reg [1:0] /* synopsys enum outputs */ out, triout;

always @ (posedge clk) state = next_state;
assign triout = trienable ? out : 'bz;
```

If you must select individual bits of an enumerated type, you can declare a temporary variable of the same size as the enumerated type. Assign the enumerated type to the variable, then select individual bits of the temporary variable. Example 8-16 shows how this is done.

Example 8-16 Supported Bit-Select From Enumerated Type

```
parameter [2:0] /* synopsys enum states */
    s0 = 3'd0, s1 = 3'd1, s2 = 3'd2, s3 = 3'd3,
    s4 = 3'd4, s5 = 3'd5, s6 = 3'd6, s7 = 3'd7;
reg [2:0] /* synopsys enum states */ state, next_state;
wire [2:0] temporary;

assign temporary = state;
assign high_bit = temporary[2]; //supported
```

Note:

Selecting individual bits from an enumerated type is not recommended.

If you declare a port as a reg and as an enumerated type, you must declare the enumeration when you declare the port. Example 8-17 shows the declaration of the enumeration.

Example 8-17 Enumerated Type Declaration for a Port

```
module good_example (a,b);

parameter [1:0] /* synopsys enum colors */
    green = 2'b00, white = 2'b11;
input a;
output [1:0] /* synopsys enum colors */ b;
reg [1:0] b;
.
.
endmodule
```

Example 8-18 shows the wrong way to declare a port as an enumerated type, because the enumerated type declaration appears with the reg declaration instead of with the output port declaration. This code does not export enumeration information to FPGA Compiler II / FPGA Express.

Example 8-18 *Incorrect Enumerated Type Declaration for a Port*

```
module bad_example (a,b);  
  
parameter [1:0] /* synopsys enum colors */  
    green = 2'b00, white = 2'b11;  
input a;  
output [1:0] b;  
reg [1:0] /* synopsys enum colors */ b;  
.  
.  
endmodule
```

Component Implication

In Verilog, you cannot instantiate modules in behavioral code. To include an embedded netlist in your behavioral code, use the directives `// synopsys map_to_module` and `// synopsys return_port_name` for FPGA Compiler II / FPGA Express to recognize the netlist as a function being implemented by another module. When this subprogram is invoked in the behavioral code, FPGA Compiler II / FPGA Express instantiates the module (see Example 8-19).

The first directive, `// synopsys map_to_module`, flags a function for implementation as a distinct component. The syntax is

```
// synopsys map_to_module modulename
```


The second directive identifies a return port (functions in Verilog do not have output ports). To instantiate the function as a component, the return port must have a name. The syntax is

```
// synopsys return_port_name portname
```

Note:

Remember that if you add a `map_to_module` directive to a function, the contents of the function are parsed and ignored whereas the indicated module is instantiated. Ensure that the functionality of the module instantiated in this way and the function it replaces are the same; otherwise, pre-synthesis and post-synthesis simulation do not match.

Example 8-19 illustrates the `map_to_module` and `return_port_name` directives.

Example 8-19 Component Implication

```
module mux_inst (a, b, c, d, e);
input a, b, c, d;
output e;

    function mux_func;
// synopsys map_to_module mux_module
// synopsys return_port_name mux_ret
input in1, in2, cntrl;
    /*
    ** the contents of this function are ignored for
    ** synthesis, but the behavior of this function
    ** must match the behavior of mux_module for
    ** simulation purposes
    */
    begin
    if (cntrl) mux_func = in1;
    else mux_func = in2;
    end

    endfunction

assign e = a & mux_func (b, c, d);
// this function call actually instantiates component (module) mux_module
```

```
endmodule

module mux_module (in1, in2, cntrl, mux_ret);
input in1, in2, cntrl;
output mux_ret;

and and2_0 (wire1, in1, cntrl);
not not1 (not_cntrl, cntrl);
and and2_1 (wire2, in2, not_cntrl);
or or2 (mux_ret, wire1, wire2);

endmodule
```

A

Examples

This appendix presents examples that demonstrate basic concepts of Synopsys FPGA Compiler II / *FPGA Express*:

- Count Zeros—Combinational Version
- Count Zeros—Sequential Version
- Drink Machine—State Machine Version
- Drink Machine—Count Nickels Version
- Carry-Lookahead Adder

Count Zeros—Combinational Version

Using this circuit is one possible solution to a design problem. Given an 8-bit value, the circuit must determine two things:

- The presence of a value containing exactly one sequence of zeros
- The number of zeros in the sequence (if any)

The circuit must complete this computation in a single clock cycle. The input to the circuit is an 8-bit value, and the two outputs the circuit produces are the number of zeros found and an error indication.

A valid value contains only one series of zeros. If more than one series of zeros appears, the value is invalid. A value consisting of all ones is a valid value. If a value is invalid, the count of zeros is set to zero. For example,

- The value 00000000 is valid, and the count is eight zeros.
- The value 11000111 is valid, and the count is three zeros.
- The value 00111110 is invalid.

A Verilog description and a schematic of the circuit are shown in Example A-1 and Figure A-1.

Example A-1 Count Zeros—Combinational Version

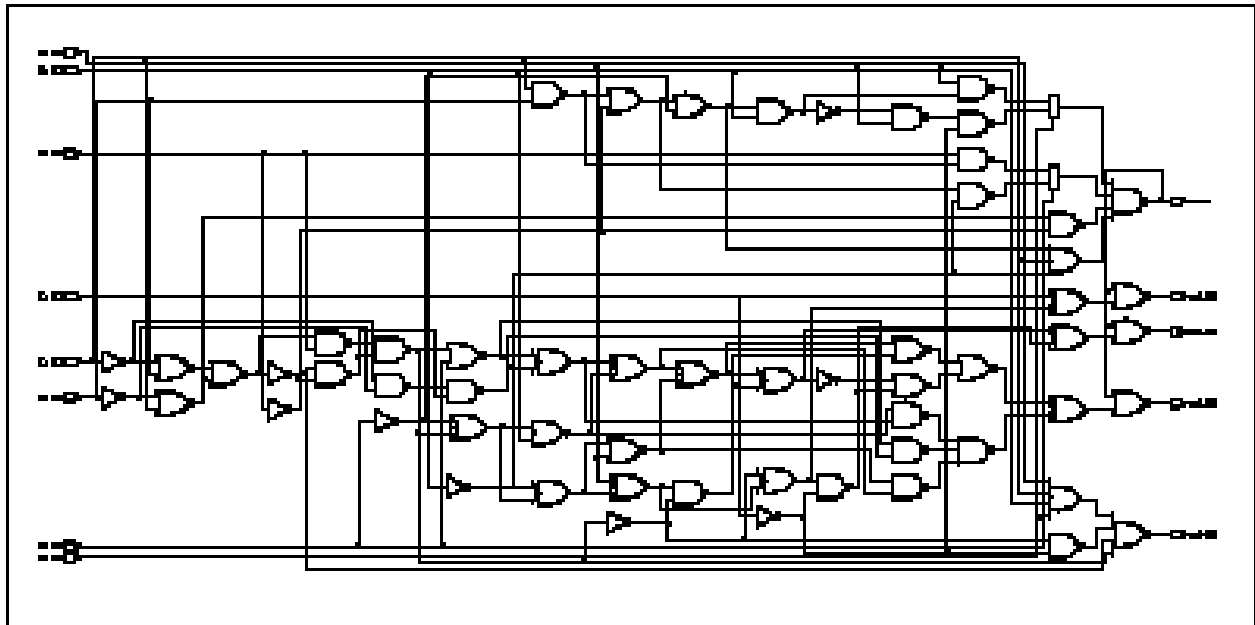
```
module count_zeros(in, out, error);
    input  [7:0] in;
    output [3:0] out;
    output error;
    function legal;
    input [7:0] x;
    reg seenZero, seenTrailing;
    integer i;
    begin : _legal_block
        legal = 1; seenZero = 0; seenTrailing = 0;
        for ( i=0; i <= 7; i=i+1 )
            if ( seenTrailing && (x[i] == 1'b0) ) begin
                legal = 0;
                disable _legal_block;
            end
            else if ( seenZero && (x[i] == 1'b1) )
                seenTrailing = 1;
            else if ( x[i] == 1'b0 )
                seenZero = 1;
        end
    endfunction

    function [3:0] zeros;
    input [7:0] x;
    reg [3:0] count;
    integer i;

    begin
        count = 0;
        for ( i=0; i <= 7; i=i+1 )
            if ( x[i] == 1'b0 ) count = count + 1;
        zeros = count;
    end
    endfunction

    wire is_legal = legal(in);
    assign error = ! is_legal;
    assign out    = is_legal ? zeros(in) : 1'b0;
endmodule
```

Figure A-1 Count Zeros—Combinational Version Block Diagram



This example shows two Verilog functions: `legal` and `zeros`. The function `legal` determines if the value is valid. It returns a 1-bit value: either 1 for a valid value or 0 for an invalid value. The function `zeros` cycles through all bits of the value, counts the number of zeros, and returns the appropriate value. The two functions are controlled by continuous assignment statements at the bottom of the module definition. This example shows a combinational (parallel) approach to counting zeros; the next example shows a sequential (serial) approach.

Count Zeros—Sequential Version

Example A-2 and Figure A-2 show a sequential (clocked) solution to the “count zeros” design problem. The circuit specification is slightly different from the specification in the combinational solution. The circuit now accepts the 8-bit string serially, 1 bit per clock cycle, using the data and clk inputs. The other two inputs are

- reset, which resets the circuit
- read, which causes the circuit to begin accepting data

The circuit’s three outputs are

- is_legal, which is true if the data is a valid value
- data_ready, which is true at the first invalid bit or when all 8 bits have been processed
- zeros, which is the number of zeros if is_legal is true

Example A-2 Count Zeros—Sequential Version

```
module count_zeros(data,reset,read,clk,zeros,is_legal,
                  data_ready);

    parameter TRUE=1, FALSE=0;

    input  data, reset, read, clk;
    output is_legal, data_ready;
    output [3:0] zeros;
    reg    [3:0] zeros;

    reg is_legal, data_ready;
    reg seenZero, new_seenZero;
    reg seenTrailing, new_seenTrailing;
    reg new_is_legal;
    reg new_data_ready;
    reg [3:0] new_zeros;
    reg [2:0] bits_seen, new_bits_seen;
```

```

always @ ( data or reset or read or is_legal
or data_ready or seenTrailing or
seenZero or zeros or bits_seen ) begin
    if ( reset ) begin
        new_data_ready    = FALSE;
        new_is_legal      = TRUE;
        new_seenZero      = FALSE;
        new_seenTrailing  = FALSE;
        new_zeros         = 0;
        new_bits_seen     = 0;
    end
    else begin
        new_is_legal      = is_legal;
        new_seenZero      = seenZero;
        new_seenTrailing  = seenTrailing;
        new_zeros         = zeros;
        new_bits_seen     = bits_seen;
        new_data_ready    = data_ready;
        if ( read ) begin
            if ( seenTrailing && (data == 0) )
                begin
                    new_is_legal    = FALSE;
                    new_zeros       = 0;
                    new_data_ready  = TRUE;
                end
            else if ( seenZero && (data == 1'b1) )
                new_seenTrailing = TRUE;
            else if ( data == 1'b0 ) begin
                new_seenZero = TRUE;
                new_zeros = zeros + 1;
            end

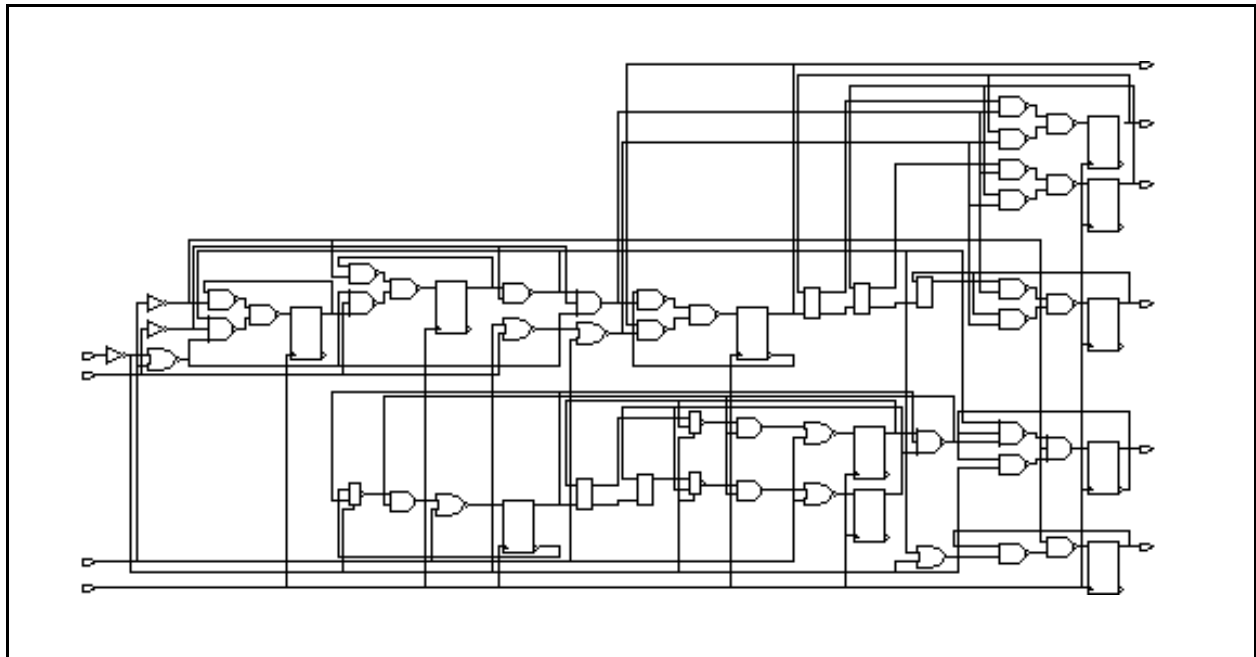
            if ( bits_seen == 7 )
                new_data_ready = TRUE;
            else
                new_bits_seen = bits_seen+1;
        end
    end
end

always @ ( posedge clk) begin
    zeros = new_zeros;
    bits_seen = new_bits_seen;
    seenZero = new_seenZero;
    seenTrailing = new_seenTrailing;
    is_legal = new_is_legal;
    data_ready = new_data_ready;
end
endmodule

```

Examples

Figure A-2 Count Zeros—Sequential Version Block Diagram



Drink Machine—State Machine Version

The next design is a vending control unit for a soft drink vending machine. The circuit reads signals from a coin-input unit and sends outputs to a change-dispensing unit and a drink-dispensing unit.

Input signals from the coin-input unit are `nickel_in` (nickel deposited), `dime_in` (dime deposited), and `quarter_in` (quarter deposited).

Outputs to the vending control unit are `collect` (collect coins), to the coin-input unit; `nickel_out` (nickel change) and `dime_out` (dime change), to the change-dispensing unit; and `dispense` (dispense drink), to the drink-dispensing unit.

The price of a drink is 35 cents. The Verilog description for this design,

shown in Example A-3, uses a state machine description style. The description includes the `state_vector` directive, which enables FPGA Compiler II / *FPGA Express* to extract an equivalent state machine.

Example A-3 Drink Machine—State Machine Version

```
`define vend_a_drink {D,dispense,collect} = {IDLE,2'b11}

module drink_machine(nickel_in, dime_in, quarter_in,
                    collect, nickel_out, dime_out,
                    dispense, reset, clk) ;
    parameter IDLE=0,FIVE=1,TEN=2,TWENTY_FIVE=3,
              FIFTEEN=4,THIRTY=5,TWENTY=6,OWE_DIME=7;

    input  nickel_in, dime_in, quarter_in, reset, clk;
    output collect, nickel_out, dime_out, dispense;

    reg collect, nickel_out, dime_out, dispense;
    reg [2:0] D, Q; /* state */
// synopsys state_vector Q
always @ ( nickel_in or dime_in or quarter_in or reset )
    begin
        nickel_out = 0;
        dime_out   = 0;
        dispense   = 0;
        collect    = 0;

        if ( reset ) D = IDLE;
        else begin
            D = Q;

            case ( Q )
            IDLE:
                if (nickel_in)      D = FIVE;
                else if (dime_in)   D = TEN;
                else if (quarter_in) D = TWENTY_FIVE;
            FIVE:
                if(nickel_in)      D = TEN;
                else if (dime_in)   D = FIFTEEN;
                else if (quarter_in) D = THIRTY;
            TEN:
                if (nickel_in)      D = FIFTEEN;
                else if (dime_in)   D = TWENTY;
                else if (quarter_in) `vend_a_drink;
            TWENTY_FIVE:
                if( nickel_in)      D = THIRTY;
                else if (dime_in)   `vend_a_drink;
            endcase
        end
    end
endmodule
```

```

        else if (quarter_in) begin

            `vend_a_drink;
            nickel_out = 1;
            dime_out = 1;
        end

    FIFTEEN:
        if (nickel_in)          D = TWENTY;
        else if (dime_in)      D = TWENTY_FIVE;
        else if (quarter_in) begin
            `vend_a_drink;
            nickel_out = 1;
        end

    THIRTY:
        if (nickel_in)          `vend_a_drink;
        else if (dime_in)      begin
            `vend_a_drink;
            nickel_out = 1;
        end
        else if (quarter_in) begin
            `vend_a_drink;
            dime_out = 1;
            D = OWE_DIME;
        end

    TWENTY:
        if (nickel_in)          D = TWENTY_FIVE;
        else if (dime_in)      D = THIRTY;
        else if (quarter_in) begin
            `vend_a_drink;
            dime_out = 1;
        end

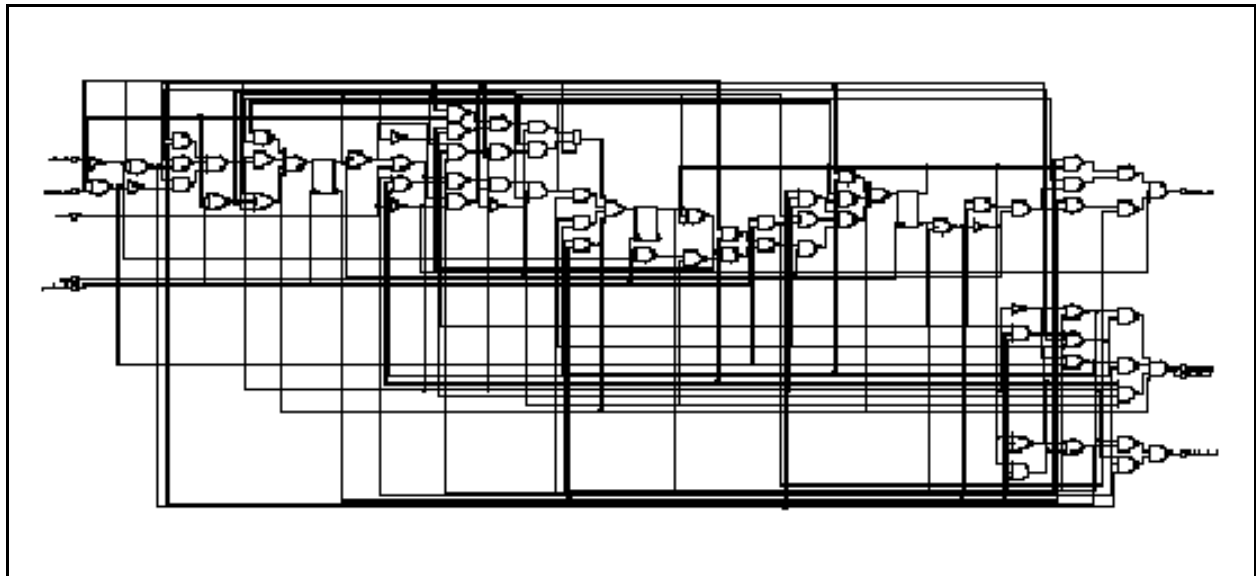
    OWE_DIME:
        begin
            dime_out = 1;
            D = IDLE;
        end
    endcase
end

end

always @ (posedge clk ) begin
    Q = D;
end
endmodule

```

Figure A-3 Drink Machine—State Machine Version Block Diagram



Drink Machine—Count Nickels Version

Example A-4 uses the same design parameters as Example A-4 with the same input and output signals. In this version, a counter counts the number of nickels deposited. This counter is incremented by one if the deposit is a nickel, by two if it's a dime, and by five if it's a quarter.

Example A-4 Drink Machine—Count Nickels Version

```
module drink_machine(nickel_in,dime_in,quarter_in,collect,
    nickel_out,dime_out,dispense,reset,clk);

    input  nickel_in, dime_in, quarter_in, reset, clk;
    output nickel_out, dime_out, collect, dispense;

    reg nickel_out, dime_out, dispense, collect;
    reg [3:0] nickel_count, temp_nickel_count;
    reg temp_return_change, return_change;
```

```

always @ ( nickel_in or dime_in or quarter_in or
collect or temp_nickel_count or
reset or nickel_count or return_change) begin
    nickel_out = 0;
    dime_out    = 0;
    dispense    = 0;
    collect     = 0;
    temp_nickel_count = 0;
    temp_return_change = 0;

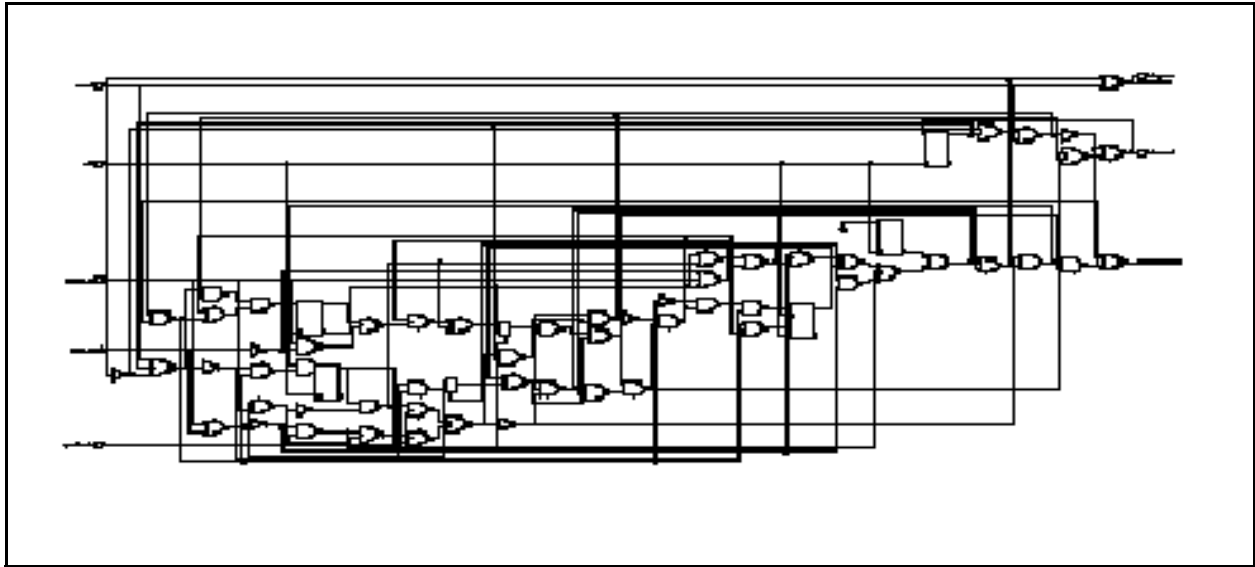
    // Check whether money has come in
    if (! reset) begin
        temp_nickel_count = nickel_count;
        if (nickel_in)
            temp_nickel_count = temp_nickel_count + 1;
        else if (dime_in)
            temp_nickel_count = temp_nickel_count + 2;
        else if (quarter_in)
            temp_nickel_count = temp_nickel_count + 5;

        // correct amount deposited?
        if (temp_nickel_count >= 7) begin
            temp_nickel_count = temp_nickel_count - 7;
            dispense = 1;
            collect = 1;
        end
        // return change
        if (return_change || collect) begin
            if (temp_nickel_count >= 2) begin
                dime_out = 1;
                temp_nickel_count = temp_nickel_count - 2;
                temp_return_change = 1;
            end

            if (temp_nickel_count == 1) begin
                nickel_out = 1;
                temp_nickel_count = temp_nickel_count - 1;
            end
        end
    end
end
end
always @ (posedge clk ) begin
    nickel_count = temp_nickel_count;
    return_change = temp_return_change;
end
endmodule

```

Figure A-4 Drink Machine—Count Nickels Version Block Diagram



Carry-Lookahead Adder

Figure A-5 and Example A-5 show how to build a 32-bit carry-lookahead adder. The adder is built by partitioning of the 32-bit input into eight slices of 4 bits each. The PG module computes propagate and generate values for each of the eight slices.

Propagate (output P from PG) is 1 for a bit position if that position propagates a carry from the next-lower position to the next-higher position. Generate (output G) is 1 for a bit position if that position generates a carry to the next-higher position, regardless of the carry-in from the next-lower position.

The carry-lookahead logic reads the carry-in, propagate, and generate information computed from the inputs. It computes the carry value for each bit position. This logic makes the addition operation an XOR of the inputs and the carry values.

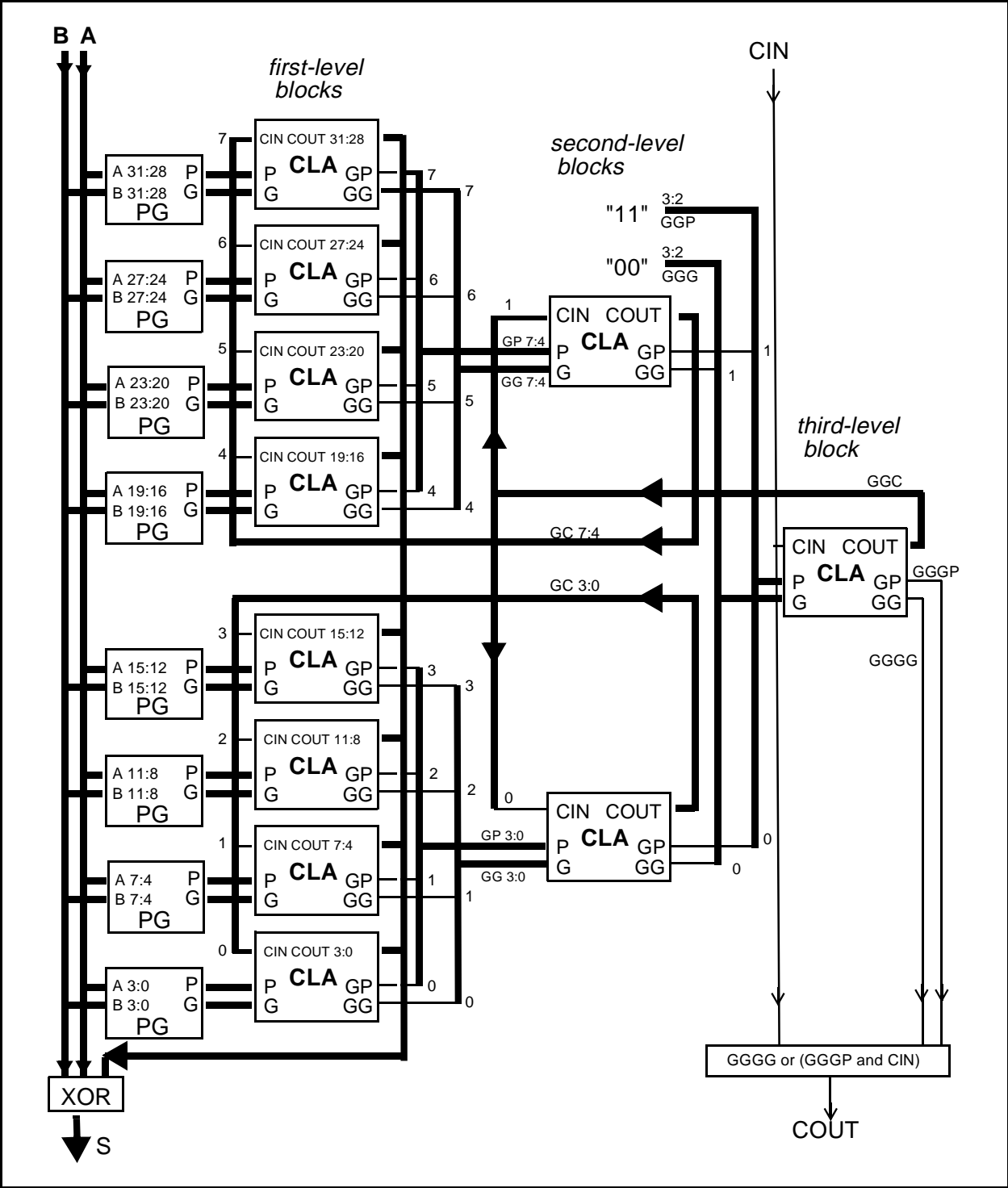
The following list shows the order in which the carry values are computed by a three-level tree of 4-bit carry-lookahead blocks (illustrated in Figure A-4):

1. The first level of the tree computes the 32 carry values and the 8 group propagate and generate values. Each of the first-level group propagate and generate values tells if that 4-bit slice propagates and generates carry values from the next-lower group to the next-higher. The first-level lookahead blocks read the group carry computed at the second level.
2. At the second level of the tree, the lookahead blocks read the group propagate and generate information from the four first-level blocks and then compute their own group propagate and generate information. They also read group carry information computed at the third level to compute the carries for each of the third-level blocks.
3. At the third level of the tree, the third-level block reads the propagate and generate information of the second level to compute a propagate and generate value for the entire adder. It also reads the external carry to compute each second-level carry. The carry-out for the adder is 1 if the third-level generate is 1 or if the third-level propagate is 1 and the external carry is 1.

The third-level carry-lookahead block can process four second-level blocks. Because there are only two second-level blocks in Figure A-4, the high-order 2 bits of the computed carry are ignored, the high-order 2 bits of the generate input to the third-level are set to 00 (zero), and the propagate high-order bits are set to 11. This causes the unused portion to propagate carries but not to generate them.

Figure A-5 shows the three levels of a block diagram of the 32-bit carry-lookahead adder. Example A-5 shows the code for the adder.

Figure A-5 Carry-Lookahead Adder Block Diagram



Examples

Example A-5 Carry-Lookahead Adder

```
`define word_size 32
`define word [`word_size-1:0]

`define n 4
`define slice [`n-1:0]

`define s0 (1*`n)-1:0*`n
`define s1 (2*`n)-1:1*`n
`define s2 (3*`n)-1:2*`n
`define s3 (4*`n)-1:3*`n
`define s4 (5*`n)-1:4*`n
`define s5 (6*`n)-1:5*`n
`define s6 (7*`n)-1:6*`n
`define s7 (8*`n)-1:7*`n

module cla32_4(a, b, cin, s, cout);
input `word a, b;
input cin;
output `word s;
output cout;

    wire [7:0] gg, gp, gc; // Group generate, propagate,
                          // carry
    wire [3:0] ggg, ggp, ggc; // Second-level gen., prop.
    wire gggg, gggp; // Third-level gen., prop.

    bitslice i0(a[`s0], b[`s0], gc[0], s[`s0], gp[0], gg[0]);
    bitslice i1(a[`s1], b[`s1], gc[1], s[`s1], gp[1], gg[1]);
    bitslice i2(a[`s2], b[`s2], gc[2], s[`s2], gp[2], gg[2]);
    bitslice i3(a[`s3], b[`s3], gc[3], s[`s3], gp[3], gg[3]);

    bitslice i4(a[`s4], b[`s4], gc[4], s[`s4], gp[4], gg[4]);
    bitslice i5(a[`s5], b[`s5], gc[5], s[`s5], gp[5], gg[5]);
    bitslice i6(a[`s6], b[`s6], gc[6], s[`s6], gp[6], gg[6]);
    bitslice i7(a[`s7], b[`s7], gc[7], s[`s7], gp[7], gg[7]);

    cla c0(gp[3:0], gg[3:0], ggc[0], gc[3:0], ggp[0], ggg[0]);
    cla c1(gp[7:4], gg[7:4], ggc[1], gc[7:4], ggp[1], ggg[1]);

    assign ggp[3:2] = 2'b11;
    assign ggg[3:2] = 2'b00;
    cla c2(ggp, ggg, cin, ggc, gggp, gggg);
    assign cout = gggg | (gggp & cin);
endmodule

// Compute sum and group outputs from a, b, cin
```

```

module bitslice(a, b, cin, s, gp, gg);
input `slice a, b;
input cin;
output `slice s;
output gp, gg;

    wire `slice p, g, c;
    pg i1(a, b, p, g);
    cla i2(p, g, cin, c, gp, gg);
    sum i3(a, b, c, s);
endmodule

// compute propagate and generate from input bits

module pg(a, b, p, g);
input `slice a, b;
output `slice p, g;

    assign p = a | b;
    assign g = a & b;
endmodule

// compute sum from the input bits and the carries

module sum(a, b, c, s);
input `slice a, b, c;
output `slice s;

    wire `slice t = a ^ b;
    assign s = t ^ c;
endmodule

// n-bit carry-lookahead block

module cla(p, g, cin, c, gp, gg);
input `slice p, g; // propagate and generate bits
input cin; // carry in
output `slice c; // carry produced for each bit
output gp, gg; // group generate and group propagate

    function [99:0] do_cla;
    input `slice p, g;
    input cin;

    begin : label
    integer i;
    reg gp, gg;
    reg `slice c;

```

Examples

```
gp = p[0];
gg = g[0];
c[0] = cin;
for(i = 1; i < 'n'; i = i+1) begin
    gp = gp & p[i];
    gg = (gg & p[i]) | g[i];
    c[i] = (c[i-1] & p[i-1]) | g[i-1];
end
do_cla = {c, gp, gg};
end
endfunction

assign {c, gp, gg} = do_cla(p, g, cin);
endmodule
```

Examples

A-18

B

Verilog Syntax

This appendix contains a syntax description of the Verilog language as supported by Synopsys FPGA Compiler II / FPGA *Express*. It covers the following topics:

- Syntax
- Lexical Conventions
- Verilog Keywords
- Unsupported Verilog Language Constructs

Syntax

This section presents the syntax of the supported Verilog language in Backus-Naur form (BNF) and the syntax formalism.

Note:

The BNF syntax convention used in this section differs from the Synopsys syntax convention used elsewhere in this manual.

BNF Syntax Formalism

White space separates lexical tokens.

name

is a keyword.

<name>

is a syntax construct definition.

<name>

is a syntax construct item.

<name>?

is an optional item.

<name>*

is zero, one, or more items.

<name>+

is one or more items.

<port> <,<port>>*

is a comma-separated list of items.

::=

gives a syntax definition to an item.

||=

refers to an alternative syntax construct.

BNF Syntax

```
<source_text>
  ::= <description>*

<description>
  ::= <module>

<module>
  ::= module <name_of_module> <list_of_ports>? ;
         <module_item>*
         endmodule

<name_of_module>
  ::= <IDENTIFIER>

<list_of_ports>
  ::= ( <port> <,<port>>* )
  ||= ( )

<port>
  ::= <port_expression>?
  ||= . <name_of_port> ( <port_expression>? )

<port_expression>
  ::= <port_reference>
  ||= { <port_reference> <,<port_reference>>* }

<port_reference>
  ::= <name_of_variable>
  ||= <name_of_variable> [ <expression> ]
  ||= <name_of_variable> [ <expression> : <expression> ]

<name_of_port>
  ::= <IDENTIFIER>

<name_of_variable>
  ::= <IDENTIFIER>
```

```

<module_item>
 ::= <parameter_declaration>
  || = <input_declaration>
  || = <output_declaration>
  || = <inout_declaration>
  || = <net_declaration>
  || = <reg_declaration>
  || = <integer_declaration>
  || = <gate_instantiation>
  || = <module_instantiation>
  || = <continuous_assign>
  || = <function>

<function>
 ::= function <range>? <name_of_function> ;
      <func_declaration>*
      <statement_or_null>
      endfunction

<name_of_function>
 ::= <IDENTIFIER>

<func_declaration>
 ::= <parameter_declaration>
  || = <input_declaration>
  || = <reg_declaration>
  || = <integer_declaration>

<always>
 ::= always @ ( <identifier> or <identifier> )
  || = always @ ( posedge <identifier> )
  || = always @ ( negedge <identifier> )
  || = always @ ( <edge> or <edge> or ... )

<edge>
 ::= posedge <identifier>
  || = negedge <identifier>

<parameter_declaration>
 ::= parameter <range>? <list_of_assignments> ;

```



```

<input_declaration>
    ::= input <range>? <list_of_variables> ;

<output_declaration>
    ::= output <range>? <list_of_variables> ;

<inout_declaration>
    ::= inout <range>? <list_of_variables> ;

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

<NETTYPE>
    ::= wire
    ||= wor
    ||= wand
    ||= tri

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

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

<integer_declaration>
    ::= integer <list_of_integer_variables> ;

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

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

<name_of_variable>
    ::= <IDENTIFIER>

```

```

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

<register_variable>
    ::= <IDENTIFIER>

<list_of_integer_variables>
    ::= <integer_variable> <, <integer_variable>> *

<integer_variable>
    ::= <IDENTIFIER>

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

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

<STRENGTH0>
    ::= supply0
       || = strong0
       || = pull0
       || = weak0
       || = highz0

<STRENGTH1>
    ::= supply1
       || = strong1
       || = pull1
       || = weak1
       || = highz1

<range>
    ::= [ <expression> : <expression> ]

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

```

```

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

<GATETYPE>
    ::= and
    | = nand
    | = or
    | = nor
    | = xor
    | = xnor
    | = buf
    | = not

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

<name_of_gate_instance>
    ::= <IDENTIFIER>

<terminal>
    ::= <identifier>
    | = <expression>

<module_instantiation>
    ::= <name_of_module> <parameter_value_assignment>?
        <module_instance> < , <module_instance>>* ;

<name_of_module>
    ::= <IDENTIFIER>

<parameter_value_assignment>
    ::= #( <expression> < , <expression>>* )

<module_instance>
    ::= <name_of_module_instance>
        ( <list_of_module_terminals>? )

<name_of_module_instance>
    ::= <IDENTIFIER>

```

```

<list_of_module_terminals>
  ::= <module_terminal>? <,<module_terminal>>*
  ||= <named_port_connection> <,<named_port_connection>>*

<module_terminal>
  ::= <identifier>
  ||= <expression>

<named_port_connection>
  ::= . IDENTIFIER ( <identifier> )
  ||= . IDENTIFIER ( <expression> )

<statement>
  ::= <assignment>
  ||= if ( <expression> )
      <statement_or_null>
  ||= if ( <expression> )
      <statement_or_null>
      else
      <statement_or_null>
  ||= case ( <expression> )
      <case_item>+
      endcase
  ||= casex ( <expression> )
      <case_item>+
      endcase
  ||= casez ( <expression> )
      <case_item>+
      endcase
  ||= for ( <assignment> ; <expression> ; <assignment> )
      <statement>
  ||= <seq_block>
  ||= disable <IDENTIFIER> ;
  ||= forever <statement>
  ||= while ( <expression> ) <statement>

<statement_or_null>
  ::= statement
  ||= ;

<assignment>
  ::= <lvalue> = <expression>

```

```

<case_item>
    ::= <expression> <,<expression>>* :
<statement_or_null>
    ||= default : <statement_or_null>
    ||= default <statement_or_null>

<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>

<lvalue>
    ::= <IDENTIFIER>
    ||= <IDENTIFIER> [ <expression> ]
    ||= <concatenation>

<expression>
    ::= <primary>
    ||= <UNARY_OPERATOR> <primary>
    ||= <expression> <BINARY_OPERATOR>
    ||= <expression> ? <expression> : <expression>

<UNARY_OPERATOR>
    ::= !
    ||= ~
    ||= &
    ||= ~&
    ||= |
    ||= ~|
    ||= ^

```

```
|| = ~^
|| = -
|| = +
```

<BINARY_OPERATOR>

```
::= +
|| = -
|| = *
|| = /
|| = %
|| = ==
|| = !=
|| = &&
|| = ||
|| = <
|| = <=
|| = >
|| = >=
|| = &
|| = |
|| = <<
|| = >>
```

<primary>

```
::= <number>
|| = <identifier>
|| = <identifier> [ <expression> ]
|| = <identifier> [ <expression> : <expression> ]
|| = <concatenation>
|| = <multiple_concatenation>
|| = <function_call>
|| = ( <expression> )
```

<number>

```
::= <NUMBER>
|| = <BASE> <NUMBER>
|| = <SIZE> <BASE> <NUMBER>
```

<NUMBER>

A number can have any of these characters:
0123456789abcdefxzABCDEFXZ.

```
<SIZE>
 ::= 'b
    | 'B
    | 'o
    | 'O
    | 'd
    | 'D
    | 'h
    | 'H
```

```
<SIZE>
```

A size can have any number of these digits: 0123456789

```
<concatenation>
 ::= { <expression> <,<expression>>* }

<multiple_concatenation>
 ::= { <expression> { <expression> <,<expression>>* } }

<function_call>
 ::= <name_of_function> ( <expression> <,<expression>>* )

<name_of_function>
 ::= <IDENTIFIER>

<identifier>
```

An identifier is any sequence of letters, digits, and the underscore character (_), where the first character is a letter or an underscore. Uppercase and lowercase letters are treated as different characters. Identifiers can be any size, and all characters are significant. Escaped identifiers start with the backslash character (\) and end with a space.

The leading backslash character (\) is not part of the identifier. Use escaped identifiers to include any printable ASCII characters in an identifier.

```
<delay>
 ::= # <NUMBER>
 || = # <identifier>
 || = # ( <expression> <,<expression>>* )
```

Lexical Conventions

The lexical conventions FPGA Compiler II / *FPGA Express* uses are nearly identical to those of the Verilog language. The types of lexical tokens FPGA Compiler II / *FPGA Express* uses are described in the following subsections:

- White Space
- Comments
- Numbers
- Identifiers
- Operators
- Macro Substitution
- include Construct
- Simulation Directives
- Verilog System Functions

White Space

White space separates words in the input description and can contain spaces, tabs, new lines, and form feeds. You can place white space anywhere in the description. FPGA Compiler II / FPGA *Express* ignores white space.

Comments

You can enter comments anywhere in a Verilog description, in two forms:

- Beginning with two slashes `//`
FPGA Compiler II / FPGA *Express* ignores all text between these characters and the end of the current line.
- Beginning with the two characters `/*` and ending with `*/`
FPGA Compiler II / FPGA *Express* ignores all text between these characters, so you can continue comments over more than one line.

Note:

You cannot nest comments.

Numbers

You can declare numbers in several different radices and bit-widths. A radix is the base number on which a numbering system is built. For example, the binary numbering system has a radix of 2, octal has a radix of 8, and decimal has a radix of 10.

You can use these three number formats:

- A simple decimal number that is a sequence of digits in the range of 0 to 9. All constants declared this way are assumed to be 32-bit numbers.
- A number that specifies the bit-width as well as the radix. These numbers are the same as those in the previous format, except that they are preceded by a decimal number that specifies the bit-width.
- A number followed by a two-character sequence prefix that specifies the number's size and radix. The radix determines which symbols you can include in the number. Constants declared this way are assumed to be 32-bit numbers. Any of these numbers can include underscores (`_`), which improve readability and do not affect the value of the number. Table B-1 summarizes the available radices and valid characters for the number.

Table B-1 Verilog Radices

Name	Character Prefix	Valid Characters
Binary	'b	0 1 x X z Z _ ?
Octal	'o	0-7 x X z Z _ ?
Decimal	'd	0-9 _
Hexadecimal	'h	0-9 a-f A-F x X z Z _ ?

Example B-1 shows some valid number declarations.

Example B-1 Valid Verilog Number Declarations

```
391           // 32-bit decimal number
'h3a13       // 32-bit hexadecimal number
10'o1567     // 10-bit octal number
3'b010      // 3-bit binary number
4'd9        // 4-bit decimal number
40'hFF_FFFF_FFFF // 40-bit hexadecimal number
2'bxx       // 2-bits don't care
```

```
3'bzzz           // 3-bits high-impedance
```

Identifiers

Identifiers are user-defined words for variables, function names, module names, and instance names. Identifiers can be composed of letters, digits, and the underscore character (`_`). The first character of an identifier cannot be a number. Identifiers can be any length. Identifiers are case-sensitive, and all characters are significant.

Identifiers that contain special characters, begin with numbers, or have the same name as a keyword can be specified as an escaped identifier. An escaped identifier starts with the backslash character (`\`), followed by a sequence of characters, followed by white space.

Some escaped identifiers are shown in Example B-2.

Example B-2 Sample Escaped Identifiers

```
\a+b           \3state  
\module       \ (a&b) |c
```

The Verilog language supports the concept of hierarchical names, which can be used to access variables of submodules directly from a higher-level module. These are partially supported by FPGA Compiler II / FPGA *Express*. (For more information, see “Unsupported Verilog Language Constructs” on page B-20.)

Operators

Operators are one- or two-character sequences that perform operations on variables. Some examples of operators are `+`, `~^`, `<=`, and `>>`. Operators are described in detail in “Operators” on page 4-3.

Macro Substitution

Macro substitution assigns a string of text to a macro variable. The string of text is inserted into the code where the macro is encountered. The definition begins with the back quotation mark (`'`), followed by the keyword *define*, followed by the name of the macro variable. All text from the macro variable until the end of the line is assigned to the macro variable.

You can declare and use macro variables anywhere in the description. The definitions can carry across several files that are read into FPGA Compiler II / *FPGA Express* at the same time. To make a macro substitution, type a back quotation mark (`'`) followed by the macro variable name.

Some sample macro variable declarations are shown in Example B-3.

Example B-3 Macro Variable Declarations

```
`define highbits      31:29
`define bitlist       {first, second, third}
wire [31:0] bus;
`bitlist = bus[`highbits];
```

Text macros are not supported when used with sized constants, as shown in Example B-4.

Example B-4 Macro With Sized Constants

```
`define SIZE 4

module test (in,out);
output [3:0] out;
input [3:0] in;

assign out = `SIZE'b0101; //text macro from `define statement
                        //cannot be used with a sized constant

endmodule
```

include Construct

The include construct in Verilog is similar to the #include directive in C. You can use this construct to include Verilog code, such as type declarations and functions, from one module in another module. Example B-5 shows an application of the include construct.

Example B-5 Including a File Within a File

Contents of file1.v

```
`define WORDSIZE 8
function [WORDSIZE-1:0] fastadder;
.
.
endfunction
```

Contents of secondfile

```
module secondfile (in1,in2,out)
`include file1.v
wire [WORDSIZE-1:0] temp;
assign temp = fastadder (in1,in2);
.
.
endmodule
```

Included files can include other files, with up to 24 levels of nesting. You cannot use the include construct recursively.

Simulation Directives

Simulation directives refer to special commands that affect the operation of the Verilog HDL Simulator. You can include these directives in your design description, because FPGA Compiler II / FPGA *Express* parses and ignores them:

```
`accelerate
`celldefine
`default_nettype
`endcelldefine
`endprotect
`expand_vectornets
`noaccelerate
`noexpand_vectornets
`noremove_netnames
`nounconnected_drive
`protect
`remove_netnames
`resetall
`timescale
`unconnected_drive
```

Verilog System Functions

Verilog system functions are special functions Verilog HDL Simulators implement to generate input or output during simulation. Their names start with a dollar sign (\$). These functions are parsed and ignored by FPGA Compiler II / FPGA *Express*.

Verilog Keywords

Verilog uses keywords, shown in Table B-2, to interpret an input file. You cannot use these words as user variable names unless you use an escaped identifier. For more information, see “Identifiers” on page B-15.

Table B-2 Verilog Keywords

always	force	or	triereg
and	forever	output	table
assign	fork	parameter	task
begin	function	pmos	time
buf	highz0	posedge	tran
bufif0	highz1	primitive	tranif0
bufif1	if	pull0	tranif1
case	initial	pull1	tri
casex	inout	rcmos	triand
casez	input	reg	tri0
cmos	integer	release	tri1
deassign	join	repeat	vectored
default	large	rnmos	wait
defparam	medium	rpmos	wand
disable	module	rtran	weak0
end	nand	rtranif0	weak1
endcase	negedge	rtranif1	while
endfunction	nmos	scalared	wire
endmodule	nor	small	wor
endprimitive	not	strong0	xnor
endtable	notif0	strong1	xor
endtask	notif1	supply0	
event	pulldown	supply1	
for	pullup	trior	

Unsupported Verilog Language Constructs

FPGA Compiler II / FPGA *Express* does not support the following Verilog constructs:

- Unsupported definitions and declarations
 - primitive definition
 - time declaration
 - event declaration
 - triand, trior, tri1, tri0, and trireg net types
 - Ranges and arrays for integers
- Unsupported statements
 - defparam statement
 - initial statement
 - repeat statement
 - delay control
 - event control
 - wait statement
 - fork statement
 - deassign statement
 - force statement
 - release statement

- Unsupported operators
 - Case equality and inequality operators (=== and !==)
 - Division and modulus operators for variables
- Unsupported gate-level constructs
 - nmos, pmos, cmos, rnmos, rpmos, rcmos, pullup, pulldown, tranif0, tranif1, rtran, rtrainf0, and rtrainf1 gate types
- Unsupported miscellaneous constructs
 - Hierarchical names within a module

Constructs added to the Verilog Simulator in versions after Verilog 1.6 might not be supported.

If you use an unsupported construct in a Verilog description, FPGA Compiler II / *FPGA Express* issues a syntax error such as

```
event is not supported
```


Glossary

anonymous type

A predefined or underlying type with no name, such as universal integers.

ASIC

Application-specific integrated circuit.

behavioral view

The set of Verilog statements that describe the behavior of a design by using sequential statements. These statements are similar in expressive capability to those found in many other programming languages. See also the *data flow view*, *sequential statement*, and *structural view* definitions.

bit-width

The width of a variable, signal, or expression in bits. For example, the bit-width of the constant 5 is 3 bits.

character literal

Any value of type CHARACTER, in single quotation marks.

computable

Any expression whose (constant) value FPGA Compiler II / FPGA Express can determine during translation.

constraints

The designer's specification of design performance goals. FPGA Compiler II / FPGA *Express* uses constraints to direct the optimization of a design to meet area and timing goals.

convert

To change one type to another. Only integer types and subtypes are convertible, along with same-sized arrays of convertible element types.

data flow view

The set of VHDL/Verilog statements that describe the behavior of a design by using concurrent statements. These descriptions are usually at the level of Boolean equations combined with other operators and function calls. See also the *behavioral view* and *structural view* definitions.

design constraints

See *constraints*.

flip-flop

An edge-sensitive memory device.

HDL

Hardware Description Language.

identifier

A sequence of letters, underscores, and numbers. An identifier cannot be a VHDL/Verilog reserved word, such as type or loop. An identifier must begin with a letter or an underscore.

latch

A level-sensitive memory device.

netlist

A network of connected components that together define a design.

optimization

The modification of a design in an attempt to improve some performance aspect. FPGA Compiler II / FPGA *Express* optimizes designs and tries to meet specified design constraints for area and speed.

port

A signal declared in the interface list of an entity.

reduction operator

An operator that takes an array of bits and produces a single-bit result, namely the result of the operator applied to each successive pair of array elements.

register

A memory device containing one or more flip-flops or latches used to hold a value.

resource sharing

The assignment of a similar VHDL/Verilog operation (for example, +) to a common netlist cell. Netlist cells are the resources—they are equivalent to built hardware.

RTL

Register transfer level, a set of structural and data flow statements.

sequential statement

A set of VHDL/Verilog statements that execute in sequence.

signed value

A value that can be positive, zero, or negative.

structural view

The set of VHDL/Verilog statements used to instantiate primitive and hierarchical components in a design. A VHDL/Verilog design at the structural level is also called a netlist. See also the *behavioral view* and *data flow view* definitions.

subtype

A type declared as a constrained version of another type.

synthesis

The creation of optimized circuits from a high-level description. When VHDL/Verilog is used, synthesis is a two-step process: translation from VHDL/Verilog to gates and optimization of those gates for a specific FPGA library.

technology library

A library of cells available to FPGA Compiler II / *FPGA Express* during the synthesis process. A technology library can contain area, timing, and functional information on each cell.

translation

The mapping of high-level language constructs onto a lower-level form. FPGA Compiler II / *FPGA Express* translates RTL VHDL/Verilog descriptions to gates.

type

In VHDL/Verilog, the mechanism by which objects are restricted in the values they are assigned and the operations that can be applied to them.

unsigned

A value that can be only positive or zero.

variable

An electrical quantity that can be used to transmit information. A signal is declared with a type and receives its value from one or more drivers. Signals are created in Verilog through either wire or reg declarations.

VHDL

VHSIC hardware description language.

VHSIC

Very high speed integrated circuit, a high-technology program of the United States Department of Defense.

Index

Symbols

! (logical NOT operator) 4-8
!= (inequality operator) 4-6
% (modulus operator) 4-4
& (binary bitwise AND operator) 4-9
& (reduction AND operator) 4-10
&& (logical AND operator) 4-8
-(subtraction operator) 4-4
* (multiplication operator) 4-4
+ (addition operator) 4-4
/ (division operator) 4-4
// synopsys enum 8-10
// synopsys full_case 8-5
// synopsys parallel_case 8-4
 circuitry synthesized for 8-4
// synopsys state_vector 8-8
// synopsys translate_off 8-2
// synopsys translate_on 8-2
== (equality operator) 4-6
> (greater than operator) 4-5
>= (greater than or equal to operator) 4-5
?: (conditional operator) 4-12
^ (binary bitwise XOR operator) 4-9
^ (reduction XOR operator) 4-10
^~ (binary bitwise XNOR operator) 4-9

^~ (reduction XNOR operator) 4-10
{ } (concatenation operator) 4-13
| (binary bitwise OR operator) 4-9
| (reduction OR operator) 4-10
|| (logical OR operator) 4-8
~ (unary negation operator) 4-9
~& (reduction NAND operator) 4-10
~^ (binary bitwise XNOR operator) 4-9
~^ (reduction XNOR operator) 4-10
~| (reduction NOR operator) 4-10

A

adder tree
 balanced 7-8
addition operator (+) 4-4
always block 5-34
 clocks 5-35
 edge syntax B-4
 event expression 5-34, 5-35
 event specification 5-37
 grouping triggers 5-34
 in functional descriptions 2-3
 in modules 3-7
 negedge in 5-36
 posedge in 5-36
 syntax B-4

- AND binary bitwise operator (&) 4-9
- AND logical operator (&&) 4-8
- AND reduction operator (&) 4-10
- and, connection list 3-21
- apparently sequential constructs 5-2
- arithmetic optimization
 - considering overflow from carry bits 7-11
 - introduction 7-7
- assign 3-15
- async_set_reset 6-4
- async_set_reset_local_all 6-5
- asynchronous designs
 - optimization 7-34
- asynchronous preload 5-36

B

- balanced adder tree 7-8
- begin-end 5-14
- begin-end pair 5-4
- bidirectional port 3-15
- binary bitwise AND operator (&) 4-9
- binary bitwise OR operator (|) 4-9
- binary bitwise XNOR operator (^~ or ~^) 4-9
- binary bitwise XOR operator (^) 4-9
- binary numbers B-13
- binary operators 4-3, B-10
- bit-select
 - definition 4-17
 - from an enumerated type 8-15
- bit-width
 - expression 4-19
 - in module instantiation 3-17
 - prefix for numbers B-14
 - specifying in numbers B-14
- bitwise operator 4-9
 - binary AND 4-9
 - binary OR (|) 4-9
 - binary XNOR (^~ or ~^) 4-9
 - binary XOR (^) 4-9

- unary negation (~) 4-9
- block
 - begin in 5-14
 - end in 5-14
 - named 5-14
 - sequential 5-14
 - statements 5-14
 - syntax B-9
 - variables in named 5-15
- buf 3-21
- buffer instantiation 3-22

C

- call
 - function 5-3
- carry-lookahead adder chain 7-4
- case
 - avoiding latch and register inference 5-20
 - case item 5-19
 - circuitry synthesized 8-4
 - default 5-20
 - latch inference 5-20, 8-6
 - multiple expressions in 5-19
 - register inference 5-20
 - statement 5-18
 - full case 5-20
 - parallel case 5-21
- case item 5-19, 5-23, 5-26
 - syntax B-9
- casex
 - case item 5-23
 - statement 5-22
- casez
 - case item 5-26
 - statement 5-25
- charge strength, syntax of B-6
- circuitry
 - efficient 7-1
- combinational feedback path 7-32
- combinational logic

- apparently sequential constructs 5-2
 - in functional descriptions 2-3
- comments
 - HDL Compiler directives 8-2
 - lexical conventions B-13
- common subexpressions
 - sharing 7-13
- component implication
 - distinct component
 - syntax 8-16
 - instantiation 8-16
 - registers 6-1
- component organization 7-3
- concatenation
 - in procedural assignment 5-10
 - operand 3-4, 4-19
 - operator 4-19
 - syntax B-11
- concatenation operator ({}) 4-13
 - number of operands 4-3
 - repetition multiplier 4-13
 - unsized constants 4-13
- conditional operator 4-12
 - nested 4-12
 - number of operands 4-3
- conditional statement 5-15
- conditionally assigned variable
 - reading 5-18, 6-19
- connection list 3-17
 - terminals 3-17
- constant
 - in number operands 4-16
 - sized 4-16
 - unsized 4-16, B-14
- constant propagation 7-31
- constant-valued expression
 - definition 4-2
 - in range specifications 3-8
 - represented in parameters 3-8
 - synthesized circuitry 4-2
- construct 3-7

- unsupported B-20
- context-determined operands 4-19
- continuous assignment 2-3
 - drive strength in 3-16
 - driving a wire 3-9
 - in a wire declaration 3-15
 - in function declarations 5-3
 - in modules 3-7
 - left side of 3-16
 - right side of 3-16
 - syntax B-5

D

- data assignments 3-7
- data declarations 3-7
- decimal numbers B-13
- declarations
 - function 5-3
 - input 5-5
 - integer 5-9
 - parameter 5-8
 - register 5-6
- decrementing loop 5-27
- default 5-20
- define B-16
- definitions
 - register inference 6-1
- delay
 - gate-level options 3-21
 - syntax B-12
- delay value 3-11
- description style 2-6
- descriptions
 - logically equivalent 7-1
- design
 - efficiency 7-34
- Design Compiler
 - restructuring 1-5
 - synthesis and optimization 1-5
- design flow 1-6

- design methodology 2-6
- directives
 - enum 8-10
 - full_case
 - syntax 8-5
 - simulation B-18
- disable 5-31
 - in named block 5-31
- division operator (/) 4-4
- don't care inference
 - simulation 7-29
 - synthesis 7-29
- don't care values 7-28
- dot operator (.) 3-6
- drive strength
 - in a continuous assignment 3-16
 - syntax B-6

E

- edge
 - syntax B-4
- efficiency
 - of descriptions 7-1
- endfunction
 - keyword 5-4
- enum directive 8-10
- equality operator (==) 4-6
- escaped identifier B-15
- event
 - always block 5-35
 - specification
 - in always blocks 5-37
- event expression
 - always block 5-34
- examples
 - three-state component
 - registered input 6-58
 - two-phase clocked design 6-19
- explicit state style
 - for state machines 7-19

- expression tree 7-7
 - subexpressions in 7-10
- expressions
 - bit-width 4-19
 - compile-time evaluation 7-31
 - context determined 4-19
 - definition 4-1
 - legal 4-1
 - self-determined 4-19
 - syntax B-9
 - using parentheses in 7-9

F

- falling edge 5-35
- feedback paths 7-32
- finite state machines
 - describing with explicit state style 7-19
 - describing with implicit state style 7-19
 - using state information 7-16
- flip-flop
 - definition 6-1
 - implying edge-triggered 5-34
- for
 - duplicating statements 5-28
 - nested 5-27
 - range expression 5-27
- for loops 5-27
 - begin statement 5-27
 - end statement 5-27
- full case 5-20
- full_case directive 8-5
- fully specified variable 7-32
- function
 - declaration 5-3
 - continuous assignments 5-3
 - module terminals in 5-3
 - ignored B-18
 - keyword 5-4
 - local variables 5-8
 - outputs 5-5

- range specification 5-4
- syntax B-4
- function call 5-3
 - operand 4-3, 4-18
 - syntax B-11
- function declaration
 - in functional descriptions 2-3
 - syntax B-4
- function definition
 - in modules 3-7
- function name
 - syntax B-4, B-11
- function statement
 - begin-end blocks 5-14
 - case statements 5-18
 - casex statements 5-22
 - casez statements 5-25
 - disable statement 5-31
 - for loop 5-27
 - forever 5-30
 - if ... else construct 5-15
 - procedural assignment 5-10
 - supported types 5-9
 - while loop 5-29
- functional description 1-7, 2-3
 - always blocks in 2-3
 - combinational logic in 2-3
 - construction and use 5-1
 - function declarations in 2-3
 - mixing with structural descriptions 2-4
 - sequential logic in 2-3

G

- gate
 - connecting to inout 3-15
- gate instance
 - name, syntax B-7
 - syntax B-7
- gate instantiation
 - in modules 3-7

- syntax B-7
- gate types B-7
- gate-level constructs 2-3
- gate-level modeling 3-20
 - delay options 3-21
 - instance names 3-21
- global variable
 - integer 5-9

H

- hardware description languages 1-2
- HDL
 - definition 1-2
- HDL Compiler
 - design knowledge 7-7
 - efficient translations 7-7
- HDL Compiler directives
 - circuitry synthesized for parallel_case 8-4
 - definition 8-1
 - enum 8-10
 - full_case 8-5
 - full_case used with parallel_case 8-6
 - parallel_case 8-4
 - parallel_case used with full_case 8-6
 - state_vector 8-8
 - translate_off 8-2
 - translate_on 8-2
- HDL synthesis 1-2
- hexadecimal numbers B-13
- hierarchical boundaries 2-2
- hierarchical constructs 2-3
- hierarchical names
 - not supported B-15
- high-impedance state 6-51

I

- identifier B-15
 - escaped B-15
 - lowercase sensitivity B-11

- syntax B-11
 - uppercase sensitivity B-11
- if ... else construct 5-15
- ignored functions B-18
- implicit state style
 - for state machines 7-19
- implying registers 6-1
- include construct
 - example B-17
- incompletely specified variable 7-32
- incrementing loop 5-27
- inequality operator (!=) 4-6
- inference report
 - description 6-51
 - example 6-2
- infinite loops 5-30
- inout
 - connecting to gate 3-15
 - connecting to module 3-15
 - declaration 3-7
 - syntax of B-5
 - statement 3-15
 - wire 3-15
- input
 - declaration 3-7, 5-5
 - ports 3-14
 - range specifications 5-5
 - signal 5-5
 - statement 3-14
 - structural data type 3-8
 - wire 3-14
- input declaration
 - definition 5-5
 - syntax B-5
- input statement 3-7, 5-5
- instance names
 - in gate-level modeling 3-21
- integer 5-9
 - declaration 5-9
 - syntax of B-5
 - in procedural assignment 5-10

- integer variable 5-9
 - global 5-9
 - local 5-9
 - size 5-9
- internal design format 1-4

K

- keywords B-19

L

- language constructs 2-6
- latch
 - definition 6-1
- latch inference
 - avoiding 8-5, 8-7
 - local variables 6-11
- least significant bit 3-8
- lexical conventions B-12
- local variable 5-8
 - integer 5-9
- logic
 - combinational 5-2
 - multipath branch 5-18, 5-22, 5-25
- logical AND operator (&&) 4-8
- logical NOT operator (!) 4-8
- logical OR operator (||) 4-8
- loop
 - decrementing 5-27
 - incrementing 5-27
- lsb (least significant bit) 3-8

M

- macro substitution B-16
- macro variable B-16
- memory construct 5-7
 - two-dimensional array 5-7
- modeling

- gate-level 3-20
- module 3-2, 3-7
 - connecting to inout 3-15
 - connection list 3-17
 - constructs 3-7
 - instance name, syntax B-7
 - instance, syntax B-7
 - instantiation 3-17
 - instantiation, syntax B-7
 - name, syntax B-3, B-7
 - syntax B-3
 - terminals 3-17
- module definition
 - in structural descriptions 2-3
- module instantiation 3-17
 - bit-widths 3-17
 - in structural descriptions 2-3
 - name-based 3-18
 - named notation 3-18
 - positional notation 3-18
 - position-based 3-18
- module statement
 - module instantiations 3-7
- module terminals
 - in function declarations 5-3
- modulus operator (%) 4-4
- most significant bit 3-8
- msb (most significant bit) 3-8
- multipath branch 5-18
 - in casex statement 5-22
 - in casez statement 5-25
- multiplexer
 - creating with case and parallel_case 8-4
- multiplication operator (*) 4-4

N

- named block
 - construct 5-14
 - disable used in 5-31
 - syntax B-9

- variables in 5-15
- named notation 3-18
- NAND reduction operator (~&) 4-10
- negative edge 5-35
- negedge 5-35, 5-36
- net types B-5
- netlist connection
 - in structural descriptions 2-3
- NOR reduction operator (~|) 4-10
- NOT logical operator (!) 4-8
- number 4-16
 - binary B-13
 - decimal B-13
 - formats B-13
 - hexadecimal B-13
 - octal B-13
 - operand in expressions 4-16
 - sized 4-16
 - specifying bit-width B-14
 - syntax B-10
 - unsized 4-16

O

- octal numbers B-13
- operand 4-1, 4-16
 - bit-select 4-17
 - concatenation 3-4, 4-19
 - constants 4-16
 - constant-valued 4-4
 - context-determined 4-19
 - function call 4-3, 4-18
 - in expressions 4-16
 - number 4-16
 - part-select 4-18
 - register 4-17
 - self-determined 4-19
 - variable 4-4
 - wire 4-17
- operator 4-1, 7-15, B-15
 - addition (+) 4-4

- arithmetic 4-4
- binary 4-3, B-10
- binary bitwise AND (&) 4-9
- binary XNOR bitwise operator($\wedge\sim$ or $\sim\wedge$) 4-9
- bitwise 4-9
- bitwise binary XOR (\wedge) 4-9
- bitwise OR (|) 4-9
- concatenation ({}) 4-3, 4-13, 4-19
- conditional 4-3
- conditional (?:) 4-12
- definition 4-3
- division (/) 4-4
- dot (.) 3-6
- equality (==) 4-6
- inequality (!=) 4-6
- lexical conventions B-15
- logical and (&&) 4-8
- logical not (!) 4-8
- logical or (||) 4-8
- modulus (%) 4-4
- multiplication (*) 4-4
- nested conditional 4-12
- precedence 4-15
- reduction AND (&) 4-10
- reduction NAND ($\sim\&$) 4-10
- reduction NOR($\sim|$) 4-10
- reduction OR (|) 4-10
- reduction XNOR($\sim\wedge$) 4-10
- reduction XOR (\wedge) 4-10
- relational 4-5
- shift left (<) 4-11
- shift right (>>) 4-11
- subtraction (-) 4-4
- unary 4-3, B-9
- unary bitwise negation (\sim) 4-9
- OR binary bitwise operator (|) 4-9
- OR logical operator (||) 4-8
- OR reduction operator (|) 4-10
- output
 - assigning to a function name 5-5
 - declaration 3-7, 5-5

- syntax of B-5
 - of functions 5-5
- port 3-14
- reg 3-14
- returning multiple 5-5
- statement 3-7, 3-14
- wire 3-14
- overflow characteristics
 - arithmetic optimization 7-11

P

- parallel case 5-21
- parallel_case directive 8-4
 - circuitry synthesized for 8-4
- parameter
 - declaration 3-7, 5-8
 - syntax of B-4
 - local variables 5-8
 - name 3-8
 - range 3-8
 - sized 3-8
 - variables 5-8
- parameterized design 3-19
- part-select 4-18
 - operand 4-18
- physical circuit description 3-1
- port
 - dot operator 3-6
 - explicit instantiation 3-18
 - explicitly renaming 3-6
 - expression 3-4
 - bit-select 3-4
 - concatenation 3-4
 - identifier 3-4, 3-5
 - part-select 3-4
 - syntax B-3
 - implicit instantiation 3-5, 3-18
 - input 3-14
 - list 3-4
 - syntax of B-3

- name
 - syntax B-3
- output 3-14
- renaming inside module 3-6
- syntax B-3
- port declarations 3-14
- posedge 5-35, 5-36
- positional notation 3-18
- positive edge 5-35
- precedence, operator 4-15
- preload 5-36
- priority encoder 8-4
- procedural assignment
 - concatenation in 5-10
 - integer 5-10
 - left side 5-10
 - register 5-10
 - right side 5-10
 - statement 5-10

R

- radices B-13
- range
 - constant-valued expressions 3-8
 - expression in for loops 5-27
 - specification 3-8, 5-4
 - in function declarations 5-4
 - in inputs 5-5
 - syntax B-6
- reading
 - conditionally assigned variables 5-18
- reduction operator
 - AND (&) 4-10
 - NAND (~&) 4-10
 - NOR (~|) 4-10
 - OR (|) 4-10
 - XNOR (^~ or ~^) 4-10
 - XOR (^) 4-10
- reg 5-6
- register

- declaration 5-6
 - syntax of B-5
- definition of 6-1
- holding state information 5-7
- in procedural assignments 5-10
- inference 6-1
- operand 4-17
- output 3-14
- register inference 2-8
 - avoiding extra registers 7-24
 - D latch 6-9
 - definition 6-1
 - edge expressions 6-21
 - efficient circuits 7-24
 - SR latch 6-7
 - templates 6-2
 - wait statement 6-21
- relational operators 4-5
- resource allocation 7-1
- resource sharing 7-1
- ripple carry adder chain 7-4
- rising edge 5-35

S

- sequential
 - block 5-14
 - logic
 - in functional descriptions 2-3
- sequential logic 2-4
- sharing
 - common subexpressions
 - automatically determined 7-13
- signals
 - edge detection 6-21
- simulation
 - directives B-18
 - don't care values 7-29
 - place in the design process 1-7
 - test vectors 1-7
- size syntax B-11

- state information
 - holding with a register 5-7
 - using for efficiency 7-16
- state machines
 - explicit state style 7-19
 - implicit state style 7-19
- state_vector 8-8
- statements 3-7
- structural data types 3-8
- structural description
 - construction 3-1
 - elements of 2-3
 - in design flow 1-7
 - mixing with functional description 2-4
- structure
 - before optimization 7-3
 - controlling with parentheses 7-5
 - preservation 7-3
- subexpressions
 - in an expression tree 7-10
- subtraction operator (-) 4-4
- sync_set_reset 6-5
- sync_set_reset_local 6-5
- sync_set_reset_local_all 6-5
- syntax B-1
 - component implication
 - distinct component 8-16
 - full_case directive 8-5
 - of charge strength B-6
 - Verilog B-1
- synthesis policy 2-6
- system functions, Verilog B-18

T

- task
 - construct 5-33
 - statement 5-32
 - in modules 3-7
- terminal
 - expression 3-17

- in function declaration 5-3
- syntax B-7
- test vectors
 - simulation 1-7
- three-state
 - gate 6-57
 - registered input 6-58
- three-state buffer instantiation 3-22
- three-state gate 6-58
- translate_off 8-2
- translate_on 8-2
- translation 8-2
 - restart 8-2
 - suspend 8-2
- triggers 5-34
- two-phase design 6-19

U

- unary negation bitwise operator (~) 4-9
- unary operator
 - definition 4-3
 - syntax B-9
- unassigned variables 5-18
- underscore B-11
 - in numbers B-14
- unsupported
 - Verilog constructs B-20

V

- variable
 - conditionally assigned 5-18, 6-19
 - in named blocks 5-15
 - integer 5-9
 - local in parameters 5-8
 - operand 4-4
 - reading 5-18
 - registering 6-48
- verification of description implementation 1-7
- Verilog

- relational operators 4-5
- Verilog constructs
 - unsupported B-20
- Verilog hardware descriptions 1-4
- Verilog HDL description 1-1
- Verilog keywords B-19
- Verilog syntax B-1
- Verilog system function B-18
- VHDL
 - register inference 2-7

W

- wait statement
 - creating registers 6-21
- wand 3-10
 - wired-AND 3-10
- white space
 - lexical convention B-13
- wire 4-17
 - continuous assignment 3-15
 - declaration 3-7, 3-9, 3-10
 - driving with a continuous assignment 3-9
 - high impedance 3-10
 - inout 3-15
 - input 3-14

- operand 4-17
- output 3-14
- structural data type 3-8
- undriven 3-10
- use in a function 3-9
- wired-AND 3-10
- wired-OR 3-11
- wired-AND 3-10
- wired-OR 3-11
- wor
 - wired-OR 3-12
- wor data type
 - wired-OR 3-11

X

- XNOR binary bitwise operator ($\wedge\sim$ or $\sim\wedge$) 4-9
- xnor connection list 3-21
- XNOR reduction operator ($\wedge\sim$ or $\sim\wedge$) 4-10
- XOR binary bitwise operator (\wedge) 4-9
- XOR reduction operator (\wedge) 4-10

Z

- z undriven wire 3-10