

FPGA Compiler II / FPGA *Express* VHDL 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 VHDL Reference Manual, Version 1999.05

About This Manual

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

VHDL is defined by IEEE Standard 1076 and the United States Department of Defense Standard MIL-STD-454L. Appendix B and Appendix C summarize the level of Synopsys support for all VHDL packages and constructs.

Audience

This manual is written for logic designers and electronic engineers who are familiar with Synopsys synthesis products. A basic knowledge of VHDL or other high-level programming language is also necessary.

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*

For more information about VHDL and its use, see the following publications:

- *IEEE Standard VHDL Language Reference Manual*, IEEE Std 1076-1987.
- *Introduction to HDL-Based Design Using VHDL*. Steve Carlson. Synopsys, Inc., 1990.
- *VHDL*. Douglas L. Perry. McGraw-Hill, Inc., 1991.

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. 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) 594-2539.

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>object_name</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>pin1 [pin2, . . . , pinN]</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>(Control-c)</code>	Indicates a keyboard combination, such as holding down the Control 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. Using FPGA Compiler II / FPGA *Express* with VHDL

Hardware Description Languages	1-2
Typical uses for HDLs	1-3
Advantages of HDLs	1-3
About VHDL	1-4
FPGA Compiler II / FPGA <i>Express</i> Design Process	1-7
Using FPGA Compiler II / FPGA <i>Express</i> to Compile a VHDL Design	1-8
Design Methodology	1-9

2. Design Descriptions

Entities	2-2
Entity Generic Specifications	2-3
Entity Port Specifications	2-4
Architecture	2-5
Declarations	2-10

Components	2-10
Concurrent Statements	2-17
Constants	2-18
Processes	2-19
Signals	2-21
Subprograms	2-22
Types	2-31
Examples of Architectures for NAND2 Entity	2-33
Configurations	2-34
Packages	2-35
Package Uses	2-35
Package Structure	2-36
Package Declarations	2-37
Package Body	2-39
Resolution Functions	2-40
 3. Data Types	
Enumeration Types	3-3
Enumeration Overloading	3-4
Enumeration Encoding	3-4
Enumeration Encoding Values	3-7
Integer Types	3-8
Array Types	3-9
Constrained Arrays	3-10
Unconstrained Arrays	3-10
Array Attributes	3-12

Record Types	3-13
Record Aggregates	3-14
Predefined VHDL Data Types	3-16
Data Type BOOLEAN	3-18
Data Type BIT	3-18
Data Type CHARACTER	3-18
Data Type INTEGER	3-19
Data Type NATURAL	3-19
Data Type POSITIVE	3-19
Data Type STRING	3-19
Data Type BIT_VECTOR	3-19
Unsupported Data Types.	3-20
Physical Types	3-20
Floating-Point Types.	3-20
Access Types.	3-20
File Types.	3-20
Synopsys Data Types	3-21
Subtypes	3-21
 4. Expressions	
Operators	4-2
Logical Operators.	4-3
Relational Operators	4-5
Adding Operators.	4-8
Unary (Signed) Operators	4-10

Multiplying Operators	4-11
Miscellaneous Arithmetic Operators	4-12
Operands	4-14
Operand Bit-Width	4-15
Computable Operands	4-16
Aggregates	4-18
Attributes	4-20
Expressions	4-21
Function Calls	4-22
Identifiers	4-23
Indexed Names	4-24
Literals	4-26
Numeric Literals	4-26
Character Literals	4-26
Enumeration Literals	4-27
String Literals	4-27
Qualified Expressions	4-29
Records and Fields	4-30
Slice Names	4-32
Limitations on Null Slices	4-33
Limitations on Noncomputable Slices	4-34
Type Conversions	4-34
 5. Sequential Statements	
Assignment Statements and Targets	5-2
Simple Name Targets	5-3
Indexed Name Targets	5-4

Slice Targets	5-7
Field Targets	5-8
Aggregate Targets	5-9
Variable Assignment Statements.	5-11
Signal Assignment Statements	5-12
if Statements	5-15
Evaluating Conditions	5-15
Using the if Statement to Infer Registers and Latches.	5-16
case Statements	5-17
Using Different Expression Types	5-18
Invalid case Statements.	5-21
loop Statements	5-22
Basic loop Statements	5-23
while...loop Statements	5-24
for...loop Statements	5-25
Steps in the Execution of a for...loop Statement.	5-27
for...loop Statements and Arrays	5-28
next Statements.	5-30
exit Statements	5-33
Subprograms	5-35
Subprogram Always a Combinational Circuit.	5-35
Subprogram Declaration and Body	5-35
Subprogram Calls	5-37
Procedure Calls	5-39
Function Calls.	5-41

return Statement	5-43
Procedures and Functions as Design Components.	5-45
Example With Component Implication Directives	5-47
Example Without Component Implication Directives	5-49
wait Statements	5-50
Inferring Synchronous Logic	5-51
Combinational Versus Sequential Processes	5-55
null Statements	5-58
 6. Concurrent Statements	
process Statements.	6-2
Combinational Process Example.	6-5
Sequential Process Example.	6-6
Driving Signals.	6-8
block Statements	6-10
Nested Blocks	6-11
Guarded Blocks	6-12
Concurrent Versions of Sequential Statements.	6-13
Concurrent Procedure Calls.	6-14
Concurrent Signal Assignments.	6-17
Simple Concurrent Signal Assignments.	6-17
Conditional Signal Assignment.	6-18
Selected Signal Assignments.	6-20
Component Instantiation Statements	6-22
Direct Instantiation.	6-25

generate Statements	6-26
for...generate Statement	6-26
Steps in the Execution of a for...generate Statement	6-27
Common Usage of a for...generate Statement	6-29
if...generate Statements	6-31
 7. Register and Three-State Inference	
Register Inference	7-1
The inference Report	7-3
Latch Inference Warnings	7-4
Controlling Register Inference	7-4
Attributes That Control Register Inference	7-5
Inferring Latches	7-8
Inferring Set/Reset (SR) Latches	7-8
Inferring D Latches	7-10
Inferring Master-Slave Latches	7-20
Inferring Flip-Flops	7-21
Inferring D Flip-Flops	7-22
Inferring JK Flip-Flops	7-41
Inferring Toggle Flip-Flops	7-45
Getting the Best Results	7-51
Understanding Limitations of Register Inference	7-57
Three-State Inference	7-59
Reporting Three-State Inference	7-59
Controlling Three-State Inference	7-60
Inferring Three-State Drivers	7-60
Inferring a Simple Three-State Driver	7-60

Three-State Driver With Registered Enable	7-65
Three-State Driver Without Registered Enable	7-67
Understanding the Limitations of Three-State Inference	7-69
8. Writing Circuit Descriptions	
How Statements Are Mapped to Logic	8-2
Design Structure	8-3
Adding Structure	8-3
Using Variables and Signals.	8-4
Using Parentheses	8-5
Using Design Knowledge.	8-6
Optimizing Arithmetic Expressions	8-6
Arranging Expression Trees for Minimum Delay	8-7
Sharing Common Subexpressions.	8-12
Changing an Operator Bit-Width	8-14
Using State Information	8-17
Propagating Constants	8-21
Sharing Complex Operators	8-22
Asynchronous Designs	8-23
Don't Care Inference	8-29
Using don't care Default Values.	8-32
Differences Between Simulation and Synthesis	8-33
Synthesis Issues	8-34
Feedback Paths and Latches.	8-34
Fully Specified Variables	8-35
Asynchronous Behavior	8-37

Understanding Superset Issues and Error Checking	8-38
9. FPGA Compiler II / FPGA <i>Express</i> Directives	
Notation for FPGA Compiler II / FPGA <i>Express</i> Directives	9-2
FPGA Compiler II / FPGA <i>Express</i> Directives	9-2
Translation Stop and Start Pragma Directives	9-3
synthesis_off and synthesis_on Directives	9-3
Resolution Function Directives	9-5
Component Implication Directives	9-5
A. Examples	
Moore Machine	A-2
Mealy Machine	A-5
Read-Only Memory	A-7
Waveform Generator	A-10
Smart Waveform Generator	A-13
Definable-Width Adder-Subtractor	A-16
Count Zeros—Combinational Version	A-19
Count Zeros—Sequential Version	A-22
Soft Drink Machine—State Machine Version	A-24
Soft Drink Machine—Count Nickels Version	A-29
Carry-Lookahead Adder	A-32
Carry Value Computations	A-32
Implementation	A-39

Serial-to-Parallel Converter—Counting Bits	A-40
Input Format.	A-41
Implementation Details	A-42
Serial-to-Parallel Converter—Shifting Bits.	A-47
Programmable Logic Arrays	A-51
 B. Synopsys Packages	
std_logic_1164 Package	B-2
std_logic_arith Package	B-3
Using the Package.	B-4
Modifying the Package.	B-5
Data Types.	B-6
UNSIGNED	B-6
SIGNED	B-7
Conversion Functions	B-8
Arithmetic Functions	B-10
Comparison Functions.	B-13
Shift Functions.	B-15
ENUM_ENCODING Attribute.	B-17
pragma built_in	B-17
Two-Argument Logic Functions	B-18
One-Argument Logic Functions	B-19
Type Conversion.	B-19
numeric_std Package	B-20
Understanding the Limitations of numeric_std package	B-21
Using the Package.	B-21

Data Types	B-22
Conversion Functions	B-22
Resize Function	B-23
Arithmetic Functions	B-23
Comparison Functions	B-24
Defining Logical Operators Functions	B-26
Shift Functions	B-27
Rotate Functions	B-28
Shift and Rotate Operators	B-28
std_logic_misc Package	B-30
ATTRIBUTES Package	B-31

C. VHDL Constructs

VHDL Construct Support.	C-2
Design Units	C-3
Data Types	C-4
Declarations	C-5
Specifications	C-6
Names	C-7
Identifiers and Extended Identifiers	C-8
Specifics of Identifiers	C-8
Specifics of Extended Identifiers	C-8
Operators	C-9
Shift and Rotate Operators	C-10
xnor Operator	C-11
Operands and Expressions	C-12

Sequential Statements	C-13
Concurrent Statements	C-15
Predefined Language Environment	C-16
VHDL Reserved Words	C-17

List of Figures

Figure 1-1	VHDL Hardware Model.	1-5
Figure 1-2	Design Flow	1-9
Figure 2-1	3-Bit Counter Synthesized Circuit.	2-8
Figure 2-2	Design Using Resolved Signal	2-43
Figure 4-1	Design Schematic for Logical Operators	4-4
Figure 4-2	Relational Operators Design Illustrating Example 4-4 . .	4-8
Figure 4-3	Design Array Illustrating Example 4-5.	4-9
Figure 4-4	Design Illustrating Unary Negation From Example 4-6. .	4-10
Figure 4-5	Design Illustrating Multiplying Operators From Example 4-7	4-12
Figure 4-6	Design With Arithmetic Operators From Example 4-8 . .	4-13
Figure 4-7	Design Illustrating Use of Indexed Names From Example 4-16	4-25
Figure 4-8	Design Illustrating Use of Slices From Example 4-24. . .	4-33
Figure 5-1	Design Illustrating Indexed Name Targets From Example 5-3	5-6
Figure 5-2	Schematic Design From Example 5-8	5-16

Figure 5-3	Schematic Design From Example 5-9	5-19
Figure 5-4	Schematic Design From Example 5-10	5-20
Figure 5-5	Schematic Design From Example 5-12	5-28
Figure 5-6	Schematic Design of Array From Example 5-13.	5-29
Figure 5-7	Schematic Design From Example 5-14	5-31
Figure 5-8	Schematic Design From Example 5-16	5-34
Figure 5-9	Schematic Design From Example 5-18	5-41
Figure 5-10	Schematic Design From Example 5-20	5-45
Figure 5-11	Schematic Design With Component Implication Directives	5-48
Figure 5-12	Schematic Design Without Component Implication Directives	5-50
Figure 5-13	Schematic Design From Example 5-30	5-57
Figure 5-14	Schematic Design From Example 5-31	5-59
Figure 6-1	Modulo-10 Counter Process Design	6-6
Figure 6-2	Modulo-10 Counter Process With wait Statement Design	6-8
Figure 6-3	Two Three-State Buffers Driving the Same Signal	6-9
Figure 6-4	Schematic of Nested Blocks	6-12
Figure 6-5	Concurrent CHECK Procedure Design.	6-16
Figure 6-6	Conditional Signal Assignment Design.	6-19
Figure 6-7	Selected Signal Assignment Design.	6-21
Figure 6-8	A Simple Netlist Design	6-24
Figure 6-9	An 8-Bit Array Design	6-29
Figure 6-10	Design of COMP Components Connecting Bit Vectors A and B	6-30

Figure 6-11	Design of N-Bit Serial-to-Parallel Converter	6-33
Figure 7-1	SR Latch	7-10
Figure 7-2	D Latch	7-13
Figure 7-3	D Latch With Asynchronous Set	7-15
Figure 7-4	D Latch With Asynchronous Reset	7-17
Figure 7-5	D Latch With Asynchronous Set and Reset	7-19
Figure 7-6	Two-Phase Clocks	7-21
Figure 7-7	Positive Edge-Triggered D Flip-Flop	7-25
Figure 7-8	Positive Edge-Triggered D Flip-Flop Using rising_edge .	7-27
Figure 7-9	Negative Edge-Triggered D Flip-Flop	7-28
Figure 7-10	Negative Edge-Triggered D Flip-Flop Using falling_edge	7-29
Figure 7-11	D Flip-Flop With Asynchronous Set	7-30
Figure 7-12	D Flip-Flop With Asynchronous Reset	7-32
Figure 7-13	D Flip-Flop With Asynchronous Set and Reset	7-34
Figure 7-14	D Flip-Flop With Synchronous Set	7-35
Figure 7-15	D Flip-Flop With Synchronous Reset	7-37
Figure 7-16	D Flip-Flop With Synchronous and Asynchronous Load	7-38
Figure 7-17	MultipleFlip-FlopswithAsynchronousandSynchronousControls	7-40
Figure 7-18	JK Flip-Flop.	7-43
Figure 7-19	JK Flip-Flop With Asynchronous Set and Reset.	7-45
Figure 7-20	Toggle Flip-Flop With Asynchronous Set	7-47
Figure 7-21	Toggle Flip-Flop With Asynchronous Reset	7-49
Figure 7-22	Toggle Flip-Flop With Enable and Asynchronous Reset.	7-51

Figure 7-23	Circuit With Six Inferred Flip-Flops	7-54
Figure 7-24	Circuit With Three Inferred Flip-Flops	7-56
Figure 7-25	Schematic of Simple Three-State Driver	7-61
Figure 7-26	One Three-State Driver Inferred From a Single Process	7-63
Figure 7-27	Two Three-State Drivers Inferred From Separate Processes	7-65
Figure 7-28	Three-State Driver With Registered Enable	7-67
Figure 7-29	Three-State Driver Without Registered Enable.	7-69
Figure 8-1	Ripple Carry and Carry-Lookahead Chain Design	8-5
Figure 8-2	Diagram of 4-Input Adder	8-5
Figure 8-3	Diagram of 4-Input Adder With Parentheses	8-6
Figure 8-4	Default Expression Tree	8-7
Figure 8-5	Balanced Adder Tree (Same Arrival Times for All Signals)	8-8
Figure 8-6	Expression Tree With Minimum Delay (Signal A Arrives Last)	8-9
Figure 8-7	Expression Tree With Subexpressions Dictated by Parentheses	8-10
Figure 8-8	Default Expression Tree With 4-Bit Temporary Variable .	8-11
Figure 8-9	Expression Tree With 5-Bit Intermediate Result	8-12
Figure 8-10	Function With One Adder Schematic	8-15
Figure 8-11	Using TEMP Declaration to Save Circuit Area	8-16
Figure 8-12	Schematic of Simple State Machine With Two Processes	8-19
Figure 8-13	Schematic of an Improved State Machine	8-21
Figure 8-14	Schematic of Synchronous Counter With Reset and Enable	8-24

Figure 8-15	Design With AND Gate on Clock and Enable Signals . .	8-26
Figure 8-16	Design With Asynchronous Reset	8-26
Figure 8-17	Schematic of Incorrect Asynchronous Design With Gated Clock 8-28	
Figure 8-18	Seven-Segment LED Decoder With Don't Care Type. . .	8-30
Figure 8-19	Seven-Segment LED Decoder With 0 LED Default . . .	8-32
Figure A-1	Moore Machine Specification	A-2
Figure A-2	Moore Machine Schematic	A-4
Figure A-3	Mealy Machine Specification	A-5
Figure A-4	Mealy Machine Schematic	A-7
Figure A-5	ROM Schematic	A-9
Figure A-6	Waveform Example.	A-10
Figure A-7	Waveform Generator Schematic.	A-12
Figure A-8	Waveform for Smart Waveform Generator Example. . .	A-13
Figure A-9	Smart Waveform Generator Schematic	A-16
Figure A-10	6-Bit Adder-Subtractor Schematic	A-19
Figure A-11	Count Zeros—Combinational Schematic	A-21
Figure A-12	Count Zeros—Sequential Schematic	A-24
Figure A-13	Soft Drink Machine—State Machine Schematic.	A-28
Figure A-14	Soft Drink Machine—Count Nickels Version Schematic. .	A-31
Figure A-15	Carry-Lookahead Adder Block Diagram	A-34
Figure A-16	Sample Waveform Through the Converter	A-42
Figure A-17	Serial-to-Parallel Converter—Counting Bits Schematic .	A-47
Figure A-18	Serial-to-Parallel Converter—Shifting Bits Schematic . .	A-50

Figure A-19	Programmable Logic Array Schematic	A-55
-------------	--	------

List of Tables

Table 3-1	Array Index Attributes	3-12
Table 4-1	Predefined VHDL Operators	4-3
Table 7-1	SR Latch Truth Table (NAND Type)	7-9
Table 7-2	Truth Table for JK Flip-Flop	7-42
Table B-1	UNSIGNED,SIGNED,andBIT_VECTORComparisonFunctions B-4	
Table B-2	Number of Bits Returned by + and –	B-13
Table C-1	VHDL Reserved Words	C-17

List of Examples

Example 2-1	VHDL Entity Specification	2-3
Example 2-2	Interface for an N-Bit Counter	2-5
Example 2-3	An Implementation of a 3-Bit Counter	2-7
Example 2-4	Incorrect Use of a Port Name in Declaring Signals or Constants 2-9	
Example 2-5	Component Declaration of a 2-Input AND Gate	2-11
Example 2-6	Component Declaration of an N-Bit Adder	2-11
Example 2-7	Equivalent Named and Positional Association	2-15
Example 2-8	Structural Description of a 3-Bit Counter	2-15
Example 2-9	Constant Declarations	2-18
Example 2-10	Variable Declarations	2-20
Example 2-11	Signal Declarations	2-21
Example 2-12	Two Subprogram Declarations	2-25
Example 2-13	Two Subprogram Calls	2-26
Example 2-14	Two Subprogram Bodies	2-29
Example 2-15	Subprogram Overloading	2-29
Example 2-16	Operator Overloading	2-30

Example 2-17	Variable Declarations	2-31
Example 2-18	Structural Architecture for Entity NAND2	2-33
Example 2-19	Data Flow Architecture for Entity NAND2	2-34
Example 2-20	RTL Architecture for Entity NAND2	2-34
Example 2-21	Sample Package Declarations	2-38
Example 2-22	Resolved Signal and Its Resolution Function	2-42
Example 3-1	Enumeration Type Definitions	3-4
Example 3-2	Enumeration Literal Overloading.	3-4
Example 3-3	Automatic Enumeration Encoding.	3-5
Example 3-4	Using the ENUM_ENCODING Attribute	3-6
Example 3-5	Integer Type Definitions.	3-8
Example 3-6	Declaration of Array of Arrays	3-9
Example 3-7	Constrained Array Type Definition.	3-10
Example 3-8	Unconstrained Array Type Definition.	3-11
Example 3-9	Use of Array Attributes	3-13
Example 3-10	Record Type Declaration and Use	3-13
Example 3-11	Simple Record Type	3-15
Example 3-12	Named Aggregate for Example 3-11.	3-15
Example 3-13	Use of others in an Aggregate	3-16
Example 3-14	Positional Aggregate	3-16
Example 3-15	Record Aggregate Equivalent to Example 3-16	3-16
Example 3-16	Record Aggregate With Set of Choices	3-16
Example 3-17	FPGA Compiler II / FPGA <i>Express</i> STANDARD Package	3-17

Example 3-18	Valid and Invalid Assignments Between INTEGER Subtypes 3-22	
Example 3-19	Attributes and Functions Operating on a Subtype . . .	3-23
Example 4-1	Operator Precedence	4-3
Example 4-2	Logical Operators	4-4
Example 4-3	True Relational Expressions	4-7
Example 4-4	Relational Operators	4-7
Example 4-5	Adding Operators	4-9
Example 4-6	Unary (Signed) Operators	4-10
Example 4-7	Multiplying Operators With Powers of 2	4-11
Example 4-8	Miscellaneous Arithmetic Operators	4-13
Example 4-9	Computable and Noncomputable Expressions	4-17
Example 4-10	Simple Aggregate	4-19
Example 4-11	Equivalent Aggregates	4-20
Example 4-12	Equivalent Aggregates Using the others Expression .	4-20
Example 4-13	Function Calls	4-23
Example 4-14	Sample Extended Identifiers	4-23
Example 4-15	Identifiers	4-24
Example 4-16	Indexed Name Operands	4-25
Example 4-17	Numeric Literals	4-26
Example 4-18	Enumeration Literals	4-27
Example 4-19	Character String Literals	4-28
Example 4-20	Bit String Literals	4-29
Example 4-21	A Qualified Decimal Literal	4-30

Example 4-22	Qualified Aggregates and Enumeration Literals	4-30
Example 4-23	Record and Field Access	4-31
Example 4-24	Slice Name Operands	4-32
Example 4-25	Null and Noncomputable Slices	4-34
Example 4-26	Valid and Invalid Type Conversions.	4-35
Example 5-1	Simple Name Targets	5-4
Example 5-2	Indexed Name Targets	5-5
Example 5-3	Computable and Noncomputable Indexed Name Targets	5-5
Example 5-4	Slice Targets	5-7
Example 5-5	Field Targets	5-8
Example 5-6	Aggregate Targets.	5-10
Example 5-7	Variable and Signal Assignments	5-14
Example 5-8	if Statement.	5-16
Example 5-9	case Statement With Enumerated Type	5-18
Example 5-10	case Statement With Integers.	5-20
Example 5-11	Invalid case Statements	5-21
Example 5-12	for...loop Statement With Equivalent Code Fragments	5-27
Example 5-13	for...loop Statement Operating on an Entire Array . . .	5-28
Example 5-14	next Statement	5-30
Example 5-15	Named next Statement	5-32
Example 5-16	Comparator That Uses the exit Statement	5-34
Example 5-17	Subprogram Declarations and Bodies	5-37
Example 5-18	Procedure Call to Sort an Array	5-40

Example 5-19	Function Definition With Two Calls	5-42
Example 5-20	Use of Multiple return Statements.	5-44
Example 5-21	Using Component Implication Directives on a Function	5-47
Example 5-22	Using Gates to Implement a Function.	5-49
Example 5-23	Equivalent wait Statements.	5-51
Example 5-24	wait for a Positive Edge.	5-51
Example 5-25	Loop That Uses a wait Statement.	5-52
Example 5-26	Multiple wait Statements	5-52
Example 5-27	wait Statements and State Logic.	5-53
Example 5-28	Synchronous Reset That Uses wait Statements.	5-54
Example 5-29	Invalid Uses of wait Statements	5-54
Example 5-30	Parity Tester That Uses the wait Statement	5-56
Example 5-31	null Statement.	5-58
Example 6-1	Modulo-10 Counter Process	6-5
Example 6-2	Modulo-10 Counter Process With wait Statement	6-7
Example 6-3	Multiple Drivers of a Signal	6-9
Example 6-4	Nested Blocks	6-11
Example 6-5	Guarded Blocks.	6-12
Example 6-6	Level-Sensitive Latch Using Guarded Blocks	6-13
Example 6-7	Concurrent Procedure Call and Equivalent Process.	6-14
Example 6-8	Procedure Definition for Example 6-9	6-15
Example 6-9	Concurrent Procedure Calls	6-16
Example 6-10	Concurrent Signal Assignment	6-17
Example 6-11	Conditional Signal Assignment	6-19

Example 6-12	Process Equivalent to Conditional Signal Assignment	6-19
Example 6-13	Selected Signal Assignment	6-21
Example 6-14	Process Equivalent to Selected Signal Assignment . .	6-22
Example 6-15	Component Declaration and Instantiations	6-24
Example 6-16	A Simple Netlist.	6-24
Example 6-17	Component Instantiation Statement	6-25
Example 6-18	Direct Component Instantiation Statement	6-26
Example 6-19	for...generate Statement	6-28
Example 6-20	for...generate Statement Operating on an Entire Array	6-30
Example 6-21	Typical Use of if...generate Statements.	6-32
Example 7-1	Inference Report for a JK Flip-Flop	7-3
Example 7-2	SR Latch	7-9
Example 7-3	Inference Report for an SR Latch	7-10
Example 7-4	Latch Inference	7-11
Example 7-5	Fully Specified Signal: No Latch Inference	7-11
Example 7-6	Function: No Latch Inference	7-11
Example 7-7	D Latch	7-13
Example 7-8	Inference Report for a D Latch	7-13
Example 7-9	D Latch With Asynchronous Set	7-14
Example 7-10	Inference Report for D Latch With Asynchronous Set	7-15
Example 7-11	D Latch With Asynchronous Reset	7-16
Example 7-12	Inference Report for D Latch With Asynchronous Reset	7-16
Example 7-13	D Latch With Asynchronous Set and Reset	7-18

Example 7-14	Inference Report for D Latch With Asynchronous Set and Reset	7-18
Example 7-15	Invalid Use of a Conditionally Assigned Variable	7-19
Example 7-16	Two-Phase Clocks.	7-20
Example 7-17	Inference Reports for Two-Phase Clocks	7-21
Example 7-18	Using a wait Statement to Infer a Flip-Flop	7-23
Example 7-19	Using an if Statement to Infer a Flip-Flop	7-23
Example 7-20	Positive Edge-Triggered D Flip-Flop	7-25
Example 7-21	Inference Report for Positive Edge-Triggered D Flip-Flop	7-25
Example 7-22	Positive Edge-Triggered D Flip-Flop Using rising_edge	7-26
Example 7-23	Inference Report for a Positive Edge-Triggered D Flip-Flop Using rising_edge	7-26
Example 7-24	Negative Edge-Triggered D Flip-Flop	7-27
Example 7-25	Inference Report for Negative Edge-Triggered D Flip-Flop	7-28
Example 7-26	Negative Edge-Triggered D Flip-Flop Using falling_edge	7-28
Example 7-27	Inference Report for a Negative Edge-Triggered D Flip-Flop Using falling_edge.	7-29
Example 7-28	D Flip-Flop With Asynchronous Set	7-30
Example 7-29	Inference Report for a D Flip-Flop With Asynchronous Set	7-30
Example 7-30	D Flip-Flop With Asynchronous Reset	7-31
Example 7-31	Inference Report for a D Flip-Flop With Asynchronous Reset	7-31

Example 7-32	D Flip-Flop With Asynchronous Set and Reset.	7-33
Example 7-33	Inference Report for a D Flip-Flop With Asynchronous Set and Reset	7-33
Example 7-34	D Flip-Flop With Synchronous Set	7-35
Example 7-35	Inference Report for a D Flip-Flop With Synchronous Set	7-35
Example 7-36	D Flip-Flop With Synchronous Reset	7-36
Example 7-37	Inference Report for a D Flip-Flop With Synchronous Reset	7-36
Example 7-38	D Flip-Flop With Synchronous and Asynchronous Load	7-37
Example 7-39	Inference Report for a D Flip-Flop With Synchronous and Asynchronous Load	7-38
Example 7-40	Multiple Flip-Flops: Asynchronous and Synchronous Controls	7-39
Example 7-41	Inference Reports for Example 7-40	7-40
Example 7-42	JK Flip-Flop.	7-42
Example 7-43	Inference Report for JK Flip-Flop	7-43
Example 7-44	JK Flip-Flop With Asynchronous Set and Reset	7-44
Example 7-45	Inference Report for JK Flip-Flop With Asynchronous Set and Reset.	7-45
Example 7-46	Toggle Flip-Flop With Asynchronous Set	7-46
Example 7-47	Inference Report for Toggle Flip-Flop With Asynchronous Set	7-47
Example 7-48	Toggle Flip-Flop With Asynchronous Reset	7-48
Example 7-49	Inference Report for a Toggle Flip-Flop With Asynchronous Reset	7-48

Example 7-50	Toggle Flip-Flop With Enable and Asynchronous Reset	7-50
Example 7-51	Inference Report for Toggle Flip-Flop With Enable and Asynchronous Reset	7-50
Example 7-52	Circuit With Six Inferred Flip-Flops	7-52
Example 7-53	Inference Report for Circuit With Six Inferred Flip-Flops	7-53
Example 7-54	Circuit With Three Inferred Flip-Flops	7-55
Example 7-55	Inference Report for Circuit With Three Inferred Flip-Flops	7-55
Example 7-56	Delays in Registers	7-57
Example 7-57	Three-State Inference Report	7-59
Example 7-58	Simple Three-State Driver	7-61
Example 7-59	Inference Report for Simple Three-State Driver	7-61
Example 7-60	Inferring One Three-State Driver From a Single Process	7-62
Example 7-61	Single Process Inference Report	7-62
Example 7-62	Inferring Two Three-State Drivers From Separate Processes	7-64
Example 7-63	Inference Report for Two Three-State Drivers From Separate Processes	7-64
Example 7-64	Inferring a Three-State Driver With Registered Enable	7-66
Example 7-65	Inference Report for Three-State Driver With Registered Enable	7-66
Example 7-66	Three-State Driver Without Registered Enable	7-68
Example 7-67	Inference Report for Three-State Driver Without Registered Enable	7-68

Example 7-68	Incorrect Use of the Z Value in an Expression	7-70
Example 7-69	Correct Use of the Z Value in an Expression	7-70
Example 8-1	Four Logic Blocks	8-2
Example 8-2	Ripple Carry Chain	8-4
Example 8-3	Carry-Lookahead Chain	8-4
Example 8-4	4-Input Adder	8-5
Example 8-5	4-Input Adder Structured With Parentheses	8-6
Example 8-6	Simple Arithmetic Expression	8-7
Example 8-7	Parentheses in an Arithmetic Expression	8-9
Example 8-8	Adding Numbers of Different Bit-Widths	8-11
Example 8-9	Simple Additions With a Common Subexpression . . .	8-12
Example 8-10	Sharing Common Subexpressions—Increases Area .	8-13
Example 8-11	Common Subexpressions	8-14
Example 8-12	Function With One Adder	8-15
Example 8-13	Using Design Knowledge to Simplify an Adder.	8-16
Example 8-14	A Simple State Machine	8-17
Example 8-15	A Better Implementation of a State Machine	8-20
Example 8-16	Equivalent Statements	8-22
Example 8-17	Fully Synchronous Counter With Reset and Enable. .	8-24
Example 8-18	Design With Gated Clock and Asynchronous Reset. .	8-25
Example 8-19	Incorrect Design (Counter With Asynchronous Load). .	8-27
Example 8-20	Incorrect Asynchronous Design With Gated Clock . . .	8-28
Example 8-21	Using don't care Type for Seven-Segment LED Decoder	8-29

Example 8-22	Seven-Segment Decoder Without Don't Care Type . .	8-31
Example 8-23	Fully Specified Variables	8-35
Example 9-1	Using synthesis_on and synthesis_off Directives	9-4
Example A-1	Implementation of a Moore Machine.	A-3
Example A-2	Implementation of a Mealy Machine	A-5
Example A-3	Implementation of a ROM in Random Logic	A-8
Example A-4	Implementation of a Waveform Generator	A-11
Example A-5	Implementation of a Smart Waveform Generator	A-14
Example A-6	MATH Package for Example A-7	A-17
Example A-7	Implementation of a 6-Bit Adder-Subtractor	A-18
Example A-8	Count Zeros—Combinational	A-20
Example A-9	Count Zeros—Sequential	A-22
Example A-10	Soft Drink Machine—State Machine	A-25
Example A-11	Soft Drink Machine—Count Nickels	A-29
Example A-12	Carry-Lookahead Adder	A-35
Example A-13	Serial-to-Parallel Converter—Counting Bits	A-45
Example A-14	Serial-to-Parallel Converter—Shifting Bits	A-48
Example A-15	Programmable Logic Array	A-53
Example B-1	New Function Based on a std_logic_arith Package Function B-5	
Example B-2	UNSIGNED Declarations	B-7
Example B-3	SIGNED Declarations	B-8
Example B-4	Conversion Functions	B-8
Example B-5	Binary Arithmetic Functions	B-11

Example B-6	Unary Arithmetic Functions.	B-12
Example B-7	Using the Carry-Out Bit.	B-13
Example B-8	Ordering Functions	B-14
Example B-9	Equality Functions.	B-14
Example B-10	Shift Functions	B-15
Example B-11	Shift Operations	B-16
Example B-12	Using a built_in pragma	B-17
Example B-13	Built-In AND for Arrays	B-18
Example B-14	Built-In NOT for Arrays	B-19
Example B-15	Use of SYN_FEED_THRU	B-20
Example B-16	numeric_std Conversion Functions.	B-22
Example B-17	numeric_std Resize Function	B-23
Example B-18	numeric_std Binary Arithmetic Functions	B-24
Example B-19	numeric_std Unary Arithmetic Functions	B-24
Example B-20	numeric_std Ordering Functions.	B-25
Example B-21	numeric_std Equality Functions	B-26
Example B-22	numeric_std Logical Operators Functions.	B-26
Example B-23	numeric_std Shift Functions	B-27
Example B-24	numeric_std Rotate Functions	B-28
Example B-25	numeric_std Shift Operators	B-28
Example B-26	Some numeric_std Shift Functions and Shift Operators	B-29
Example B-27	Boolean Reduction Functions	B-30
Example B-28	Boolean Reduction Operations	B-31

Example C-1	Sample Extended Identifiers	C-9
Example C-2	Sample Showing Use of Shift and Rotate Operators .	C-11
Example C-3	Sample Showing Use of xnor Operator.	C-11

1

Using FPGA Compiler II / FPGA *Express* with VHDL

FPGA Compiler II / FPGA *Express* translates a VHDL description to an internal gate-level equivalent format. This format is then optimized for a given FPGA technology.

This chapter contains the following sections:

- Hardware Description Languages
- About VHDL
- FPGA Compiler II / FPGA Express Design Process
- Using FPGA Compiler II / FPGA Express to Compile a VHDL Design
- Design Methodology

The United States Department of Defense, as part of its Very High Speed Integrated Circuit (VHSIC) program, developed VHSIC HDL (VHDL) in 1982. VHDL describes the behavior, function, inputs, and outputs of a digital circuit design. VHDL is similar in style and syntax to modern programming languages, but includes many hardware-specific constructs.

FPGA Compiler II / FPGA *Express* reads and parses the supported VHDL syntax. Appendix C, "VHDL Constructs", lists all VHDL constructs and includes the level of Synopsys support provided for each construct.

Hardware Description Languages

Hardware description languages (HDLs) are used to describe the architecture and behavior of discrete electronic systems.

HDLs were developed to deal with increasingly complex designs. An analogy is often made to the development of software description languages, from machine code (transistors and solder), to assembly language (netlists), to high-level languages (HDLs).

Top-down, HDL-based system design is most useful in large projects, where several designers or teams of designers are working concurrently. HDLs provide structured development. After major architectural decisions have been made, and major components and their connections have been identified, work can proceed independently on subprojects.

Typical uses for HDLs

HDLs typically support a mixed-level description, where structural or netlist constructs can be mixed with behavioral or algorithmic descriptions. With this mixed-level capability, you can describe system architectures at a high level of abstraction; then incrementally refine a design into a particular component-level or gate-level implementation. Alternatively, you can read an HDL design description into FPGA Compiler II / *FPGA Express*, then direct the compiler to synthesize a gate-level implementation automatically.

Advantages of HDLs

A design methodology that uses HDLs has several fundamental advantages over a traditional gate-level design methodology. Among the advantages are the following:

- You can verify design functionality early in the design process and immediately simulate a design written as an HDL description. Design simulation at this higher level, before implementation at the gate level, allows you to test architectural and design decisions.
- FPGA Compiler II / *FPGA Express* provides logic synthesis and optimization, so you can automatically convert a VHDL description to a gate-level implementation in a given technology. This methodology eliminates the former gate-level design bottleneck and reduces circuit design time and errors introduced when hand-translating a VHDL specification to gates.

With FPGA Compiler II / FPGA *Express logic optimization*, you can automatically transform a synthesized design to a smaller and faster circuit. You can apply information gained from the synthesized and optimized circuits back to the VHDL description, perhaps to fine-tune architectural decisions.

- HDL descriptions provide technology-independent documentation of a design and its functionality. An HDL description is more easily read and understood than a netlist or schematic description. Because the initial HDL design description is technology-independent, you can later reuse it to generate the design in a different technology, without having to translate from the original technology.
- VHDL, like most high-level software languages, provides strong *type checking*. A component that expects a four-bit-wide signal type cannot be connected to a three- or five-bit-wide signal; this mismatch causes an error when the HDL description is compiled. If a variable's range is defined as 1 to 15, an error results from assigning it a value of 0. Incorrect use of types has been shown to be a major source of errors in descriptions. Type checking catches this kind of error in the HDL description even before a design is generated.

About VHDL

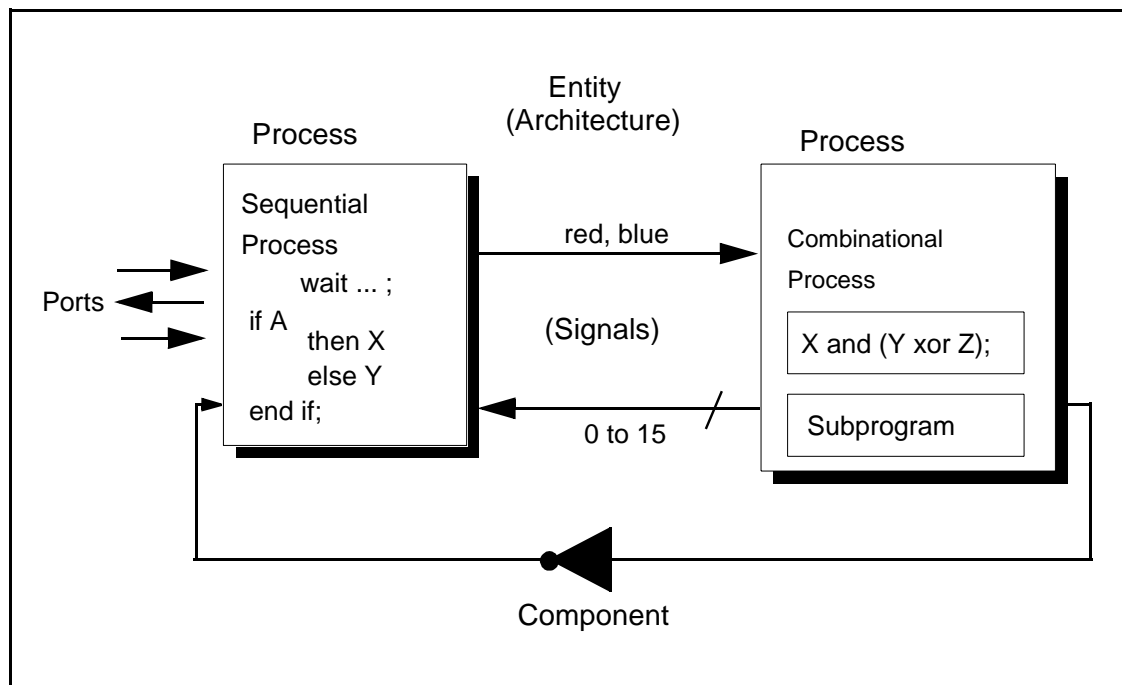
VHDL is one of a few HDLs in widespread use today. VHDL is recognized as a standard HDL by the Institute of Electrical and Electronics Engineers (IEEE Standard 1076, ratified in 1987) and by the United States Department of Defense (MIL-STD-454L).

VHDL divides *entities* (components, circuits, or systems) into an external or visible part (entity name and connections) and an internal or hidden part (entity algorithm and implementation). After you define the external interface to an entity, other entities can use that entity when they all are being developed. This concept of internal and external views is central to a VHDL view of system design. An entity is defined, relative to other entities, by its connections and behavior. You can explore alternate implementations (*architectures*) of an entity without changing the rest of the design.

After you define an entity for one design, you can reuse it in other designs as needed. You can develop libraries of entities for use by many designs or for a family of designs.

The VHDL hardware model is shown in Figure 1-1.

Figure 1-1 VHDL Hardware Model



A VHDL *entity* (design) has one or more input, output, or inout *ports* that are connected (wired) to neighboring systems. An entity is composed of interconnected entities, *processes*, and *components*, all of which operate concurrently. Each entity is defined by a particular *architecture*, which is composed of VHDL constructs such as arithmetic, signal assignment, or component instantiation statements.

In VHDL independent *processes* model sequential (clocked) circuits, using flip-flops and latches, and combinational (unclocked) circuits, using only logic gates. Processes can define and call (*instantiate*) *subprograms* (subdesigns). Processes communicate with each other by *signals* (wires).

A signal has a source (driver), one or more destinations (receivers), and a user-defined *type*, such as “color” or “number between 0 and 15”.

VHDL provides a broad set of constructs. With VHDL, you can describe discrete electronic systems of varying complexity (systems, boards, chips, or modules) with varying levels of abstraction.

VHDL language constructs are divided into three categories by their level of abstraction: *behavioral*, *dataflow*, and *structural*. These categories are described as follows:

behavioral

The functional or algorithmic aspects of a design, expressed in a sequential VHDL process.

dataflow

The view of data as flowing through a design, from input to output. An operation is defined in terms of a collection of data transformations, expressed as concurrent statements.

structural

The view closest to hardware; a model where the components of a design are interconnected. This view is expressed by component instantiations.

FPGA Compiler II / FPGA *Express* Design Process

FPGA Compiler II / FPGA *Express* performs three functions:

- Translates VHDL to an internal format
- Optimizes the block-level representation through various optimization methods
- Maps the design's logical structure for a specific FPGA technology library

FPGA Compiler II / FPGA *Express* synthesizes VHDL descriptions according to the VHDL *synthesis policy* defined in Chapter 2, "Design Descriptions". The Synopsys VHDL synthesis policy has three parts: design methodology, design style, and language constructs. You use the VHDL synthesis policy to produce high quality VHDL-based designs.

Using FPGA Compiler II / FPGA *Express* to Compile a VHDL Design

When a VDL 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 can restructure part or all of the design. You can control the degree of restructuring. Options include:

- Fully preserving the design's hierarchy
- Allowing full modules to be moved up or down in the hierarchy
- Allowing certain modules to be combined with others
- Compressing the entire design into one module (called *flattening* the design) if it is beneficial to do so

The following section describes the design process that uses FPGA Compiler II / FPGA *Express* with a VHDL simulator.

Design Methodology

Figure 1-2 shows a typical design process that uses FPGA Compiler II / FPGA Express and a VHDL simulator.

Figure 1-2 Design Flow

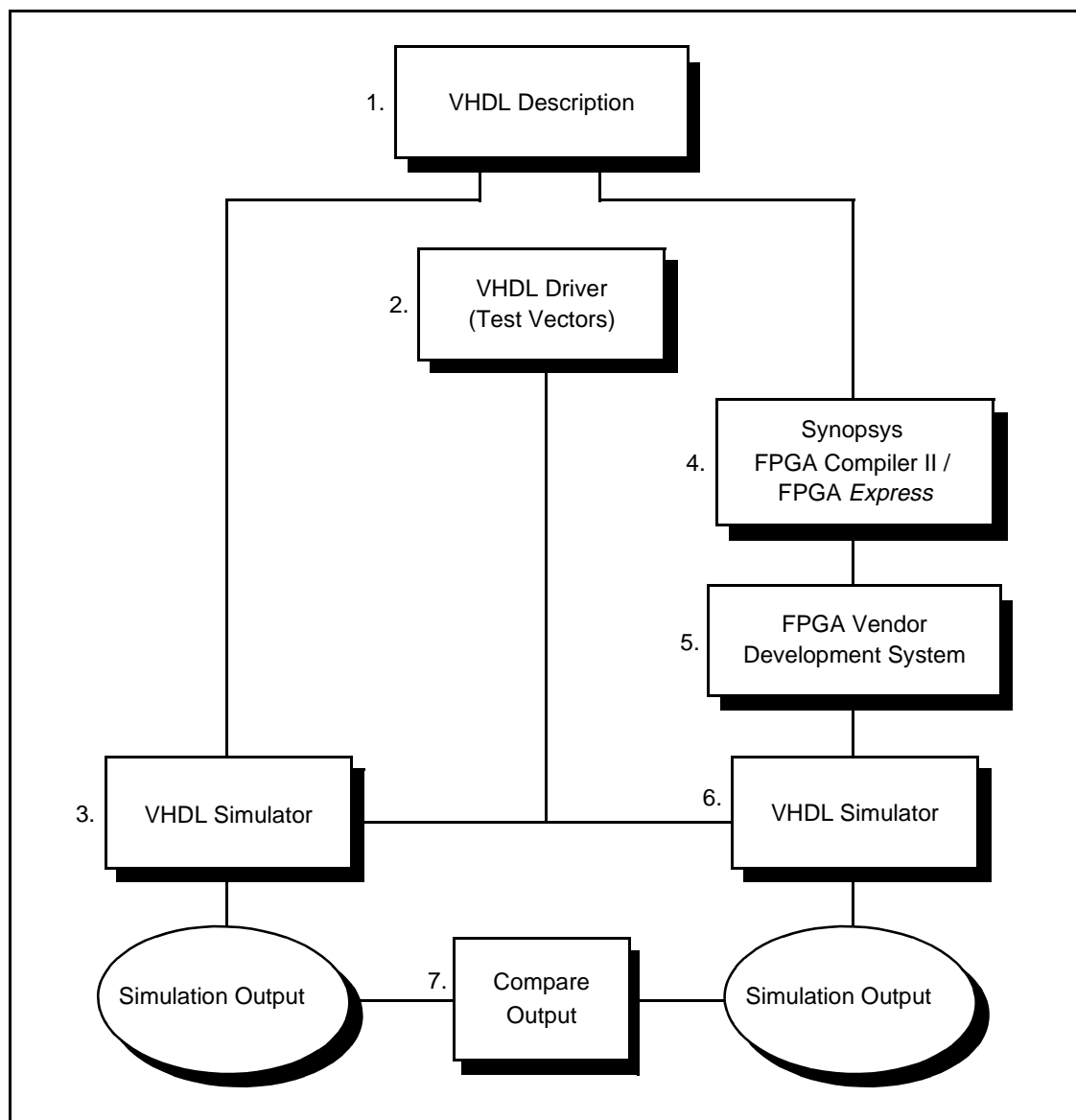


Figure 1-2 illustrates the following steps:

1. Write a design description in VHDL. This description can be a combination of structural and functional elements (as shown in Chapter 2, "Design Descriptions"). This description is used with both FPGA Compiler II / *FPGA Express* and the VHDL simulator.
2. Provide VHDL test drivers for the simulator. For information on writing these drivers, see the appropriate simulator manual. The drivers supply test vectors for simulation and other output data.
3. Simulate the design by using a VHDL simulator. Verify that the description is correct.
4. Use FPGA Compiler II / *FPGA Express* to synthesize and optimize the VHDL design description into a gate-level netlist. FPGA Compiler II / *FPGA Express* generates optimized netlists to satisfy timing constraints for a targeted FPGA architecture.
5. Use your FPGA development system to link the FPGA technology-specific version of the design to the VHDL simulator. The development system includes simulation models and interfaces required for the design flow.
6. Simulate the technology-specific version of the design with the VHDL simulator. You can use the original VHDL 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) against the original VHDL description simulation (step 3) to verify that the implementation is correct.

2

Design Descriptions

Each VHDL structural design can have four parts, which this chapter discusses in the following major sections:

- Entities
- Architecture
- Configurations
- Packages

This chapter also contains the section “Resolution Functions” on page 2-40.

Entities

An entity defines the input and output ports of a design. A design can contain more than one entity. Each entity has its own architecture statement.

The syntax is

```
entity entity_name is [ generic ( generic_declarations );]  
    [ port ( port_declarations ) ;]  
end [ entity_name ] ;
```

entity_name

The name of the entity.

- *generic_declarations* determine local constants used for sizing or timing the entity.
- *port_declarations* determine the number and type of input and output ports.

You cannot use the declaration of other in the entity specification.

An entity serves as an interface to other designs, by defining entity characteristics that must be known to FPGA Compiler II / FPGA *Express* before it can connect the entity to other entities and components.

For example, before you can connect a counter to other entities, you must specify the number and types of its input and output ports, as shown in Example 2-1.

Example 2-1 VHDL Entity Specification

```
entity NAND2 is
  port(A, B: in BIT;      -- Two inputs, A and B
        Z: out BIT);     -- One output, Z = (A and B)'
end NAND2;
```

Entity Generic Specifications

Generic specifications are entity parameters. Generics can specify the bit-widths of components—such as adders—or can provide internal timing values.

A generic can have a default value. It receives a nondefault value only when the entity is instantiated (see “Component Instantiation Statements” on page 2-13) or configured (see “Configurations” on page 2-34). Inside an entity, a generic is a constant value.

The syntax is

```
generic(
  constant_name : type [ := value ]
  { ; constant_name : type [ := value ] }
);
```

constant_name

The name of a generic constant.

- type is a previously defined data type.
- Optional value is the default value of constant_name.

Entity Port Specifications

Port specifications define the number and type of ports in the entity.

The syntax is

```
port(  
  port_name : mode port_type  
  { ; port_name : mode port_type }  
);
```

port_name

The name of the port.

mode

Any of these four values:

in

Can only be read.

out

Can only be assigned a value.

inout

Can be read and assigned a value. The value read is that of the port's incoming value, not the assigned value (if any).

buffer

Similar to out but can be read. The value read is the assigned value. It can have only one driver. For more information about drivers, see "Driving Signals" on page 6-8.

port_type

A previously defined data type.

Example 2-2 shows an entity specification for a 2-input N-bit comparator with a default bit-width of 8.

Example 2-2 Interface for an N-Bit Counter

```
-- Define an entity (design) called COMP
-- that has 2 N-bit inputs and one output.

entity COMP is
  generic(N:  INTEGER := 8);          -- default is 8 bits

  port(X, Y:  in  BIT_VECTOR(0 to N-1);
        EQUAL: out BOOLEAN);
end COMP;
```

Architecture

Architecture, which determines the implementation of an entity, can range in abstraction from an algorithm (a set of sequential statements within a process) to a structural netlist (a set of component instantiations).

The syntax is

```
architecture architecture_name of entity_name is
  { block_declarative_item }
begin
  { concurrent_statement }
end [ architecture_name ] ;
```

architecture_name

The name of the architecture.

entity_name

The name of the entity being implemented.

block_declarative_item

Any of the following statements:

- use statement (see “Package Uses” on page 2-35)
- subprogram declaration (“Subprogram Declarations” on page 2-23)
- subprogram body (“Subprogram Body” on page 2-26)
- type declaration (see “Types” on page 2-31)
- subtype declaration (see “Subtypes” on page 2-32)
- constant declaration (see “Constants” on page 2-18)
- signal declaration (see “Signals” on page 2-21)
- component declaration (see “Subtypes” on page 2-32)
- concurrent statement
Defines a unit of computation that reads signals, performs computations, and assigns values to signals (see “Concurrent Statements” on page 2-17).

Example 2-3 shows a description for a 3-bit counter that contains an entity specification and an architecture statement:

- Entity specification for COUNTER3
- Architecture statement, MY_ARCH

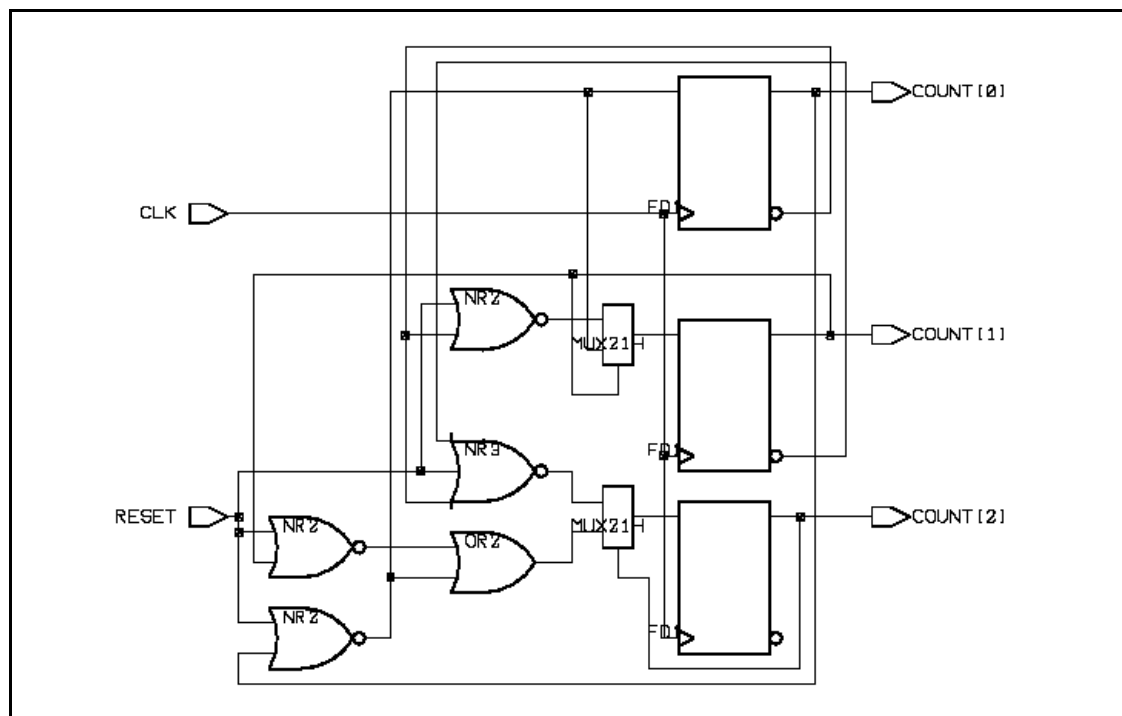
Figure 2-1 shows a schematic of the design.

Example 2-3 An Implementation of a 3-Bit Counter

```
entity COUNTER3 is
port ( CLK : in bit;
      RESET: in bit;
      COUNT: out integer range 0 to 7);
end COUNTER3;
architecture MY_ARCH of COUNTER3 is
signal COUNT_tmp : integer range 0 to 7;

begin
  process
  begin
    wait until (CLK'event and CLK = '1');
    -- wait for the clock
    if RESET = '1' or COUNT_tmp = 7 then
      -- Check for RESET or max. count
      COUNT_tmp <= 0;
    else COUNT_tmp <= COUNT_tmp + 1;
      -- Keep counting
    end if;
  end process;
  COUNT <= COUNT_tmp;
end MY_ARCH;
```

Figure 2-1 3-Bit Counter Synthesized Circuit



Note:

In an architecture, you must not give constants or signals the same name as any of the entity's ports in the entity specification.

If you declare a constant or signal with a port's name, the new declaration hides that port name. If the new declaration lies directly in the architecture declaration (as shown in Example 2-4) and not in an inner block, FPGA Compiler II / *FPGA Express* reports an error.

Example 2-4 Incorrect Use of a Port Name in Declaring Signals or Constants

```
entity X is
  port(SIG, CONST: in  BIT;
        OUT1, OUT2: out BIT);
end X;

architecture EXAMPLE of X is
  signal  SIG  : BIT;
  constant CONST: BIT := '1';
begin
  ...
end EXAMPLE;
```

The error messages generated for Example 2-4 are

```
    signal  SIG  : BIT;
           ^
Error:  (VHDL-1872) line 13
      Illegal redeclaration of SIG.

    constant CONST: BIT := '1';
           ^
Error:  (VHDL-1872) line 14
      Illegal redeclaration of CONST.
```

Declarations

An architecture consists of a declaration section where you declare

- Components
- Concurrent statements
- Constants
- Processes
- Signals
- Subprograms
- Types

Components

If your design consists only of VHDL entity statements, every component declaration in the architecture or package statement has to correspond to an entity.

Components declared in an architecture are local to that architecture.

The syntax is

```
component identifier
    [ generic( generic_declarations ); ]
    [ port( port_declarations ); ]
end component ;
```

identifier

The name of the component.

You cannot use names preceded by GTECH_ for components other than ones provided by Synopsys. However, you can use GTECH to precede a name if it is used without an underscore, as in GTECHBUSTBUF.

generic_declaration

Determines local constants used for sizing or timing the component.

port_declaration

Determines the number and type of input and output ports.

Example 2-5 shows a simple component declaration statement.

Example 2-5 Component Declaration of a 2-Input AND Gate

```
component AND2
  port(I1, I2: in BIT;
        O1:      out BIT);
end component;
```

Example 2-6 shows a component declaration statement that uses a generic parameter.

Example 2-6 Component Declaration of an N-Bit Adder

```
component ADD
  generic(N: POSITIVE);

  port(X, Y:      in  BIT_VECTOR(N-1 downto 0);
        Z:        out BIT_VECTOR(N-1 downto 0);
        CARRY:    out BIT);
end component;
```

The component declaration makes a design entity (AND2 in Example 2-5, ADD in Example 2-6) usable within an architecture. You must declare a component in an architecture or package before you can instantiate it.

Sources of Components

A declared component can come from

- The same VHDL source file
- A different VHDL source file
- Another format, such as EDIF or XNF
- A component from a technology library

Consistency of Component Ports

FPGA Compiler II / *FPGA Express* checks for consistency among its VHDL entities. For other entities, the port names are taken from the original design description, as follows:

- For components in a technology library, the port names are the input and output pin names.
- For EDIF designs, the port names are the EDIF port names.

The bit-widths of each port must match.

- For VHDL components, *FPGA Compiler II / FPGA Express* verifies matching.
- For components from other sources, *FPGA Compiler II / FPGA Express* checks when linking the component to the VHDL description.

Component Instantiation Statements

You use a component instantiation statement to define a design hierarchy or build a netlist in VHDL. A netlist is a structural description of a design.

To form a netlist, use component instantiation statements to instantiate and connect components. A component instantiation statement create a new level of design hierarchy.

The syntax of the component instantiation statement is

```
instance_name : component_name
[ generic map (
    generic_name => expression
    { , generic_name => expression }
) ]
port map (
    [ port_name => ] expression
    { , [ port_name => ] expression }
);
```

instance_name

The name of this instance of component type *component_name*, as in

```
U1 : ADD
```

generic map (optional)

Maps nondefault values onto generics. Each *generic_name* is the name of a generic exactly as declared in the corresponding component declaration statement. Each expression evaluates to an appropriate value.

```
U1 : ADD generic map (N => 4)
```

port map

Maps the component's ports onto connections. Each port_name is the name of a port, exactly as declared in the corresponding component declaration statement. Each expression evaluates to a signal value.

```
U1 : ADD generic map (N => 4)
      port map (X, Y, Z, CARRY) ;
```

FPGA Compiler II / *FPGA Express* uses the following two rules to select which entity and architecture to associate with a component instantiation:

1. Each component declaration must have an entity—a VHDL entity, a design entity from another source or format, or a library component—with the same name. This entity is used for each component instantiation associated with the component declaration.
2. A VHDL entity may have only one architecture associated with it. If multiple architectures are available, add only one of these files to the Design Sources window.

Mapping Generic Values

When you instantiate a component with generics, you can map generics to values. A generic without a default value must be instantiated with a generic map value.

For example, a 4-bit instantiation of the component ADD from Example 2-6 on page 2-11 might use the following generic map:

```
U1:  ADD generic map (N => 4)
      port map (X, Y, Z, CARRY);
```

Mapping Port Connections

The port map maps component ports to actual signals.

Use named or positional association to specify port connections in component instantiation statements, as follows:

- To identify the specific ports of the component, use named association. The `port_name =>` construction identifies the ports.
- To list the component port expressions in the declared port order, use positional association.

Example 2-7 shows named and positional association for the U5 component instantiation statement in Example 2-8.

Example 2-7 Equivalent Named and Positional Association

```
U5: or2 port map (O => n6, I1 => n3, I2 => n1);  
    -- Named association
```

```
U5: or2 port map (n3, n1, n6);  
    -- Positional association
```

Note:

When you use positional association, the instantiated port expressions (signals) must be in the same order as the ports in the component declaration statement.

Example 2-8 shows a structural netlist description for the COUNTER3 design entity from Example 2-3 on page 2-7.

Example 2-8 Structural Description of a 3-Bit Counter

```
architecture STRUCTURE of COUNTER3 is  
    component DFF  
        port(CLK, DATA: in BIT;  
            Q: out BIT);  
    end component;
```

```

component AND2
    port(I1, I2: in BIT;
          O: out BIT);
end component;
component OR2
    port(I1, I2: in BIT;
          O: out BIT);
end component;
component NAND2
    port(I1, I2: in BIT;
          O: out BIT);
end component;
component XNOR2
    port(I1, I2: in BIT;
          O: out BIT);
end component;
component INV
    port(I: in BIT;
          O: out BIT);
end component;

signal N1, N2, N3, N4, N5, N6, N7, N8, N9: BIT;

begin
    u1: DFF port map(CLK, N1, N2);
    u2: DFF port map(CLK, N5, N3);
    u3: DFF port map(CLK, N9, N4);
    u4: INV port map(N2, N1);
    u5: OR2 port map(N3, N1, N6);
    u6: NAND2 port map(N1, N3, N7);
    u7: NAND2 port map(N6, N7, N5);
    u8: XNOR2 port map(N8, N4, N9);
    u9: NAND2 port map(N2, N3, N8);
    COUNT(0) <= N2;
    COUNT(1) <= N3;
    COUNT(2) <= N4;
end STRUCTURE;

```

Concurrent Statements

Each concurrent statement in an architecture defines a unit of computation that

- Reads signals
- Performs a computation that is based on the values of the signals
- Assigns the computed values to the signals

Concurrent statements all compute their values at the same time. Although the order of concurrent statements has no effect on the order in which FPGA Compiler II / *FPGA Express* executes them, concurrent statements coordinate their processing by communicating with each other through signals.

The five kinds of concurrent statements are

Block

Groups a set of concurrent statements.

Component instantiation

Creates an instance of an entity, connecting its interface ports to signals or interface ports of the entity being defined. See “Component Instantiation Statements” on page 2-13.

Procedure call

Calls algorithms that compute and assign values to signals.

Process

Defines sequential algorithms that read the values of signals and compute new values to assign to other signals. For a discussion of processes, see “Processes” on page 2-19.”

Signal assignments

Assign computed values to signals or interface ports.

Concurrent statements are described further in Chapter 6, "Concurrent Statements".

Constants

Constant declarations create named values of a given type. The value of a constant can be read but not changed.

Constant declarations are allowed in architectures, packages, entities, blocks, processes, and subprograms.

Constants declared in an architecture are local to that architecture.

Example 2-9 shows some constant declarations.

Example 2-9 Constant Declarations

```
constant WIDTH: INTEGER := 8;  
constant X      : NEW_BIT := 'X';
```

You can use constants in expressions, as described in "Identifiers" on page 4-23 and "Literals" on page 4-26, and as source values in assignment statements, as described in "Assignment Statements and Targets" on page 5-2.

Processes

A process, which is declared within an architecture, is a concurrent statement. But it is made up of sequentially executed statements that define algorithms. The sequential statements can be any of the following, all of which are discussed in Chapter 5, "Sequential Statements":

- case statement
- exit statement
- if statement
- loop statement
- next statement
- null statement
- Procedure call
- Signal assignment
- Variable assignment
- wait statement

Processes, like all other concurrent statements, read and write signals and the values of interface ports to communicate with the rest of the architecture and with the enclosing system.

Processes are unique in that they behave like concurrent statements to the rest of the design, but they are internally sequential. In addition, only processes can define variables to hold intermediate values in a sequence of computations.

Because the statements in a process are sequentially executed, several constructs, such as if and loop statements, are provided to control the order of execution.

Variable Declarations

Variable declarations define a named value of a given type.

Example 2-10 shows some variable declarations.

Example 2-10 Variable Declarations

```
variable A, B: BIT;  
variable INIT: NEW_BIT;
```

You can use variables in expressions, as described in Chapter 4, "Expressions".

You assign values to variables by using variable assignment statements, as described in "Variable Assignment Statements" on page 5-11.

FPGA Compiler II / *FPGA Express* does not support variable initialization. If you try to initialize a variable, FPGA Compiler II / *FPGA Express* generates the following message:

```
Warning: Initial values for signals are not supported for  
synthesis. They are ignored on line %n (VHDL-2022)
```

Note:

Variables are declared and used only in processes and subprograms, because processes and subprograms cannot declare signals for internal use.

Signals

Signals connect the separate concurrent statements of an architecture to each other, and to other parts of a design, through interface ports.

Signal declarations create new named signals (wires) of a given type. Signals can be given default (initial) values, but these initial values are ignored for synthesis.

Signals with multiple drivers (signals driven by wired logic) can have associated resolution functions, as described in “Resolution Functions” on page 2-40.

Example 2-11 shows two signal declarations.

Example 2-11 Signal Declarations

```
signal A, B: BIT;  
signal INIT: INTEGER := -1;
```

Note:

Ports are also signals, with the restriction that out ports cannot be read and in ports cannot be assigned a value. You create signals either with port declarations or with signal declarations. You create ports only with port declarations.

You can declare signals in architectures, entities, and blocks and can use them in processes and subprograms. Processes and subprograms cannot declare signals for internal use.

You can use signals in expressions, as described in Chapter 5, “Sequential Statements”. Signals are assigned values by signal assignment statements, as described in “Signal Assignment Statements” on page 5-12.

Subprograms

Subprograms use sequential statements to define algorithms and are useful for performing repeated calculations, often in different parts of an architecture (see “Subprograms” on page 5-35). Subprograms declared in an architecture are local to that architecture.

Subprograms differ from processes, in that subprograms cannot directly read or write signals from the rest of the architecture. All communication is through the subprogram's interface. Each subprogram call has its own set of interface signals.

Signal declarations create new named signals (wires) of a given type. Signals can be given default (initial) values, but these initial values are ignored for synthesis.

Signals with multiple drivers (signals driven by wired logic) can have associated resolution functions, as described in “Resolution Functions” on page 2-40.

Subprograms also differ from component instantiation statements, in that the use of a subprogram by an entity or another subprogram does not create a new level of design hierarchy.

There are two types of subprograms, which can have zero or more parameters:

Procedure Subprogram

A procedure returns zero or more values through its interface.

Function Subprogram

A function returns a single value directly.

A subprogram has two parts:

- Declaration
- Body

Note:

When you declare a subprogram in a package, the subprogram declaration must be in the package declaration and the subprogram body must be in the package body.

When you declare a subprogram in an architecture, the program body must be in the architecture body but there is no corresponding subprogram declaration.

Subprogram Declarations

A declaration lists the names and types of the subprogram's parameters and, for functions, the type of the subprogram's return value.

Procedure Declaration Syntax

The syntax of a procedure declaration is

```
procedure proc_name [ ( parameter_declarations ) ] ;
```

proc_name

The name of the procedure.

parameter_declarations

Specify the number and type of input and output ports. The syntax is

```
[ parameter_name      : mode parameter_type  
  { ; parameter_name : mode parameter_type } ]
```

parameter_name

The name of a parameter.

mode

Procedure parameters can be any of these four modes:

in

Can only be read.

out

Can only be assigned a value.

inout

Can be read and assigned a value. The value read is that of the port's incoming value, not the assigned value (if any).

buffer

Similar to out but can be read. The value read is the assigned value. A buffer can have only one driver. For more information about drivers, see "Driving Signals" on page 6-8.

parameter_type

A previously defined data type.

Function Declaration Syntax

The syntax of a function declaration is

```
function func_name [ ( parameter_declarations )  
    return type_name ;
```

func_name

The name of the function.

type_name

The type of the function's returned value. Signal parameters of type range cannot be passed to a subprogram.

parameter_declarations

Specify the number and type of input and output ports. The syntax is

```
[ parameter_name      : mode parameter_type  
  { ; parameter_name : mode parameter_type } ]
```

parameter_name

The name of a parameter.

mode

Function parameters can only use the in mode:

in

Can only be read.

parameter_type

A previously defined data type.

Declaration Examples

Example 2-12 shows sample subprogram declarations for a function and a procedure.

Example 2-12 Two Subprogram Declarations

```
type BYTE    is array (7 downto 0) of BIT;  
type NIBBLE  is array (3 downto 0) of BIT;  
  
function IS_EVEN(NUM: in INTEGER) return BOOLEAN;  
  -- Returns TRUE if NUM is even.  
  
procedure BYTE_TO_NIBBLES(B: in BYTE;  
                          UPPER, LOWER: out NIBBLE);  
  -- Splits a BYTE into UPPER and LOWER halves.
```

When FPGA Compiler II / *FPGA Express* calls a subprogram, it substitutes actual parameters for the declared formal parameters.

Actual parameters are

- Constant values
- Names of signals, variables, constants, or ports

An actual parameter must support the type and mode of the formal parameter. For example, FPGA Compiler II / *FPGA Express* does not accept an input port as an out actual parameter and uses a constant only as an in actual parameter.

Example 2-13 shows some calls to the subprogram declarations from Example 2-12.

Example 2-13 Two Subprogram Calls

```
signal INT : INTEGER;
variable EVEN : BOOLEAN;
. . .
INT <= 7;
EVEN := IS_EVEN(INT);
. . .

variable TOP, BOT: NIBBLE;
. . .
BYTE_TO_NIBBLES("00101101", TOP, BOT);
```

Subprogram Body

A subprogram body defines an implementation of a subprogram's algorithm.

Procedure Body Syntax

The syntax of a procedure body is

```
procedure procedure_name [ (parameter_declarations) ] is
    { subprogram_declarative_item }
begin
    { sequential_statement }
end [ procedure_name ] ;
```

procedure_name

Name of the procedure

subprogram_declarative_item

A *subprogram_declarative_item* can be any of the following statements:

- use clause
- type declaration
- subtype declaration
- constant declaration
- variable declaration
- attribute declaration
- attribute specification
- subprogram declaration (for local or nested subprograms)
- subprogram body (for locally declared subprograms)

Function Body Syntax

The syntax of a function body is

```
function function_name [ (parameter_declarations) ]  
    return type_name is  
    { subprogram_declarative_item }  
begin  
    { sequential_statement }  
end [ function_name ] ;
```

function_name

Name of the function

subprogram_declarative_item

A *subprogram_declarative_item* can be any of the following statements:

- use clause
- type declaration
- subtype declaration
- constant declaration
- variable declaration
- attribute declaration
- attribute specification
- subprogram declaration (for local or nested subprograms)
- subprogram body (for locally declared subprograms)

Example 2-14 shows subprogram bodies for the sample subprogram declarations in Example 2-12 on page 2-25.

Example 2-14 Two Subprogram Bodies

```
function IS_EVEN(NUM: in INTEGER)
    return BOOLEAN is
begin
    return ((NUM rem 2) = 0);
end IS_EVEN;
procedure BYTE_TO_NIBBLES(B: in BYTE;
                           UPPER, LOWER: out NIBBLE) is
begin
    UPPER := NIBBLE(B(7 downto 4));
    LOWER := NIBBLE(B(3 downto 0));
end BYTE_TO_NIBBLES;
```

Subprogram Overloading

You can overload subprograms, which means that one or more subprograms can have the same name. Each subprogram that uses a given name must have a different parameter profile.

A parameter profile specifies a subprogram's number and type of parameters. This information determines which subprogram is called when more than one subprogram has the same name. Overloaded functions are also distinguished by the type of their return values.

Example 2-15 shows two subprograms with the same name (IS_ODD) but different parameter profiles.

Example 2-15 Subprogram Overloading

```
type SMALL is range 0 to 100;
type LARGE is range 0 to 10000;

function IS_ODD(NUM: SMALL) return BOOLEAN;
function IS_ODD(NUM: LARGE) return BOOLEAN;

signal A_NUMBER: SMALL;
signal B: BOOLEAN;
. . .
B <= IS_ODD(A_NUMBER); -- Will call the first function above
```

Operator Overloading

You can overload predefined operators such as +, and, and mod. By using overloading, you can adapt predefined operators to work with your own data types.

For example, you can declare new logic types rather than use the predefined types BIT and INTEGER. However, you cannot use predefined operators with these new types unless you overload the operators for the types.

Example 2-16 shows how some predefined operators are overloaded for a new logic type.

Example 2-16 Operator Overloading

```
type NEW_BIT is ('0', '1', 'X');
    -- New logic type

function "and"(I1, I2: in NEW_BIT) return NEW_BIT;
function "or" (I1, I2: in NEW_BIT) return NEW_BIT;
    -- Declare overloaded operators for new logic type
. . .
signal A, B, C: NEW_BIT;
. . .

C <= (A and B) or C;
```

VHDL requires that overloaded operator declarations enclose the operator name or symbol in double quotation marks, because operator name and symbol are infix operators (they are used between operands). If you declare the overloaded operators without quotation marks, a VHDL tool considers them functions rather than operators.

Variable Declarations

Variable declarations define a named value of a given type.

You can use variables in expressions, as described in “Identifiers” on page 4-23 and “Literals” on page 4-26. You assign values to variables by using variable assignment statements, as described in “Variable Assignment Statements” on page 5-11.”

FPGA Compiler II / FPGA *Express* does not support variable initialization. If you try to initialize a variable, FPGA Compiler II / FPGA *Express* generates the following message:

```
Warning: Initial values for signals are not supported for  
synthesis. They are ignored on line %n (VHDL-2022)
```

Example 2-17 shows some variable declarations.

Example 2-17 Variable Declarations

```
variable A, B: BIT;  
variable INIT: NEW_BIT;
```

Note:

Variables are declared and used only in processes and subprograms, because processes and subprograms cannot declare signals for internal use.

To use these declarations in more than one entity or architecture, place them in a package, as described in “Packages” on page 2-35.

Types

You declare each signal with a type that determines the kind of data it carries. Types declared in an architecture are local to that architecture.

You can use type declarations in architectures, packages, entities, blocks, processes, and subprograms.

Type declarations define the name and characteristics of a type. Types and type declarations are fully described in Chapter 3, "Data Types". A type is a named set of values, such as the set of integers or the set of colors (red, green, and blue). An object of a given type, such as a signal, can have any value of that type.

You can see an example of a type declaration for type `NEW_BIT` in Example 2-16 on page 2-30.

Subtypes

Use subtype declarations to define the name and characteristics of a constrained subset of another type or subtype. A subtype is fully compatible with its parent type, but only over the subtype's range.

The following subtype declaration (`NEW_LOGIC`) is a subrange of the type declaration in Example 2-16 on page 2-30.

```
subtype NEW_LOGIC is NEW_BIT range '0' to '1';
```

You can use subtype declarations wherever you use type declarations: in architectures, packages, entities, blocks, processes, and subprograms.

Examples of Architectures for NAND2 Entity

Example 2-18, Example 2-19, and Example 2-20 show three different architectures for the entity NAND2. The three examples define equivalent implementations of NAND2. After optimization and synthesis, they all produce the same circuit, a 2-input NAND gate in the target technology. The architecture description style you use for this entity depends on your own preferences.

Example 2-18 shows how the entity NAND2 can be implemented by using two components from a technology library. The entity inputs A and B are connected to AND gate U0, producing an intermediate I signal. Signal I is then connected to inverter U1, producing the entity output Z.

Example 2-18 Structural Architecture for Entity NAND2

```
architecture STRUCTURAL of NAND2 is
    signal I: BIT;

    component AND_2          -- From a technology library
        port(I1, I2: in BIT;
              O1: out BIT);
    end component;

    component INVERT         -- From a technology library
        port(I1: in BIT;
              O1: out BIT);
    end component;

begin
    U0: AND_2 port map (I1 => A, I2 => B, O1 => I);
    U1: INVERT port map (I1 => I, O1 => Z);
end STRUCTURAL;
```

Example 2-19 shows how you can define the entity NAND2 by its logical function.

Example 2-19 Data Flow Architecture for Entity NAND2

```
architecture DATAFLOW of NAND2 is
begin
    Z <= A nand B;
end DATAFLOW;
```

Example 2-20 shows another implementation of NAND2.

Example 2-20 RTL Architecture for Entity NAND2

```
architecture RTL of NAND2 is
begin
    process(A, B)
    begin
        if (A = '1') and (B = '1') then
            Z <= '0';
        else
            Z <= '1';
        end if;
    end process;
end RTL;
```

Configurations

Configurations are not currently supported by FPGA Compiler II /
FPGA Express.

Packages

A package is a collection of declarations that more than one design can use.

You can collect constants, data types, component declarations, and subprograms into a VHDL package that can then be used by more than one design or entity.

A package must contain at least one of the following constructs:

Constant

Declares systemwide parameters, such as data-path widths.

VHDL data type declaration

Defines data types used throughout a design. All entities in a design must use common interface types, such as common address bus types.

Component declaration

Specifies the interfaces to entities that can be instantiated in the design.

Subprogram

Defines algorithms that can be called anywhere in a design.

Packages are often sufficiently general that they are usable in many different designs. For example, the `std_logic_1164` package defines data types `std_logic` and `std_logic_vector`.

Package Uses

The use statement allow an entity to use the declarations in a package.

The syntax is

```
use LIBRARY_NAME.PACKAGE_NAME.ALL;
```

LIBRARY_NAME

The name of a VHDL library.

PACKAGE_NAME

The name of the included package.

A use statement is usually the first statement in a package or entity specification source file.

Note:

Synopsys does not support different packages with the same name when they exist in different libraries. No two packages can have the same name.

Package Structure

Packages have two parts: the declaration and the body.

Package declaration

Holds public information, including constant, type, and subprogram declarations.

Package body

Holds private information, including local types and subprogram implementations (bodies).

Note:

When a package declaration contains subprogram declarations, a corresponding package body must define the subprogram bodies.

Package Declarations

Package declarations collect information that are needed by one or more entities in a design. This information includes data type declarations, signal declarations, subprogram declarations, and component declarations.

Note:

Signals declared in packages cannot be shared across entities. If two entities both use a signal from a given package, each entity has its own copy of that signal.

Although you can declare all this information explicitly in each design entity or architecture in a system, it is often easier to declare system information in a separate package. Each design entity in the system can then use the system's package.

The syntax of a package declaration is

```
package package_name is
    { package_declarative_item }
end [ package_name ] ;
```

package_name

The name of this package.

package_declarative_item

Any of the following statements:

- use clause (to include other packages)
- type declaration
- subtype declaration
- constant declaration

- signal declaration
- subprogram declaration
- component declaration

Example 2-21 shows some package declarations.

Example 2-21 Sample Package Declarations

```
package EXAMPLE is

    type BYTE is range 0 to 255;
    subtype NIBBLE is BYTE range 0 to 15;

    constant BYTE_FF: BYTE := 255;

    signal ADDEND: NIBBLE;

    component BYTE_ADDER
        port(A, B:          in BYTE;
             C:             out BYTE;
             OVERFLOW: out BOOLEAN);
    end component;

    function MY_FUNCTION (A: in BYTE) return BYTE;

end EXAMPLE;
```

To use the previous example declarations, add a use statement at the beginning of your design description as follows:

```
use WORK.EXAMPLE.ALL;
entity . . .
architecture . . .
```

Appendix B, "Synopsys Packages", contains more examples of packages and their declarations.

Package Body

A package body includes

- The implementations (bodies) of subprograms declared in the package declaration
- Internal support subprograms

But designs or entities that use the package never see this information.

The syntax of a package body is

```
package body package_name is {  
    { package_body_declarative_item }  
end [ package_name ] ;
```

package_name

The name of the associated package.

package_body_declarative_item

Any of the following statements:

- use clause
- subprogram declaration
- subprogram body
- type declaration
- subtype declaration
- constant declaration

Appendix B, "Synopsys Packages", shows a package declaration and body example that comes with FPGA Compiler II / *FPGA Express*.

Resolution Functions

Resolution functions are used with signals that can be connected (wired together). For example, if two drivers directly connect to a signal, the resolution function determines whether the signal value is the AND, OR, or three-state function of the driving values.

Use resolution functions to assign the driving values when there are multiple drivers. For simulation, you can write an arbitrary function to resolve bus conflicts.

Note:

A resolution function might change the value of a resolved signal even if all drivers have the same value.

The resolution function for a signal is part of that signal's subtype declaration. You create a resolved signal in four steps:

1. Declare the signal's base type.

```
type SIGNAL_TYPE is ...  
-- signal's base type is SIGNAL_TYPE
```

2. Declare the resolution function.

```
function res_function (DATA: ARRAY_TYPE)  
  return SIGNAL_TYPE is  
-- declaration of the resolution function  
-- ARRAY_TYPE must be an unconstrained array of  
-- SIGNAL_TYPE
```

3. Declare the resolved signal's subtype as a subtype of the base type, which includes the name of the resolution function.

```
subtype res_type is res_function SIGNAL_TYPE;  
-- name of the subtype is res_type  
-- name of function is res_function  
-- signal type is res_type (a subtype of SIGNAL_TYPE)
```

4. Declare resolved signals as resolved subtypes.

```
signal resolved_signal_name:res_type;  
-- resolved_signal_name is a resolved signal
```

FPGA Compiler II / *FPGA Express* does not support arbitrary resolution functions. Only wired AND, wired OR, and three-state functions are allowed. FPGA Compiler II / *FPGA Express* requires that you mark all resolution functions with a special directive indicating the kind of resolution being performed.

FPGA Compiler II / *FPGA Express* considers the directive only when creating hardware. The body of the resolution function is parsed but ignored; using unsupported VHDL constructs generates errors (see Appendix C, "VHDL Constructs").

Do not connect signals that use different resolution functions. FPGA Compiler II / *FPGA Express* supports only one resolution function per network.

The three resolution function directives are

- `synopsys resolution_method wired_and`
- `synopsys resolution_method wired_or`
- `synopsys resolution_method three_state`

Pre-synthesis and post-synthesis simulation results might not match if the body of the resolution function the simulator uses does not match the directive the synthesizer uses.

Example 2-22 shows how to create and use a resolved signal and how to use compiler directives for resolution functions. The signal's base type is the predefined type BIT. Figure 2-2 shows the design.

Example 2-22 Resolved Signal and Its Resolution Function

```
package RES_PACK is
  function RES_FUNC(DATA: in BIT_VECTOR) return BIT;
  subtype RESOLVED_BIT is RES_FUNC BIT;
end;

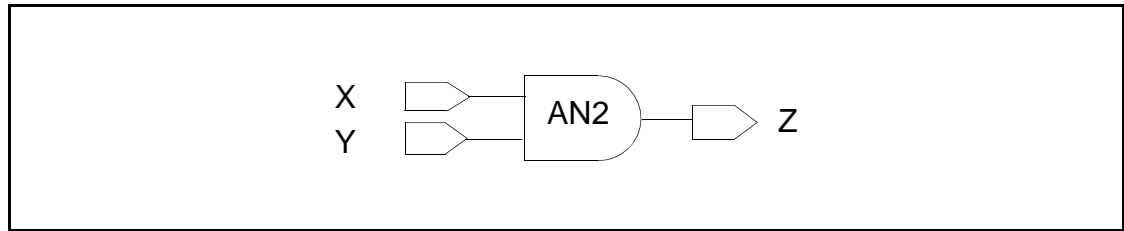
package body RES_PACK is
  function RES_FUNC(DATA: in BIT_VECTOR) return BIT is
    -- synopsys resolution_method wired_and
  begin
    -- The code in this function is ignored by the program
    -- but parsed for correct VHDL syntax

    for I in DATA'range loop
      if DATA(I) = '0' then
        return '0';
      end if;
    end loop;
    return '1';
  end;
end;

use work.RES_PACK.all;
entity WAND_VHDL is
  port(X, Y: in BIT; Z: out RESOLVED_BIT);
end WAND_VHDL;

architecture WAND_VHDL of WAND_VHDL is
begin
  Z <= X;
  Z <= Y;
end WAND_VHDL;
```

Figure 2-2 Design Using Resolved Signal



3

Data Types

VHDL is a strongly typed language. Every constant, signal, variable, function, and parameter is declared with a type, such as `BOOLEAN` or `INTEGER`, and can hold or return only a value of that type.

VHDL predefines abstract data types such as `BOOLEAN`, which are part of most programming languages, and hardware-related types, such as `BIT`, which are found in most hardware languages. VHDL predefined types are declared in the `STANDARD` package supplied with all VHDL implementations (see Example 3-17 on page 3-17). This chapter includes information about

- Enumeration Types
- Integer Types
- Array Types
- Record Types

- Predefined VHDL Data Types
- Unsupported Data Types
- Synopsys Data Types
- Subtypes

The advantage of strong typing is that VHDL tools can detect many common design errors, such as assigning an 8-bit value to a 4-bit-wide signal, or detect incrementing of an array index out of its range.

The following code shows the definition of a new type, `BYTE`, as an array of 8 bits and a variable declaration, `ADDEND`, which uses this type.

```
type BYTE is array(7 downto 0) of BIT;  
  
variable ADDEND: BYTE;
```

The predefined VHDL data types are built from the basic VHDL data types. Some VHDL types, such as `REAL` and `FILE`, are not supported for synthesis.

The examples in this chapter show type definitions and associated object declarations. Although each constant, signal, variable, function, and parameter is declared with a type, only variable and signal declarations are shown in the examples. For more information about constant, function, and parameter declarations, see “Declarations” on page 2-10.

VHDL also provides subtypes, which are defined as subsets of other types. Anywhere a type definition can appear, a subtype definition can also appear. The difference between a type and a subtype is that a subtype is a subset of a previously defined parent (or base) type or subtype. Overlapping subtypes of a given base type can be compared against and assigned to each other. All integer types, for example, are technically subtypes of the built-in integer base type (see “Integer Types” on page 3-8 and “Subtypes” on page 3-21).

Enumeration Types

You define an enumeration type by listing (enumerating) all possible values of that type.

The syntax of an enumeration type definition is

```
type type_name is ( enumeration_literal {, enumeration_literal} );
```

type_name

An identifier.

Each *enumeration_literal* is either an identifier (*enum_6*) or a character literal ('A').

An identifier is a sequence of letters, underscores, and numbers. It must start with a letter and cannot be a VHDL reserved word, such as TYPE. All VHDL reserved words are listed in “VHDL Reserved Words” on page C-17.

A character literal is any value of type CHARACTER, in single quotation marks.

Example 3-1 shows two enumeration type definitions and corresponding variable and signal declarations.

Example 3-1 Enumeration Type Definitions

```
type COLOR is (BLUE, GREEN, YELLOW, RED);
type MY_LOGIC is ('0', '1', 'U', 'Z');
variable HUE: COLOR;
signal SIG: MY_LOGIC;
. . .
HUE := BLUE;
SIG <= 'Z';
```

Enumeration Overloading

You can overload an enumeration literal by including it in the definition of two or more enumeration types. When you use such an overloaded enumeration literal, FPGA Compiler II / *FPGA Express* is usually able to determine the literal's type. However, under certain circumstances determination might be impossible. In these cases, you must qualify the literal by explicitly stating its type. (See “Enumeration Literals” on page 4-27.) Example 3-2 shows how you can qualify an overloaded enumeration literal.

Example 3-2 Enumeration Literal Overloading

```
type COLOR is (RED, GREEN, YELLOW, BLUE, VIOLET);
type PRIMARY_COLOR is (RED, YELLOW, BLUE);
. . .
A <= COLOR'(RED);
```

Enumeration Encoding

Enumeration types are ordered by enumeration value. By default, the first enumeration literal is assigned the value 0, the next enumeration literal is assigned the value 1, and so forth.

FPGA Compiler II / FPGA *Express* automatically encodes enumeration values into bit vectors that are based on each value's position. The length of the encoding bit vector is the minimum number of bits required to encode the number of enumerated values. For example, an enumeration type with five values would have a 3-bit encoding vector.

Example 3-3 shows the default encoding of an enumeration type with five values.

Example 3-3 Automatic Enumeration Encoding

```
type COLOR is (RED, GREEN, YELLOW, BLUE, VIOLET);
```

The enumeration values are encoded as follows:

```
RED      = "000"  
GREEN    = "001"  
YELLOW   = "010"  
BLUE     = "011"  
VIOLET   = "100"
```

The result is RED < GREEN < YELLOW < BLUE < VIOLET.

You can override the automatic enumeration encodings and specify your own enumeration encodings with the ENUM_ENCODING attribute. This interpretation is specific to FPGA Compiler II / FPGA *Express*.

A VHDL attribute is defined by its name and type and is then declared with a value for the attributed type, as shown in Example 3-4.

Several VHDL synthesis-related attributes are declared in the ATTRIBUTES package supplied with FPGA Compiler II / FPGA *Express*. For more information about this package, see "ATTRIBUTES Package" on page B-31.

The ENUM_ENCODING attribute must be a string containing a series of vectors, one for each enumeration literal in the associated type. The encoding vector is specified by '0's, '1's, 'D's, 'U's, and 'Z's, separated by blank spaces. The meaning of these encoding vectors is described in the next section. The first vector in the attribute string specifies the encoding for the first enumeration literal, the second vector specifies the encoding for the second enumeration literal, and so on. The ENUM_ENCODING attribute must immediately follow the type declaration.

Example 3-4 illustrates how the default encodings from Example 3-3 can be changed with the ENUM_ENCODING attribute.

Example 3-4 Using the ENUM_ENCODING Attribute

```
attribute ENUM_ENCODING: STRING;
-- Attribute definition

type COLOR is (RED, GREEN, YELLOW, BLUE, VIOLET);
attribute ENUM_ENCODING of
  COLOR: type is "010 000 011 100 001";
-- Attribute declaration
```

The enumeration values are encoded as follows:

```
RED      = "010"
GREEN    = "000"
YELLOW   = "011"
BLUE     = "100"
VIOLET   = "001"
```

The result is GREEN < VIOLET < RED < YELLOW < BLUE.

Note:

The interpretation of the ENUM_ENCODING attribute is specific to FPGA Compiler II / FPGA Express. Other VHDL tools, such as simulators, use the standard encoding (ordering).

Enumeration Encoding Values

The possible encoding values for the ENUM_ENCODING attribute are '0', '1', 'D', 'U', and 'Z':

'0'

Bit value '0'.

'1'

Bit value '1'.

'D'

Don't care (can be either '0' or '1'). To use don't care information, see "Don't Care Inference" on page 8-29.

'U'

Unknown. If 'U' appears in the encoding vector for an enumeration, you cannot use that enumeration literal except as an operand to the = and /= operators. You can read an enumeration literal encoded with a 'U' from a variable or signal, but you cannot assign it.

For synthesis, the = operator returns false and /= returns true when either of the operands is an enumeration literal whose encoding contains 'U'.

'Z'

High impedance. See "Three-State Inference" on page 7-59 for more information.

Integer Types

The maximum range of a VHDL integer type is $-(2^{31}-1)$ to $2^{31}-1$ (-2_147_483_647 ... 2_147_483_647). Integer types are defined as subranges of this anonymous built-in type. Multidigit numbers in VHDL can include underscores (_) to make them easier to read.

FPGA Compiler II / *FPGA Express* encodes an integer value as a bit vector whose length is the minimum necessary to hold the defined range. FPGA Compiler II / *FPGA Express* encodes integer ranges that include negative numbers as 2's-complement bit vectors.

The syntax of an integer type definition is

```
type type_name is range integer_range ;
```

type_name is the name of the new integer type, and *integer_range* is a subrange of the anonymous integer type.

Example 3-5 shows some integer type definitions.

Example 3-5 Integer Type Definitions

```
type PERCENT is range -100 to 100;
  -- Represented as an 8-bit vector
  --   (1 sign bit, 7 value bits)

type INTEGER is range -2147483647 to 2147483647;
  -- Represented as a 32-bit vector
  --   This is the definition of the INTEGER type
```


You cannot directly access the bits of an INTEGER or explicitly state the bit-width of the type. For these reasons, Synopsys provides overloaded functions for arithmetic. These functions are defined in the `std_logic_signed` and `std_logic_unsigned` packages, described in “`std_logic_arith` Package” on page B-3.

Array Types

An array is an object that is a collection of elements of the same type. VHDL supports N-dimensional arrays, but FPGA Compiler II / FPGA *Express* supports only one-dimensional arrays. Array elements can be of any type. An array has an index whose value selects each element. The index range determines how many elements are in the array and their ordering (low to high or high downto low). An index can be of any integer type.

You can declare multidimensional arrays by building one-dimensional arrays where the element type is another one-dimensional array, as shown in Example 3-6.

Example 3-6 Declaration of Array of Arrays

```
type BYTE    is array (7 downto 0) of BIT;
type VECTOR is array (3 downto 0) of BYTE;
```

VHDL provides constrained as well as unconstrained arrays. The difference between the two comes from the index range in the array type definition.

Constrained Arrays

A constrained array's index range is explicitly defined; an example is the integer range (1 to 4). When you declare a variable or signal of the type constrained array, the variable or signal has the same index range as the constrained array.

The syntax of a constrained array type definition is

```
type array_type_name is array ( integer_range ) of type_name;
```

array_type_name

The name of the new constrained array type.

integer_range

A subrange of another integer type.

type_name

The type of each array element.

Example 3-7 shows a constrained array type definition.

Example 3-7 Constrained Array Type Definition

```
type BYTE is array (7 downto 0) of BIT;  
-- A constrained array whose index range is  
-- (7, 6, 5, 4, 3, 2, 1, 0)
```

Unconstrained Arrays

You define an unconstrained array's index range as a type; for example, INTEGER. This definition implies that the index range can be any contiguous subset of that type's values. When you declare an array variable or signal of this type, you also define its actual index range. Different declarations can have different index ranges.

The syntax of an unconstrained array type definition is

```
type array_type_name is
    array (range_type_name range <>)
        of element_type_name ;
```

array_type_name

The name of the new unconstrained array type.

range_type_name

The name of a range type or subtype.

element_type_name

The type of each array element.

Example 3-8 shows an unconstrained array type definition and a declaration that uses it.

Example 3-8 Unconstrained Array Type Definition

```
type BIT_VECTOR is array(INTEGER range <>) of BIT;
    -- An unconstrained array definition
. . .
variable MY_VECTOR : BIT_VECTOR(5 downto -5);
```

The advantage of using unconstrained arrays is that a VHDL tool can recall the index range of each declaration. You can use array attributes to determine the range (bounds) of a signal or variable of an unconstrained array type. With this information, you can write routines that use variables or signals of an unconstrained array type, independent of any one array variable's or signal's bounds. The next section describes array attributes and how they are used.

Array Attributes

FPGA Compiler II / FPGA *Express* supports the following predefined VHDL attributes for use with arrays:

- left
- right
- high
- low
- length
- range
- reverse_range

These attributes all return a value corresponding to part of an array's range. Table 3-1 shows the values of the array attributes for the variable MY_VECTOR in Example 3-8.

Table 3-1 Array Index Attributes

Attribute Expression	Value
MY_VECTOR'left	5
MY_VECTOR'right	–5
MY_VECTOR'high	5
MY_VECTOR'low	–5
MY_VECTOR'length	11
MY_VECTOR'range	(5 downto –5)
MY_VECTOR'reverse_range	(–5 to 5)

Example 3-9 shows the use of array attributes in a function that ORs together all elements of a given bit vector (declared in Example 3-8) and returns that value.

Example 3-9 Use of Array Attributes

```
function OR_ALL (X: in BIT_VECTOR) return BIT is
  variable OR_BIT: BIT;
begin
  OR_BIT := '0';
  for I in X'range loop
    OR_BIT := OR_BIT or X(I);
  end loop;

  return OR_BIT;
end;
```

Note:

This function works for a bit vector of any size.

Record Types

A record is a set of named fields of various types, unlike an array, which is composed of identical anonymous entries. A record's field can be of any previously defined type, including another record type.

Example 3-10 shows a record type declaration (BYTE_AND_IX), three signals of that type, and some assignments.

Example 3-10 Record Type Declaration and Use

```
constant LEN: INTEGER := 8;

subtype BYTE_VEC is BIT_VECTOR(LEN-1 downto 0);

type BYTE_AND_IX is
  record
    BYTE: BYTE_VEC;
    IX:   INTEGER range 0 to LEN;
  end record;

signal X, Y, Z: BYTE_AND_IX;
```

```

signal DATA: BYTE_VEC;
signal NUM:  INTEGER;
. . .

X.BYTE <= "11110000";
X.IX   <= 2;

DATA <= Y.BYTE;
NUM  <= Y.IX;

Z <= X;

```

As shown in Example 3-10, you can read values from or assign values to records in two ways:

- By individual field name

```

X.BYTE <= DATA;
X.IX   <= LEN;

```

- From another record object of the same type

```

Z <= X;

```

A record type object's individual fields are accessed by the object name, a period, and a field name: X.BYTE or X.IX. To access an element of the BYTE field's array, use array notation: X.BYTE(2).

Record Aggregates

Record aggregates (constants) have the same syntax as array aggregates (see "Aggregates" on page 4-18). They can appear anywhere records appear.

The following line illustrates a named record aggregate in a description:

```
X <= (BYTE => "11110000", IX => 2);
```

The following line illustrates a positional record aggregate in a description:

```
X <= ("11110000", 2);
```

You can use the others construct in a named or positional record aggregate, just as you can in an array aggregate (see “Aggregates” on page 4-18).

You can mix named and positional aggregates in a description, with the positional items listed first.

You cannot have a named item that refers to a field covered in the positional aggregate. The following four examples illustrate this caveat.

Example 3-11 Simple Record Type

```
type rec is
  record
    a: integer;
    b: integer;
    c: integer;
    d: integer;
    e: integer;
  end record
end
```

Example 3-12 Named Aggregate for Example 3-11

```
(a => 1, b => 2, c => 0, d => 3, e => 0)
```

In a named aggregate, the items can appear in any order.

Example 3-13 Use of others in an Aggregate

```
(1, 2, d => 3, others => 0)
```

Example 3-13 is equivalent to Example 3-12 or Example 3-14.

Example 3-14 Positional Aggregate

```
(1, 2, 0, 3, 0)
```

You can supply a set of choices in a description of a record aggregate, but a choice cannot be a range. See Example 3-15 and Example 3-16.

Example 3-15 Record Aggregate Equivalent to Example 3-16

```
(b => 2, c => 2, d => 2, a => 1, e => 3)
```

Example 3-16 Record Aggregate With Set of Choices

```
(b | c | d => 2, a => 1, e => 3)
```

Predefined VHDL Data Types

IEEE VHDL describes two site-specific packages, each containing a standard set of types and operations: the STANDARD package and the TEXTIO package.

The STANDARD package of data types is included in all VHDL source files by an implicit use clause. The TEXTIO package defines types and operations for communication with a standard programming environment (terminal and file I/O). This package is not needed for synthesis; therefore, FPGA Compiler II / FPGA Express does not support it.

The FPGA Compiler II / FPGA *Express* implementation of the STANDARD package is listed in Example 3-17. This STANDARD package is a subset of the IEEE VHDL STANDARD package. Differences are described in “Unsupported Data Types” on page 3-20.

Example 3-17 *FPGA Compiler II / FPGA Express STANDARD Package*

```
package STANDARD is
  type BOOLEAN is (FALSE, TRUE);
  type BIT is ('0', '1');
  type CHARACTER is (
    NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
    BS, HT, LF, VT, FF, CR, SO, SI,
    DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
    CAN, EM, SUB, ESC, FSP, GSP, RSP, USP,

    ' ', '!', '"', '#', '$', '%', '&', '\'',
    '(', ')', '*', '+', ',', '-', '.', '/',
    '0', '1', '2', '3', '4', '5', '6', '7',
    '8', '9', ':', ';', '<', '=', '>', '?',

    '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
    'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
    'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
    'X', 'Y', 'Z', '[', '\', ']', '^', '_',

    '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
    'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
    'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
    'x', 'y', 'z', '{', '|', '}', '~', DEL);

  type INTEGER is range -2147483647 to 2147483647;
  subtype NATURAL is INTEGER range 0 to 2147483647;
  subtype POSITIVE is INTEGER range 1 to 2147483647;
  type STRING is array (POSITIVE range <>)
    of CHARACTER;
  type BIT_VECTOR is array (NATURAL range <>)
    of BIT;
end STANDARD;
```

Data Type BOOLEAN

The BOOLEAN data type is actually an enumerated type with two values, false and true, where false < true. Logical functions, such as equality (=) and comparison (<) functions, return a BOOLEAN value.

Convert a BIT value to a BOOLEAN value as follows:

```
BOOLEAN_VAR := (BIT_VAR = '1');
```

Data Type BIT

The BIT data type represents a binary value as one of two characters, '0' or '1'. Logical operations such as “and” can take and return BIT values.

Convert a BOOLEAN value to a BIT value as follows:

```
if (BOOLEAN_VAR) then
    BIT_VAR := '1';
else
    BIT_VAR := '0';
end if;
```

Data Type CHARACTER

The CHARACTER data type enumerates the ASCII character set. Nonprinting characters are represented by a three-letter name, such as NUL for the null character. Printable characters are represented by themselves, in single quotation marks, as follows:

```
variable CHARACTER_VAR: CHARACTER;
. . .
CHARACTER_VAR := 'A';
```

Data Type INTEGER

The INTEGER data type represents positive and negative whole numbers.

Data Type NATURAL

The NATURAL data type is a subtype of INTEGER that is used for representing natural (nonnegative) numbers.

Data Type POSITIVE

The POSITIVE data type is a subtype of INTEGER that is used for representing positive (nonzero, nonnegative) numbers.

Data Type STRING

The STRING data type is an unconstrained array of characters. A STRING value is enclosed in double quotation marks as follows:

```
variable STRING_VAR: STRING(1 to 7);  
.  
.  
.  
STRING_VAR := "Rosebud";
```

Data Type BIT_VECTOR

The BIT_VECTOR data type represents an array of BIT values.

Unsupported Data Types

Some data types are either not useful for synthesis or are not supported. The following sections list and describe these unsupported data types.

Appendix C, "VHDL Constructs" describes the level of FPGA Compiler II / FPGA *Express* support for each VHDL construct.

Physical Types

FPGA Compiler II / FPGA *Express* does not support physical types, such as units of measure (for example, ns).

Floating-Point Types

FPGA Compiler II / FPGA *Express* does not support floating-point types, such as REAL.

Access Types

FPGA Compiler II / FPGA *Express* does not support access (pointer) types because no equivalent hardware construct exists.

File Types

FPGA Compiler II / FPGA *Express* does not support file (disk file) types, such as a hardware file type RAM or ROM.

Synopsys Data Types

The `std_logic_arith` package provides arithmetic operations and numeric comparisons on array data types. The package also defines two major data types: `UNSIGNED` and `SIGNED`. These data types, unlike the predefined `INTEGER` type, provide access to the individual bits (wires) of a numeric value. For more information, see “`std_logic_arith` Package” on page B-3.

Subtypes

A subtype is defined as a subset of a previously defined type or subtype. A subtype definition can appear anywhere a type definition is allowed.

Using subtypes is a powerful way to use VHDL type checking to ensure valid assignments and meaningful data handling. Subtypes inherit all operators and subprograms defined for their parent (base) types.

Subtypes are also used for resolved signals to associate a resolution function with the signal type (see “Subtypes” on page 2-32, for more information).

For example, note in Example 3-17 that `NATURAL` and `POSITIVE` are subtypes of `INTEGER` and that they can be used with any `INTEGER` function. They can be added, multiplied, compared, and assigned to each other if the values are within the appropriate subtype’s range. All `INTEGER` types and subtypes are actually subtypes of an anonymous predefined numeric type.

Example 3-18 shows some valid and invalid assignments between NATURAL and POSITIVE values.

Example 3-18 Valid and Invalid Assignments Between INTEGER Subtypes

```
variable NAT: NATURAL;  
variable POS: POSITIVE;  
.  
.  
.  
POS := 5;  
NAT := POS + 2;  
.  
.  
.  
NAT := 0;  
POS := NAT;           -- Invalid; out of range
```

For example, the type BIT_VECTOR is defined as

```
type BIT_VECTOR is array(NATURAL range <>) of BIT;
```

If your design uses only 16-bit vectors, you can define a subtype MY_VECTOR as

```
subtype MY_VECTOR is BIT_VECTOR(0 to 15);
```

Example 3-19 shows that all functions and attributes that operate on BIT_VECTOR also operate on MY_VECTOR.

Example 3-19 Attributes and Functions Operating on a Subtype

```
type BIT_VECTOR is array(NATURAL range <>) of BIT;
subtype MY_VECTOR is BIT_VECTOR(0 to 15);
. . .
signal  VEC1, VEC2: MY_VECTOR;
signal  S_BIT: BIT;
variable UPPER_BOUND: INTEGER;
. . .
if (VEC1 = VEC2)
. . .
VEC1(4) <= S_BIT;
VEC2 <= "0000111100001111";
. . .
RIGHT_INDEX := VEC1'high;
```


4

Expressions

In VHDL, expressions perform arithmetic or logical computations by applying an operator to one or more operands. Operators specify the computation to perform. Operands are the data for the computation.

In the following VHDL fragment, A and B are operands, + is an operator, and A + B is an expression.

```
C := A + B;  -- Computes the sum of two values
```

You can use expressions in many places in a design description. You can

- Assign them to variables or signals or use them as the initial values of constants
- Use them as operands to other operators
- Use them for the return value of functions

- Use them for the IN parameters in a subprogram call
- Assign them to the OUT parameters in a procedure body
- Use them to control the actions of statements such as if, loop, and case

This chapter discusses the use of expressions in a design description, in the following major sections:

- Operators
- Operands

Operators

A VHDL operator is characterized by

- Name
- Computation (function)
- Number of operands
- Type of operands (such as Boolean or character)
- Type of result value

You can define new operators, like functions, for any type of operand and result value. The predefined VHDL operators are listed in Table 4-1.

Table 4-1 Predefined VHDL Operators

Type	Operators						Precedence
Logical	and	or	nand	nor	xor	xnor	Lowest
Relational	=	/=	<	<=	>	>=	
Adding	+	-	&				
Unary (sign)	+	-					
Multiplying	*	/	mod	rem			
Miscellaneous	**	abs	not				Highest

Each line in the table lists operators with the same precedence. Each line's operators have greater precedence than those on the previous line. An operator's precedence determines whether it is applied before or after adjoining operators.

Example 4-1 shows some expressions and how they are interpreted.

Example 4-1 Operator Precedence

$A + B * C$ = $A + (B * C)$
`not BOOL and (NUM = 4)` = `(not BOOL) and (NUM = 4)`

VHDL allows existing operators to be overloaded—that is, applied to new types of operands. The logical operator called `and`, for example, can be overloaded to work with a new logic type. For more information, see “Operator Overloading” on page 2-30.

Logical Operators

Operands of a logical operator must be of the same type. The logical operators—`and`, `or`, `nand`, `nor`, `xor`, `xnor`, `not`—accept operands of type `BIT` or type `BOOLEAN` and one-dimensional arrays of `BIT` or `BOOLEAN`. Array operands must be the same size. A logical operator applied to two array operands is applied to pairs of the two arrays' elements.

Example 4-2 shows some logical signal declarations and logical operations on them. Figure 4-1 illustrates the resulting design.

Example 4-2 Logical Operators

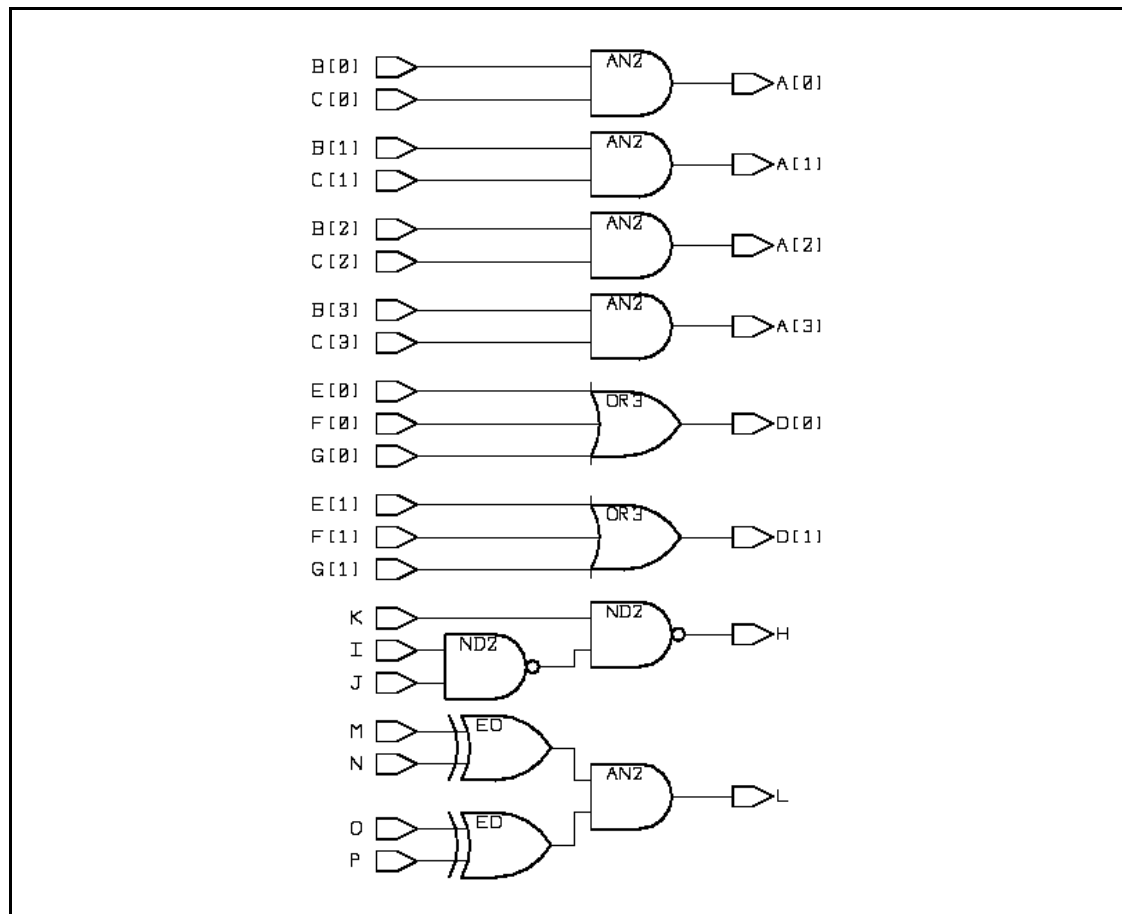
```

signal A, B, C:      BIT_VECTOR(3 downto 0);
signal D, E, F, G:   BIT_VECTOR(1 downto 0);
signal H, I, J, K:   BIT;
signal L, M, N, O, P: BOOLEAN;

A <= B and C;
D <= E or F or G;
H <= (I nand J) nand K;
L <= (M xor N) and (O xor P);

```

Figure 4-1 Design Schematic for Logical Operators



Normally, to use more than two operands in an expression, you must use parentheses to group the operands. An exception is that you can combine, without parentheses, a sequence that uses only one of the following operators:

- and
- or
- xor
- xnor

The following expression uses the same operator—and—in the sequence:

A and B and C and D

However, a sequence that contains more than one of these operators requires parentheses to indicate which two operands are to be paired. In the following sequence, and is the first operator, or is the second.

A and B or C

Parentheses should be used in one of two ways, as shown:

(A and B) or C

or

A and (B or C)

Relational Operators

Relational operators, such as = or >, compare two operands of the same base type and return a Boolean value.

IEEE VHDL defines the equality (=) and inequality (/=) operators for all types. Two operands are equal if they represent the same value. For array and record types, IEEE VHDL compares corresponding elements of the operands.

IEEE VHDL defines the ordering operators (<, <=, >, and >=) for all enumerated types, integer types, and one-dimensional arrays of enumeration or integer types.

The internal order of a type's values determines the result of the ordering operators. Integer values are ordered from negative infinity to positive infinity. Enumerated values are in the same order as they were declared, unless you have changed the encoding.

Note:

If you set the encoding of your enumerated types (see "Enumeration Encoding" on page 3-4), the ordering operators compare your encoded value ordering, not the declaration ordering. Because this interpretation is specific to FPGA Compiler II / *FPGA Express*, a VHDL simulator still uses the declaration's order of enumerated types.

Arrays are ordered alphabetically. FPGA Compiler II / *FPGA Express* determines the relative order of two array values by comparing each pair of elements in turn, beginning from the left bound of each array's index range. If a pair of array elements is not equal, the order of the different elements determines the order of the arrays. For example, bit vector "101011" is less than "1011", because the fourth bit of each vector is different, and '0' is less than '1'.

If the two arrays have different lengths and the shorter one matches the first part of the longer one, the shorter comes before the longer. Thus, the bit vector "101" is less than "101000". Arrays are compared from left to right, regardless of their index ranges (to or downto).

Example 4-3 shows several expressions that evaluate to true.

Example 4-3 True Relational Expressions

```
'1'    =  '1'  
"101"  =  "101"  
"1"    >  "011"    -- Array comparison  
"101"  <  "110"
```

To interpret bit vectors such as "011" as signed or unsigned binary numbers, use the relational operators defined in the `std_logic_arith` package (listed in Appendix B, "Synopsys Packages"). The third line in Example 4-3 evaluates false if the operands are of type `UNSIGNED`.

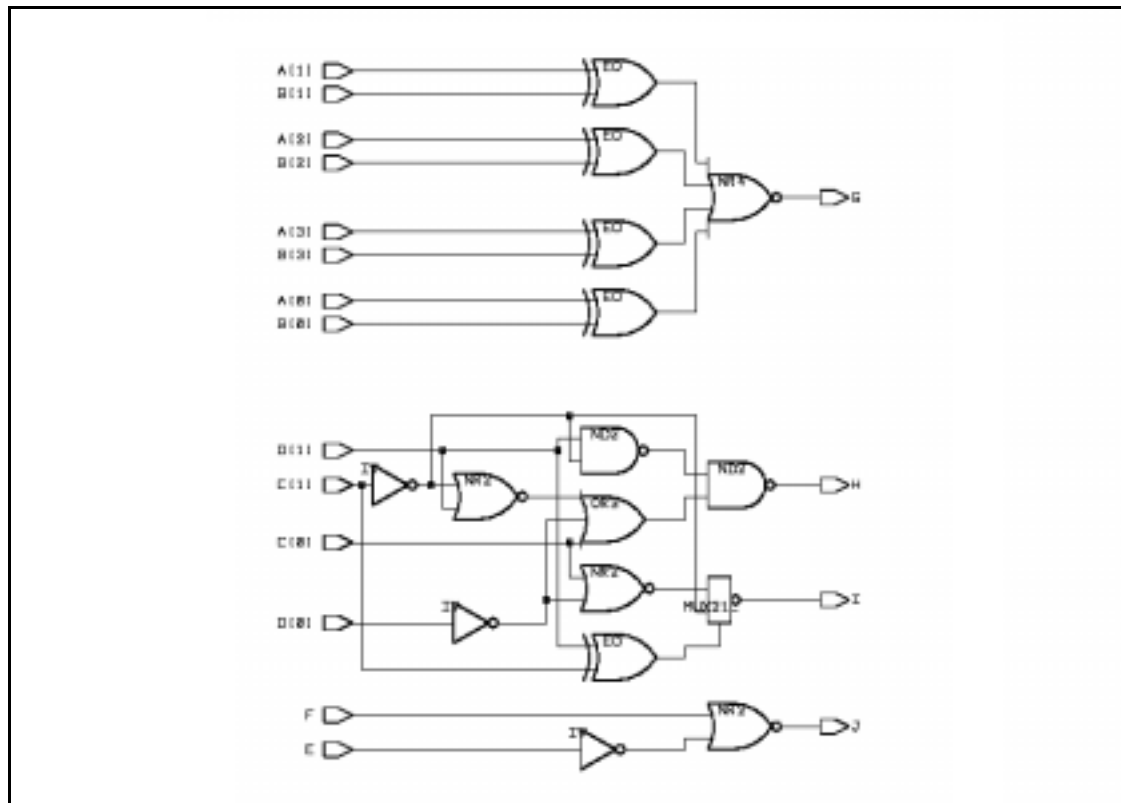
```
UNSIGNED' "1"  <  UNSIGNED' "011"    -- Numeric comparison
```

Example 4-4 shows some relational expressions. Figure 4-2 illustrates the resulting synthesized circuits.

Example 4-4 Relational Operators

```
signal A, B: BIT_VECTOR(3 downto 0);  
signal C, D: BIT_VECTOR(1 downto 0);  
signal E, F, G, H, I, J: BOOLEAN;  
  
G <= (A = B);  
H <= (C < D);  
I <= (C >= D);  
J <= (E > F);
```

Figure 4-2 Relational Operators Design Illustrating Example 4-4



Adding Operators

Adding operators include arithmetic and concatenation operators.

The arithmetic operators $+$ and $-$ are predefined for all integer operands. These addition and subtraction operators perform conventional arithmetic. Example 4-5 uses the $+$ operator.

The concatenation operator $\&$ is predefined for all one-dimensional array operands. The concatenation operator builds arrays by combining the operands. Each operand of $\&$ can be an array or an

element of an array. Use & to add a single element to the beginning or end of an array, to combine two arrays, or to build an array out of elements, as shown in Example 4-5 and Figure 4-3.

Example 4-5 Adding Operators

```

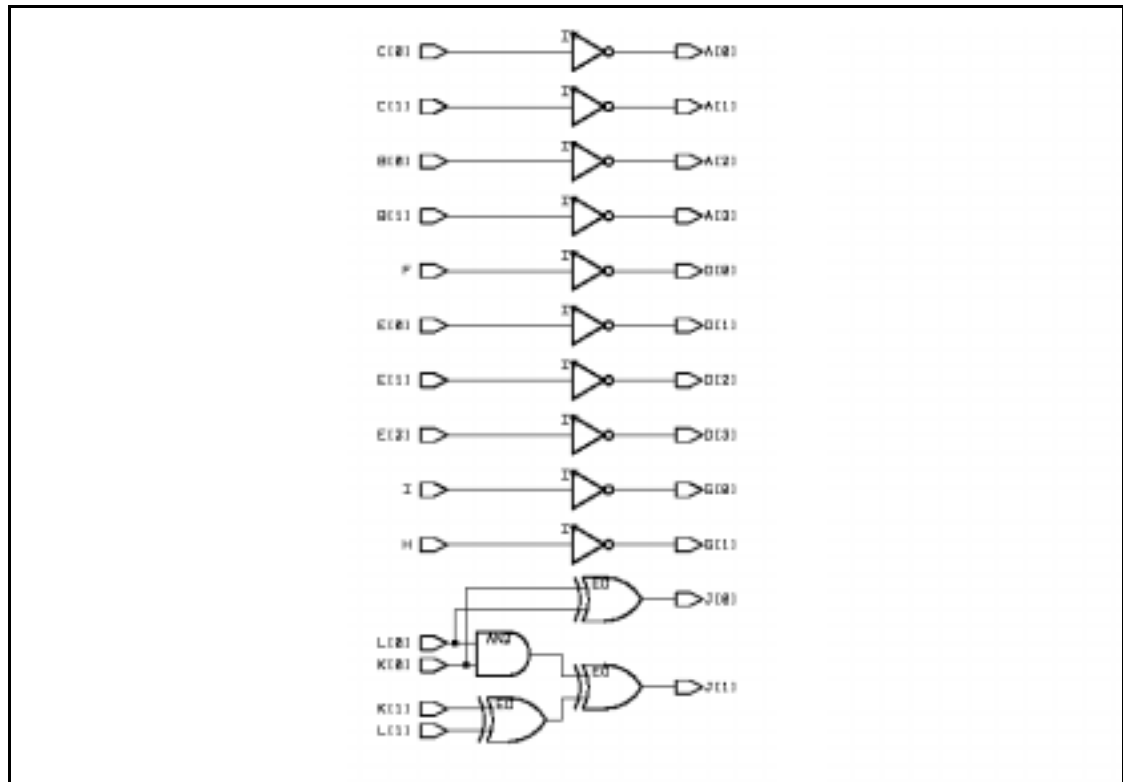
signal A, D:    BIT_VECTOR(3 downto 0);
signal B, C, G: BIT_VECTOR(1 downto 0);
signal E:      BIT_VECTOR(2 downto 0);
signal F, H, I: BIT;

signal J, K, L: INTEGER range 0 to 3;

A <= not B & not C;  -- Array & array
D <= not E & not F;  -- Array & element
G <= not H & not I;  -- Element & element
J <= K + L;          -- Simple addition

```

Figure 4-3 Design Array Illustrating Example 4-5



Unary (Signed) Operators

A unary operator has only one operand. FPGA Compiler II / FPGA *Express* predefines unary operators + and – for all integer types. The + operator has no effect. The – operator negates its operand. For example,

$$\begin{aligned} 5 &= +5 \\ 5 &= -(-5) \end{aligned}$$

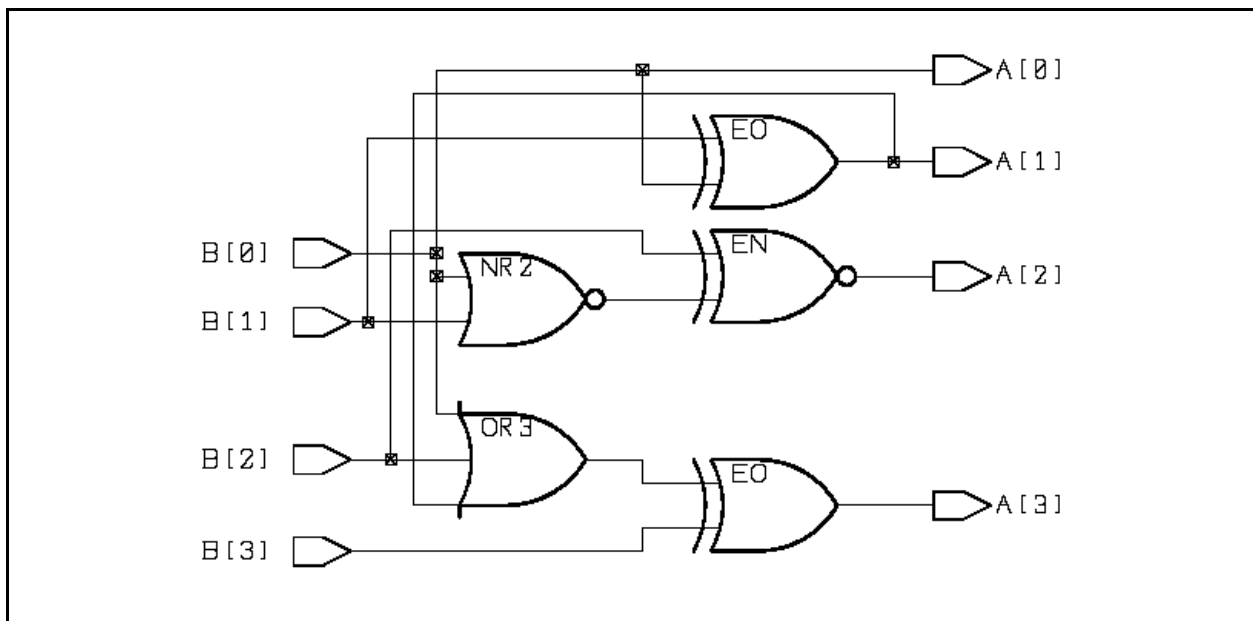
Example 4-6 shows how unary negation is synthesized, and Figure 4-4 illustrates the resulting design.

Example 4-6 Unary (Signed) Operators

```
signal A, B: INTEGER range -8 to 7;

A <= -B;
```

Figure 4-4 Design Illustrating Unary Negation From Example 4-6



Multiplying Operators

FPGA Compiler II / *FPGA Express* predefines the multiplying operators (*, /, mod, and rem) for all integer types.

FPGA Compiler II / *FPGA Express* places some restrictions on the supported values for the right-hand operands of the multiplying operators, as follows:

*

Integer multiplication: no restrictions.

/

Integer division: The right-hand operand must be a computable power of 2 and cannot be negative (see “Computable Operands” on page 4-16). This operator is implemented as a bit shift.

mod

Modulus: same as /.

rem

Remainder: same as /.

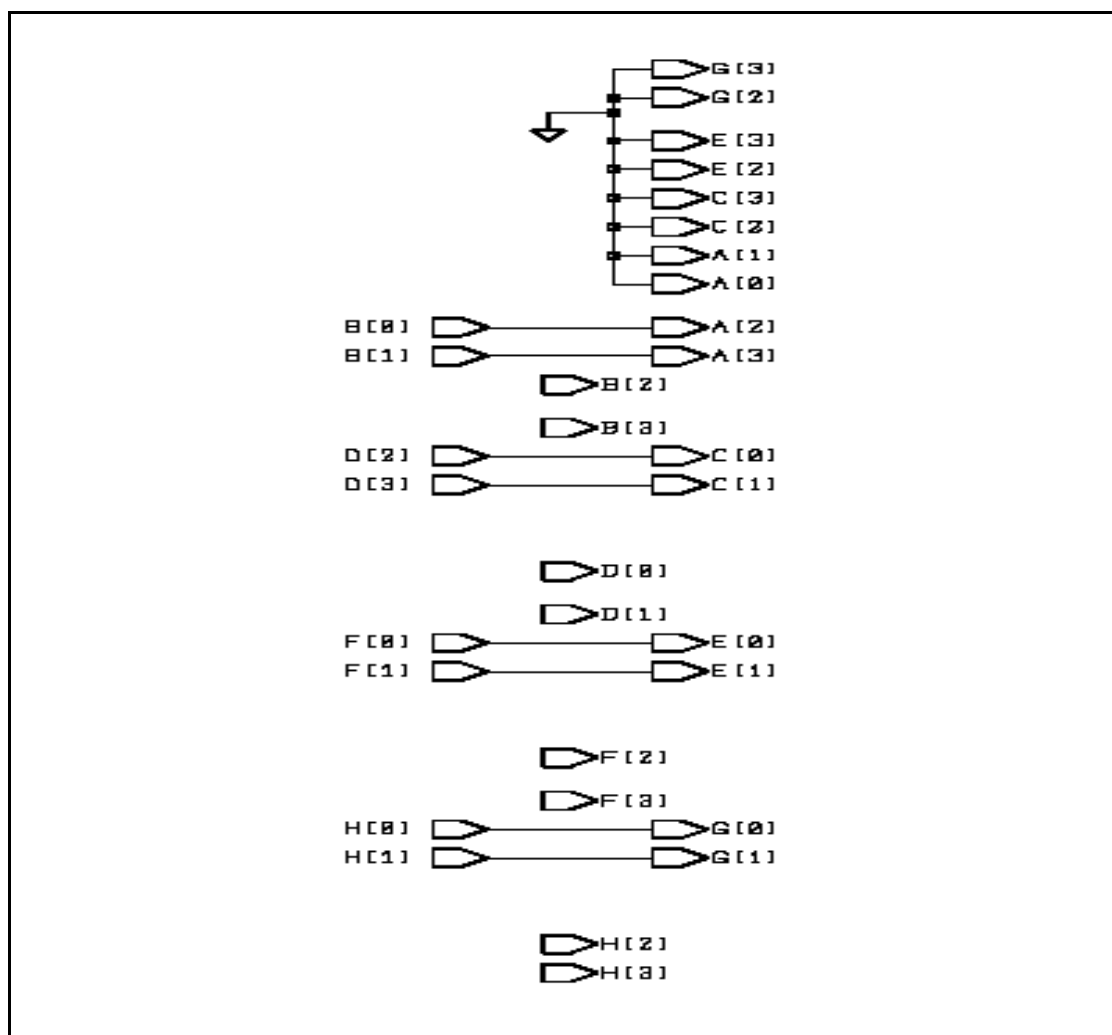
Example 4-7 shows some uses of the multiplying operators whose right-hand operands are all powers of 2. Figure 4-5 illustrates the resulting synthesized circuit design.

Example 4-7 Multiplying Operators With Powers of 2

```
signal A, B, C, D, E, F, G, H: INTEGER range 0 to 15;
```

```
A <= B * 4;  
C <= D / 4;  
E <= F mod 4;  
G <= H rem 4;
```

Figure 4-5 Design Illustrating Multiplying Operators From Example 4-7



Miscellaneous Arithmetic Operators

FPGA Compiler II / FPGA *Express* predefines the absolute value (abs) and exponentiation (**) operators for all integer types. There is one restriction placed on the ** operator: When you're using ** exponentiation, the left operand must be the computable value 2 (see "Computable Operands" on page 4-16).

Example 4-8 shows how these operators are used. Figure 4-6 illustrates the synthesized design.

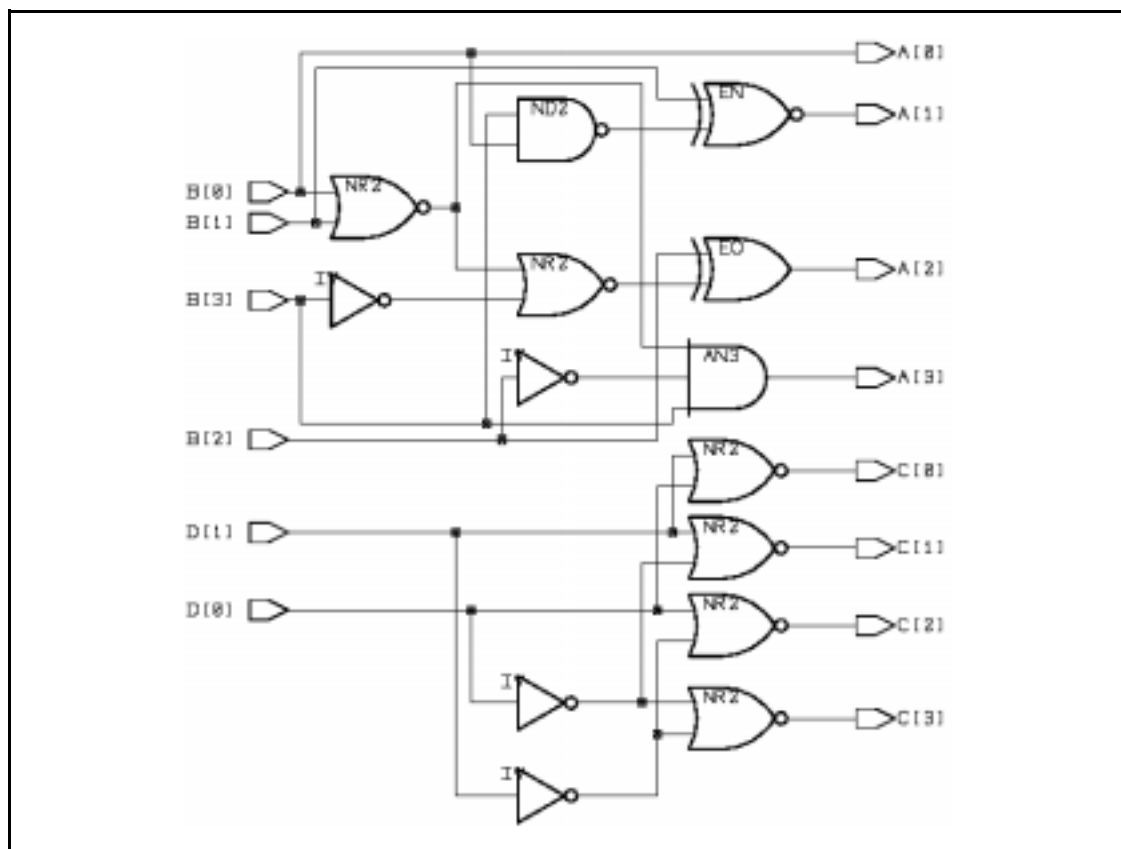
Example 4-8 Miscellaneous Arithmetic Operators

```

signal A, B: INTEGER range -8 to 7;
signal C:    INTEGER range 0 to 15;
signal D:    INTEGER range 0 to 3;
A <= abs(B);
C <= 2 ** D;

```

Figure 4-6 Design With Arithmetic Operators From Example 4-8



Operands

The operands specify the data used by the operator to compute its value. An operand returns its value to the operator.

There are many categories of operands. The simplest operand is a literal, such as the number 7, or an identifier, such as a variable or signal name. Operands can themselves be expressions. You create expression operands by surrounding an expression with parentheses.

The operand categories are

Aggregates

`my_array_type'(others => 1)`

Attributes

`my_array'range`

Expressions

`(A nand B)`

Function calls

`LOOKUP_VAL(my_var_1, my_var_2)`

Identifiers

`my_var, my_sig`

Indexed names

`my_array(7)`

Literals

`'0', "101", 435, 16#FF3E#`

Qualified expressions

`BIT_VECTOR('1' & '0')`

Records and fields

my_record.a_field

Slice names

my_array(7 to 11)

Type conversions

THREE_STATE('0')

The next two sections discuss operand bit-widths and explain computable operands. The sections following them describe the operand categories listed here.

Operand Bit-Width

FPGA Compiler II / *FPGA Express* uses the bit-width of the largest operand to determine the bit-width needed to implement an operator in a circuit. For example, an INTEGER operand is 32 bits wide by default. An addition of two INTEGER operands causes FPGA Compiler II / *FPGA Express* to build a 32-bit adder.

To use hardware resources efficiently, always indicate the bit-width of numeric operands. For example, use a subrange of INTEGER when declaring types, variables, or signals.

```
type      ENOUGH:  INTEGER range 0 to 255;
variable WIDE:    INTEGER range -1024 to 1023;
signal    NARROW:  INTEGER range 0 to 7;
```

Note:

During optimization, FPGA Compiler II / *FPGA Express* removes hardware for unused bits.

Computable Operands

Some operators, such as the division operator, restrict their operands to be computable. A computable operand is one whose value can be determined by FPGA Compiler II / *FPGA Express*. Computability is important because noncomputable expressions can require logic gates to determine their value.

Following are examples of computable operands:

- Literal values
- for...loop parameters, when the loop's range is computable
- Variables assigned a computable expression
- Aggregates that contain only computable expressions
- Function calls whose return value is computable
- Expressions with computable operands
- Qualified expressions when the expression is computable
- Type conversions when the expression is computable
- The value of the and or nand operators when one of the operands is a computable '0'
- The value of the or operator or the nor operator when one of the operands is a computable '1'

Additionally, a variable is given a computable value if it is an OUT or INOUT parameter of a procedure that assigns it a computable value.

Following are examples of noncomputable operands:

- Signals

- Ports
- Variables assigned different computable values that depend on a noncomputable condition
- Variables assigned noncomputable values

Example 4-9 shows some definitions and declarations, followed by several computable and noncomputable expressions.

Example 4-9 Computable and Noncomputable Expressions

```

signal S: BIT;
. . .
function MUX(A, B, C: BIT) return BIT is
begin
    if (C = '1') then
        return(A);
    else
        return(B);
    end if;
end;

procedure COMP(A: BIT; B: out BIT) is
begin
    B := not A;
end;

process(S)
    variable V0, V1, V2: BIT;
    variable V_INT:      INTEGER;
    subtype MY_ARRAY is BIT_VECTOR(0 to 3);
    variable V_ARRAY:    MY_ARRAY;
begin
    V0 := '1';           -- Computable (value is '1')
    V1 := V0;            -- Computable (value is '1')
    V2 := not V1;        -- Computable (value is '0')

    for I in 0 to 3 loop
        V_INT := I;      -- Computable (value depends on iteration)
    end loop;

```

```

V_ARRAY := MY_ARRAY'(V1, V2, '0', '0');
-- Computable ("1000")
V1 := MUX(V0, V1, V2); -- Computable (value is '1')
COMP(V1, V2);
V1 := V2; -- Computable (value is '0')
V0 := S and '0'; -- Computable (value is '0')
V1 := MUX(S, '1', '0'); -- Computable (value is '1')
V1 := MUX('1', '1', S); -- Computable (value is '1')

if (S = '1') then
    V2 := '0'; -- Computable (value is '0')
else
    V2 := '1'; -- Computable (value is '1')
end if;
V0 := V2; -- Non-computable; V2 depends on S
V1 := S; -- Non-computable; S is signal
V2 := V1; -- Non-computable; V1 is no longer
computable
end process;

```

Aggregates

Aggregates create array literals, by giving a value to each element of an instance of an array type. Aggregates can also be considered array literals, because they specify an array type and the value of each array element. The syntax is

```
type_name'([choice =>] expression{, [choice =>] expression})
```

type_name

A constrained array type (as required by FPGA Compiler II / FPGA Express in the previous example), an element index, a sequence of indexes, or the others expression. Each expression provides a value for the chosen elements and must evaluate to a value of the element's type.

Example 4-10 shows an array type definition and an aggregate representing a literal of that array type. The two sets of assignments have the same result.

Example 4-10 Simple Aggregate

```
subtype MY_VECTOR is BIT_VECTOR(1 to 4);
signal X: MY_VECTOR;
variable A, B: BIT;

X <= MY_VECTOR('1', A nand B, '1', A or B) -- Aggregate
                                           -- assignment

X(1) <= '1';                               -- Element assignment
X(2) <= A nand B;
X(3) <= '1';
X(4) <= A or B;
```

You can specify an element's index by using either positional or named notation. With positional notation, each element receives the value of its expression in order, as shown in Example 4-10.

By using named notation, the choice `=>` construct specifies one or more elements of the array. The choice can contain an expression, such as `(1 mod 2) =>`, to indicate a single element index or a range, such as `3 to 5 =>` or `7 downto 0 =>`, to indicate a sequence of element indexes.

An aggregate can use both positional and named notation.

It is not necessary to specify all element indexes in an aggregate. All unassigned values are given a value by inclusion of the others `=>` expression as the last element of the list.

Example 4-11 shows several aggregates representing the same value.

Example 4-11 Equivalent Aggregates

```
subtype MY_VECTOR is BIT_VECTOR(1 to 4);

MY_VECTOR('1', '1', '0', '0');
MY_VECTOR(2 => '1', 3 => '0', 1 => '1', 4 => '0');
MY_VECTOR('1', '1', others => '0');
MY_VECTOR(3 => '0', 4 => '0', others => '1');
MY_VECTOR(3 to 4 => '0', 2 downto 1 => '1');
MY_VECTOR(3 to 4 => '0', others => '1');
```

The others expression can be the only expression in the aggregate. Example 4-12 shows two equivalent aggregates.

Example 4-12 Equivalent Aggregates Using the others Expression

```
MY_VECTOR(others => '1');
MY_VECTOR('1', '1', '1', '1');
```

For information on using an aggregate as the target of an assignment statement, see “Assignment Statements and Targets” on page 5-2.

Attributes

VHDL defines attributes for various types. A VHDL attribute takes a variable or signal of a given type and returns a value. The syntax of an attribute is

object'attribute

FPGA Compiler II / FPGA Express supports the following predefined VHDL attributes for use with arrays, as described in “Array Types” on page 3-9.

- left
- right

- high
- low
- length
- range
- reverse_range

FPGA Compiler II / *FPGA Express* also supports the following predefined VHDL attributes for use with wait and if statements, as described in Chapter 7, "Register and Three-State Inference".

- event
- stable

In addition to supporting the previous predefined VHDL attributes, FPGA Compiler II / *FPGA Express* has a defined set of synthesis-related attributes. You can include these FPGA Compiler II / *FPGA Express*-specific attributes in your VHDL design description to direct FPGA Compiler II / *FPGA Express* during optimization.

Expressions

Operands can themselves be expressions. You create expression operands by surrounding an expression with parentheses, such as (A nand B).

Function Calls

A function call executes a named function with the given parameter values. The value returned to an operator is the function's return value. The syntax of a function call is

```
function_name ( [parameter_name =>] expression  
                {, [parameter_name =>] expression } ) ;
```

function_name

Name of a defined function. The optional *parameter_names* are the names of formal parameters as defined by the function. Each expression provides a value for its parameter and must evaluate to a type appropriate for that parameter.

You can specify parameters in positional or named notation, as you can with aggregate values.

In positional notation, the *parameter_name* => construct is omitted. The first expression provides a value for the function's first parameter, the second expression is for the second parameter, and so on.

In named notation, *parameter_name* => is specified before an expression; the named parameter gets the value of that expression.

You can mix positional and named expressions in the same function call if you put all positional expressions before named parameter expressions.

Example 4-13 shows a function declaration and several equivalent function calls.

Example 4-13 Function Calls

```
function FUNC(A, B, C: INTEGER) return BIT;  
.  
.  
.  
FUNC(1, 2, 3)  
FUNC(B => 2, A => 1, C => 7 mod 4)  
FUNC(1, 2, C => -3+6)
```

Identifiers

Identifiers are probably the most common operand. An identifier is the name of a constant, variable, function, signal, entity, port, subprogram, or parameter and returns that object's value to an operand.

Identifiers that contain special characters, begin with numbers, or have the same name as a keyword can be specified as an extended identifier. An extended identifier starts with a backslash character (\), followed by a sequence of characters, followed by another backslash character (\).

Example 4-14 shows some extended identifiers.

Example 4-14 Sample Extended Identifiers

```
\a+b\           \3state\  
\type\          \ (a&b) | c\
```

Example 4-15 shows several kinds of identifiers and their usages. All identifiers appear in bold type.

Example 4-15 Identifiers

```
entity EXAMPLE is
  port (INT_PORT:    in INTEGER;
        BIT_PORT:   out BIT);
end;

. . .
signal  BIT_SIG: BIT;
signal  INT_SIG: INTEGER;
. . .
INT_SIG  <= INT_PORT;    -- Signal assignment from port
BIT_PORT <= BIT_SIG;    -- Signal assignment to port

function FUNC(INT_PARAM:  INTEGER)
  return INTEGER;
end function;

. . .
constant CONST:    INTEGER := 2;
variable VAR:      INTEGER;
. . .
VAR := FUNC(INT_PARAM => CONST);  -- Function call
```

Indexed Names

An indexed name identifies one element of an array variable or signal. The syntax of an indexed name is

identifier (expression)

identifier

Name of a signal or variable of an array type. The expression must return a value within the array's index range. The value returned to an operator is the specified array element.

If the expression is computable (see "Computable Operands" on page 4-16), the operand is synthesized directly. If the expression

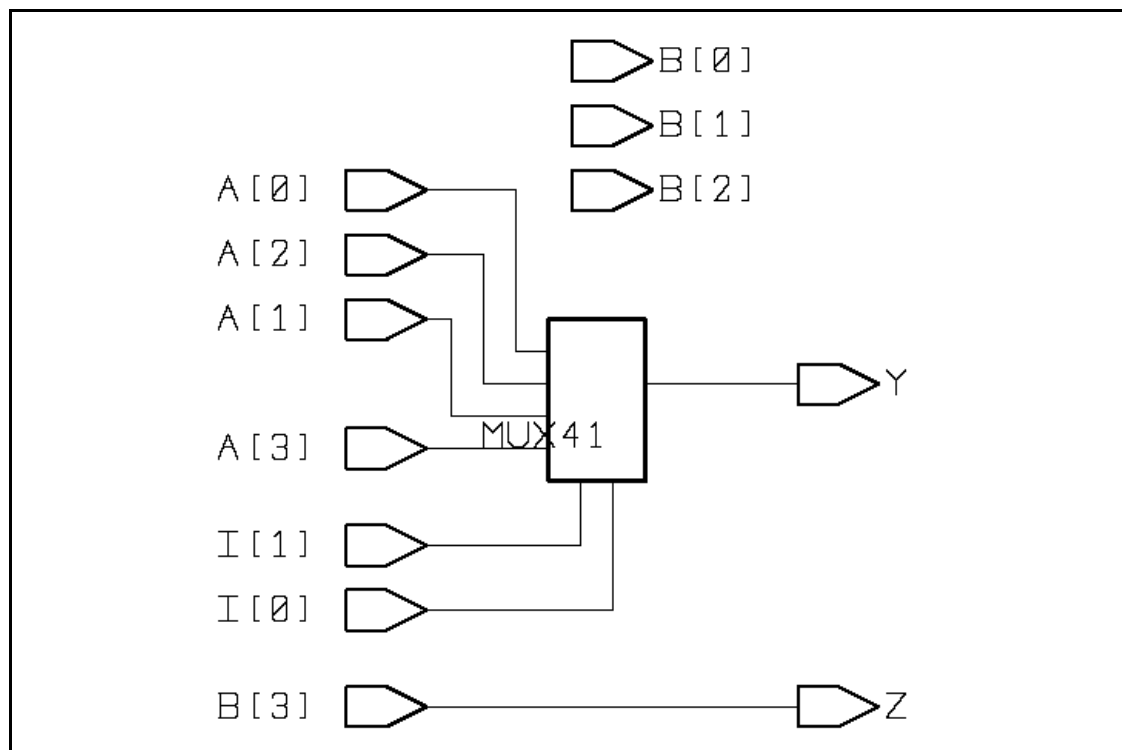
is noncomputable, a circuit is synthesized that extracts the specified element from the array.

Example 4-16 shows two indexed names, one computable and one not. Figure 4-7 illustrates the resulting synthesized circuit design.

Example 4-16 Indexed Name Operands

```
signal A, B: BIT_VECTOR(0 to 3);  
signal I:    INTEGER range 0 to 3;  
signal Y, Z: BIT;  
  
Y <= A(I);  -- Noncomputable index expression  
Z <= B(3);  -- Computable index expression
```

Figure 4-7 Design Illustrating Use of Indexed Names From Example 4-16



You can also use indexed names as assignment targets; see “Assignment Statements and Targets” on page 5-2.

Literals

A literal (constant) operand can be a numeric literal, a character literal, an enumeration literal, or a string literal. The following sections describe these four kinds of literals.

Numeric Literals

Numeric literals are constant integer values. The two kinds of numeric literals are decimal and based. A decimal literal is written in base 10. A based literal can be written in a base from 2 to 16 and is composed of the base number, an octothorpe (#), the value in the given base, and another octothorpe (#). For example, 2#101# is decimal 5.

The digits in either kind of numeric literal can be separated by underscores. Example 4-17 shows several different numeric literals, all representing the same value, which is 170.

Example 4-17 Numeric Literals

```
170
1_7_0
10#170#
2#1010_1010#
16#AA#
```

Character Literals

Character literals are single characters enclosed in single quotation marks—for example, 'A'. Character literals are used both as values for operators and in defining enumerated types, such as CHARACTER and BIT. See “Enumeration Types” on page 3-3 for the valid character types.

Enumeration Literals

Enumeration literals are values of enumerated types. The two kinds of enumeration literals are character literals and identifiers. Character literals are described earlier. Enumeration identifiers are those listed in an enumeration type definition. For example,

```
type SOME_ENUM is (ENUM_ID_1, ENUM_ID_2, ENUM_ID_3);
```

If two enumerated types use the same literals, those literals are overloaded. You must qualify overloaded enumeration literals when you use them in an expression, unless their type can be determined from context (see “Qualified Expressions” on page 4-29). For more information, see “Enumeration Types” on page 3-3.

Example 4-18 defines two enumerated types and shows some enumeration literal values.

Example 4-18 Enumeration Literals

```
type ENUM_1 is (AAA, BBB, 'A', 'B', ZZZ);
type ENUM_2 is (CCC, DDD, 'C', 'D', ZZZ);

AAA          -- Enumeration identifier of type ENUM_1
'B'          -- Character literal of type ENUM_1
CCC          -- Enumeration identifier of type ENUM_2
'D'          -- Character literal of type ENUM_2
ENUM_1'(ZZZ) -- Qualified because overloaded
```

String Literals

String literals are one-dimensional arrays of characters enclosed in double quotation marks (" "). The two kinds are

- Character strings, which are sequences of characters in double quotation marks, for example, "ABCD".

- Bit strings, which are similar to character strings but represent binary, octal, or hexadecimal values. For example, B"1101", O"15", and X"D" all represent the decimal value 13.

A string literal's type is a one-dimensional array of an enumerated type. Each of the characters in the string represents one element of the array.

Example 4-19 shows some character string literals.

Example 4-19 Character String Literals

```
"10101"  
"ABCDEF"
```

Note:

Null string literals ("") are not supported.

Bit strings, like based numeric literals, are composed of a base specifier character, a double quotation mark, a sequence of numbers in the given base, and another double quotation mark. For example, B"0101" represents the bit vector 0101. A bit string literal consists of the base specifier B, O, or X, followed by a string literal. It is interpreted as a bit vector, a one-dimensional array of the predefined type BIT. The base specifier determines the interpretation of the bit string as follows:

B (binary)

The value is in binary digits (bits 0 or 1). Each bit in the string represents one BIT in the generated bit vector (array).

O (octal)

The value is in octal digits (0 to 7). Each octal digit in the string represents three BITs in the generated bit vector (array).

X (hexadecimal)

The value is in hexadecimal digits (0 to 9 and A to F). Each hexadecimal digit in the string represents four BITS in the generated bit vector (array).

You can separate the digits in a bit string literal value with underscores (_) for readability. Example 4-20 shows three bit string literals representing the value AAA.

Example 4-20 Bit String Literals

```
X"AAA"  
B"1010_1010_1010"  
O"5252"
```

Qualified Expressions

Qualified expressions state the type of an ambiguous operand. You cannot use qualified expressions for type conversion (see “Type Conversions” on page 4-34).

The syntax of a qualified expression is

```
type_name' (expression)
```

type_name

The name of a defined type. The expression must evaluate to a value of an appropriate type.

Note:

FPGA Compiler II / FPGA *Express* requires a single quotation mark (tick) between *type_name* and (*expression*). If the single quotation mark is not there, the construction is interpreted as a type conversion (described in the next section).

Example 4-21 shows a qualified expression that resolves an overloaded function by qualifying the type of a decimal literal parameter.

Example 4-21 A Qualified Decimal Literal

```
type R_1 is range 0 to 10;  -- Integer 0 to 10
type R_2 is range 0 to 20;  -- Integer 0 to 20

function FUNC(A: R_1) return BIT;
function FUNC(A: R_2) return BIT;

FUNC(5) -- Ambiguous; could be of type R_1, R_2, or INTEGER

FUNC(R_1'(5))  -- Unambiguous
```

Example 4-22 shows how qualified expressions resolve ambiguities in aggregates and enumeration literals.

Example 4-22 Qualified Aggregates and Enumeration Literals

```
type ARR_1 is array(0 to 10) of BIT;
type ARR_2 is array(0 to 20) of BIT;
. . .
(others => '0')          -- Ambiguous; could be of type ARR_1 or ARR_2

ARR_1'(others => '0')  -- Qualified; unambiguous
-----
type ENUM_1 is (A, B);
type ENUM_2 is (B, C);
. . .
B                      -- Ambiguous; could be of type ENUM_1 or ENUM_2

ENUM_1'(B)             -- Qualified; unambiguous
```

Records and Fields

Records are composed of named fields of any type. For more information, see “Record Types” on page 3-13.

In an expression, you can refer to a whole record or to a single field. The syntax of field names is

`record_name.field_name`

`record_name`

Name of the record variable or signal. A `record_name` is different for each variable or signal of that record type.

`field_name`

Name of a field in that record type. A `field_name` is separated from the `record_name` by a period (.). A `field_name` is the field name defined for that record type.

Example 4-23 shows a record type definition and record and field access.

Example 4-23 Record and Field Access

```
type BYTE_AND_IX is
  record
    BYTE: BIT_VECTOR(7 downto 0);
    IX:   INTEGER range 0 to 7;
  end record;

signal X: BYTE_AND_IX;
. . .
X          -- record
X.BYTE     -- field: 8-bit array
X.IX       -- field: integer
```

A field can be of any type, including an array, record, or aggregate type. Refer to an element of a field by using that type's notation; for example,

```
X.BYTE(2)          -- one element from array field BYTE
X.BYTE(3 downto 0) -- 4-element slice of array field BYTE
```

Slice Names

Slice names identify a sequence of elements of an array variable or signal. The syntax is

identifier (expression direction expression)

identifier

Name of a signal or variable of an array type. Each expression must return a value within the array's index range and must be computable (see "Computable Operands" on page 4-16).

The direction must be either to or downto. The direction of a slice must be the same as the direction of an identifier's array type. If the left and right expressions are equal, they define a single element.

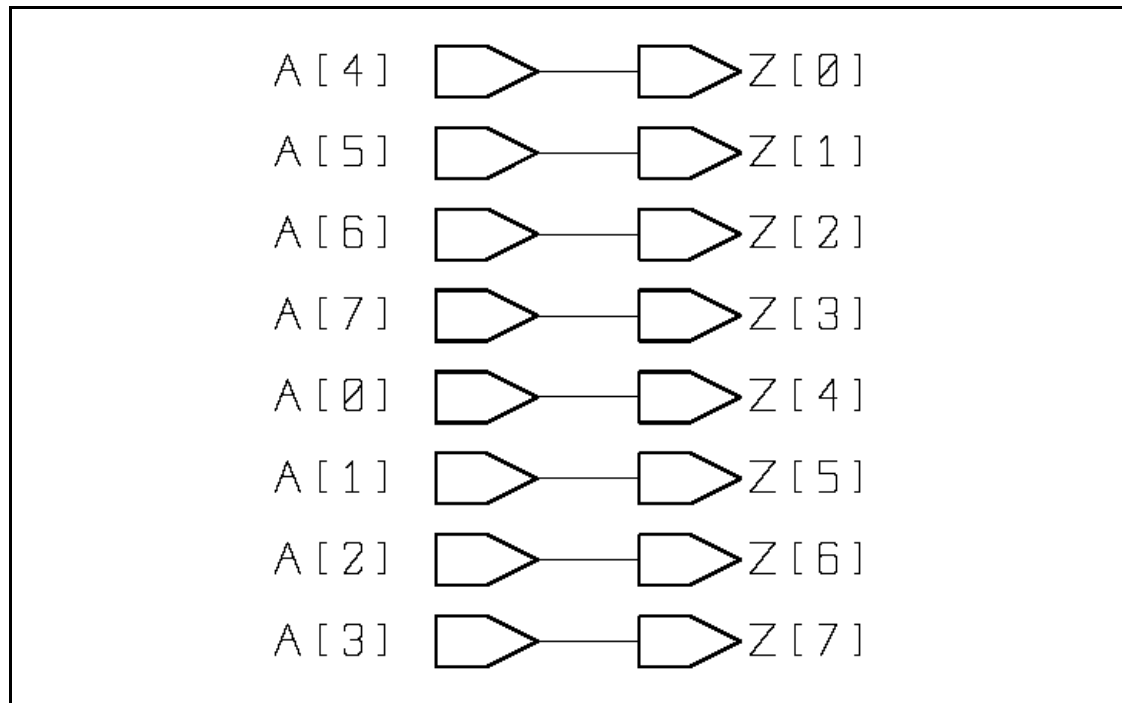
The value returned to an operator is a subarray containing the specified array elements.

Example 4-24 uses slices to assign an 8-bit input to an 8-bit output, exchanging the lower and upper 4 bits. Figure 4-8 illustrates the resulting synthesized circuit design. Slices are also used as assignment targets. This usage is described in "Assignment Statements and Targets" on page 5-2.

Example 4-24 Slice Name Operands

```
signal A, Z: BIT_VECTOR(0 to 7);  
  
Z(0 to 3) <= A(4 to 7);  
Z(4 to 7) <= A(0 to 3);
```


Figure 4-8 Design Illustrating Use of Slices From Example 4-24



Limitations on Null Slices

Synthesis does not support null slices, which are indicated by

- A null range, such as (4 to 3)
- A range with the wrong direction, such as UP_VAR(3 downto 2) when the UP_VAR declared range is ascending (Example 4-25)

Example 4-25 shows three null slices and one noncomputable slice.

Example 4-25 Null and Noncomputable Slices

```
subtype DOWN is BIT_VECTOR(4 downto 0);
subtype UP   is BIT_VECTOR(0 to 7);
. . .
variable UP_VAR:    UP;
variable DOWN_VAR: DOWN;
. . .
UP_VAR(4 to 3)      -- Null slice (null range)
UP_VAR(4 downto 0)  -- Null slice (wrong direction)
DOWN_VAR(0 to 1)    -- Null slice (wrong direction)
variable I: INTEGER range 0 to 7;
. . .
UP_VAR(I to I+1)    -- Noncomputable slice
```

Limitations on Noncomputable Slices

Synthesis does not allow noncomputable slices—slices whose range contains a noncomputable expression.

Type Conversions

Type conversions change an expression's type.

The syntax of a type conversion is

type_name(expression)

type_name

The name of a defined type. The expression must evaluate to a value of a type that is convertible into type *type_name*.

- Type conversions can convert between integer types or between similar array types.
- Two array types are similar if they have the same length and have convertible or identical element types.

- Enumerated types are not convertible.

Example 4-26 shows some type definitions and associated signal declarations, followed by valid and invalid type conversions.

Example 4-26 Valid and Invalid Type Conversions

```

type INT_1 is range 0 to 10;
type INT_2 is range 0 to 20;

type ARRAY_1 is array(1 to 10) of INT_1;
type ARRAY_2 is array(11 to 20) of INT_2;

subtype MY_BIT_VECTOR is BIT_VECTOR(1 to 10);
type BIT_ARRAY_10 is array(11 to 20) of BIT;
type BIT_ARRAY_20 is array(0 to 20) of BIT;

signal S_INT:      INT_1;
signal S_ARRAY:    ARRAY_1;
signal S_BIT_VEC:  MY_BIT_VECTOR;
signal S_BIT:      BIT;
    -- Legal type conversions

INT_2(S_INT)
    -- Integer type conversion

BIT_ARRAY_10(S_BIT_VEC)
    -- Similar array type conversion
    -- Illegal type conversions

BOOLEAN(S_BIT);
    -- Can't convert between enumerated types

INT_1(S_BIT);
    -- Can't convert enumerated types to other types

BIT_ARRAY_20(S_BIT_VEC);
    -- Array lengths not equal

ARRAY_1(S_BIT_VEC);
    -- Element types are not convertible

```


5

Sequential Statements

FPGA Compiler II / FPGA *Express* interprets sequential statements, such as `A := 3`, in the order in which they appear in the code. VHDL sequential statements can appear only in processes and subprograms. This chapter discusses the different types of sequential statements, in the following sections:

- Assignment Statements and Targets
- Variable Assignment Statements
- Signal Assignment Statements
- if Statements
- case Statements
- loop Statements
- next Statements

- exit Statements
- Subprograms
- return Statement
- wait Statements
- null Statements

Assignment Statements and Targets

Use an assignment statement to assign a value to a variable or signal. The syntax is

```
target := expression; -- Variable assignment
```

```
target <= expression; -- Signal assignment
```

target

The target can be a variable or a signal (or part of a variable or a signal, such as a subarray) that receives the value of the expression. The expression must evaluate to the same type as the target. See Chapter 4, "Expressions" for more information.

There are five kinds of targets:

- Simple names, such as `my_var`
- Indexed names, such as `my_array_var(3)`
- Slices, such as `my_array_var(3 to 6)`
- Field names, such as `my_record.a_field`
- Aggregates, such as `(my_var1, my_var2)`

The difference in syntax between variable assignments and signal assignments is that

- Variables use the `:=` operator

Variables are local to a process or subprogram, and their assignments take effect immediately.

- Signals use the `<=` operator

Signals need to be global in a process or subprogram, and their assignments take effect at the end of a process.

Signals are the only means of communication between processes. For more information on semantic differences, see “Signal Assignment Statements” on page 5-12.

The following descriptions refer to variable as well as signal targets.

Simple Name Targets

The syntax for an assignment to a simple name (identifier) target is

```
identifier := expression; -- Variable assignment
```

```
identifier <= expression; -- Signal assignment
```

identifier

The name of a signal or variable. The assigned expression must have the same type as the signal or variable. For array types, all elements of the array are assigned values.

Example 5-1 shows some assignments to simple name targets.

Example 5-1 Simple Name Targets

```
variable A, B: BIT;
signal    C: BIT_VECTOR(1 to 4);

-- Target      Expression
A  :=  '1'; -- Variable A is assigned '1'
B  :=  '0'; -- Variable B is assigned '0'
C  <= "1100"; -- Signal array C is assigned bit value "1100"
```

Indexed Name Targets

The syntax for an assignment to an indexed name (identifier) target is

```
identifier(index_expression) := expression; -- Variable assignment
identifier(index_expression) <= expression; -- Signal assignment
```

identifier

The name of an array type signal or variable. The `index_expression` must evaluate to an index value for the identifier array's index type and bounds. It does not have to be computable (see Chapter 4, "Expressions"), but more hardware is synthesized if it is not.

The assigned expression must have the array's element type.

In Example 5-2, the array variable A elements are assigned values as indexed names.

Example 5-3 shows two indexed name targets. One is computable, the other is not. Figure 5-1 illustrates the corresponding design.

Example 5-2 Indexed Name Targets

```
variable A: BIT_VECTOR(1 to 4);

-- Target      Expression;
A(1)   := '1';   -- Assigns '1' to the first element of array A.
A(2)   := '1';   -- Assigns '1' to the second element of array A
A(3)   := '0';   -- Assigns '0' to the third element of array A
A(4)   := '0';   -- Assigns '0' to the fourth element of array A
```

Example 5-3 Computable and Noncomputable Indexed Name Targets

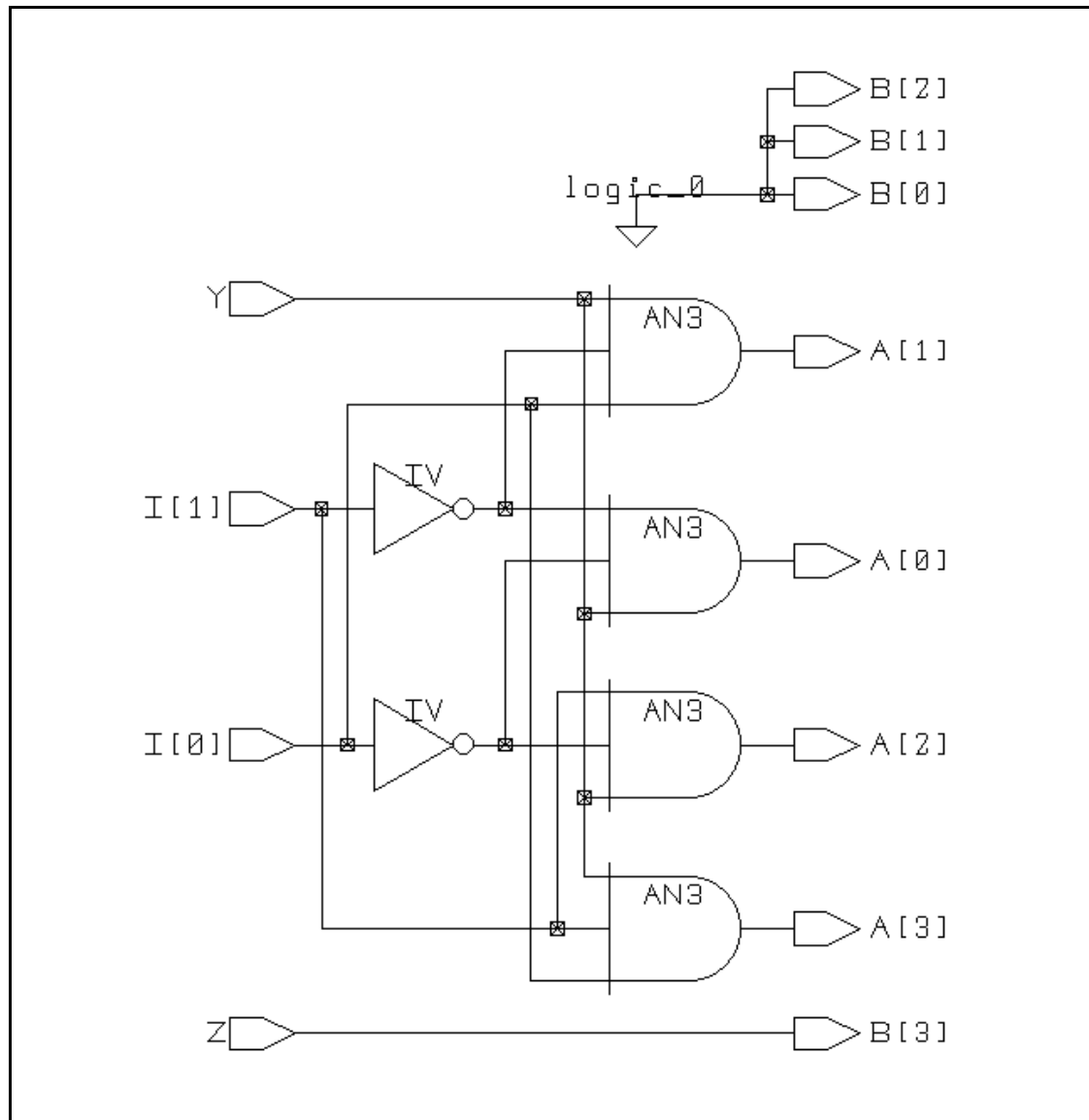
```
entity example5_3 is
    port (
        signal A, B: out BIT_VECTOR(0 to 3);
        signal I: in INTEGER range 0 to 3;
        signal Y, Z: in BIT
    );
end example5_3;

architecture behave of example5_3 is

begin
    process (I,Y,Z)
    begin
        A    <= "0000";
        B    <= "0000";
        A(I) <= Y;  -- Noncomputable index expression
        B(3) <= Z;  -- Computable index expression

    end process;
end behave;
```

Figure 5-1 Design Illustrating Indexed Name Targets From Example 5-3



Slice Targets

The syntax for an assignment to a slice target is

```
identifier(index_expr_1 direction index_expr_2)
```

identifier

The name of an array type signal or variable. Each *index_expr* expression must evaluate to an index value for the identifier array's index type and bounds. Both *index_expr* expressions must be computable (see Chapter 4, "Expressions") and must lie within the bounds of the array. The direction must match the identifier array type's direction, either to or downto.

The assigned expression must have the array's element type.

In Example 5-4, array variables A and B are assigned the same value.

Example 5-4 Slice Targets

```
variable A, B: BIT_VECTOR(1 to 4);  
-- Target      Expression  
A(1 to 2) := "11";  
    -- Assigns "11" to the first two elements of array A  
A(3 to 4) := "00";  
    -- Assigns "00" to the last two elements of array A  
B(1 to 4) := "1100";  
    -- Assigns "1100" to array B
```

Field Targets

The syntax for an assignment to a field target is

identifier.field_name

identifier

The name of a record type signal or variable. *field_name* is the name of a field in that record type, preceded by a period (.). The assigned expression must have the identified field's type. A field can be of any type, including an array, record, or aggregate type.

Example 5-5 assigns values to the fields of record variables A and B.

Example 5-5 Field Targets

```
type REC is
  record
    NUM_FIELD:    INTEGER range -16 to 15;
    ARRAY_FIELD:  BIT_VECTOR(3 to 0);
  end record;

variable A, B: REC;

-- Target      Expression
A.NUM_FIELD    := -12;
  -- Assigns -12 to record A's field NUM_FIELD
A.ARRAY_FIELD  := "0011";
  -- Assigns "0011" to record A's field ARRAY_FIELD
A.ARRAY_FIELD(3) := '1';
  -- Assigns '1' to the most significant bit of
  -- record A's field ARRAY_FIELD
B               := A;
  -- Assigns values of record A to corresponding fields of B
```

For more information, see “Record Types” on page 3-13.

Aggregate Targets

The syntax for an assignment to an aggregate target is

```
([choice =>] identifier
{,[choice =>] identifier}) := array_expression;
-- Variable assignment

([choice =>] identifier
{,[choice =>] identifier}) <= array_expression;
-- Signal assignment
```

aggregate assignment

Assigns the `array_expression` element values to one or more variable or signal identifiers.

Each (optional) choice is an index expression selecting an element or a slice of the assigned `array_expression`. Each identifier must have the `array_expression` element type. An identifier can be an array type.

You can assign array element values to the identifiers by position or by name. In positional notation, the choice `=>` construct is not used. Identifiers are assigned array element values in order, from the left array bound to the right array bound.

In named notation, the choice `=>` construct identifies specific elements of the assigned array. A choice index expression indicates a single element (such as 3). The identifier's type must match the assigned expression's element type.

Positional and named notation can be mixed, but positional associations must come before named associations, as in Example 5-6.

Example 5-6 Aggregate Targets

```
signal A, B, C, D: BIT;
signal S: BIT_VECTOR(1 to 4);
. . .
variable E, F: BIT;
variable G: BIT_VECTOR(1 to 2);
variable H: BIT_VECTOR(1 to 4);

-- Positional notation
S      <= ('0', '1', '0', '0');
(A, B, C, D) <= S;      -- Assigns '0' to A
                        -- Assigns '1' to B
                        -- Assigns '0' to C
                        -- Assigns '0' to D

-- Named notation
(3 => E,      4 => F,
 2 => G(1), 1 => G(2)) := H;      -- Assigns H(1) to G(2)
                                -- Assigns H(2) to G(1)
                                -- Assigns H(3) to E
                                -- Assigns H(4) to F
```

Variable Assignment Statements

A variable assignment changes the value of a variable. The syntax is

target := expression;

target

Names the variables that receive the value of expression.

See “Assignment Statements and Targets” on page 5-2 for a description of variable assignment targets.

expression

Determines the assigned value; its type must be compatible with the target.

For more information about expressions, see Chapter 4, “Expressions”.

When a variable is assigned a value, the assignment takes place immediately. A variable keeps its assigned value until another assignment takes place.

Example 5-7 on page 5-14 shows the different effects of variable and signal assignments.

Signal Assignment Statements

A signal assignment changes the value being driven on a signal by the current process. The syntax is

```
target <= expression;
```

target

Names the signals that receive the value of expression.

See “Assignment Statements and Targets” on page 5-2 for a description of variable assignment targets.

expression

Determines the assigned value; its type must be compatible with target.

For more information about expressions, see Chapter 4, “Expressions”.

Signals and variables act in different ways when they receive assigned values. The differences lie in the way the two kinds of assignments take effect and how that influences the value FPGA Compiler II / FPGA *Express* reads from either variables or signals.

variable assignment

When a variable receives an assigned value, the assignment changes the value of the variable from that point on. That value is kept until the variable is assigned a different value.

signal assignment

When a signal receives an assigned value, the assignment does not necessarily take effect, because the value of a signal is determined by the processes (or other concurrent statements) that drive the signal.

- If several values are assigned to a given signal in one process, only the last assignment is effective. Even if a signal in a process is assigned, then read, and then assigned again, the value read (either inside or outside the process) is the last assignment value.
- If several processes (or other concurrent statements) assign values to one signal, the drivers are wired together. The resulting circuit depends on the expressions and the target technology. It might be invalid, wired AND, wired OR, or a three-state bus. For more information on this topic, see Chapter 6, "Concurrent Statements".

Example 5-7 shows the different effects of variable and signal assignments.

Example 5-7 Variable and Signal Assignments

```
signal S1, S2: BIT;
signal S_OUT : BIT_VECTOR(1 to 8);
. . .
process( S1, S2 )
    variable V1, V2: BIT;
begin
    V1 := '1';    -- This sets the value of V1
    V2 := '1';    -- This sets the value of V2
    S1 <= '1';    -- This assignment is the driver for S1
    S2 <= '1';    -- This has no effect because of the
                  -- assignment later in this process

    S_OUT(1) <= V1; -- Assigns '1', the value assigned above
    S_OUT(2) <= V2; -- Assigns '1', the value assigned above
    S_OUT(3) <= S1; -- Assigns '1', the value assigned above
    S_OUT(4) <= S2; -- Assigns '0', the value assigned below

    V1 := '0';    -- This sets the new value of V1
    V2 := '0';    -- This sets the new value of V2
    S2 <= '0';    -- This assignment overrides the previous
                  -- one since it is the last assignment to
                  -- this signal in this process

    S_OUT(5) <= V1; -- Assigns '0', the value assigned above
    S_OUT(6) <= V2; -- Assigns '0', the value assigned above
    S_OUT(7) <= S1; -- Assigns '1', the value assigned above
    S_OUT(8) <= S2; -- Assigns '0', the value assigned above
end process;
```

if Statements

The if statement executes a sequence of statements. The sequence depends on the value of one or more conditions. The syntax is

```
if condition then  
[    { sequential_statement }  
  elsif condition then ]  
    { sequential_statement }  
[ else  
    { sequential_statement } ]  
end if;
```

Each condition must be a Boolean expression. Each branch of an if statement can have one or more `sequential_statements`.

Evaluating Conditions

An if statement evaluates each condition in order. Only the first true condition causes the execution of the if statement's branch statements. The remainder of the if statement is skipped.

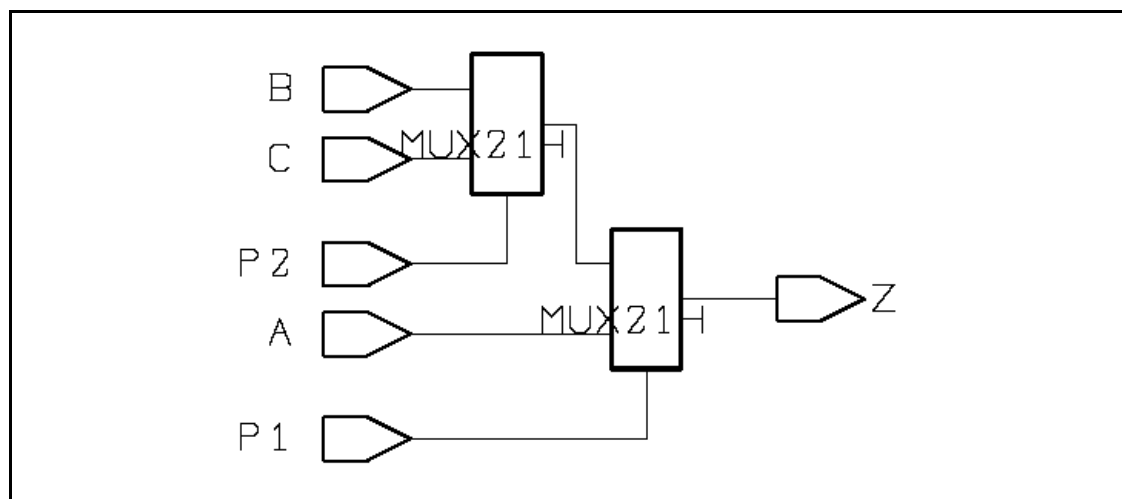
If none of the conditions is true and the else clause is present, those statements are executed. If none of the conditions is true and no else clause is present, none of the statements is executed.

Example 5-8 shows an if statement. Figure 5-2 illustrates the corresponding circuit.

Example 5-8 if Statement

```
signal A, B, C, P1, P2, Z: BIT;  
  
if (P1 = '1') then  
    Z <= A;  
elsif (P2 = '0') then  
    Z <= B;  
else  
    Z <= C;  
end if;
```

Figure 5-2 Schematic Design From Example 5-8



Using the if Statement to Infer Registers and Latches

Some forms of the if statement can be used like the wait statement, to test for signal edges and therefore imply synchronous logic. This usage causes FPGA Compiler II / *FPGA Express* to infer registers or latches, as described in Chapter 7, "Register and Three-State Inference".

case Statements

The case statement executes one of several sequences of statements, depending on the value of a single expression. The syntax is

```
case expression is
    when choices =>
        { sequential_statement }
    { when choices =>
        { sequential_statement } }
end case;
```

expression

Must evaluate to an INTEGER, an enumerated type, or an array of enumerated types such as BIT_VECTOR. Each of the choices must be of the form

```
choice { | choice }
```

choice

Each choice can be either a static expression (such as 3) or a static range (such as 1 to 3). The type of *choice_expression* determines the type of each choice. Each value in the range of *choice_expression*'s type must be covered by one choice.

The final choice can be others, as in Example 5-10 on page 5-20, which matches all remaining (unchosen) values in the range of expression's type. The others choice, if present, matches expression only if no other choices match.

The case statement evaluates *expression* and compares that value with each choice value. The when clause with the matching choice value has its statements executed.

The following restrictions are placed on choices:

- No two choices can overlap.
- If an others choice is not present, all possible values of expression must be covered by the set of choices.

Using Different Expression Types

Example 5-9 shows a case statement that selects one of four signal assignment statements by using an enumerated expression type. Figure 5-3 illustrates the corresponding design with binary encoding specified.

Example 5-9 case Statement With Enumerated Type

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package case_enum is
type ENUM is (PICK_A, PICK_B, PICK_C, PICK_D);
end case_enum;

library work;
use work.case_enum.all;

entity example5_9 is
  port (
    signal A, B, C, D: in BIT;
    signal VALUE: ENUM;
    signal Z: out BIT
  );
end example5_9;

architecture behave of example5_9 is

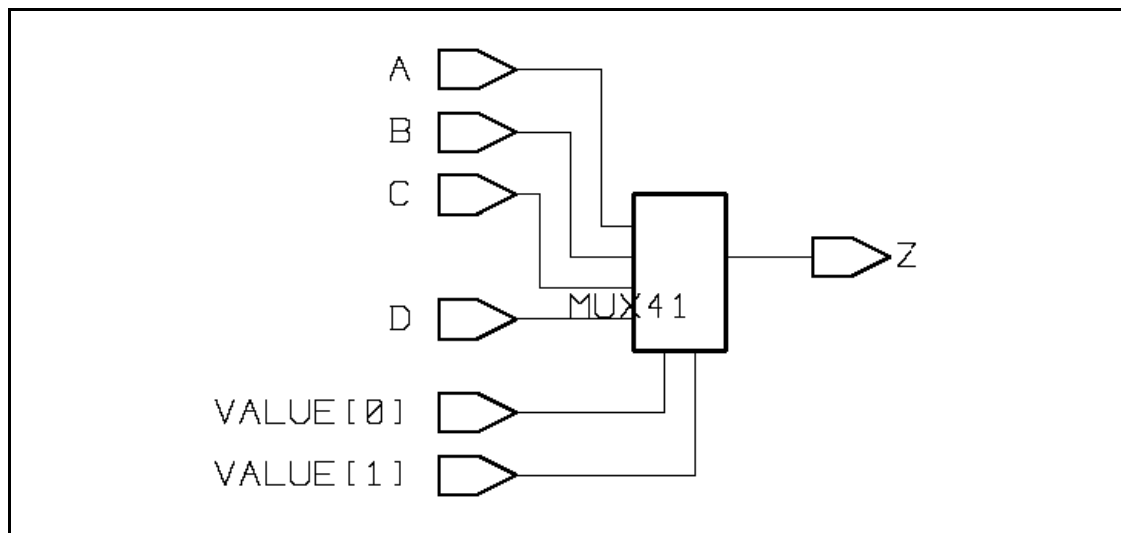
begin
  process (VALUE)
```

```

begin
  case VALUE is
    when PICK_A =>
      Z <= A;
    when PICK_B =>
      Z <= B;
    when PICK_C =>
      Z <= C;
    when PICK_D =>
      Z <= D;
  end case;
end process;
end behave;

```

Figure 5-3 Schematic Design From Example 5-9



Example 5-10 shows a case statement again used to select one of four signal assignment statements, this time by using an integer expression type with multiple choices. Figure 5-4 illustrates the corresponding design.

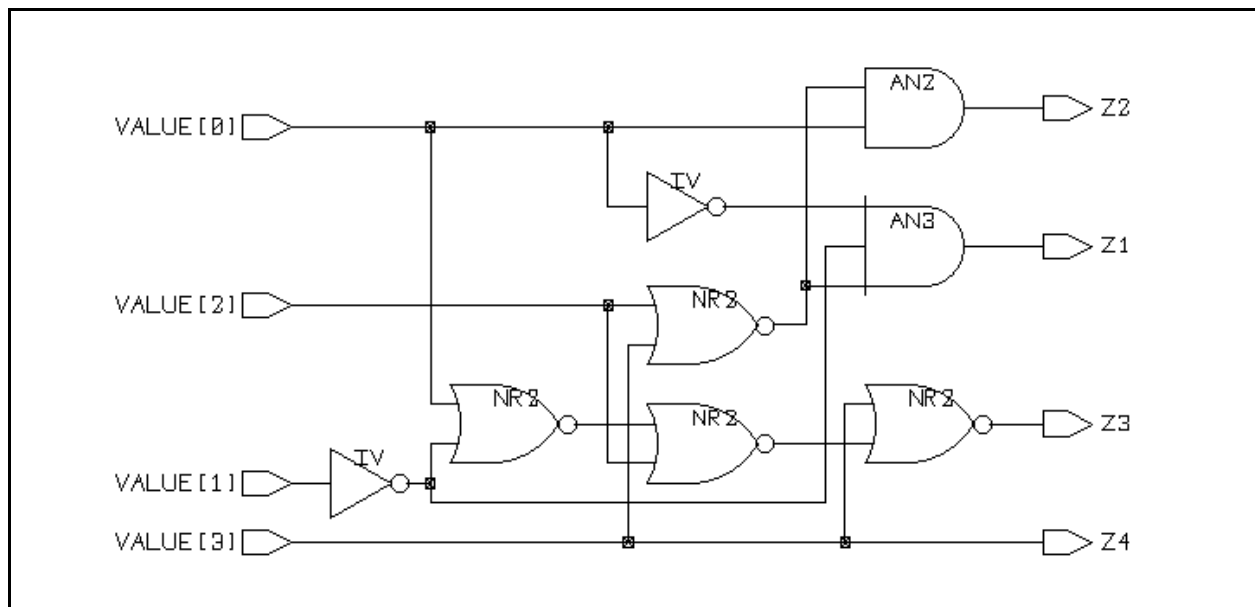
Example 5-10 case Statement With Integers

```

entity example5_10 is
    port (
        signal VALUE: in INTEGER range 0 to 15;
        signal Z1, Z2, Z3, Z4: out BIT
    );
end example5_10;
architecture behave of example5_10 is
begin
    process (VALUE)
    begin
        Z1 <= '0';
        Z2 <= '0';
        Z3 <= '0';
        Z4 <= '0';
        case VALUE is
            when 0 => -- Matches 0
                Z1 <= '1';
            when 1 | 3 => -- Matches 1 or 3
                Z2 <= '1';
            when 4 to 7 | 2 => -- Matches 2, 4, 5, 6, or 7
                Z3 <= '1';
            when others => -- Matches remaining values, 8 through 15
                Z4 <= '1';
        end case;
    end process;
end behave;

```

Figure 5-4 Schematic Design From Example 5-10



Invalid case Statements

Example 5-11 shows invalid case statements with explanatory comments.

Example 5-11 Invalid case Statements

```
signal VALUE:  INTEGER range 0 to 15;
signal OUT_1:  BIT;

case VALUE is -- Must have at least one when clause
end case;

case VALUE is -- Values 2 to 15 are not covered by choices
  when 0 =>
    OUT_1 <= '1';
  when 1 =>
    OUT_1 <= '0';
end case;

case VALUE is -- Choices 5 to 10 overlap
  when 0 to 10 =>
    OUT_1 <= '1';
  when 5 to 15 =>
    OUT_1 <= '0';
end case;
```

loop Statements

A loop statement repeatedly executes a sequence of statements. The syntax is

```
[label :] [iteration_scheme] loop
    { sequential_statement }
    { next [ label ] [ when condition ] ; }
    { exit [ label ] [ when condition ] ; }
end loop [label];
```

label

The label, which is optional, names the loop and is useful for building nested loops.

iteration_scheme

There are three types of iteration_scheme: loop, while...loop, and for...loop. They are described in the next three sections.

next and exit statements

Sequential statements used only within loops.

next statement

Skips the remainder of the current loop and continues with the next loop iteration.

exit statement

Skips the remainder of the current loop and continues with the next statement after the exited loop.

See “next Statements” on page 5-30 and “exit Statements” on page 5-33.

Basic loop Statements

The basic loop statement has no iteration scheme. FPGA Compiler II / FPGA *Express* executes enclosed statements repeatedly until it encounters an exit or next statement. The syntax statement is

```
[label :] loop
    { sequential_statement }
end loop [label];
```

loop

The label, which is optional, names this loop.

sequential_statement

Any statement described in this chapter.

Two sequential statements are used only with loops:

next statement

Skips the remainder of the current loop and continues with the next loop iteration.

exit statement

Skips the remainder of the current loop and continues with the next statement after the exited loop.

See “next Statements” on page 5-30 and “exit Statements” on page 5-33.

Note:

Noncomputable loops (loop and while...loop statements) must have at least one wait statement in each enclosed logic branch. Otherwise, a combinational feedback loop is created. See “wait Statements” on page 5-50 for more information. Conversely, computable loops (for...loop statements) must not contain wait statements. Otherwise, a race condition may result.

while...loop Statements

The while...loop statement has a Boolean iteration scheme. If the iteration condition evaluates true, FPGA Compiler II / *FPGA Express* executes the enclosed statements once. The iteration condition is then reevaluated. As long as the iteration condition remains true, the loop is repeatedly executed. When the iteration condition evaluates false, the loop is skipped and execution continues with the next loop iteration. The syntax for a while...loop statement is

```
[label :] while condition loop
    { sequential_statement }
end loop [label];
```

label

The label, which is optional, names this loop.

condition

Any Boolean expression, such as ((A = '1') or (X < Y)).

sequential_statement

Any statement described in this chapter.

Two sequential statements are used only with loops:

next statement

Skips the remainder of the current loop and continues with the next loop iteration.

exit statement

Skips the remainder of the current loop and continues with the next statement after the exited loop.

See “next Statements” on page 5-30 and “exit Statements” on page 5-33.

Note:

Noncomputable loops (loop and while...loop statements) must have at least one wait statement in each enclosed logic branch. Otherwise, a combinational feedback loop is created. See “wait Statements” on page 5-50 for more information.

for...loop Statements

The for...loop statement has an integer iteration scheme. The integer range determines the number of repetitions. The syntax for a for...loop statement is

```
[label :] for identifier in range loop  
    { sequential_statement }  
end loop [label];
```

label

The label, which is optional, names this loop.

identifier

Specific to the for...loop statement:

- Identifier is not declared elsewhere. It is automatically declared by the loop itself and is local to the loop. A loop identifier overrides any other identifier with the same name, but only within the loop.
- The identifier value can be read only inside its loop (identifier does not exist outside the loop). You cannot assign a value to a loop identifier.

range

Must be a computable integer range in either of two forms:

integer_expression to integer_expression

integer_expression downto integer_expression

Each *integer_expression* evaluates to an integer.

For more information, see Chapter 4, "Expressions"

sequential_statement

Any statement described in this chapter.

Two sequential statements are used only with loops:

next statement

Skips the remainder of the current loop and continues with the next loop iteration.

exit statement

Skips the remainder of the current loop and continues with the next statement after the exited loop.

See "next Statements" on page 5-30 and "exit Statements" on page 5-33.

Note:

Computable loops (for...loop statements) must not contain wait statements. Otherwise, a race condition may result.

Steps in the Execution of a for...loop Statement

A for...loop statement executes as follows:

1. A new integer variable, which is local to the loop, is declared with the identifier.
2. The identifier receives the first value of range, and the sequence of statements executes once.
3. The identifier receives the next value of range, and the sequence of statements executes once more.
4. Step 3 repeats until identifier receives the last value in range. The sequence of statements then executes for the last time. Execution continues with the statement following the end loop. The loop is then inaccessible.

Example 5-12 shows two equivalent code fragments. Figure 5-5 illustrates the corresponding design.

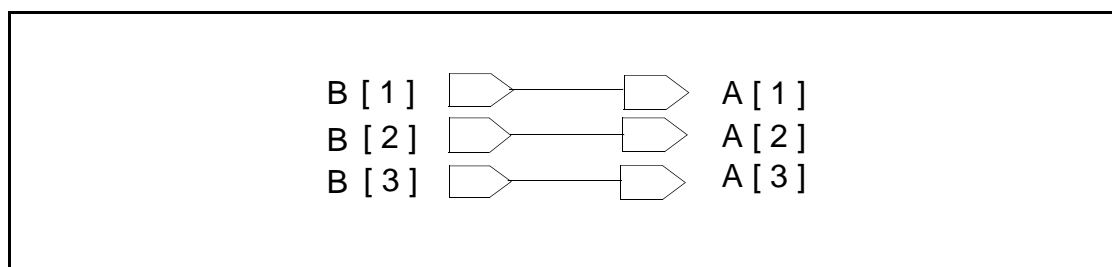
Example 5-12 for...loop Statement With Equivalent Code Fragments

```
variable A, B: BIT_VECTOR(1 to 3);

-- First fragment is a loop statement
for I in 1 to 3 loop
    A(I) <= B(I);
end loop;

-- Second fragment is three statements
A(1) <= B(1);
A(2) <= B(2);
A(3) <= B(3);
```

Figure 5-5 Schematic Design From Example 5-12



for...loop Statements and Arrays

You can use a loop statement to operate on all elements of an array, without explicitly depending on the size of the array. Example 5-13 shows how to use the VHDL array attribute 'range to invert each element of bit vector A. Figure 5-6 illustrates the corresponding design. For more information about unconstrained arrays and array attributes, see “Array Types” on page 3-9.

Example 5-13 for...loop Statement Operating on an Entire Array

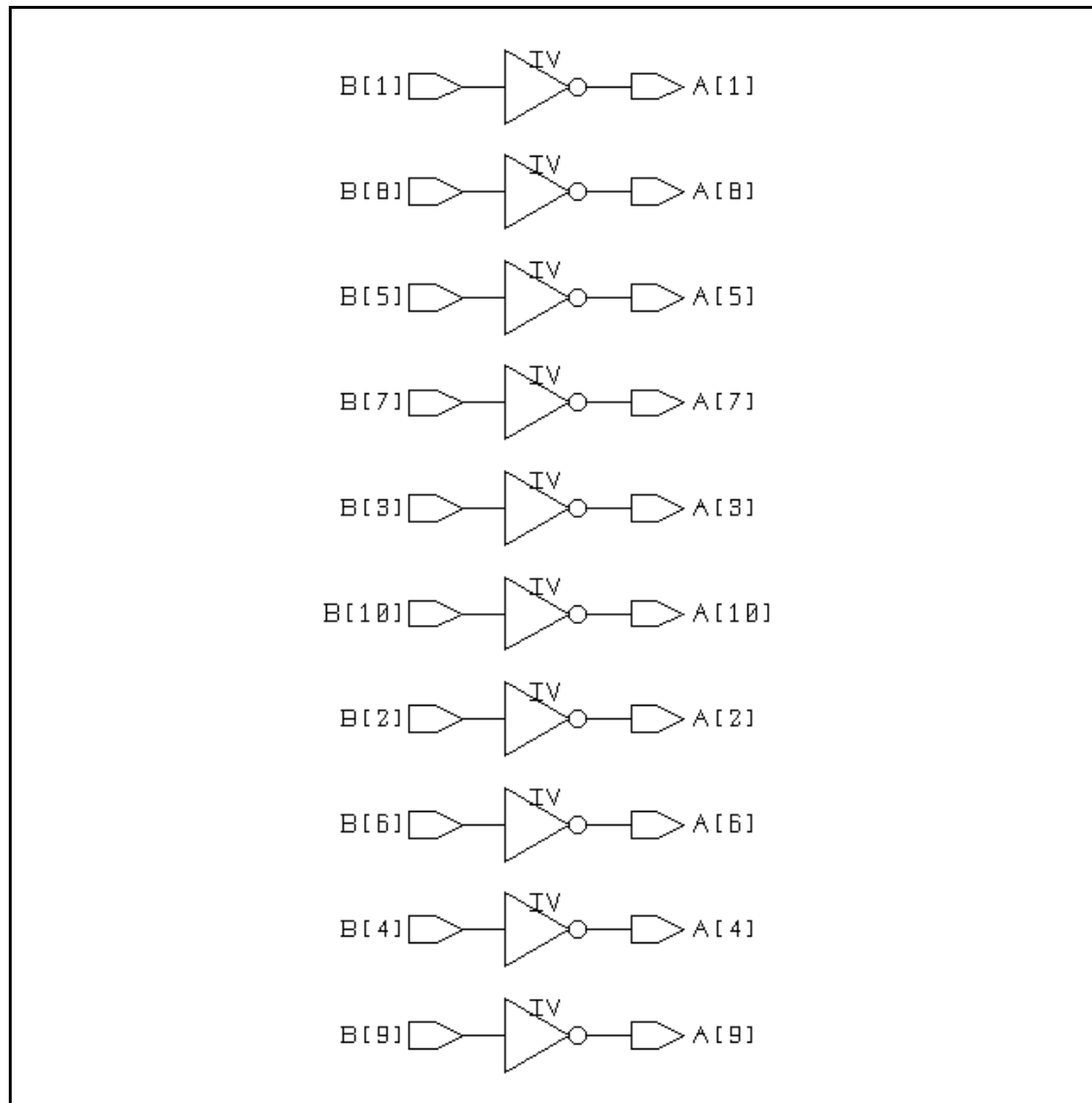
```
entity example5_13 is
    port(
        A: out BIT_VECTOR(1 to 10);
        B: in BIT_VECTOR(1 to 10)
    );
end example5_13;

architecture behave of example5_13 is
begin
    process (B)
begin

    for I in A'range loop
        A(I) <= not B(I);
    end loop;

end process;
end behave;
```


Figure 5-6 Schematic Design of Array From Example 5-13



next Statements

The next statement skips execution to the next iteration of an enclosing loop statement, called *label* in the syntax, as follows:

```
next [ label ] [ when condition ] ;
```

label

A next statement with no label terminates the current iteration of the innermost enclosing loop. When you specify a loop label, the current iteration of that named loop is terminated.

when

An optional clause that executes its next statement when its condition (a Boolean expression) evaluates true.

Example 5-14 uses the next statement to copy bits conditionally from bit vector B to bit vector A only when the next condition evaluates true. Figure 5-7 illustrates the corresponding design.

Example 5-14 *next Statement*

```
entity example5_14 is
    port(
        signal B, COPY_ENABLE: in BIT_VECTOR (1 to 8);
        signal A: out BIT_VECTOR (1 to 8)
    );
end example5_14;

architecture behave of example5_14 is

begin
    process (B, COPY_ENABLE)
    begin

        A <= "00000000";
```

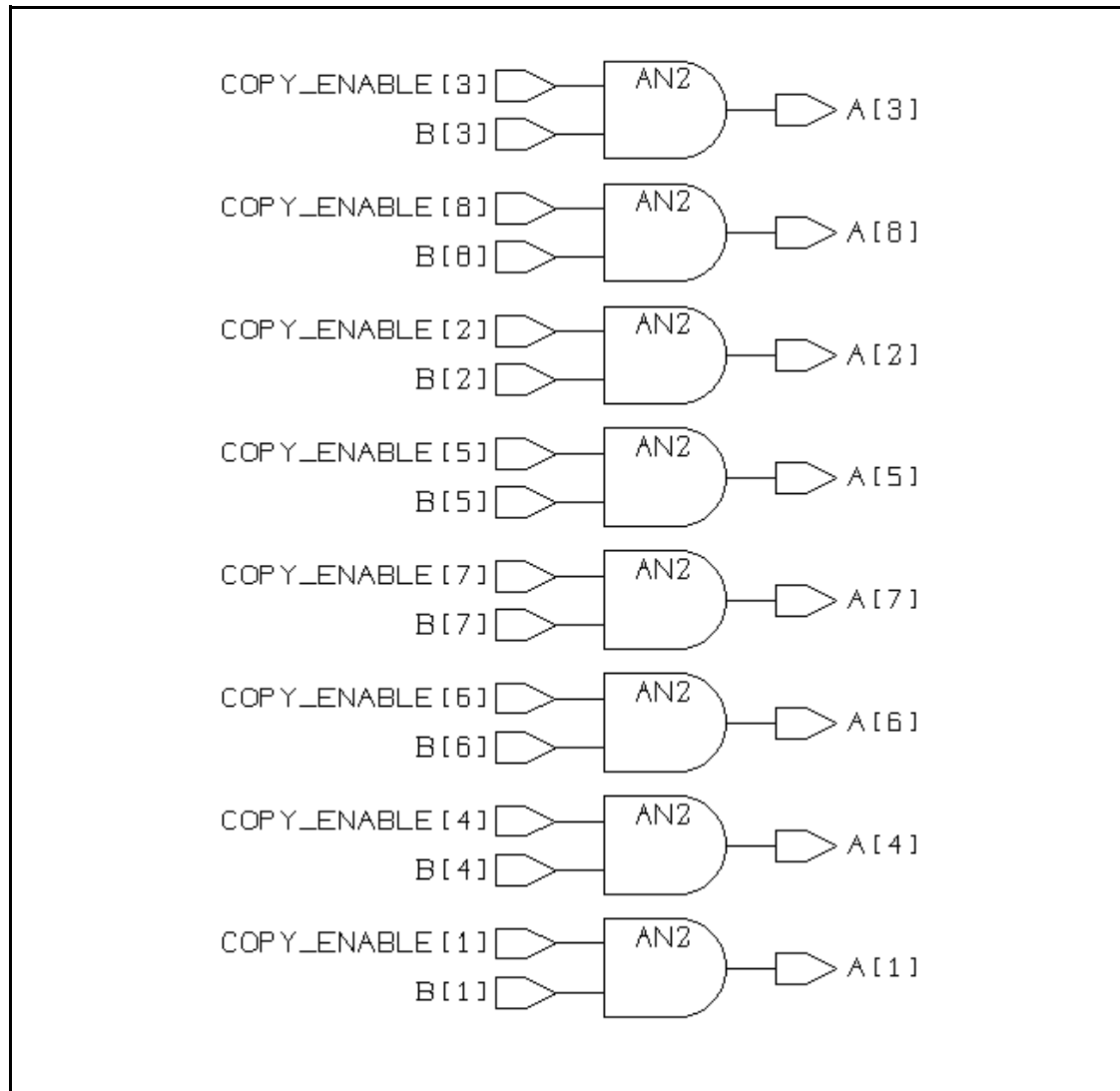
```

for I in 1 to 8 loop
    next when COPY_ENABLE(I) = '0';
    A(I) <= B(I);
end loop;

end process;
end behave;

```

Figure 5-7 Schematic Design From Example 5-14



Example 5-15 shows the use of nested next statements in named loops. This example processes

- The first element of vector X against the first element of vector Y
- The second element of X against each of the first two elements of Y
- The third element of X against each of the first three elements of Y

The processing continues in this fashion until it is completed.

Example 5-15 Named next Statement

```
signal X, Y: BIT_VECTOR(0 to 7);

A_LOOP: for I in X'range loop
. . .
  B_LOOP: for J in Y'range loop
    . . .
    next A_LOOP when I < J;
    . . .
  end loop B_LOOP;
. . .
end loop A_LOOP;
```

exit Statements

The exit statement completes execution of an enclosing loop statement, called label in the syntax. The completion is conditional if the statement includes a condition, such as the when condition in the following syntax:

```
exit [ label ] [ when condition ] ;
```

label

An exit statement with no label terminates the current iteration of the innermost enclosing loop. When you specify a loop label, the current iteration of that named loop is terminated, as shown previously in Example 5-15.

when

An optional clause that executes its next statement when its condition (a Boolean expression) evaluates true.

Note:

The exit statement and the next statement have identical syntax, and they both skip the remainder of the enclosing (or named) loop. The difference between them is that exit terminates its loop and next continues with the next loop iteration (if any).

Example 5-16 compares two bit vectors. An exit statement exits the comparison loop when a difference is found. Figure 5-8 illustrates the corresponding design.

Example 5-16 Comparator That Uses the exit Statement

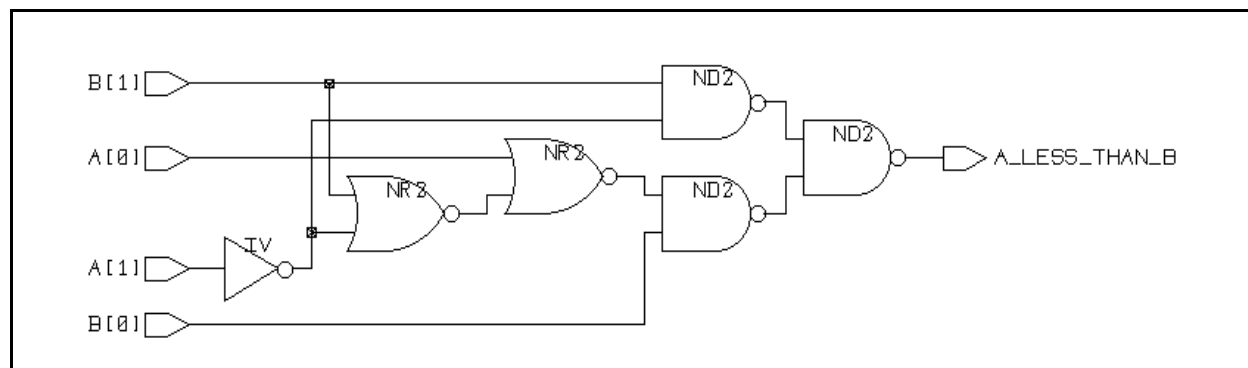
```
entity example5_16 is
    port(
        signal A, B: in BIT_VECTOR(1 downto 0);
        signal A_LESS_THAN_B: out BOOLEAN;
    );
end example5_16;

architecture behave of example5_16 is

begin
    process(A,B)
    begin
        A_LESS_THAN_B <= FALSE;

        for I in 1 downto 0 loop
            if (A(I) = '1' and B(I) = '0') then
                A_LESS_THAN_B <= FALSE;
                exit;
            elsif (A(I) = '0' and B(I) = '1') then
                A_LESS_THAN_B <= TRUE;
                exit;
            else
                null;          -- Continue comparing
            end if;
        end loop;
    end process;
end behave;
```

Figure 5-8 Schematic Design From Example 5-16



Subprograms

Subprograms are independent, named algorithms. A subprogram is either a procedure (zero or more in, inout, or out parameters) or a function (zero or more in parameters and one return value).

Subprograms are called by name from anywhere within a VHDL architecture or a package body. Subprograms can be called sequentially (as described later in this chapter in “Combinational Versus Sequential Processes” on page 5-55) or concurrently (as described in Chapter 6, “Concurrent Statements”).

Subprogram Always a Combinational Circuit

In hardware terms, a subprogram call is similar to module instantiation, except that a subprogram call becomes part of the current circuit. A module instantiation adds a level of hierarchy to the design. A synthesized subprogram is always a combinational circuit. (Use a process to create a sequential circuit.)

Subprogram Declaration and Body

Subprograms, like packages, have declarations and bodies. A subprogram declaration specifies the subprogram’s name, parameters, and return value (for functions). The subprogram body then implements the operation you want.

Often a package contains only type and subprogram declarations for use by other packages. The bodies of the declared subprograms are then implemented in the bodies of the declaring packages.

The advantage of the separation between declarations and bodies is that subprogram interfaces can be declared in public packages during system development. One group of developers can use the public subprograms as another group develops the corresponding bodies. You can modify package bodies, including subprogram bodies, without affecting existing users of that package's declarations.

You can also define subprograms locally inside an entity, block, or process.

FPGA Compiler II / FPGA *Express* implements procedure and function calls with combinational logic, unless you use the `map_to_entity` compiler directive (see “Procedures and Functions as Design Components” on page 5-45.) FPGA Compiler II / FPGA *Express* does not allow inference of sequential devices, such as latches or flip-flops, in subprograms.

Example 5-17 shows a package containing some procedure and function declarations and bodies. The example itself is not synthesizable; it just creates a template. Designs that instantiate procedure P, however, compile normally.

For more information about subprograms, see “Subprograms” on page 2-22.

Example 5-17 Subprogram Declarations and Bodies

```
package EXAMPLE is
  procedure P (A: in INTEGER; B: inout INTEGER);
    -- Declaration of procedure P

  function INVERT (A: BIT) return BIT;
    -- Declaration of function INVERT
end EXAMPLE;

package body EXAMPLE is
  procedure P (A: in INTEGER; B: inout INTEGER) is
    -- Body of procedure P
  begin
    B := A + B;
  end;

  function INVERT (A: BIT) return BIT is
    -- Body of function INVERT
  begin
    return (not A);
  end;
end EXAMPLE;
```

Subprogram Calls

Subprograms can have zero or more parameters. A subprogram declaration defines each parameter's name, mode, and type. These are a subprogram's formal parameters. When the subprogram is called, each formal parameter receives a value, termed the "actual" parameter. Each actual parameter's value (of an appropriate type) might come from an expression, a variable, or a signal.

The mode of a parameter specifies whether the actual parameter can be

- read from (mode in)
- written to (mode out)
- both read from and written to (mode inout)

Actual parameters that use mode out and mode inout must be variables or signals and include indexed names (A(1)) and slices (A(1 to 3)). They cannot be constants or expressions.

The two kinds of subprograms are procedures and functions:

Procedures

A procedure can have multiple parameters that use modes in, inout, and out, but a procedure does not itself return a value.

Procedures are used when you want to update some parameters (modes out and inout) or when you do not need a return value.

An example could be a procedure with one inout bit vector parameter that inverted each bit in place.

Functions

A function can have multiple parameters but only parameters that use mode in. A function returns its own function value. Part of a function definition specifies its return value type (also called the function type).

Use functions when you do not need to update the parameters and you want a single return value. For example, the arithmetic function ABS returns the absolute value of its parameter.

Procedure Calls

A procedure call executes the named procedure with the given parameter values. The syntax is

```
procedure_name [ ( [ name => ] expression  
                  { , [ name => ] expression } ) ] ;
```

expression

Each expression is called an actual parameter; expression is often just an identifier. If a name is present (positional notation), it is a formal parameter name associated with the actual parameter's expression.

Formal parameters are matched to actual parameters by a positional or named notation. A notation can mix positional and named notation, but positional parameters must precede named parameters.

A procedure call occurs in three steps:

1. FPGA Compiler II / *FPGA Express* assigns the values of the in and inout actual parameters to their associated formal parameters.
2. The procedure executes.
3. FPGA Compiler II / *FPGA Express* assigns the values of the inout and out formal parameters to the actual parameters.

In the synthesized circuit, the procedure's actual inputs and outputs are wired to the procedure's internal logic.

Example 5-18 shows a local procedure named SWAP that compares two elements of an array and exchanges them if they are out of order.

SWAP is called repeatedly to sort an array of three numbers. Figure 5-8 illustrates the corresponding design.

Example 5-18 Procedure Call to Sort an Array

```
library IEEE;
use IEEE.std_logic_1164.all;

package DATA_TYPES is
    type DATA_ELEMENT is range 0 to 1;
    type DATA_ARRAY is array (1 to 3) of DATA_ELEMENT;
end DATA_TYPES;

library IEEE;
use IEEE.std_logic_1164.all;
use WORK.DATA_TYPES.ALL;

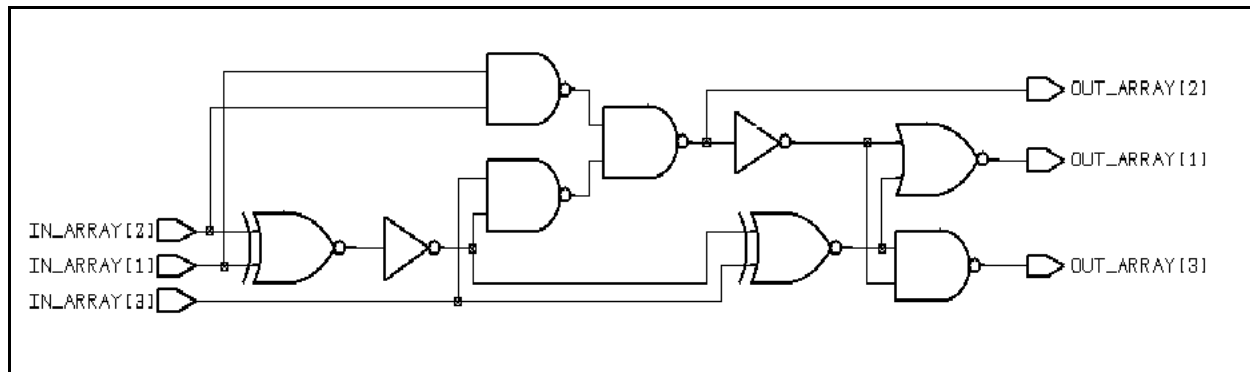
entity SORT is
    port(IN_ARRAY: in DATA_ARRAY;
         OUT_ARRAY: out DATA_ARRAY);
end SORT;

architecture EXAMPLE of SORT is
begin
    process(IN_ARRAY)
        procedure SWAP(DATA: inout DATA_ARRAY;
                       LOW, HIGH: in INTEGER) is
            variable TEMP: DATA_ELEMENT;
        begin
            if(DATA(LOW) > DATA(HIGH)) then -- Check data
                TEMP := DATA(LOW);
                DATA(LOW) := DATA(HIGH); -- Swap data
                DATA(HIGH) := TEMP;
            end if;
        end SWAP;

        variable MY_ARRAY: DATA_ARRAY;

    begin
        MY_ARRAY := IN_ARRAY; -- Read input to variable
        -- Pair-wise sort
        SWAP(MY_ARRAY, 1, 2); -- Swap first and second
        SWAP(MY_ARRAY, 2, 3); -- Swap second and third
        SWAP(MY_ARRAY, 1, 2); -- Swap 1st and 2nd again
        OUT_ARRAY <= MY_ARRAY; -- Write result to output
    end process;
end EXAMPLE;
```

Figure 5-9 Schematic Design From Example 5-18



Function Calls

A function call executes a named function with the given parameter values. The value returned to an operator is the function's return value. The syntax is

```
function_name ( [parameter_name =>] expression
                {, [parameter_name =>] expression } ) ;
```

function_name

Name of a defined function. The parameter_name, which is optional, is the name of a formal parameter as defined by the function. Each expression provides a value for its parameter and must evaluate to a type appropriate for that parameter.

You can specify parameter values in positional or named notation, as you can aggregate values.

In positional notation, the parameter_name => construct is omitted. The first expression provides a value for the function's first parameter, the second expression is for the second parameter, and so on.

In named notation, `parameter_name =>` is specified before an expression; the named parameter gets the value of that expression.

You can mix positional and named expressions in the same function call if you put all positional expressions before named parameter expressions.

Example 5-19 shows a simple function definition and two calls to that function.

Example 5-19 Function Definition With Two Calls

```
function INVERT (A : BIT) return BIT is
begin
    return (not A);
end;
...
process
    variable V1, V2, V3: BIT;
begin
    V1 := '1';
    V2 := INVERT(V1) xor 1;
    V3 := INVERT('0');
end process;
```

return Statement

The return statement terminates a subprogram. A function definition requires a return statement. In a procedure definition, a return statement is optional. The syntax is

```
return expression ;      -- Functions
return ;                  -- Procedures
```

expression

Provides the return value of a function. Every function must have at least one return statement and can have more than one. The expression type must match the declared function type. Only one return statement is reached by a given function call.

procedure

Can have one or more return statements but no expression. A return statement, if present, is the last statement executed in a procedure.

In Example 5-20, the function OPERATE returns either the and logical operator or the or logical operator of its parameters A and B. The return depends on the value of the parameter OPERATION. Figure 5-10 illustrates the corresponding design.

Example 5-20 Use of Multiple return Statements

```
package test is
    function OPERATE(A, B, OPERATION: BIT) return BIT;
end test;

package body test is

function OPERATE(A, B, OPERATION: BIT) return BIT is
begin
    if (OPERATION = '1') then
        return (A and B);
    else
        return (A or B);
    end if;
end OPERATE;
end test;

library IEEE;
use IEEE.std_logic_1164.all;
use WORK.test.all;

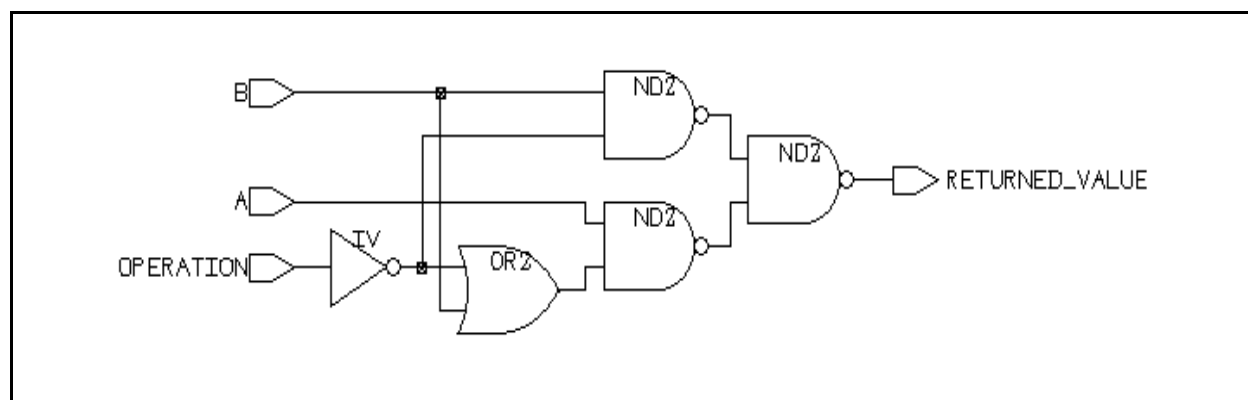
entity example5_20 is
    port(
        signal A, B, OPERATION: in BIT;
        signal RETURNED_VALUE: out BIT
    );
end example5_20;

architecture behave of example5_20 is

begin

RETURNED_VALUE <= OPERATE(A, B, OPERATION);
end behave;
```


Figure 5-10 Schematic Design From Example 5-20



Procedures and Functions as Design Components

In VHDL, entities cannot be invoked from within behavioral code. Procedures and functions cannot exist as entities (components) but must be represented by gates.

You can overcome this limitation with the compiler directive `map_to_entity`, which causes FPGA Compiler II / *FPGA Express* to implement a function or procedure as a component instantiation. Procedures and functions that use `map_to_entity` are represented as components in designs where they are called.

When you add a `map_to_entity` directive to a subprogram definition, FPGA Compiler II / *FPGA Express* assumes the existence of an entity with the identified name and the same interface.

FPGA Compiler II / *FPGA Express* does not check this assumption until it links the parent design. The matching entity must have the same input and output port names. If the subprogram is a function, you must also provide a `return_port_name` directive where the matching entity has an output port of the same name.

These two directives are called component implication directives:

```
-- pragma map_to_entity    entity_name  
-- pragma return_port_name port_name
```

Insert these directives after the function or procedure definition, as in the following example:

```
function MUX_FUNC(A,B: in TWO_BIT; C: in BIT)  
    return TWO_BIT is  
  
-- pragma map_to_entity MUX_ENTITY  
-- pragma return_port_name Z  
...
```

When FPGA Compiler II / *FPGA Express* encounters the `map_to_entity` directive, it parses but ignores the contents of the subprogram definition.

Use `--pragma synthesis_off` and `--pragma synthesis_on` to hide simulation-specific constructs in a `map_to_entity` subprogram (see “Translation Stop and Start Pragma Directives” on page 9-3 for more information about `synthesis_off` and `synthesis_on`).

The matching entity (`entity_name`) does not need to be written in VHDL. It can be in any format that *FPGA Compiler II / FPGA Express* supports.

Note:

Be aware that the behavioral description of the subprogram is not checked against the functionality of the entity overloading it. Pre-synthesis and post-synthesis simulation results might not match if differences in functionality exist between the VHDL subprogram and the overloaded entity.

Example With Component Implication Directives

Example 5-21 shows a function that uses component implication directives. Figure 5-11 illustrates the corresponding design.

Example 5-21 Using Component Implication Directives on a Function

```
package MY_PACK is
    subtype TWO_BIT is BIT_VECTOR(1 to 2);
    function MUX_FUNC(A,B: in TWO_BIT; C: in BIT) return
        TWO_BIT;
end;

package body MY_PACK is

    function MUX_FUNC(A,B: in TWO_BIT; C: in BIT) return
        TWO_BIT is

        -- pragma map_to_entity MUX_ENTITY
        -- pragma return_port_name Z

        -- contents of this function are ignored but should match the
        -- functionality of the module MUX_ENTITY, so pre- and post
        -- simulation will match
    begin
        if(C = '1') then
            return(A);
        else
            return(B);
        end if;
    end;
end;

use WORK.MY_PACK.ALL;
entity TEST is
    port(A: in TWO_BIT; C: in BIT; TEST_OUT: out TWO_BIT);
end;

architecture ARCH of TEST is
begin
    process
    begin
        TEST_OUT <= MUX_FUNC(not A, A, C);
                                -- Component implication call
    end process;
end ARCH;

use WORK.MY_PACK.ALL;
```

```

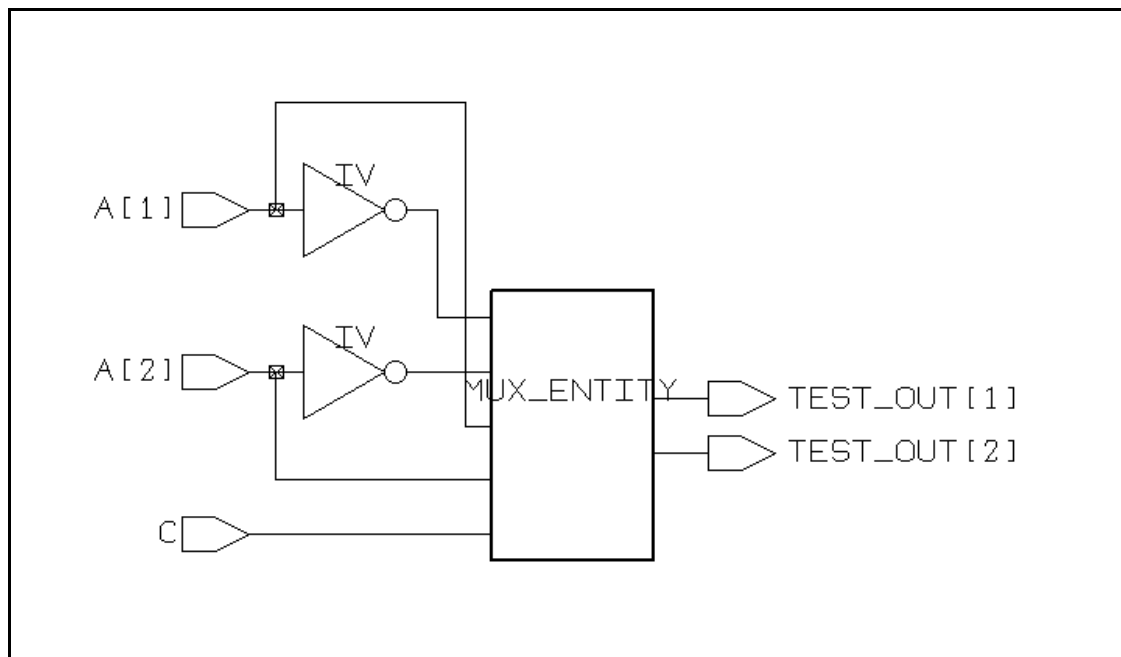
-- the following entity 'overloads' the function MUX_FUNC above

entity MUX_ENTITY is
  port(A, B: in TWO_BIT; C: in BIT; Z: out TWO_BIT);
end;

architecture ARCH of MUX_ENTITY is
begin
  process
  begin
    case C is
      when '1' => Z <= A;
      when '0' => Z <= B;
    end case;
  end process;
end ARCH;

```

Figure 5-11 *Schematic Design With Component Implication Directives*



Example Without Component Implication Directives

Example 5-22 shows the same design as Example 5-21, but without the creation of an entity for the function. The component implication directives have been removed. Figure 5-12 illustrates the corresponding design.

Example 5-22 Using Gates to Implement a Function

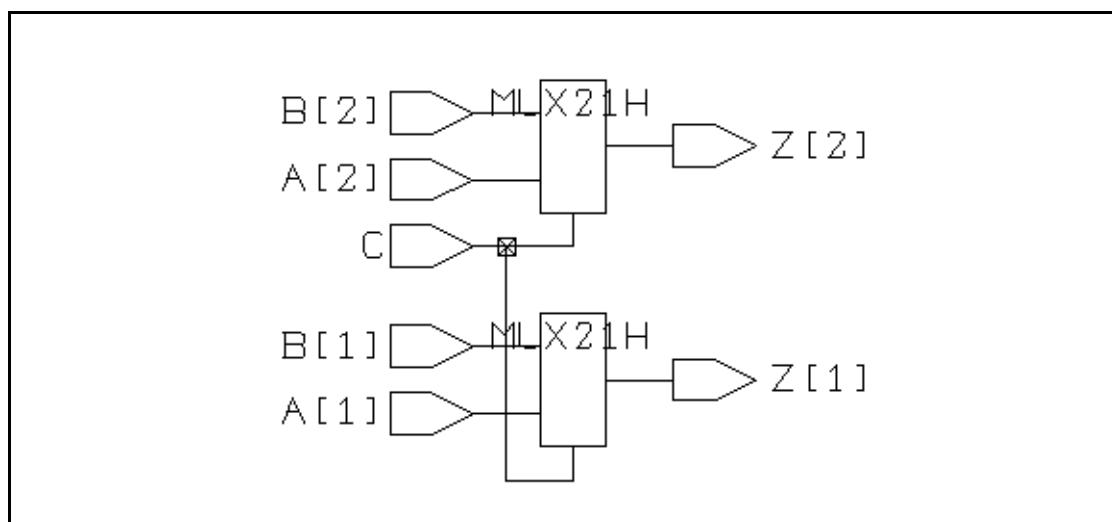
```
package MY_PACK is
    subtype TWO_BIT is BIT_VECTOR(1 to 2);
    function MUX_FUNC(A,B: in TWO_BIT; C: in BIT)
        return TWO_BIT;
end;

package body MY_PACK is
    function MUX_FUNC(A,B: in TWO_BIT; C: in BIT)
        return TWO_BIT is
    begin
        if(C = '1') then
            return(A);
        else
            return(B);
        end if;
    end;
end;

use WORK.MY_PACK.ALL;
entity TEST is
    port(A: in TWO_BIT; C: in BIT; Z: out TWO_BIT);
end;

architecture ARCH of TEST is
begin
    process
    begin
        Z <= MUX_FUNC(not A, A, C);
    end process;
end ARCH;
```

Figure 5-12 Schematic Design Without Component Implication Directives



wait Statements

A wait statement suspends a process until FPGA Compiler II / FPGA Express detects a positive-going or negative-going edge on a signal. The syntax is

```
wait until signal = value ;  
wait until signal'event and signal = value ;  
wait until not signal'stable  
         and signal = value ;
```

signal

The name of a single-bit signal—a signal of an enumerated type encoded with 1 bit (see Chapter 3, "Data Types"). The value must be one of the literals of the enumerated type. If the signal type is BIT, the awaited value is either '1', for a positive-going edge, or '0', for a negative-going edge.

Note:

Three forms of the wait statement (a subset of IEEE VHDL), shown in the syntax above and in Example 5-23, are specific to the current implementation of FPGA Compiler II / *FPGA Express*.

Inferring Synchronous Logic

A wait statement implies synchronous logic where signal is usually a clock signal. “Combinational Versus Sequential Processes” on page 5-55, describes how FPGA Compiler II / *FPGA Express* infers and implements this logic.

Example 5-23 shows three equivalent wait statements (all positive edge-triggered).

Example 5-23 Equivalent wait Statements

```
wait until CLK = '1';  
wait until CLK'event and CLK = '1';  
wait until not CLK'stable and CLK = '1';
```

When a circuit is synthesized, the hardware in the three forms of wait statements does not differ.

Example 5-24 shows a wait statement that suspends a process until the next positive edge (a 0-to-1 transition) on signal CLK.

Example 5-24 wait for a Positive Edge

```
signal CLK: BIT;  
...  
process  
begin  
    wait until CLK'event and CLK = '1';  
    -- Wait for positive transition (edge)  
    ...  
end process;
```

Note:

IEEE VHDL specifies that a process containing a wait statement must not have a sensitivity list. For more information, see “process Statements” on page 6-2.

Example 5-25 shows the use of a wait statement to describe a circuit where a value is incremented on each positive clock edge.

Example 5-26 shows the use of multiple wait statements to describe a multicycle circuit. The circuit provides an average value of its input A over four clock cycles.

Example 5-27 shows two equivalent descriptions, the first with implicit state logic and the second with explicit state logic.

Example 5-25 Loop That Uses a wait Statement

```
process
begin
  y <= 0;
  wait until (clk'event and clk = '1');
  while (y < MAX) loop
    wait until (clk'event and clk = '1');
    x <= y ;
    y <= y + 1;
  end loop;
end process;
```

Example 5-26 Multiple wait Statements

```
process
begin
  wait until CLK'event and CLK = '1';
  AVE <= A;
  wait until CLK'event and CLK = '1';
  AVE <= AVE + A;
  wait until CLK'event and CLK = '1';
  AVE <= AVE + A;
  wait until CLK'event and CLK = '1';
  AVE <= (AVE + A)/4;
end process;
```


Example 5-27 *wait Statements and State Logic*

```
--Implicit State Logic
process
begin
    wait until CLOCK'event and CLOCK = '1';
    if (CONDITION) then
        X <= A;
    else
        wait until CLOCK'event and CLOCK = '1';
    end if;
end process;

-- Explicit State Logic
type STATE_TYPE is (S0, S1);
variable STATE : STATE_TYPE;
...
process
begin
    wait until CLOCK'event and CLOCK = '1';
    case STATE is
        when S0 =>
            if (CONDITION) then
                X <= A;
                STATE := S0;
            else
                STATE := S1;
            end if;
        when S1 =>
            STATE := S0;
    end case;
end process;
```

Note:

You can use wait statements anywhere in a process except in for...loop statements and subprograms. However, if any path through the logic has one or more wait statements, all the paths must have at least one wait statement.

Example 5-28 shows how to describe a circuit with synchronous reset, using wait statements in an infinite loop. FPGA Compiler II / FPGA *Express* checks the reset signal immediately after each wait statement. The assignment statements in Example 5-28 (X <= A; and Y <= B;) represent the sequential statements that implement the circuit.

Example 5-29 shows two invalid uses of wait statements. These limitations are specific to FPGA Compiler II / *FPGA Express*.

Example 5-28 Synchronous Reset That Uses wait Statements

```
process
begin
  RESET_LOOP: loop
    wait until CLOCK'event and CLOCK = '1';
    next RESET_LOOP when (RESET = '1');
    X <= A;
    wait until CLOCK'event and CLOCK = '1';
    next RESET_LOOP when (RESET = '1');
    Y <= B;
  end loop RESET_LOOP;
end process;
```

Example 5-29 Invalid Uses of wait Statements

```
...
type COLOR is (RED, GREEN, BLUE);
attribute ENUM_ENCODING : STRING;
attribute ENUM_ENCODING of COLOR : type is "100 010 001";
signal CLK : COLOR;
...
process
begin
  wait until CLK'event and CLK = RED;
  -- Illegal: clock type is not encoded with 1 bit
  ...
end;
...

process
begin
  if (X = Y) then
    wait until CLK'event and CLK = '1';
    ...
  end if;
  -- Illegal: not all paths contain wait statements
  ...
end;
```

Combinational Versus Sequential Processes

Synthesis of a process that contains no wait statements uses combinational logic. The computations the process performs react immediately to changes in input signals.

Synthesis of a process that contains one or more wait statements uses sequential logic. The process performs computations only one time for each specified clock edge (positive or negative) and saves the results of these computations until the next clock edge by storing them in flip-flops.

The following values are stored in flip-flops:

- Signals driven by the process; see “Signal Assignment Statements” on page 5-12
- State vector values, where the state vector can be implicit or explicit (as in Example 5-27)
- Variables that might be read before they are set

Note:

As with the wait statement, some uses of the if statement can imply synchronous logic, causing FPGA Compiler II / FPGA *Express* to infer registers or latches. These methods are described in Chapter 7, “Register and Three-State Inference”.

Example 5-30 uses a wait statement to store values across clock cycles. The example code compares the parity of a data value with a stored value. The stored value (called CORRECT_PARITY) is set from the NEW_CORRECT_PARITY signal if the SET_PARITY signal is true. Figure 5-13 illustrates the corresponding design.

Example 5-30 Parity Tester That Uses the wait Statement

```
entity example5_30 is
  port(
    signal CLOCK: in BIT;
    signal SET_PARITY: in BOOLEAN;
    signal PARITY_OK: out BOOLEAN;
    signal NEW_CORRECT_PARITY: in BIT;
    signal DATA: in BIT_VECTOR(0 to 3);
  );
end example5_30;

architecture behave of example5_30 is

begin
  process
    variable CORRECT_PARITY, TEMP: BIT;
  begin
    wait until CLOCK'event and CLOCK = '1';

    -- Set new correct parity value if requested
    if (SET_PARITY) then
      CORRECT_PARITY := NEW_CORRECT_PARITY;
    end if;

    -- Compute parity of DATA
    TEMP := '0';
    for I in DATA'range loop
      TEMP := TEMP xor DATA(I);
    end loop;

    -- Compare computed parity with the correct value
    PARITY_OK <= (TEMP = CORRECT_PARITY);
  end process;
end behave;
```

Figure 5-13 Schematic Design From Example 5-30

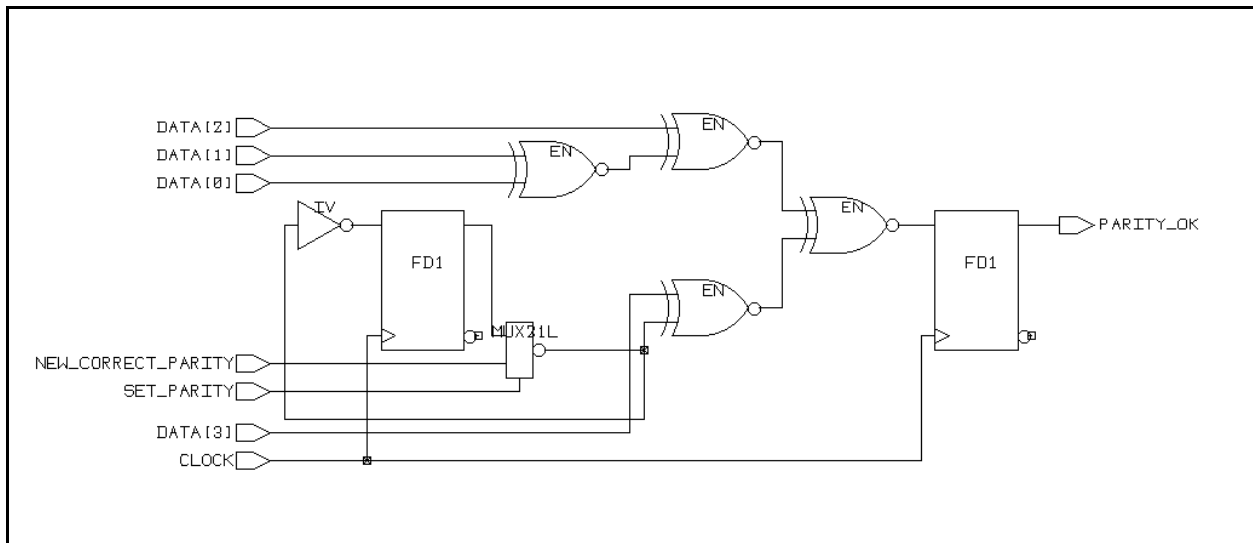


Figure 5-13 shows two flip-flops in the synthesized schematic for Example 5-30. The first (input) flip-flop holds the value of `CORRECT_PARITY`. A flip-flop is needed here because `CORRECT_PARITY` is read (when it is compared to `TEMP`) before it is set (if `SET_PARITY` is false). The second (output) flip-flop holds the value of `PARITY_OK` between clock cycles. The variable `TEMP` is not given a flip-flop because it is always set before it is read.

null Statements

The null statement explicitly states that no action is required. It is often used in case statements because all choices must be covered, even if some of the choices are ignored. The syntax is

```
null;
```

Example 5-31 shows a typical usage. Figure 5-14 illustrates the corresponding design.

Example 5-31 null Statement

```
entity example5_31 is
    port(
        signal CONTROL: in INTEGER range 0 to 7;
        signal A: in BIT;
        signal Z: out BIT
    );
end example5_31;

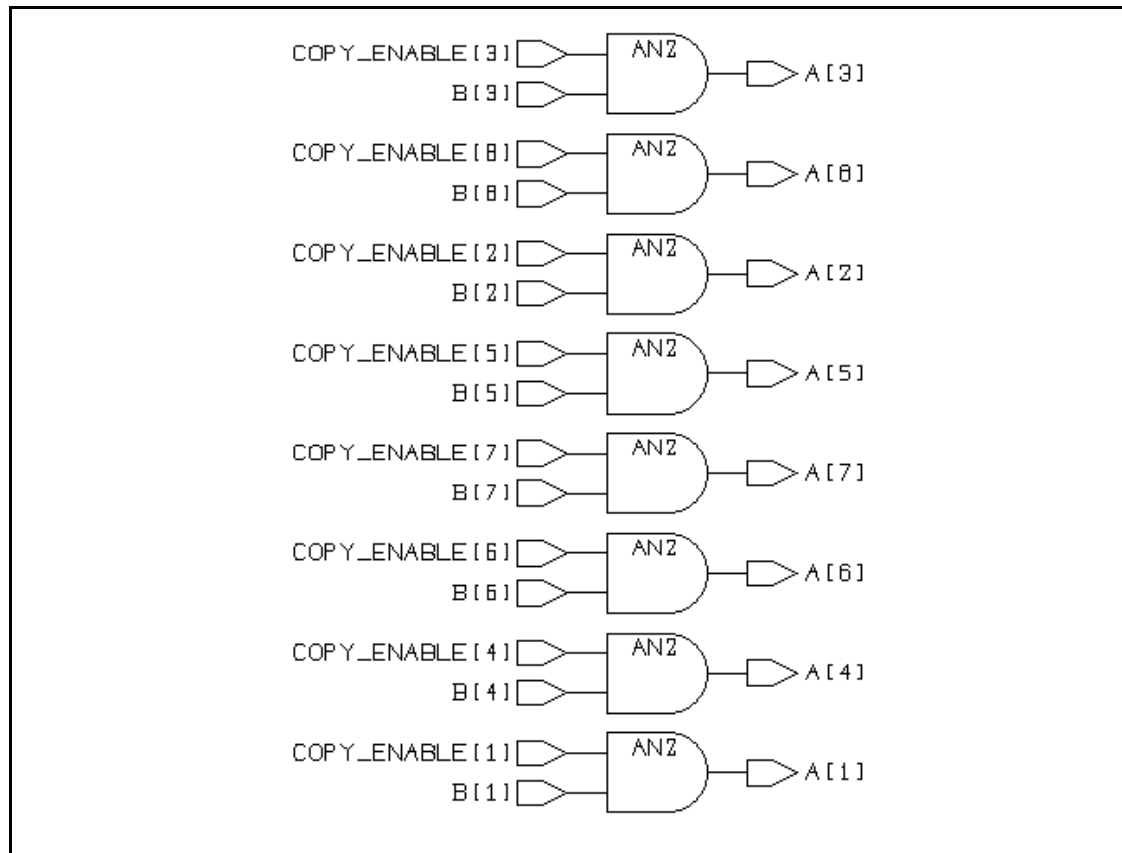
architecture behave of example 5_31 is

begin

    process (CONTROL, A)
    begin

        Z <= A;
        case CONTROL is
            when 0 | 7 =>          -- If 0 or 7, then invert A
                Z <= not A;
            when others =>
                null;              -- If not 0 or 7, then do nothing
        end case;
    end process;
end behave;
```

Figure 5-14 Schematic Design From Example 5-31



6

Concurrent Statements

A VHDL architecture construct comprises a set of interconnected concurrent statements, such as processes and blocks, that describe an overall design in terms of behavior or structure. Concurrent statements in a design execute simultaneously, unlike sequential statements, which execute one after another.

This chapter describes concurrent statements, in the following order:

- The two main concurrent statements
 - process Statements
 - block Statements
- Concurrent Versions of Sequential Statements
 - Concurrent Procedure Calls
 - Concurrent Signal Assignments

- Component Instantiation Statements
- Direct Instantiation
- generate Statements

process Statements

A process statement (which is concurrent) contains a set of sequential statements. Although all processes in a design execute concurrently, FPGA Compiler II / *FPGA Express* interprets the sequential statements within each process one at a time.

A process communicates with the rest of the design by reading values from or writing them to signals or ports outside the process.

The syntax of a process statement is

```
[ label: ] process [ ( sensitivity_list ) ]  
    { process_declarative_item }  
begin  
    { sequential_statement }  
end process [ label ] ;
```

label

A label, which is optional, names the process.

sensitivity_list

A list of all signals (including ports) read by the process. The syntax is

```
signal_name {, signal_name}
```

The circuit FPGA Compiler II / *FPGA Express* synthesizes is sensitive to all signals the process reads. To guarantee the same results from a VHDL simulator and the synthesized circuit, a process sensitivity list has to contain all signals whose changes require resimulation of that process.

Follow these guidelines when developing the sensitivity list:

- Synchronous processes (processes that compute values only on clock edges) must be sensitive to the clock signal.
- Asynchronous processes (processes that compute values on clock edges and when asynchronous conditions are true) must be sensitive to the clock signal (if any) and to inputs that affect asynchronous behavior.

FPGA Compiler II / *FPGA Express* checks sensitivity lists for completeness and issues warning messages for any signals that are read inside a process but are not in the sensitivity list. An error message is issued if a clock signal is read as data in a process.

Note:

IEEE VHDL does not allow a sensitivity list if the process has a wait statement.

process_declarative_item

Declares subprograms, types, constants, and variables local to the process. These items can be any of the following, all of which are discussed in Chapter 2, "Design Descriptions":

- use clause
- Subprogram declaration

- Subprogram body
- Type declaration
- Subtype declaration
- Constant declaration
- Variable declaration

The sequence of statements in a process defines the behavior of the process. After executing all the statements in a process, FPGA Compiler II / *FPGA Express* executes them all again.

The only exception is during simulation: If a process has a sensitivity list, the process is suspended (after its last statement) until a change occurs in one of the signals in the sensitivity list.

If a process has one or more wait statements (and, therefore, no sensitivity list), the process is suspended at the first wait statement whose wait condition is false.

The circuit synthesized for a process is either combinational (not clocked) or sequential (clocked). If a process includes a wait or if signal'event statement, its circuit contains sequential components. The wait and if statements are described in Chapter 5, "Sequential Statements".

Process statements provide a natural means of describing sequential algorithms. If the values computed in a process are inherently parallel, consider using concurrent signal assignment statements (see "Concurrent Signal Assignments" on page 6-17).

Combinational Process Example

Example 6-1 shows a process (with no wait statements) that implements a simple modulo-10 counter. The process

- Reads two signals: CLEAR and IN_COUNT
- Drives one signal, OUT_COUNT

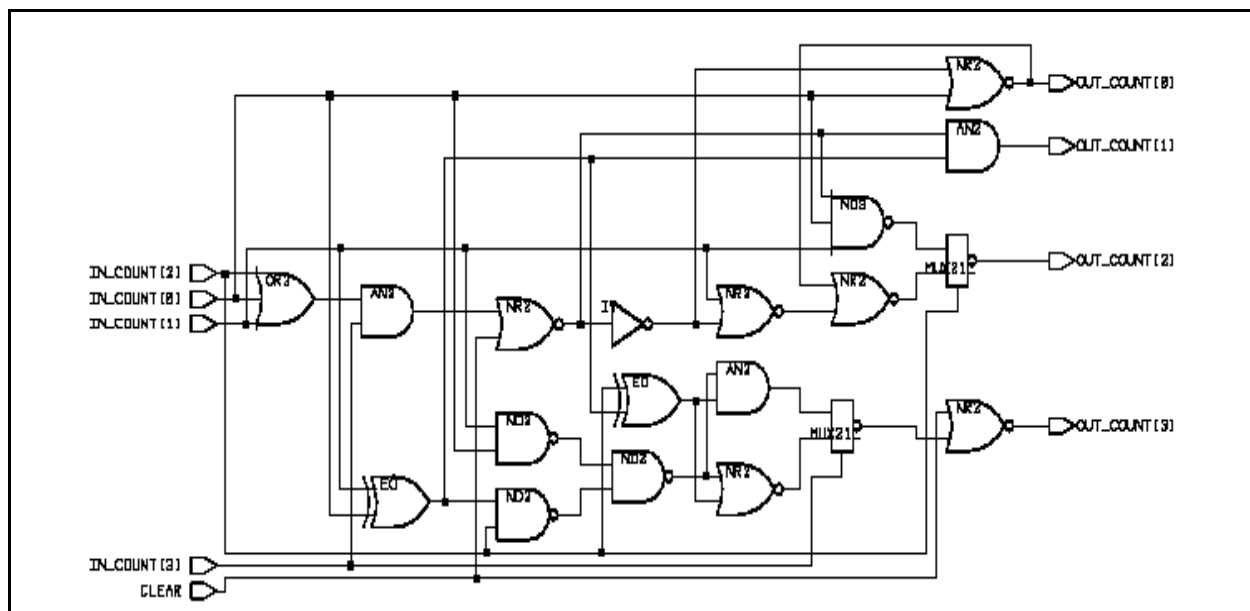
If CLEAR is '1' or IN_COUNT is 9, OUT_COUNT is set to 0 (zero). Otherwise, OUT_COUNT is set to the value of IN_COUNT plus 1 (one).

Figure 6-1 illustrates the resulting circuit design.

Example 6-1 Modulo-10 Counter Process

```
entity COUNTER is
    port (CLEAR:      in BIT;
          IN_COUNT:   in INTEGER range 0 to 9;
          OUT_COUNT:  out INTEGER range 0 to 9);
end COUNTER;
architecture EXAMPLE of COUNTER is
begin
    process(IN_COUNT, CLEAR)
    begin
        if (CLEAR = '1' or IN_COUNT = 9) then
            OUT_COUNT <= 0;
        else
            OUT_COUNT <= IN_COUNT + 1;
        end if;
    end process;
end EXAMPLE;
```

Figure 6-1 Modulo-10 Counter Process Design



Sequential Process Example

Another way to implement the counter in Example 6-1 is to use a wait statement to contain the count value internally in the process.

The process in Example 6-2 implements the counter as a sequential (clocked) process.

- On each 0-to-1 CLOCK transition, if CLEAR is '1' or COUNT is 9, COUNT is set to 0 (zero).
- Otherwise, FPGA Compiler II / FPGA *Express* increments the value of COUNT by 1.

- The value of the variable COUNT is stored in four flip-flops, which FPGA Compiler II / *FPGA Express* generates because COUNT can be read before it is set. Thus, the value of COUNT has to be maintained from the previous clock cycle. For more information on using wait statements and count values, see “wait Statements” on page 5-50.

Figure 6-2 illustrates the resulting circuit design.

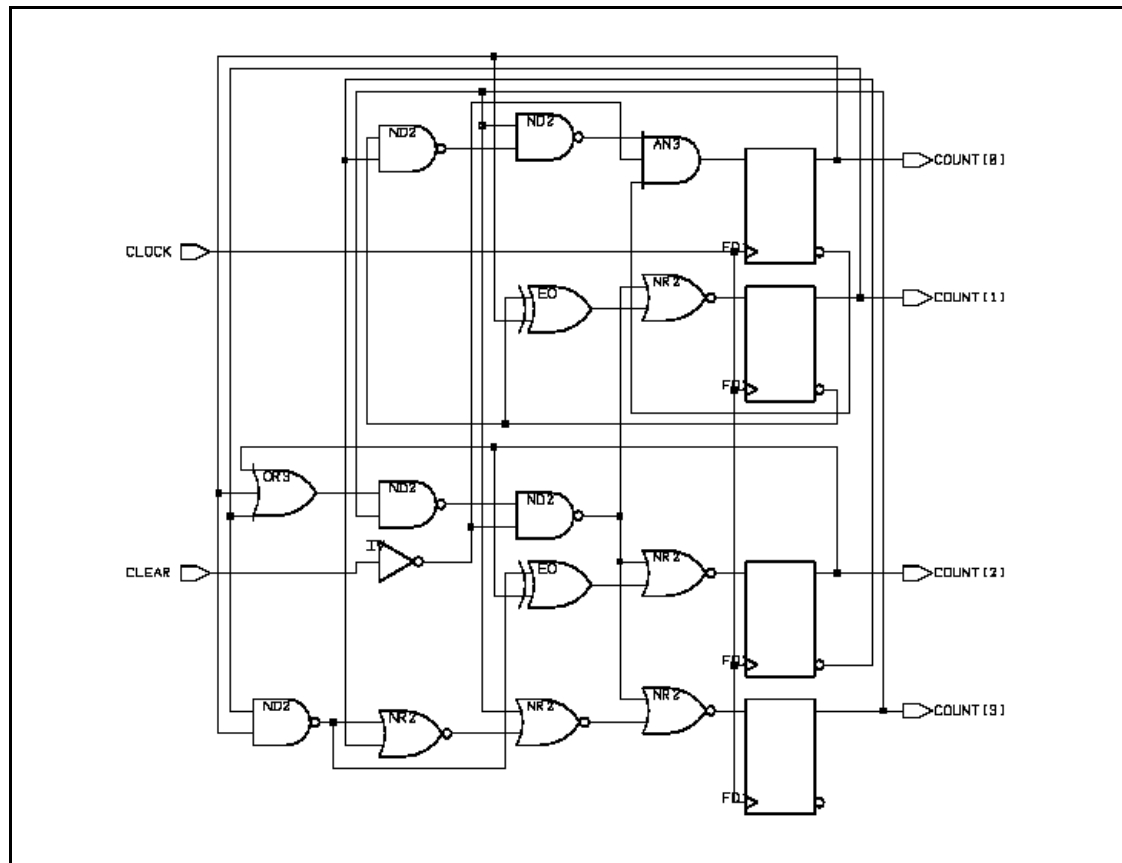
Example 6-2 Modulo-10 Counter Process With wait Statement

```
entity COUNTER is
  port (CLEAR: in BIT;
        CLOCK: in BIT;
        COUNT: buffer INTEGER range 0 to 9);
end COUNTER;

architecture EXAMPLE of COUNTER is
begin
  process
  begin
    wait until CLOCK'event and CLOCK = '1';

    if (CLEAR = '1' or COUNT >= 9) then
      COUNT <= 0;
    else
      COUNT <= COUNT + 1;
    end if;
  end process;
end EXAMPLE;
```

Figure 6-2 Modulo-10 Counter Process With wait Statement Design



Driving Signals

If a process assigns a value to a signal, the process is a driver of that signal. If more than one process or other concurrent statement drives a signal, that signal has multiple drivers.

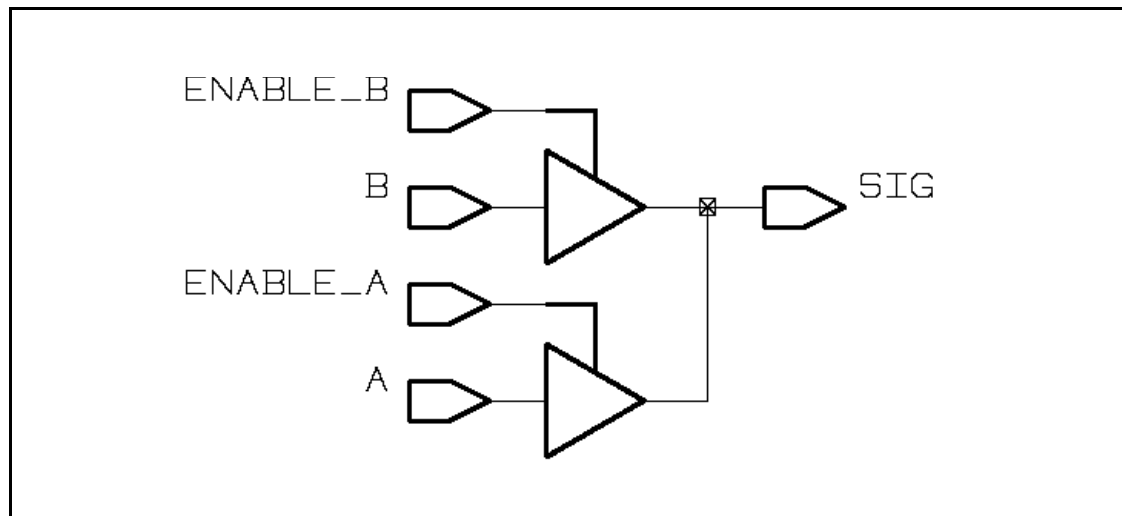
In the code fragment in Example 6-3, two three-state buffers drive the same signal (SIG). To learn to infer three-state devices in VHDL, see “Three-State Inference” on page 7-59.

Figure 6-3 shows the schematic design.

Example 6-3 Multiple Drivers of a Signal

```
A_OUT <= A when ENABLE_A else 'Z';  
B_OUT <= B when ENABLE_B else 'Z';  
process(A_OUT)  
begin  
    SIG <= A_OUT;  
end process;  
process(B_OUT)  
begin  
    SIG <= B_OUT;  
end process;
```

Figure 6-3 Two Three-State Buffers Driving the Same Signal



Bus resolution functions assign the value for a signal with multiple drivers. For more information, see “Resolution Functions” on page 2-40.

block Statements

A block statement (which is concurrent) contains a set of concurrent statements. The order of the concurrent statements does not matter, because all statements are always executing.

Note:

FPGA Compiler II / *FPGA Express* does not create a new level of design hierarchy from a block statement.

The syntax of a block statement is

```
label: block [ (expression) ]  
    { block_declarative_item }  
begin  
    { concurrent_statement }  
end block [ label ];
```

label

The label, which is required, names the block.

expression

The guard condition for the block. When this optional expression is present, FPGA Compiler II / *FPGA Express* evaluates the expression and creates a Boolean signal called GUARD.

block_declarative_item

Declares objects local to the block, which can be any of the following items:

- use clause
- subprogram declaration
- subprogram body
- type declaration

- subtype declaration
- constant declaration
- signal declaration
- component declaration

Objects declared in a block are visible to that block and to all blocks nested within it. When a child block (nested inside a parent block) declares an object with the same name as an object in the parent block, the child block's declaration overrides that of the parent.

Nested Blocks

The description in Example 6-4 uses nested blocks. Figure 6-4 shows the schematic.

Example 6-4 Nested Blocks

```

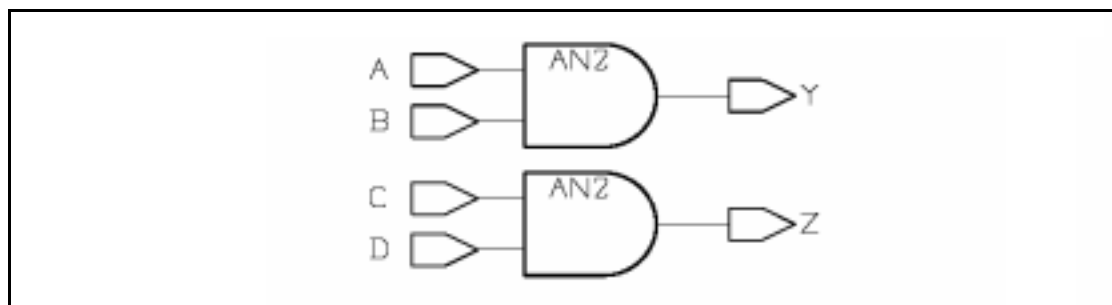
B1: block
  signal S: BIT;  -- Declaration of "S" in block B1
begin
  S <= A and B;   -- "S" from B1

  B2: block
    signal S: BIT; -- Declaration of "S", block B2
    begin
      S <= C and D; -- "S" from B2

      B3: block
        begin
          Z <= S;      -- "S" from B2
        end block B3;
      end block B2;
    Y <= S;           -- "S" from B1
  end block B1;

```

Figure 6-4 Schematic of Nested Blocks



Guarded Blocks

The description in Example 6-5 uses guarded blocks. In the example, z has the same value as a.

Example 6-5 Guarded Blocks

```
entity EG1 is
  port (a: in BIT; z: out BIT);
end;

architecture RTL of EG1 is
begin
  guarded_block: block (a = '1')
  begin
    z <= '1' when guard else '0';
  end block;
end RTL;
```

A concurrent assignment within a block statement can use the guarded keyword. In such a case, the guard expression conditions the signal assignment. The description in Example 6-6 produces a level-sensitive latch.

Example 6-6 Level-Sensitive Latch Using Guarded Blocks

```
entity EG2 is
    port (d, g: in BIT; q: out BIT);
end;

architecture RTL of EG2 is
begin
    guarded_block: block (g = '1')
    begin
        q <= guarded d;
    end block;
end RTL;
```

Note:

Do not use the 'event or 'stable attributes with the guard expression if you want to produce an edge-triggered latch using a guarded block. The presence of either attribute prevents it.

Concurrent Versions of Sequential Statements

This section describes concurrent versions of sequential statements in the form of

- Concurrent Procedure Calls
- Concurrent Signal Assignments
 - Simple Concurrent Signal Assignments
 - Conditional Signal Assignments
 - Selected Signal Assignments

Concurrent Procedure Calls

A concurrent procedure call, which is used in an architecture construct or a block statement, is equivalent to a process with a single sequential procedure call in it (see Example 6-7). The syntax is the same as that of a sequential procedure call:

```
procedure_name [ ( [ name => ] expression
                  { , [ name => ] expression } ) ] ;
```

The equivalent process reads all the in and inout parameters of the procedure. Example 6-7 shows a procedure declaration and a concurrent procedure call and its equivalent process.

Example 6-7 Concurrent Procedure Call and Equivalent Process

```
procedure ADD(signal A, B: in BIT;
              signal SUM: out BIT);

...
ADD(A, B, SUM);    -- Concurrent procedure call
...
process(A, B)      -- The equivalent process
begin
    ADD(A, B, SUM); -- Sequential procedure call
end process;
```

FPGA Compiler II / *FPGA Express* implements procedure calls (and function calls) with logic unless you use the `map_to_entity` compiler directive (see “Procedures and Functions as Design Components” on page 5-45.)

A common use for concurrent procedure calls is to obtain many copies of a procedure. For example, assume that a class of BIT_VECTOR signals must have just 1 bit with value '1' and the rest of the bits with value '0' (as in Example 6-8). Suppose you have several signals of varying widths that you want monitored at the same time (as in Example 6-9). One approach is to write a procedure to detect the error in a bit vector signal and then make a concurrent call to that procedure for each signal.

Example 6-8 shows a procedure, CHECK, that determines whether a given bit vector has exactly one element with value '1'. If this is not the case, CHECK sets its out parameter ERROR to true, as the example shows.

Example 6-8 Procedure Definition for Example 6-9

```

procedure CHECK(signal A:      in BIT_VECTOR;
                signal ERROR: out BOOLEAN) is

    variable FOUND_ONE: BOOLEAN := FALSE;
    -- Set TRUE when a '1' is seen
begin
    for I in A'range loop      -- Loop across all bits in the vector
        if A(I) = '1' then    -- Found a '1'
            if FOUND_ONE then -- Have we already found one?
                ERROR <= TRUE; -- Found two '1's
                return;        -- Terminate procedure
            end if;
            FOUND_ONE := TRUE;
        end if;
    end loop;

    ERROR <= not FOUND_ONE; -- Error will be TRUE if no '1' seen
end;
```

Example 6-9 shows the CHECK procedure called concurrently for four bit vector signals that are different sizes. Figure 6-5 illustrates the resulting circuit design.

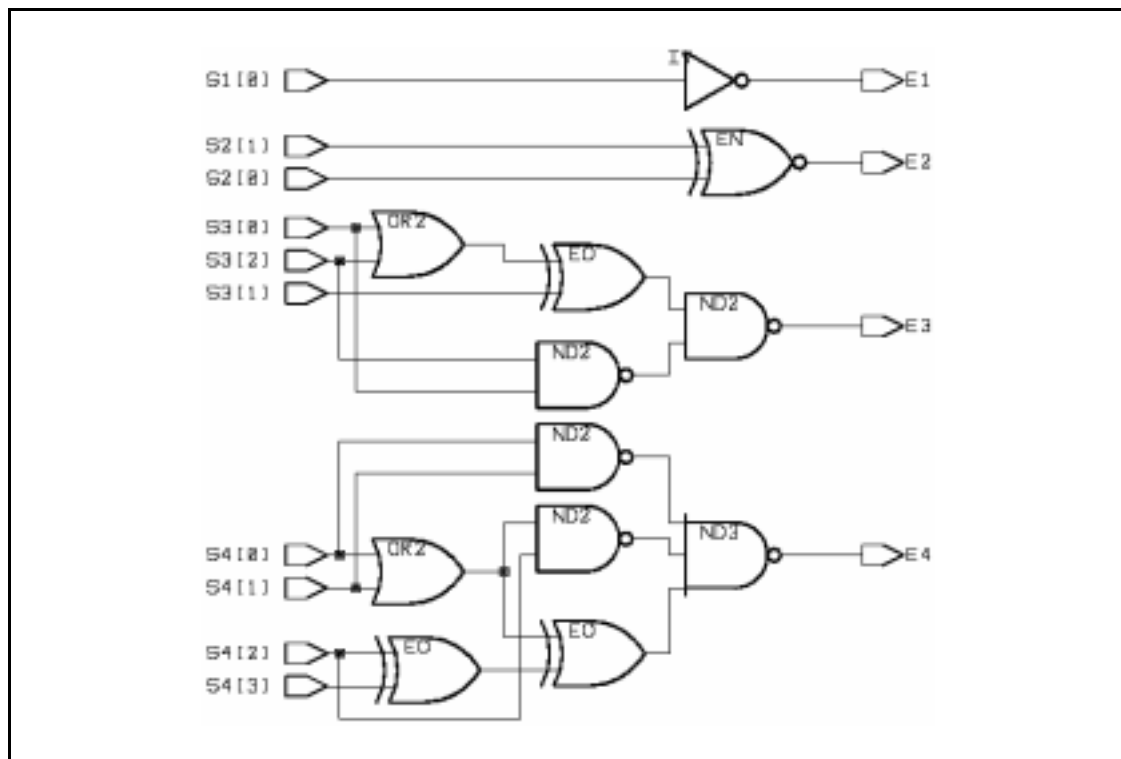
Example 6-9 Concurrent Procedure Calls

```
BLK: block
  signal S1: BIT_VECTOR(0 to 0);
  signal S2: BIT_VECTOR(0 to 1);
  signal S3: BIT_VECTOR(0 to 2);
  signal S4: BIT_VECTOR(0 to 3);

  signal E1, E2, E3, E4: BOOLEAN;

begin
  CHECK(S1, E1);  -- Concurrent procedure call
  CHECK(S2, E2);
  CHECK(S3, E3);
  CHECK(S4, E4);
end block BLK;
```

Figure 6-5 Concurrent CHECK Procedure Design



Concurrent Signal Assignments

A concurrent signal assignment is equivalent to a process containing a sequential assignment. Thus, each concurrent signal assignment defines a new driver for the assigned signal. This section discusses the three forms of concurrent signal assignment.

Simple Concurrent Signal Assignments

The syntax of the simplest form of the concurrent signal assignment is

target <= *expression*;

target

A signal that receives the value of an expression. Example 6-10 shows the value of expressions A and B concurrently assigned to signal Z.

Example 6-10 Concurrent Signal Assignment

```
BLK: block
  signal A, B, Z: BIT;
begin
  Z <= A and B;
end block BLK;
```

Conditional Signal Assignment

The syntax of the conditional signal assignment is

```
target <= { expression when condition else }  
          expression;
```

target

A signal that receives the value of an expression. The expression used is the first one whose Boolean condition is true.

When FPGA Compiler II / *FPGA Express* executes a conditional signal assignment statement, it tests each condition in the order written.

- FPGA Compiler II / *FPGA Express* assigns to the target the expression of the first condition that evaluates to true.
- If no condition evaluates to true, FPGA Compiler II / *FPGA Express* assigns the final expression to the target.
- If two or more conditions are true, FPGA Compiler II / *FPGA Express* assigns only the first one to the target.

Example 6-11 shows a conditional signal assignment. The target is the signal Z, which is assigned from one of the signals A, B, or C. The signal depends on the value of the expressions ASSIGN_A and ASSIGN_B. Figure 6-6 illustrates the resulting design.

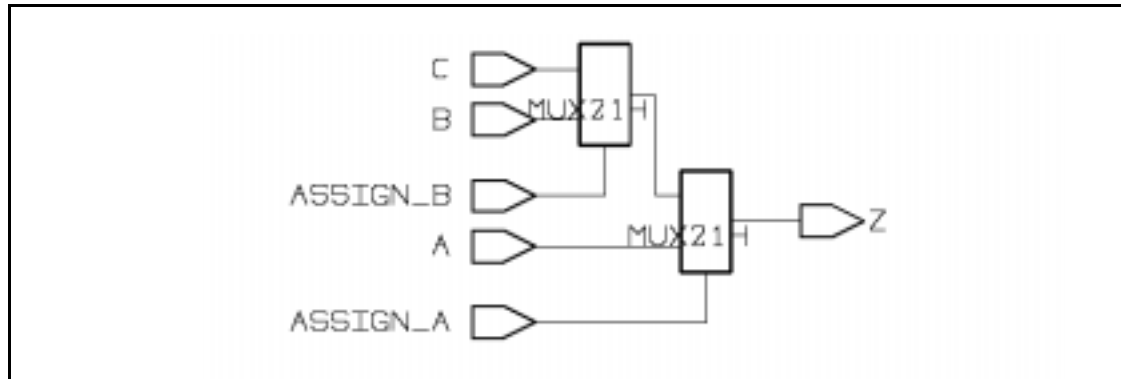
Note:

The A assignment takes precedence over B, and B takes precedence over C, because the first true condition controls the assignment.

Example 6-11 Conditional Signal Assignment

```
Z <= A when ASSIGN_A = '1' else  
  B when ASSIGN_B = '1' else  
  C;
```

Figure 6-6 Conditional Signal Assignment Design



The process in Example 6-12 is equivalent to the conditional signal assignment in Example 6-11.

Example 6-12 Process Equivalent to Conditional Signal Assignment

```
process(A, ASSIGN_A, B, ASSIGN_B, C)  
begin  
  if ASSIGN_A = '1' then  
    Z <= A;  
  elsif ASSIGN_B = '1' then  
    Z <= B;  
  else  
    Z <= C;  
  end if;  
end process;
```

Selected Signal Assignments

The syntax of the selected signal assignment is

```
with choice_expression select  
    target <= { expression when choices, }  
                expression when choices;
```

target

A signal that receives the value of an expression. The expression selected is the first one whose choices include the value of *choice_expression*.

Each choice can be either

- A static expression (such as 3)
- A static range (such as 1 to 3)

The value of each choice the target signal receives has to match the value or values of *choice_expression*.

If the value of *choice_expression* is a static range, each value in the range must be covered by one choice in the expression.

The final choice can be *others*, which matches all remaining (unchosen) values in the range of the *choice_expression* type. The *others* choice, if present, matches *choice_expression* only if none of the other choices match. You can use *others* as the final choice only if the value of *choice_expression* is a range.

The *with...select* statement evaluates *choice_expression* and compares that value with each choice value. The *when* clause with the matching choice value has its expression assigned to *target*.

The use of choices has the following restrictions:

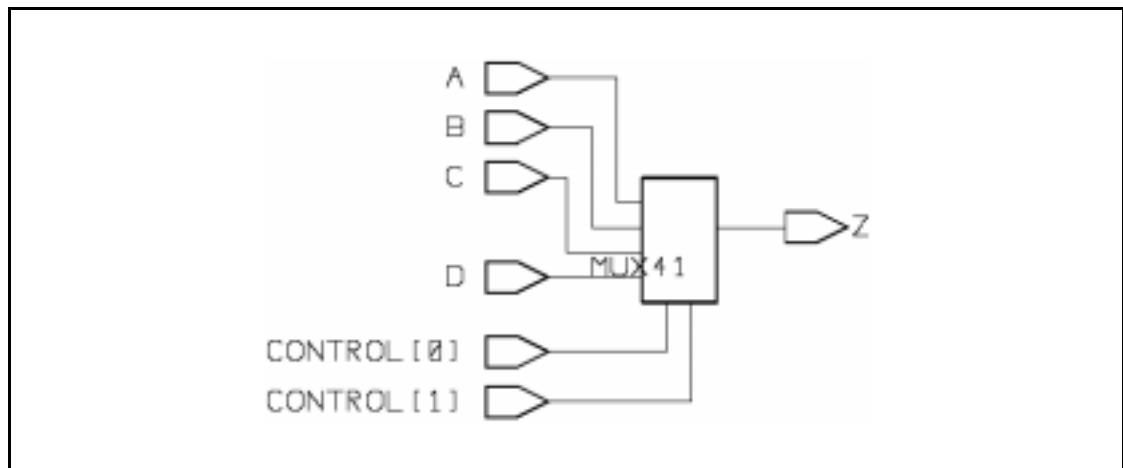
- No two choices can overlap.
- If no others choice is present, all possible values of choice_expression must be covered by the set of choices.

Example 6-13 shows target Z assigned from A, B, C, or D. The assignment depends on the current value of CONTROL. Figure 6-7 illustrates the resulting design.

Example 6-13 Selected Signal Assignment

```
signal A, B, C, D, Z: BIT;  
signal CONTROL: bit_vector(1 down to 0);  
.  
.  
.  
with CONTROL select  
    Z <= A when "00",  
        B when "01",  
        C when "10",  
        D when "11";
```

Figure 6-7 Selected Signal Assignment Design



Example 6-14 shows the process equivalent to the selected signal assignment statement in Example 6-13.

Example 6-14 Process Equivalent to Selected Signal Assignment

```
process(CONTROL, A, B, C, D)
begin
    case CONTROL is
        when 0 =>
            Z <= A;
        when 1 =>
            Z <= B;
        when 2 =>
            Z <= C;
        when 3 =>
            Z <= D;
    end case;
end process;
```

Component Instantiation Statements

The purpose of a component instantiation statement is to define a design hierarchy or build a netlist in VHDL by

- Referencing a previously defined hardware component in the current design, at the current level of hierarchy
- Referencing components not defined in VHDL, such as
 - Components from a technology library (FPGA vendor-specific)
 - Components defined in the Verilog hardware description language

The syntax is

```
instance_name : component_name port map (  
    [ port_name => ] expression  
    {, [ port_name => ] expression } );
```

instance_name

Name of this instance of the component.

component_name

Name of the component port map, which connects each port of this instance of *component_name* to a signal-valued expression in the current entity.

port_name

Name of port.

expression

Name of a signal, indexed name, slice name, or aggregate, to indicate the connection method for the component's ports.

If *expression* is the VHDL reserved word *open*, the corresponding port is left unconnected.

You can map ports to signals by named or positional notation. You can include named as well as positional connections in the port map, but you must put all positional connections before any named connections.

Note:

For named association, the component port names must match exactly the declared component's port names. For positional association, the actual port expressions must be in the same order as the declared component's port order.

Example 6-15 shows a component declaration (a 2-input NAND gate) followed by three equivalent component instantiation statements.

Example 6-15 Component Declaration and Instantiations

```
component ND2
  port(A, B: in BIT; C: out BIT);
end component;
. . .
signal X, Y, Z: BIT;
. . .
U1: ND2 port map(X, Y, Z);           -- positional
U2: ND2 port map(A => X, C => Z, B => Y); -- named
U3: ND2 port map(X, Y, C => Z);      -- mixed
```

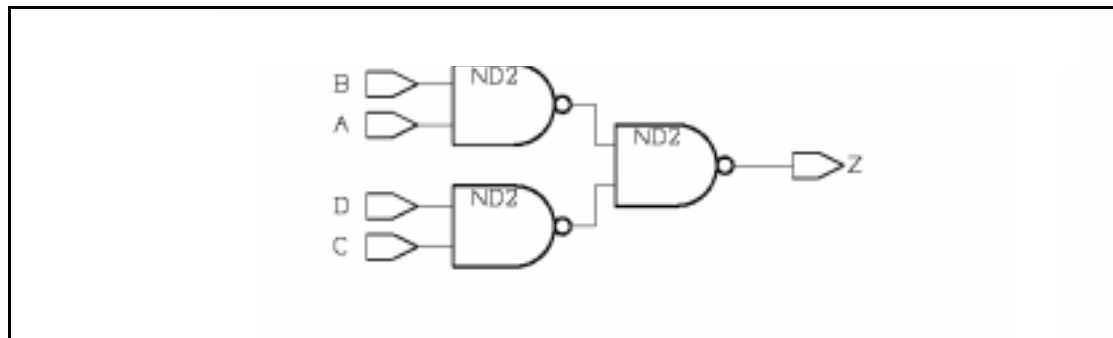
Example 6-16 shows the component instantiation statement defining a simple netlist. The three instances—U1, U2, and U3—are instantiations of the 2-input NAND gate component declared in Example 6-15.

Figure 6-8 illustrates the resulting design.

Example 6-16 A Simple Netlist

```
signal TEMP_1, TEMP_2: BIT;
. . .
U1: ND2 port map(A, B, TEMP_1);
U2: ND2 port map(C, D, TEMP_2);
U3: ND2 port map(TEMP_1, TEMP_2, Z);
```

Figure 6-8 A Simple Netlist Design



Direct Instantiation

A component instantiation statement

- Defines a subcomponent of the design entity in which it appears
- Associates signals or values with the ports of that subcomponent
- Associates values with generics of that subcomponent

Example 6-17 and Example 6-18 show the difference between a component instantiation statement and the more concise direct component instantiation statement.

Example 6-17 Component Instantiation Statement

```
ARCHITECTURE struct OF root IS
  COMPONENT leaf
    PORT (
      clk,data : in std_logic;
      Qout : out std_logic);
  END COMPONENT;
BEGIN
  u1 : leaf
    PORT MAP (
      clk => clk,
      data => d_in(0),
      Qout => q_out(0));
```

Example 6-18 shows how you can express the information in Example 6-17 in a direct component instantiation statement.

Example 6-18 Direct Component Instantiation Statement

```
ARCHITECTURE struct OF root IS
BEGIN
    u1 : entity work.leaf(rtl)
        port map (
            clk => clk,
            data => d_in(0),
            Qout => q_out(0));
```

generate Statements

A generate statement creates zero or more copies of an enclosed set of concurrent statements. The two kinds of generate statements are

`for...generate`

The number of copies is determined by a discrete range.

`if...generate`

Zero or one copy is made, conditionally.

for...generate Statement

The syntax is

```
label: for identifier in range generate
    { concurrent_statement }
end generate [ label ] ;
```

label

The label, which is required, names this statement and is useful for building nested generate statements.

identifier

Specific to the for...generate statement:

- Identifier is not declared elsewhere. It is automatically declared by the generate statement itself and is local to the statement. A for ... generate identifier overrides any other identifier with the same name, but only within the for...generate statement.
- The value of identifier can be read only inside its for...generate statement (identifier does not exist outside the statement). You cannot assign a value to a for...generate identifier.
- The value of identifier cannot be assigned to any parameter whose mode is out or inout.

range

Must be a computable integer range, in either of two forms:

integer_expression to integer_expression

integer_expression downto integer_expression

integer_expression

Each integer_expression evaluates to an integer. Each concurrent_statement can be any of the statements described in this chapter, including other generate statements.

Steps in the Execution of a for...generate Statement

A for...generate statement executes as follows:

1. A new local integer variable is declared with the name identifier.
2. The identifier receives the first value of range, and each concurrent statement executes once.

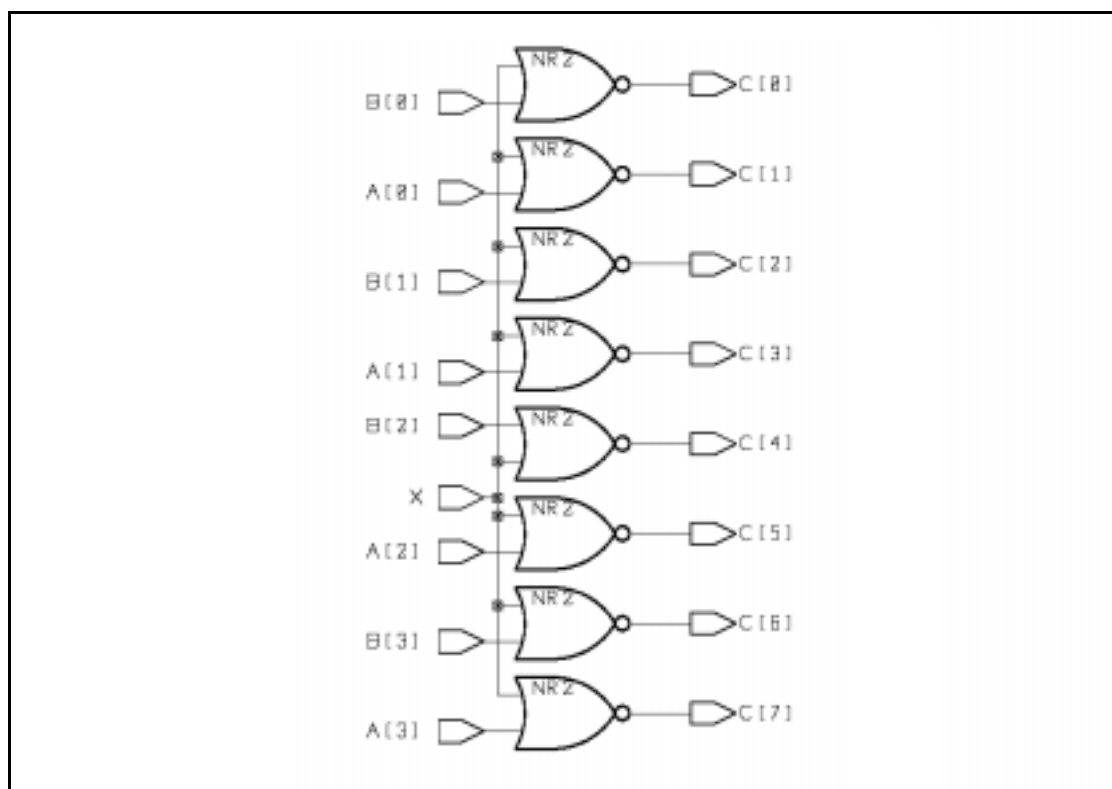
3. The identifier receives the next value of range, and each concurrent statement executes once more.
4. Step 3 repeats until the identifier receives the last value in the range and each concurrent statement executes for the last time. Execution continues with the statement following end generate. The loop identifier is deleted.

Example 6-19 shows a code fragment that combines and interleaves two 4-bit arrays, A and B, into an 8-bit array, C. Figure 6-9 illustrates the resulting design.

Example 6-19 for...generate Statement

```
signal A, B : bit_vector(3 downto 0);
signal C    : bit_vector(7 downto 0);
signal X    : bit;
. . .
GEN_LABEL: for I in 3 downto 0 generate
    C(2*I + 1) <= A(I) nor X;
    C(2*I)     <= B(I) nor X;
end generate GEN_LABEL;
```

Figure 6-9 An 8-Bit Array Design



Common Usage of a for...generate Statement

The most common usage of the generate statement is to create multiple copies of components, processes, or blocks. Example 6-20 and Figure 6-10 show this usage with components. (Example 6-21 on page 6-32 and Figure 6-11 on page 6-33 show this usage with processes.)

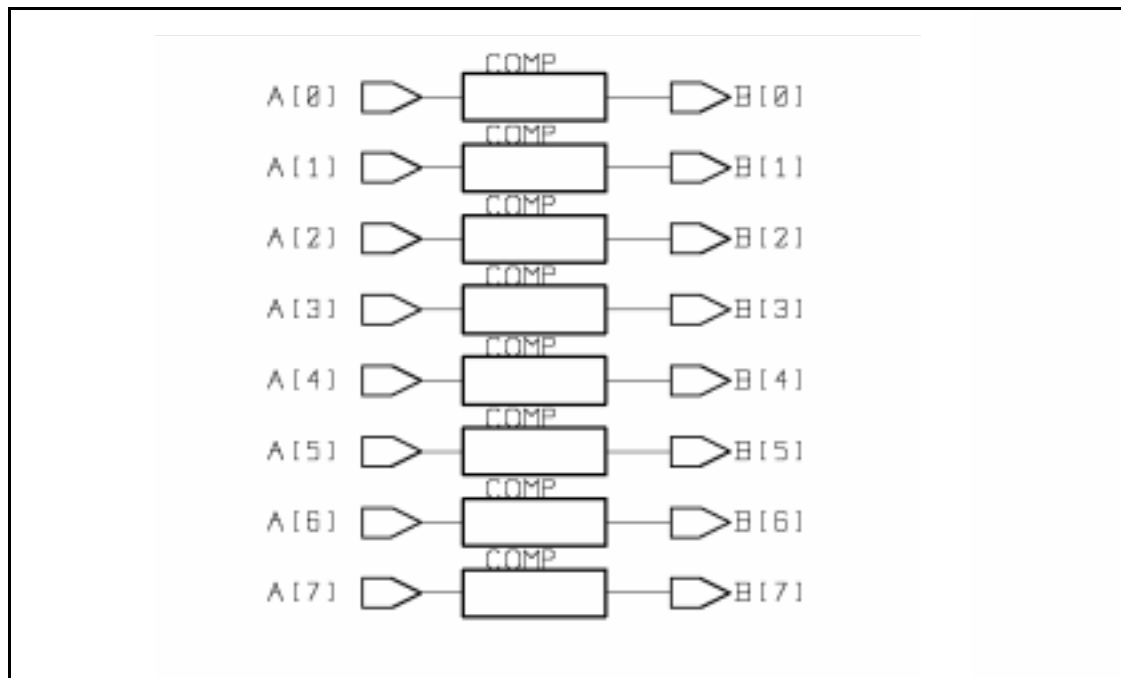
Example 6-20 shows VHDL array attribute 'range used with the for...generate statement to instantiate a set of COMP components that connect corresponding elements of bit vectors A and B. Figure 6-10 illustrates the resulting design.

Example 6-20 for...generate Statement Operating on an Entire Array

```
component COMP
  port (X : in bit;
        Y : out bit);
end component;

...
signal A, B: BIT_VECTOR(0 to 7);
...
GEN: for I in A'range generate
  U: COMP port map (X => A(I),
                   Y => B(I));
end generate GEN;
```

Figure 6-10 Design of COMP Components Connecting Bit Vectors A and B



For more information about arrays, see “Array Types” on page 3-9.

if...generate Statements

The syntax is

```
label: if expression generate
      { concurrent_statement }
end generate [ label ] ;
```

label

The label identifies (names) this statement.

expression

Any expression that evaluates to a Boolean value.

concurrent_statement

Any of the statements described in this chapter, including other generate statements.

Note:

Unlike the if statement described in “if Statements” on page 5-15, the if...generate statement has no else or elsif branches.

You can use the if...generate statement to generate a regular structure that has different circuitry at its ends. Use a for...generate statement to iterate over the desired width of a design, and use a set of if...generate statements to define the beginning, middle, and ending sets of connections.

Example 6-21 shows a technology-independent description of an N-bit serial-to-parallel converter. Data is clocked into an N-bit buffer from right to left. On each clock cycle, each bit in an N-bit buffer is shifted up 1 bit and the incoming DATA bit is moved into the low-order bit. Figure 6-11 illustrates the resulting design.

Example 6-21 Typical Use of if...generate Statements

```
entity CONVERTER is
  generic(N: INTEGER := 8);

  port(CLK, DATA: in BIT;
        CONVERT: buffer BIT_VECTOR(N-1 downto 0));
end CONVERTER;

architecture BEHAVIOR of CONVERTER is
  signal S : BIT_VECTOR(CONVERT'range);
begin

  G: for I in CONVERT'range generate

    G1: -- Shift (N-1) data bit into high-order bit
        if (I = CONVERT'left) generate
          process begin
            wait until (CLK'event and CLK = '1');
            CONVERT(I) <= S(I-1);
          end process;
        end generate G1;

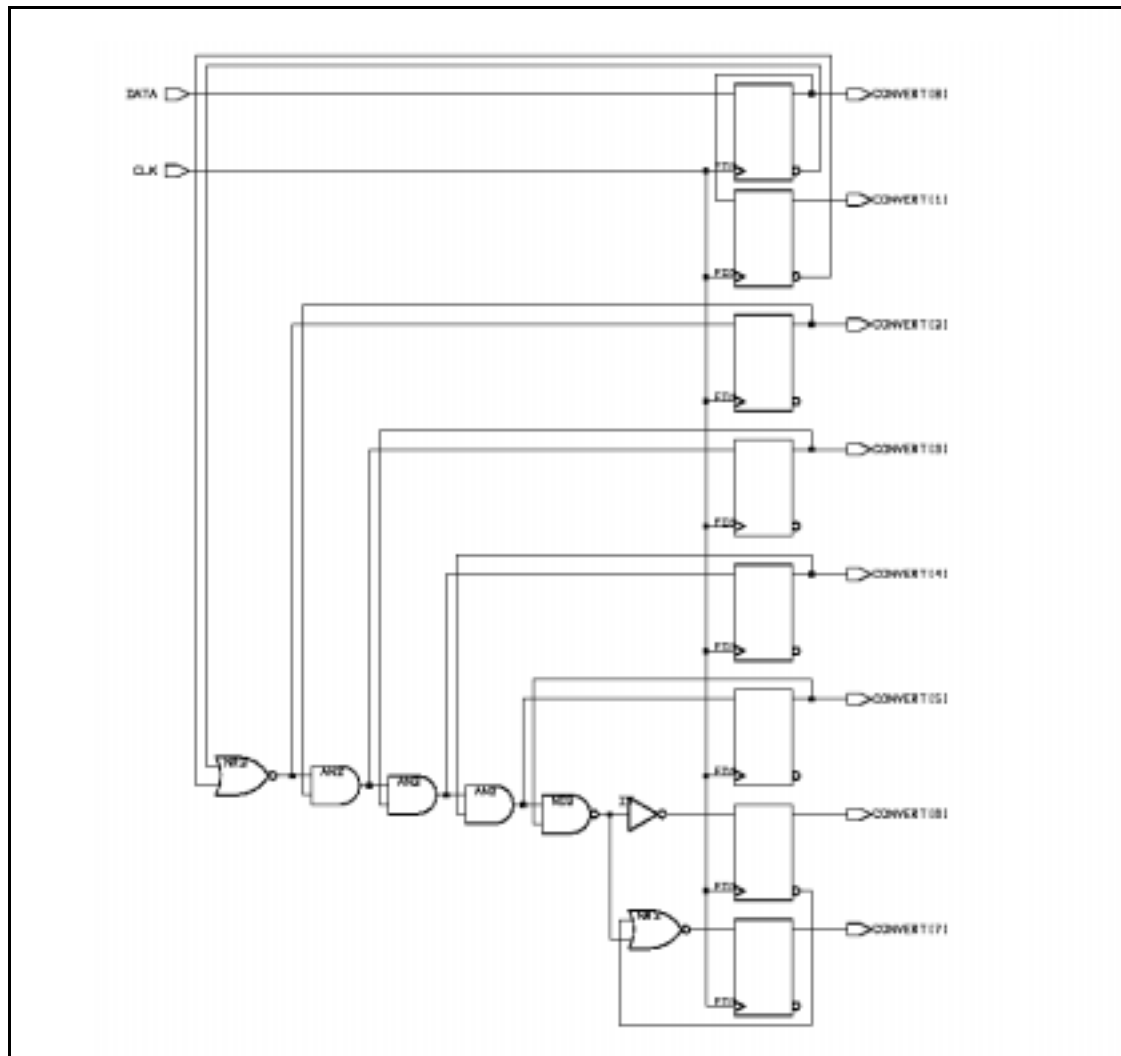
    G2: -- Shift middle bits up
        if (I > CONVERT'right and
            I < CONVERT'left) generate

          S(I) <= S(I-1) and CONVERT(I);

          process begin
            wait until (CLK'event and CLK = '1');
            CONVERT(I) <= S(I-1);
          end process;
        end generate G2;

    G3: -- Move DATA into low-order bit
        if (I = CONVERT'right) generate
          process begin
            wait until (CLK'event and CLK = '1');
            CONVERT(I) <= DATA;
          end process;
          S(I) <= CONVERT(I);
        end generate G3;
      end generate G;
end BEHAVIOR;
```


Figure 6-11 Design of N-Bit Serial-to-Parallel Converter



7

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 7-8 and “Inferring Flip-Flops” on page 7-21.

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 7-1 shows the inference report for a JK flip-flop.

Example 7-1 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 7-1,

- 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 7-1), 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 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 the output is unstable.

“Inferring Latches” on page 7-8 and “Inferring Flip-Flops” on page 7-21 provide inference reports for each register template. After you read a description in *FPGA Compiler II / FPGA Express*, check the inference report.

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

The ATTRIBUTES package in the Synopsys VHDL library defines the following attributes for controlling register inference:

- `async_set_reset`

When this is set to true on a signal, 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, in which case the signal becomes an asynchronous reset or set.

Attach this attribute to 1-bit signals by using the following syntax:

```
attribute async_set_reset of signal_name_list : signal  
is "true";
```

- `async_set_reset_local`

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

Attach this attribute to a process label by using the following syntax:

```
attribute async_set_reset_local of process_label : label  
is "signal_name_list";
```

- `async_set_reset_local_all`

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

Attach this attribute to process labels by using the following syntax:

```
attribute async_set_reset_local_all of  
process_label_list : label is "true";
```

- sync_set_reset

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

Attach this attribute to 1-bit signals by using the following syntax:

```
attribute sync_set_reset of signal_name_list : signal  
is "true";
```

- sync_set_reset_local

FPGA Compiler II / FPGA *Express* treats listed signals in the specified process as if they have the sync_set_reset attribute set to true.

Attach this attribute to a process label by using the following syntax:

```
attribute sync_set_reset_local of process_label : label  
is "signal_name_list";
```

- sync_set_reset_local_all

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

Attach this attribute to process labels by using the following syntax:

```
attribute sync_set_reset_local_all of process_label_list
: label is "true";
```

- 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* attribute prevents FPGA Compiler II / *FPGA Express* from implementing priority encoding logic for the set and reset signals.

Add an assertion to the VHDL 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 attribute to set or reset signals on sequential devices by using the following syntax:

```
attribute one_cold signal_name_list : signal is "true";
```

- 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* attribute prevents FPGA Compiler II / *FPGA Express* from implementing priority encoding logic for the set and reset signals.

Add an assertion to the VHDL 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 attribute to set or reset signals on sequential devices by using the following syntax:

```
attribute one_hot signal_name_list : signal is "true";
```

Inferring Latches

In simulation, a signal or variable holds its value until that output is reassigned. In a circuit, 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 Set/Reset (SR) Latches

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

Example 7-2 provides the VHDL code that implements the SR latch described in the truth table in Table 7-1. Example 7-3 shows the inference report generated by FPGA Compiler II / *FPGA Express*. Figure 7-1 shows the schematic for the latch.

Table 7-1 SR Latch Truth Table (NAND Type)

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

Example 7-2 SR Latch

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity sr_latch is
  port (SET, RESET : in std_logic;
        Q : out std_logic );
  attribute async_set_reset of SET, RESET :
    signal is "true";
end sr_latch;

architecture rtl of sr_latch is
begin

infer: process (SET, RESET) begin
  if (SET = '0') then
    Q <= '1';
  elsif (RESET = '0') then
    Q <= '0';
  end if;
end process infer;

end rtl;
```

Example 7-3 Inference Report for an SR Latch

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

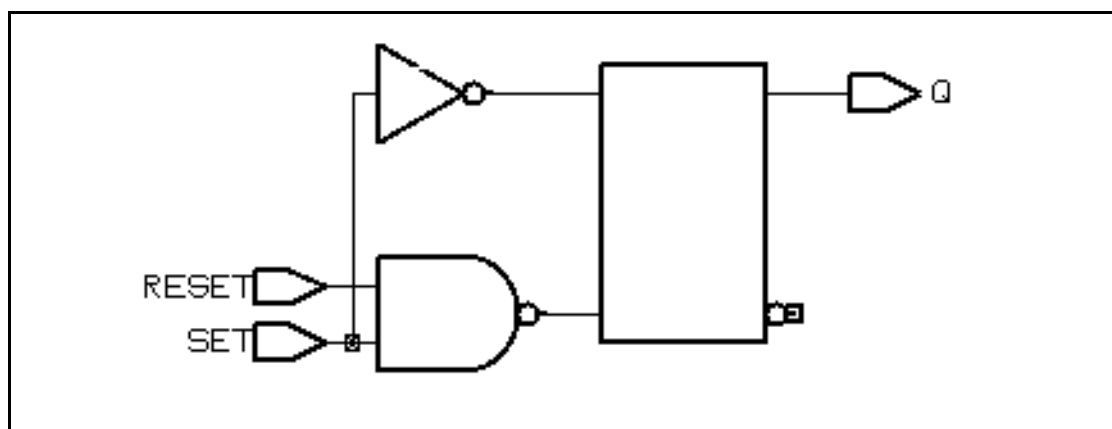
y_reg

Async-reset: RESET'

Async-set: SET'

Async-set and Async-reset ==> Q: 1

Figure 7-1 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 statement, FPGA Compiler II / *FPGA Express* infers a D latch.

For example, the if statement in Example 7-4 infers a D latch, because there is no else clause. The resulting value for output Q is specified only when input enable has a logic 1 value. As a result, output Q becomes a latched value.

Example 7-4 Latch Inference

```
process(DATA, GATE) begin
    if (GATE = '1') then
        Q <= DATA;
    end if;
end process;
```

To avoid latch inference, assign a value to the signal under all conditions, as shown in Example 7-5.

Example 7-5 Fully Specified Signal: No Latch Inference

```
process(DATA, GATE) begin
    if (GATE = '1') then
        Q <= DATA;
    else
        Q <= '0';
    end if;
end process;
```

Variables declared locally within a subprogram do not hold their value over time, because each 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 7-6, FPGA Compiler II / *FPGA Express* does not infer a latch for output Q.

Example 7-6 Function: No Latch Inference

```
function MY_FUNC(DATA, GATE : std_logic) return std_logic is
    variable STATE: std_logic;
begin
    if (GATE = '1') then
        STATE <= DATA;
    end if;
    return STATE;
end;
. . .
Q <= MY_FUNC(DATA, GATE);
```

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

- Simple D latch
- D latch with asynchronous set
- D latch with asynchronous reset
- D latch with asynchronous set and reset

Simple D Latch

When you infer a D latch, make sure that 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 7-7 provides the VHDL template for a D latch. FPGA Compiler II / *FPGA Express* generates the inference report shown in Example 7-8. Figure 7-2 shows the inferred latch.

Example 7-7 D Latch

```
library IEEE;
use IEEE.std_logic_1164.all;

entity d_latch is
  port (GATE, DATA: in std_logic;
        Q : out std_logic );
end d_latch;

architecture rtl of d_latch is
begin

  infer: process (GATE, DATA) begin
    if (GATE = '1') then
      Q <= DATA;
    end if;
  end process infer;

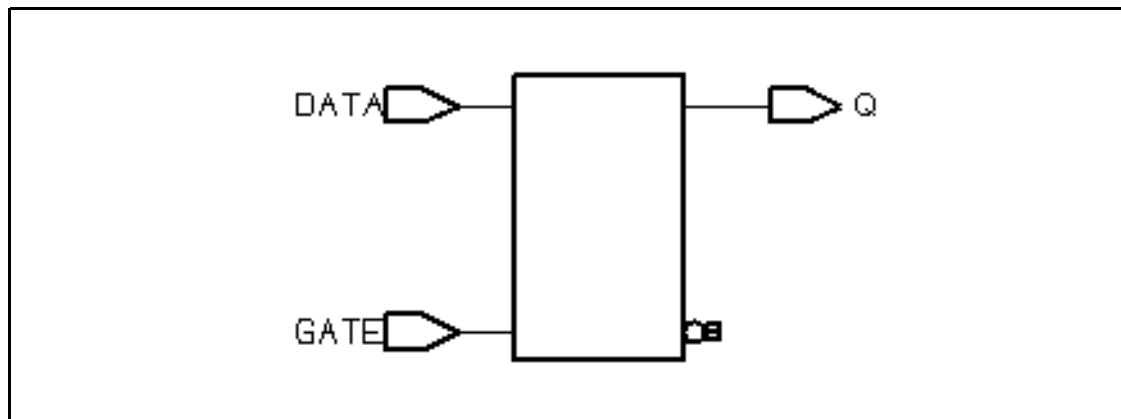
end rtl;
```

Example 7-8 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 7-2 D Latch



D Latch With Asynchronous Set

The template in this section uses the `async_set_reset` attribute to direct FPGA Compiler II / *FPGA Express* to the asynchronous set (AS) pins of the inferred latch.

Example 7-9 provides the VHDL template for a D latch with an asynchronous set. FPGA Compiler II / *FPGA Express* generates the inference report shown in Example 7-10. Figure 7-3 shows the inferred latch.

Example 7-9 D Latch With Asynchronous Set

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity d_latch_async_set is
  port (GATE, DATA, SET : in std_logic;
        Q : out std_logic );
  attribute async_set_reset of SET :
    signal is "true";
end d_latch_async_set;

architecture rtl of d_latch_async_set is
begin

  infer: process (GATE, DATA, SET) begin
    if (SET = '0') then
      Q <= '1';
    elsif (GATE = '1') then
      Q <= DATA;
    end if;
  end process infer;

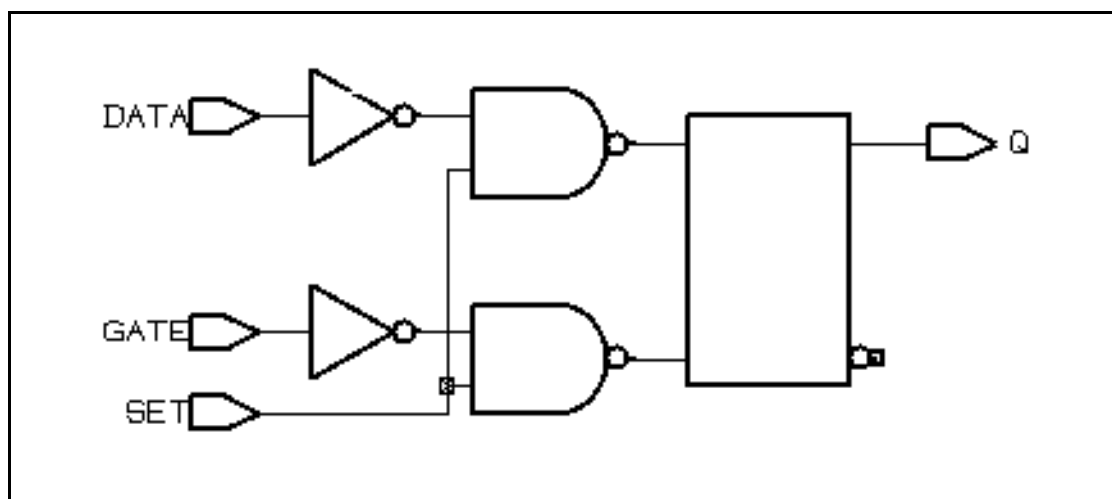
end rtl;
```


Example 7-10 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 7-3 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.

D Latch With Asynchronous Reset

The template in this section uses the `async_set_reset` attribute to direct FPGA Compiler II / *FPGA Express* to the asynchronous reset (AR) pins of the inferred latch.

Example 7-11 provides the VHDL template for a D latch with an asynchronous reset. FPGA Compiler II / *FPGA Express* generates

the inference report shown in Example 7-12. Figure 7-4 shows the inferred latch.

Example 7-11 D Latch With Asynchronous Reset

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity d_latch_async_reset is
  port (GATE, DATA, RESET : in std_logic;
        Q : out std_logic );
  attribute async_set_reset of RESET :
    signal is "true";
end d_latch_async_reset;

architecture rtl of d_latch_async_reset is
begin

  infer : process (GATE, DATA, RESET) begin
    if (RESET = '0') then
      Q <= '0';
    elsif (GATE = '1') then
      Q <= DATA;
    end if;
  end process infer;

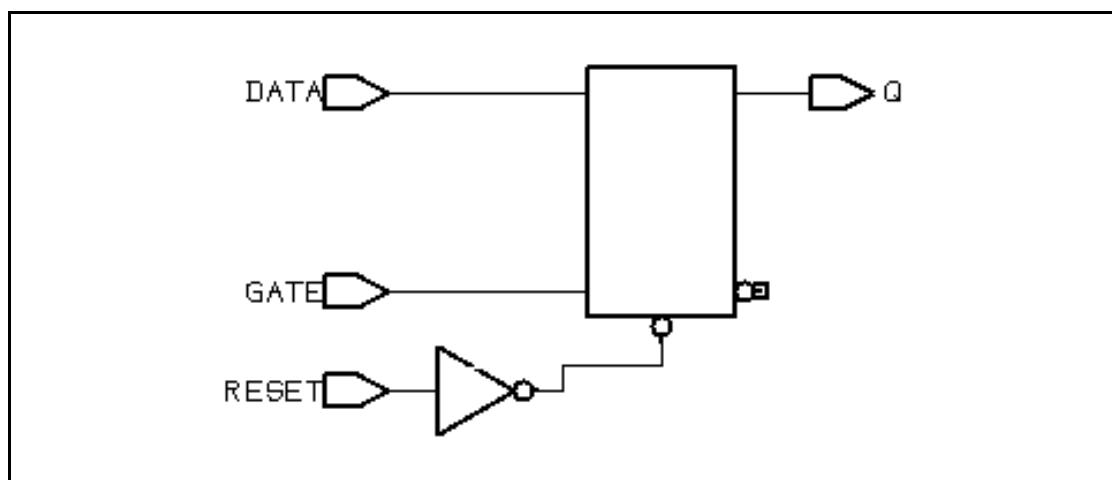
end rtl;
```

Example 7-12 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 7-4 D Latch With Asynchronous Reset



D Latch With Asynchronous Set and Reset

Example 7-13 provides the VHDL template for a D latch with an active low asynchronous set and reset. This template uses the `async_set_reset_local` attribute to direct FPGA Compiler II / FPGA Express to the asynchronous signals in the infer process.

The template in Example 7-13 uses the `one_cold` attribute to prevent priority encoding of the set and reset signals. If you do not specify the `one_cold` attribute, the set signal has priority, because it is used as the condition for the if clause. Example 7-14 shows the inference report. Figure 7-5 shows the inferred latch.

Example 7-13 D Latch With Asynchronous Set and Reset

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity d_latch_async is
  port (GATE, DATA, SET, RESET :in  std_logic;
        Q : out std_logic );
  attribute one_cold of SET, RESET :
    signal is "true";
end d_latch_async;

architecture rtl of d_latch_async is
  attribute async_set_reset_local of infer :
    label is "SET, RESET";
begin

  infer : process (GATE, DATA, SET, RESET) begin
    if (SET = '0') then
      Q <= '1';
    elsif (RESET = '0') then
      Q <= '0';
    elsif (GATE = '1') then
      Q <= DATA;
    end if;
  end process infer;
end rtl;
```

Example 7-14 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	-	-	-

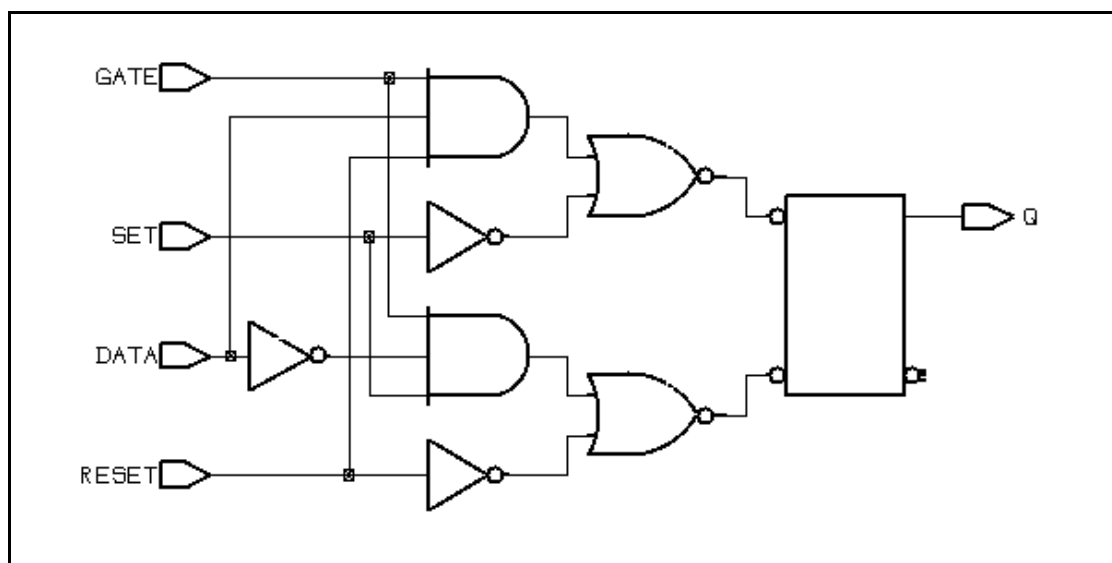
Q_reg

Async-reset: RESET'

Async-set: SET'

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

Figure 7-5 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, you cannot read a conditionally assigned variable 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 7-15 shows an invalid use of the conditionally assigned variable VALUE.

Example 7-15 Invalid Use of a Conditionally Assigned Variable

```
signal X, Y : std_logic;
. . .
process
    variable VALUE : std_logic;
begin
    if (condition) then
        VALUE <= X;
    end if;
    Y <= VALUE;  -- Invalid read of variable VALUE
end process;
```

Inferring Master-Slave Latches

You can infer two-phase systems using D latches. Example 7-16 shows a simple two-phase system with clocks MCK and SCK. Example 7-17 shows the inference reports. Figure 7-6 shows the inferred latch.

Example 7-16 Two-Phase Clocks

```
library IEEE;
use IEEE.std_logic_1164.all;

entity LATCH_VHDL is
  port(MCK, SCK, DATA: in std_logic;
       Q : out std_logic );
end LATCH_VHDL;

architecture rtl of LATCH_VHDL is
  signal TEMP : std_logic;
begin

  process (MCK, DATA) begin
    if (MCK = '1') then
      TEMP <= DATA;
    end if;
  end process;

  process (SCK, TEMP) begin
    if (SCK = '1') then
      Q <= TEMP;
    end if;
  end process;

end rtl;
```

Example 7-17 Inference Reports for Two-Phase Clocks

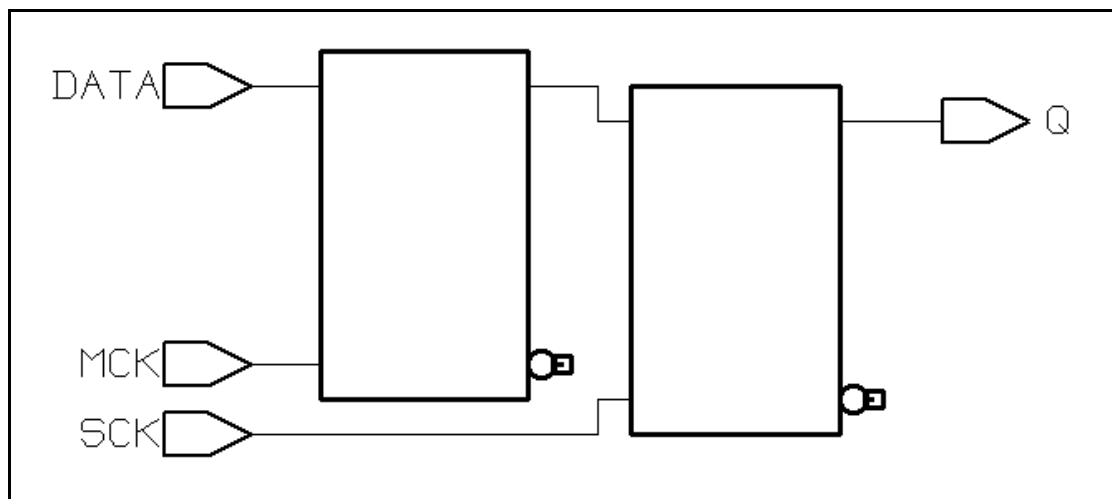
Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
TEMP_reg	Latch	1	-	-	N	N	-	-	-

TEMP_reg
reset/set: none

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

Q_reg
reset/set: none

Figure 7-6 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 give details about each.

Many FPGA devices have a dedicated global set/reset hardware resource that may 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 condition of a wait or if statement uses an edge expression (a test for the rising or falling edge of a signal). Use the following syntax to describe a rising edge:

```
SIGNAL'event and SIGNAL = '1'
```

Use the following syntax to describe a falling edge:

```
SIGNAL'event and SIGNAL = '0'
```

If you are using the IEEE std_logic_1164 package, you can use the following syntax to describe a rising edge and a falling edge:

```
if (rising_edge (CLK)) then  
if (falling_edge (CLK)) then
```

If you are using the IEEE std_logic_1164 package, you can use the following syntax for a bused clock. You can also use a member of a bus as a signal.

```
sig(3)'event and sig(3) = '1'  
  
rising_edge (sig(3))
```

A wait statement containing an edge expression causes FPGA Compiler II / *FPGA Express* to create flip-flops for all signals, and some variables are assigned values in the process. Example 7-18 shows the most common usage of the wait statement to infer a flip-flop.

Example 7-18 Using a wait Statement to Infer a Flip-Flop

```
process
begin
    wait until (edge);
    ...
end process;
```

An if statement implies flip-flops for signals and variables in the branches of the if statement. Example 7-19 shows the most-common usages of the if statement to infer a flip-flop.

Example 7-19 Using an if Statement to Infer a Flip-Flop

```
process (sensitivity_list)
begin
    if (edge)
        ...
    end if;
end process;

process (sensitivity_list)
begin
    if (...) then
        ...
    elsif (...)
        ...
    elsif (edge) then
        ...
    end if;
end process;
```

You can sometimes use wait and if statements interchangeably. If possible, use the if statement, because it provides greater control over the inferred registers.

The following sections provide code examples, inference reports, and figures for these types of D flip-flops:

- Positive edge-triggered D flip-flop
- Positive edge-triggered D flip-flop using `rising_edge`
- Negative edge-triggered D flip-flop
- Negative edge-triggered D flip-flop using `falling_edge`
- D flip-flop with asynchronous set
- D flip-flop with asynchronous reset
- D flip-flop with asynchronous set and reset
- D flip-flop with synchronous set
- D flip-flop with synchronous reset
- D flip-flop with synchronous and asynchronous load
- Multiple flip-flops with asynchronous and synchronous controls

Positive Edge-Triggered D Flip-Flop

When you infer a D flip-flop, make sure that 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 asynchronous reset or set, or with synchronous reset or set.

Example 7-20 provides the VHDL template for a positive edge-triggered D flip-flop. Example 7-21 shows the inference report. Figure 7-7 shows the inferred flip-flop.

Example 7-20 Positive Edge-Triggered D Flip-Flop

```

library IEEE ;
use IEEE.std_logic_1164.all;

entity dff_pos is
  port (DATA, CLK : in std_logic;
        Q : out std_logic );
end dff_pos;

architecture rtl of dff_pos is
begin

  infer : process (CLK) begin
    if (CLK'event and CLK = '1') then
      Q <= DATA;
    end if;
  end process infer;

end rtl;

```

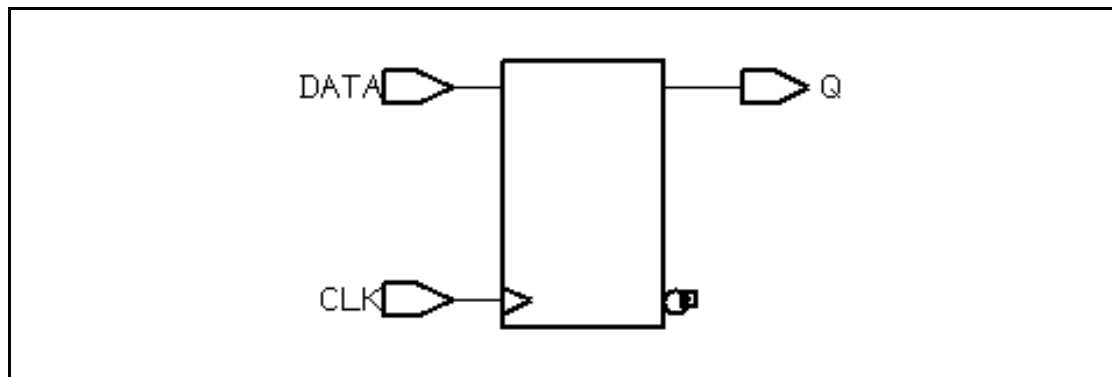
Example 7-21 Inference Report for Positive Edge-Triggered D Flip-Flop

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 7-7 Positive Edge-Triggered D Flip-Flop



Positive Edge-Triggered D Flip-Flop Using rising_edge

Example 7-22 provides the VHDL template for a positive edge-triggered D flip-flop using the IEEE_std_logic_1164 package and rising_edge.

FPGA Compiler II / FPGA *Express* generates the inference report shown in Example 7-23.

Figure 7-8 shows the inferred flip-flop.

Example 7-22 Positive Edge-Triggered D Flip-Flop Using rising_edge

```
library IEEE ;
use IEEE.std_logic_1164.all;

entity dff_pos is
  port (DATA, CLK : in std_logic;
        Q : out std_logic );
end dff_pos;

architecture rtl of dff_pos is
begin

  infer : process (CLK) begin
    if (rising_edge (CLK)) then
      Q <= DATA;
    end if;
  end process infer;

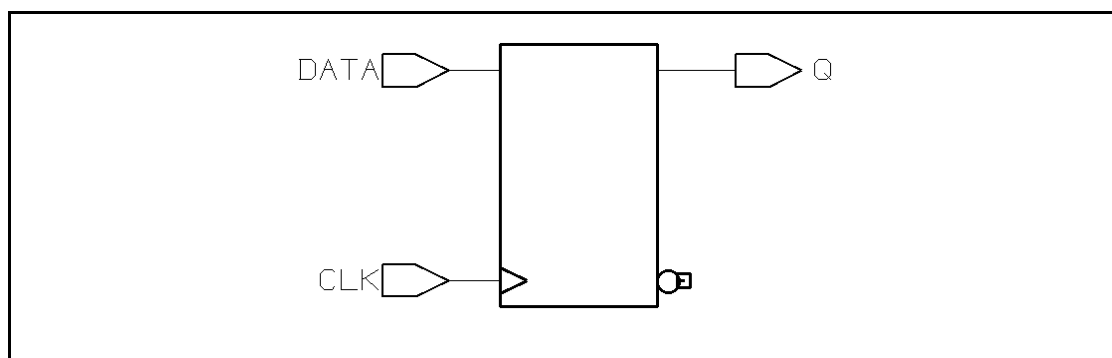
end rtl;
```

Example 7-23 Inference Report for a Positive Edge-Triggered D Flip-Flop Using rising_edge

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 7-8 Positive Edge-Triggered D Flip-Flop Using rising_edge



Negative Edge-Triggered D Flip-Flop

Example 7-24 provides the VHDL template for a negative edge-triggered D flip-flop.

FPGA Compiler II / *FPGA Express* generates the inference report shown in Example 7-25. Figure 7-9 shows the inferred flip-flop.

Example 7-24 Negative Edge-Triggered D Flip-Flop

```
library IEEE;
use IEEE.std_logic_1164.all;

entity dff_neg is
  port (DATA, CLK : in std_logic;
        Q : out std_logic );
end dff_neg;

architecture rtl of dff_neg is
begin

infer : process (CLK) begin
  if (CLK'event and CLK = '0') then
    Q <= DATA;
  end if;
end process infer;

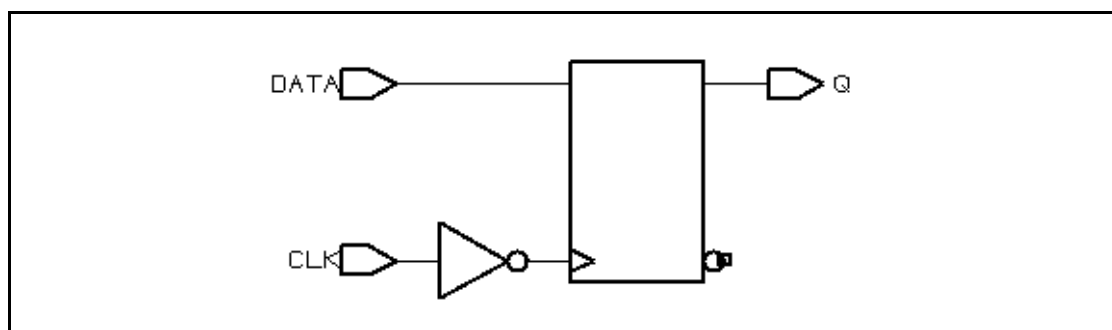
end rtl;
```

Example 7-25 Inference Report for 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 7-9 Negative Edge-Triggered D Flip-Flop



Negative Edge-Triggered D Flip-Flop Using falling_edge

Example 7-26 provides the VHDL template for a negative edge-triggered D flip-flop using the IEEE_std_logic_1164 package and falling_edge.

FPGA Compiler II / FPGA Express generates the inference report shown in Example 7-27. Figure 7-10 shows the inferred flip-flop.

Example 7-26 Negative Edge-Triggered D Flip-Flop Using falling_edge

```
library IEEE;
use IEEE.std_logic_1164.all;

entity dff_neg is
  port (DATA, CLK : in std_logic;
        Q : out std_logic );
end dff_neg;

architecture rtl of dff_neg is
begin
```

```

infer : process (CLK) begin
    if (falling_edge (CLK)) then
        Q <= DATA;
    end if;
end process infer;

end rtl;

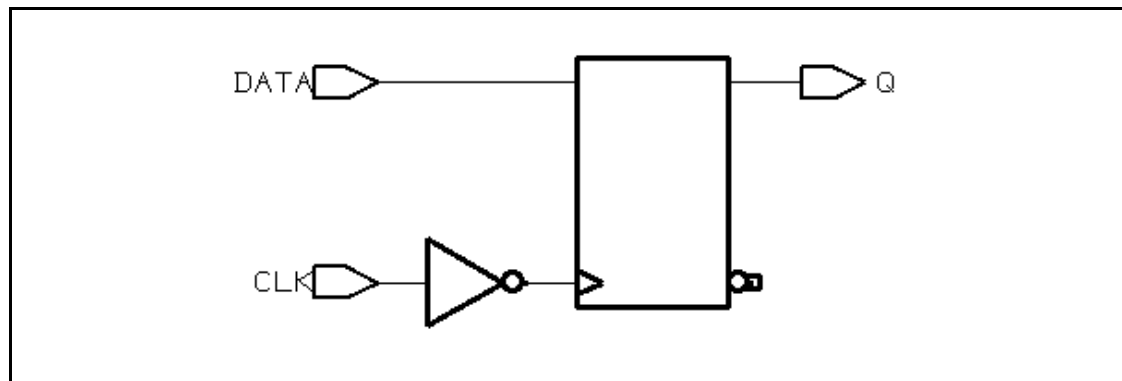
```

Example 7-27 Inference Report for a Negative Edge-Triggered D Flip-Flop Using *falling_edge*

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 7-10 Negative Edge-Triggered D Flip-Flop Using *falling_edge*



D Flip-Flop With Asynchronous Set

Example 7-28 provides the VHDL template for a D flip-flop with an asynchronous set.

FPGA Compiler II / *FPGA Express* generates the inference report shown in Example 7-29. Figure 7-11 shows the inferred flip-flop.

Example 7-28 D Flip-Flop With Asynchronous Set

```

library IEEE;
use IEEE.std_logic_1164.all;

entity dff_async_set is
  port (DATA, CLK, SET : in std_logic;
        Q : out std_logic );
end dff_async_set;

architecture rtl of dff_async_set is
begin

  infer : process (CLK, SET) begin
    if (SET = '0') then
      Q <= '1';
    elsif (CLK'event and CLK = '1') then
      Q <= DATA;
    end if;
  end process infer;
end rtl;

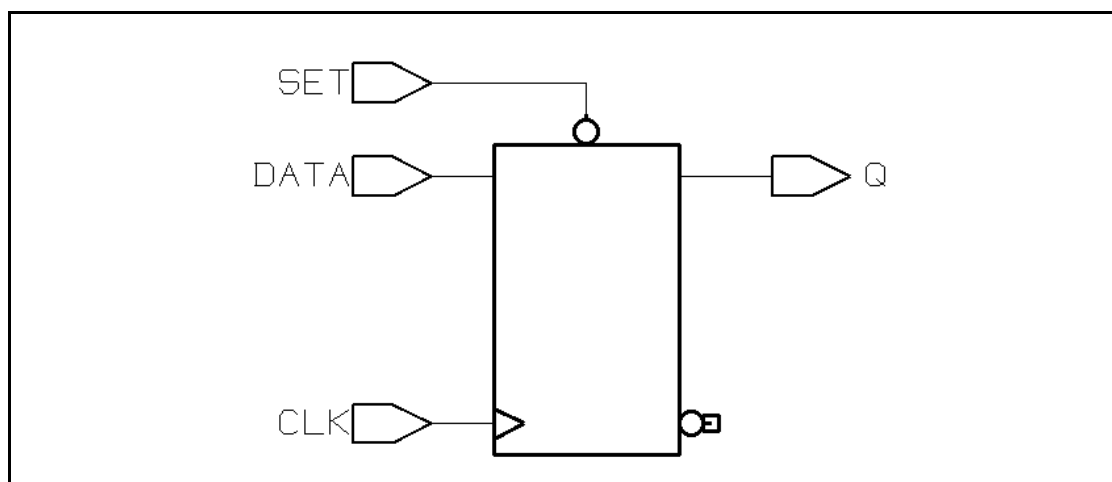
```

Example 7-29 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 7-11 D Flip-Flop With Asynchronous Set



D Flip-Flop With Asynchronous Reset

Example 7-30 provides the VHDL template for a D flip-flop with an asynchronous reset.

FPGA Compiler II / *FPGA Express* generates the inference report shown in Example 7-31. Figure 7-12 shows the inferred flip-flop.

Example 7-30 D Flip-Flop With Asynchronous Reset

```
library IEEE;
use IEEE.std_logic_1164.all;

entity dff_async_reset is
  port (DATA, CLK, RESET : in std_logic;
        Q : out std_logic );
end dff_async_reset;

architecture rtl of dff_async_reset is
begin

  infer : process ( CLK, RESET) begin
    if (RESET = '1') then
      Q <= '0';
    elsif (CLK'event and CLK = '1') then
      Q <= DATA;
    end if;
  end process infer;

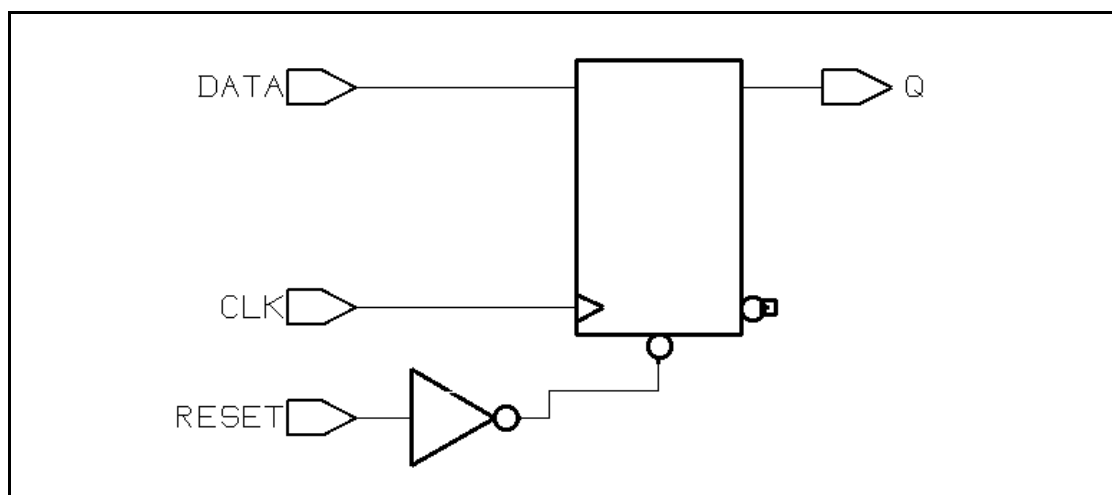
end rtl;
```

Example 7-31 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 7-12 D Flip-Flop With Asynchronous Reset



D Flip-Flop With Asynchronous Set and Reset

Example 7-32 provides the VHDL template for a D flip-flop with active high asynchronous set and reset pins.

The template in Example 7-32 uses the `one_hot` attribute to prevent priority encoding of the set and reset signals. If you do not specify the `one_hot` attribute, 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 7-33. Figure 7-13 shows the inferred flip-flop.

Note:

Most FPGA architectures do not have a register with an asynchronous set AND asynchronous reset cell available. For this reason you should avoid this construct.

Example 7-32 D Flip-Flop With Asynchronous Set and Reset

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity dff_async is
  port (DATA, CLK, SET, RESET : in std_logic;
        Q : out std_logic );
  attribute one_hot of SET, RESET : signal is "true";
end dff_async;

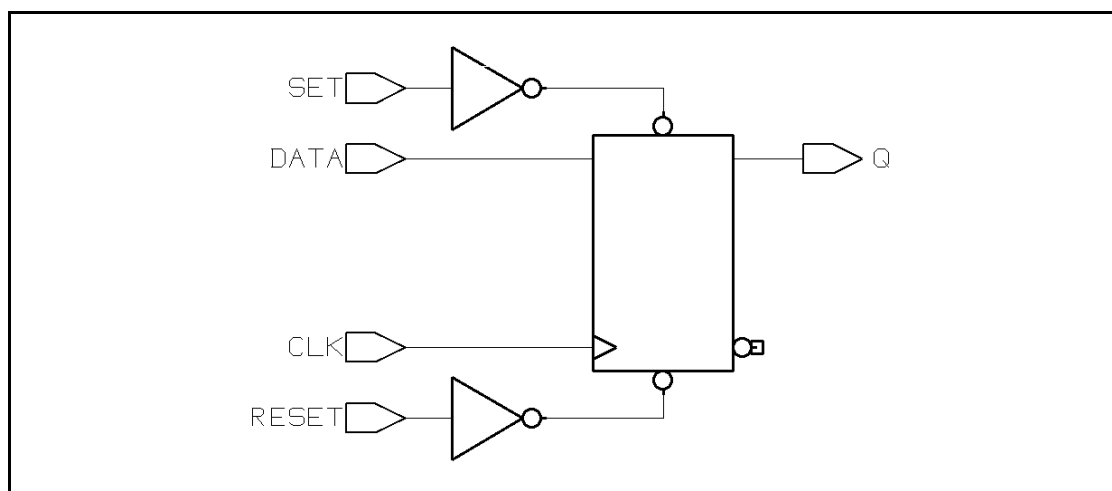
architecture rtl of dff_async is
begin
  infer : process (CLK, SET, RESET) begin
    if (RESET = '1') then
      Q <= '0';
    elsif (SET = '1') then
      Q <= '1';
    elsif (CLK'event and CLK = '1') then
      Q <= DATA;
    end if;
  end process infer;
end rtl;
```

Example 7-33 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 7-13 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 the flip-flop (see Example 7-34 and Example 7-36). The `sync_set_reset` attribute 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.

D Flip-Flop With Synchronous Set

Example 7-34 provides the VHDL template for a D flip-flop with synchronous set. FPGA Compiler II / *FPGA Express* generates the inference report shown in Example 7-35. Figure 7-14 shows the inferred flip-flop.

Example 7-34 D Flip-Flop With Synchronous Set

```

library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;
entity dff_sync_set is
  port (DATA, CLK, SET : in std_logic;
        Q : out std_logic );
  attribute sync_set_reset of SET : signal is "true";
end dff_sync_set;

architecture rtl of dff_sync_set is
begin

  infer : process (CLK) begin
    if (CLK'event and CLK = '1') then
      if (SET = '1') then
        Q <= '1';
      else
        Q <= DATA;
      end if;
    end if;
  end process infer;

end rtl;

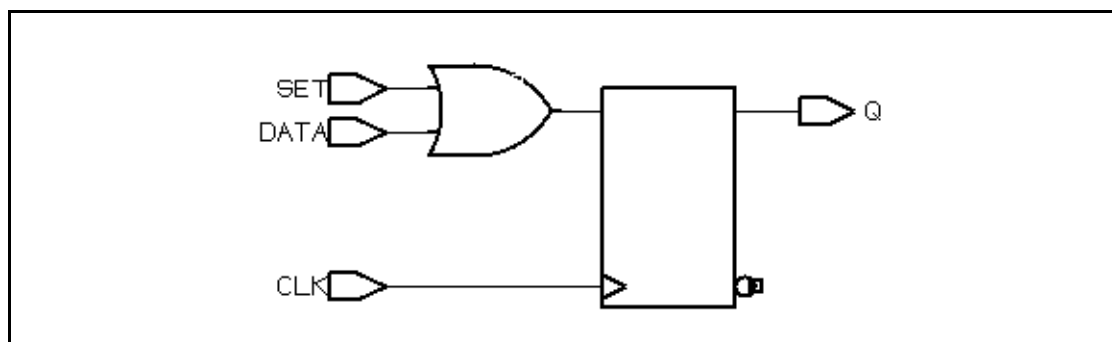
```

Example 7-35 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 7-14 D Flip-Flop With Synchronous Set



D Flip-Flop With Synchronous Reset

Example 7-36 provides the VHDL template for a D flip-flop with synchronous reset. FPGA Compiler II / *FPGA Express* generates the inference report shown in Example 7-37. Figure 7-15 shows the inferred flip-flop.

Example 7-36 D Flip-Flop With Synchronous Reset

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity dff_sync_reset is
  port (DATA, CLK, RESET : in std_logic;
        Q : out std_logic );
  attribute sync_set_reset of RESET :
    signal is "true";
end dff_sync_reset;

architecture rtl of dff_sync_reset is
begin

infer : process (CLK) begin
  if (CLK'event and CLK = '1') then
    if (RESET = '0') then
      Q <= '0';
    else
      Q <= DATA;
    end if;
  end if;
end process infer;

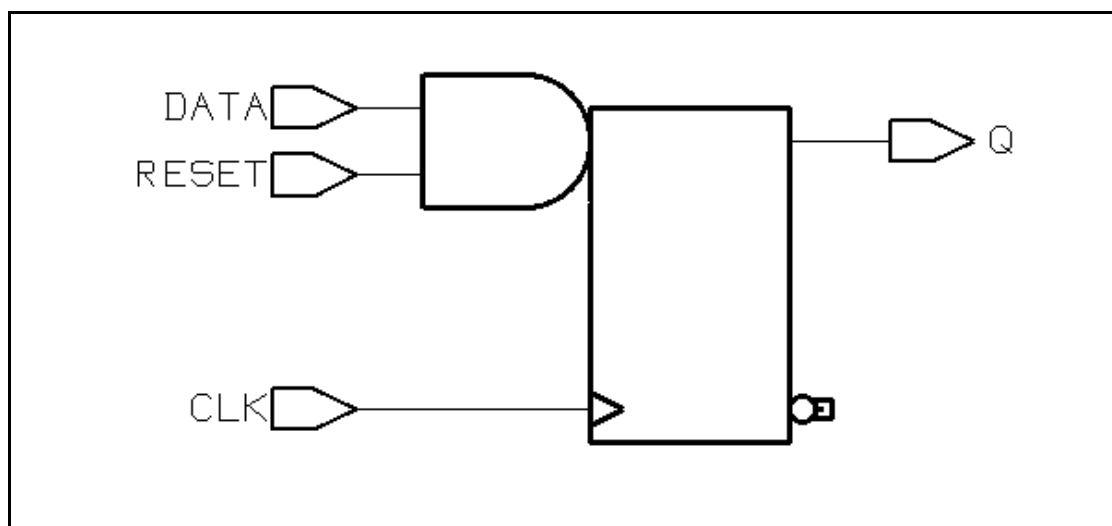
end rtl;
```

Example 7-37 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 7-15 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 both synchronous and asynchronous controls, you must check the asynchronous conditions before you check the synchronous conditions.

Example 7-38 provides the VHDL 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 7-39. Figure 7-16 shows the inferred flip-flop.

Example 7-38 D Flip-Flop With Synchronous and Asynchronous Load

```
library IEEE;
use IEEE.std_logic_1164.all;

entity dff_a_s_load is
    port(SLOAD, ALOAD, ADATA, SDATA,
         CLK : in std_logic;
         Q : out std_logic );
end dff_a_s_load;
```

```

architecture rtl of dff_a_s_load is
begin

infer: process (CLK, ALOAD) begin
    if (ALOAD = '1') then
        Q <= ADATA;
    elsif (CLK'event and CLK = '1') then
        if (SLOAD = '1') then
            Q <= SDATA;
        end if;
    end if;
end process infer;

end rtl;

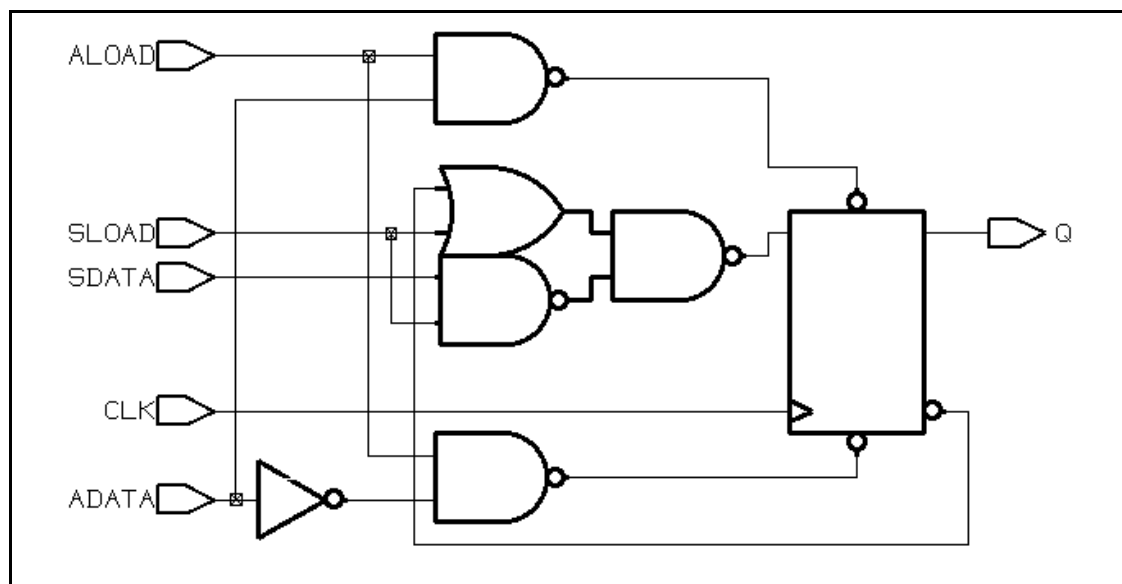
```

Example 7-39 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 7-16 D Flip-Flop With Synchronous and Asynchronous Load



Multiple Flip-Flops: Asynchronous and Synchronous Controls

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

In Example 7-40, the process `infer_sync` uses the reset signal as a synchronous reset, and the process `infer_async` uses the reset signal as an asynchronous reset. Example 7-41 shows the inference report. Figure 7-17 shows the resulting design.

Example 7-40 Multiple Flip-Flops: Asynchronous and Synchronous Controls

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity multi_attr is
  port (DATA1, DATA2, CLK, RESET, SLOAD : in std_logic;
        Q1, Q2 : out std_logic );
end multi_attr;

architecture rtl of multi_attr is
  attribute async_set_reset_local of infer_async :
    label is "RESET";
  attribute sync_set_reset_local of infer_sync :
    label is "RESET";
begin
infer_sync: process (CLK) begin
  if (CLK'event and CLK = '1') then
    if (RESET = '0') then
      Q1 <= '0';
    elsif (SLOAD = '1') then
      Q1 <= DATA1;
    end if;
  end if;
end process infer_sync;
```

```

infer_async: process (CLK, RESET) begin
  if (RESET = '0') then
    Q2 <= '0';
  elsif (CLK'event and CLK = '1') then
    if (SLOAD = '1') then
      Q2 <= DATA2;
    end if;
  end if;
end process infer_async;
end rtl;

```

Example 7-41 Inference Reports for Example 7-40

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

Q1_reg

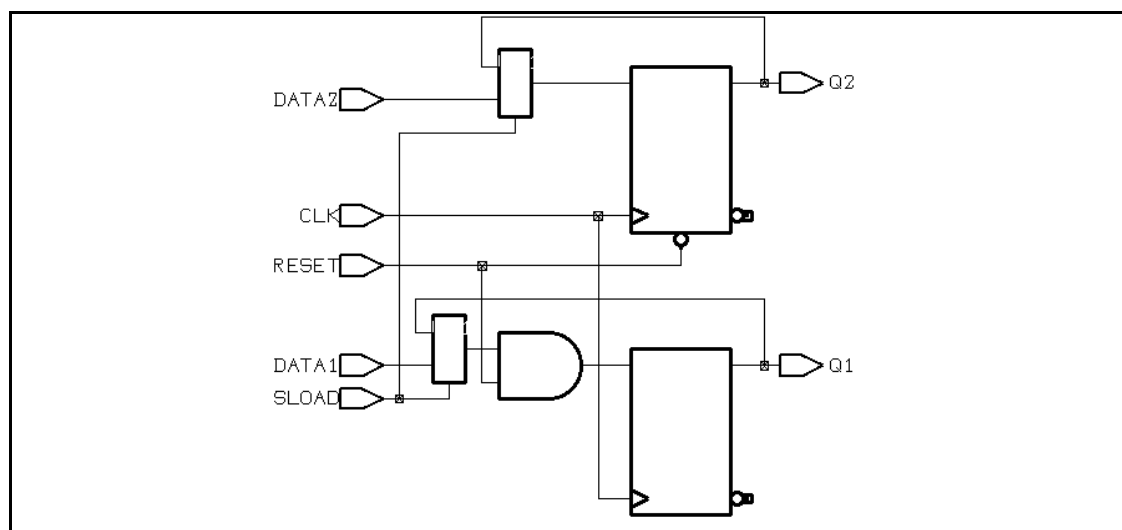
Sync-reset: RESET'

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'

Figure 7-17 Multiple Flip-Flops with Asynchronous and Synchronous Controls



A flip-flop inference has specific limitations. See “Understanding Limitations of Register Inference” on page 7-57.

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

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.

Example 7-42 provides the VHDL code that implements the JK flip-flop described in the truth table in Table 7-2.

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

Example 7-43 shows the inference report generated by FPGA Compiler II / *FPGA Express*. Figure 7-18 shows the inferred flip-flop.

Table 7-2 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 _n B
X	X	Falling	Q _n

Example 7-42 JK Flip-Flop

```

library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity jk is
  port(J, K, CLK : in std_logic;
        Q_out : out std_logic );
  attribute sync_set_reset of J, K :
    signal is "true";
end jk;

architecture rtl of jk is
  signal Q : std_logic;
begin
  infer: process
    variable JK : std_logic_vector ( 1 downto 0);
  begin
    wait until (CLK'event and CLK = '1');
    JK <= (J & K);
    case JK is
      when "01" => Q <= '0';
      when "10" => Q <= '1';
      when "11" => Q <= not (Q);
      when "00" => Q <= Q;
      when others => Q <= 'X';
    end case;
  end process infer;

  Q_out <= Q;
end rtl;

```

Example 7-43 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	N

Q_reg

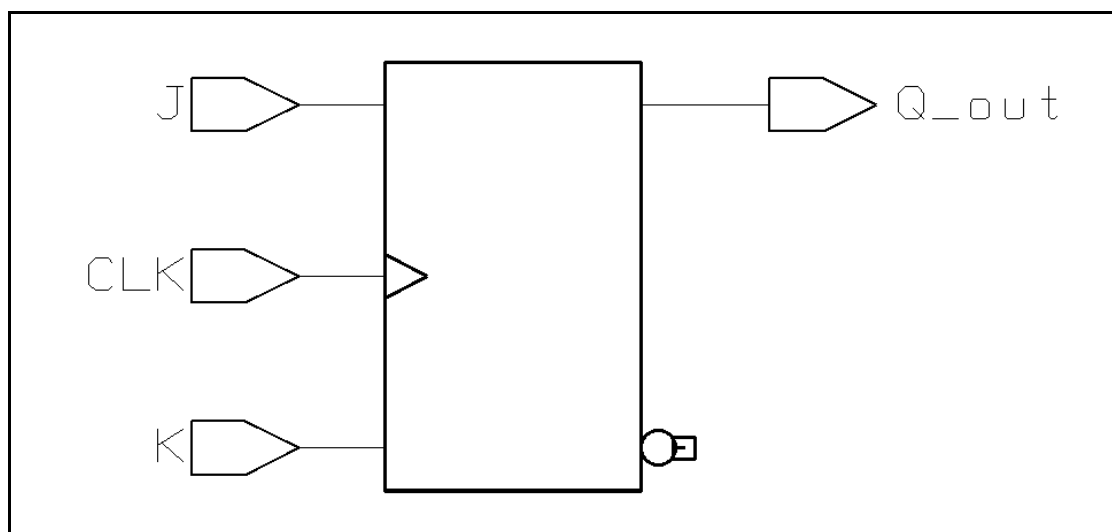
Sync-reset: J' K

Sync-set: J K'

Sync-toggle: J K

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

Figure 7-18 JK Flip-Flop



JK Flip-Flop With Asynchronous Set and Reset

Example 7-44 provides the VHDL template for a JK flip-flop with asynchronous set and reset. Use the `sync_set_reset` attribute to indicate the JK function. Use the `one_hot` attribute to prevent priority encoding of the J and K signals.

FPGA Compiler II / *FPGA Express* generates the inference report shown in Example 7-45. Figure 7-19 shows the inferred flip-flop.

Example 7-44 JK Flip-Flop With Asynchronous Set and Reset

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity jk_async_sr is
  port (SET, RESET, J, K, CLK : in std_logic;
        Q_out : out std_logic );
  attribute sync_set_reset of J, K :
    signal is "true";
  attribute one_hot of SET,RESET : signal is "true";
end jk_async_sr;

architecture rtl of jk_async_sr is
  signal Q : std_logic;
begin

  infer : process (CLK, SET, RESET)
    variable JK : std_logic_vector (1 downto 0);
  begin
    if (RESET = '1') then
      Q <= '0';
    elsif (SET = '1') then
      Q <= '1';
    elsif (CLK'event and CLK = '1') then
      JK <= (J & K);
      case JK is
        when "01" => Q <= '0';
        when "10" => Q <= '1';
        when "11" => Q <= not(Q);
        when "00" => Q <= Q;
        when others => Q <= 'X';
      end case;
    end if;
  end process infer;

  Q_out <= Q;
end rtl;
```

Example 7-45 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	N

Q_reg

Async-reset: RESET

Async-set: SET

Sync-reset: J' K

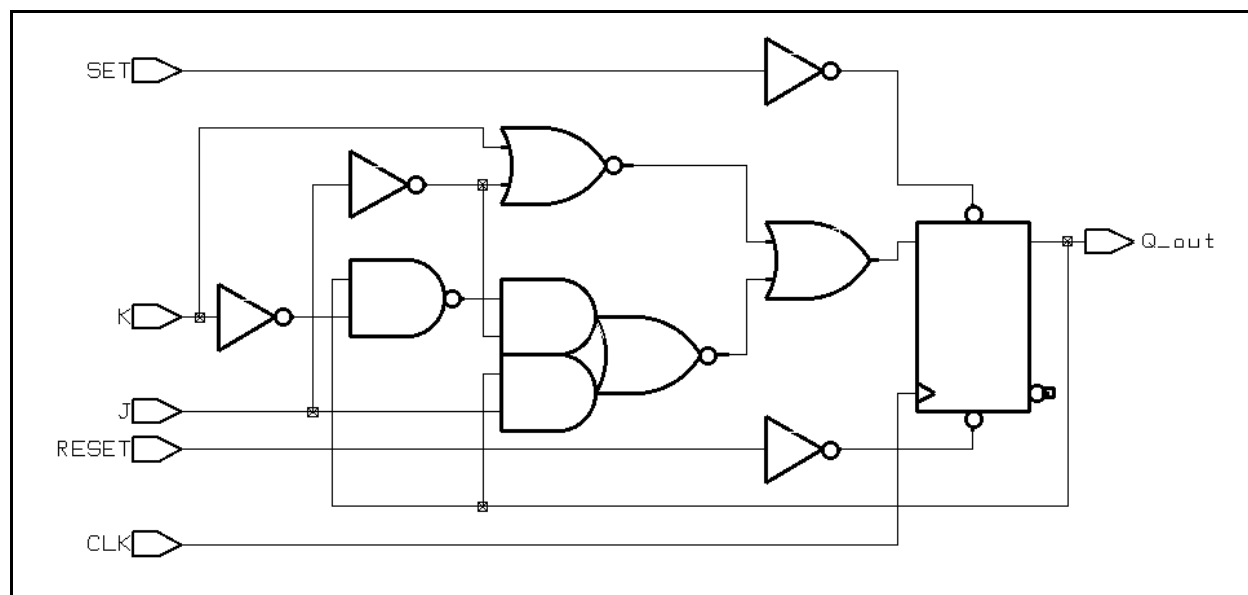
Sync-set: J K'

Sync-toggle: J K

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

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

Figure 7-19 JK Flip-Flop With Asynchronous Set and Reset



Inferring Toggle Flip-Flops

To infer toggle flip-flops, follow the coding style given in the following examples. You must include asynchronous controls in the toggle flip-flop description. Without asynchronous controls, you cannot initialize toggle flip-flops to a known state.

The following sections provide code examples, inference reports, and figures for these types of toggle flip-flops:

- Toggle flip-flop with asynchronous set
- Toggle flip-flop with asynchronous reset
- Toggle flip-flop with enable and asynchronous reset

Toggle Flip-Flop With Asynchronous Set

Example 7-46 provides the VHDL template for a toggle flip-flop with asynchronous set. Example 7-47 shows the inference report. Figure 7-20 shows the inferred flip-flop.

Example 7-46 Toggle Flip-Flop With Asynchronous Set

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
entity t_async_set is
  port(SET, CLK : in std_logic;
       Q : out std_logic );
end t_async_set;
architecture rtl of t_async_set is
  signal TMP_Q : std_logic;
begin

infer: process (CLK, SET) begin
  if (SET = '1') then
    TMP_Q <= '1';
  elsif (CLK'event and CLK = '1') then
    TMP_Q <= not (TMP_Q);
  end if;
  Q <= TMP_Q;
end process infer;

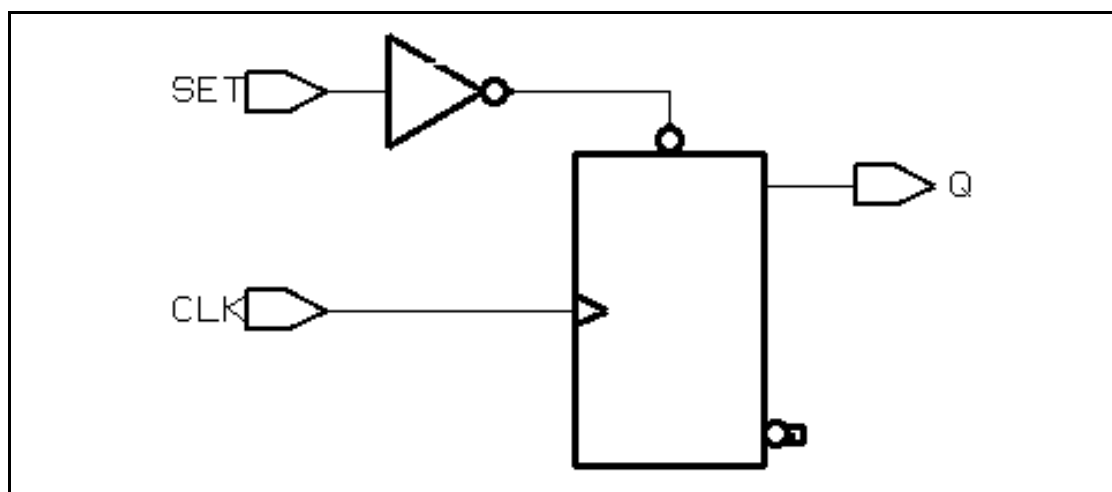
end rtl;
```


Example 7-47 Inference Report for 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 7-20 Toggle Flip-Flop With Asynchronous Set



Toggle Flip-Flop With Asynchronous Reset

Example 7-48 provides the VHDL template for a toggle flip-flop with asynchronous reset. FPGA Compiler II / *FPGA Express* generates the inference report shown in Example 7-49. Figure 7-21 shows the inferred flip-flop.

Example 7-48 Toggle Flip-Flop With Asynchronous Reset

```
library IEEE ;
use IEEE.std_logic_1164.all;

entity t_async_reset is
  port(RESET, CLK : in std_logic;
        Q : out std_logic );
end t_async_reset;

architecture rtl of t_async_reset is
  signal TMP_Q : std_logic;
begin

  infer: process (CLK, RESET) begin
    if (RESET = '1') then
      TMP_Q <= '0';
    elsif (CLK'event and CLK = '1') then
      TMP_Q <= not (TMP_Q);
    end if;
    Q <= TMP_Q;
  end process infer;

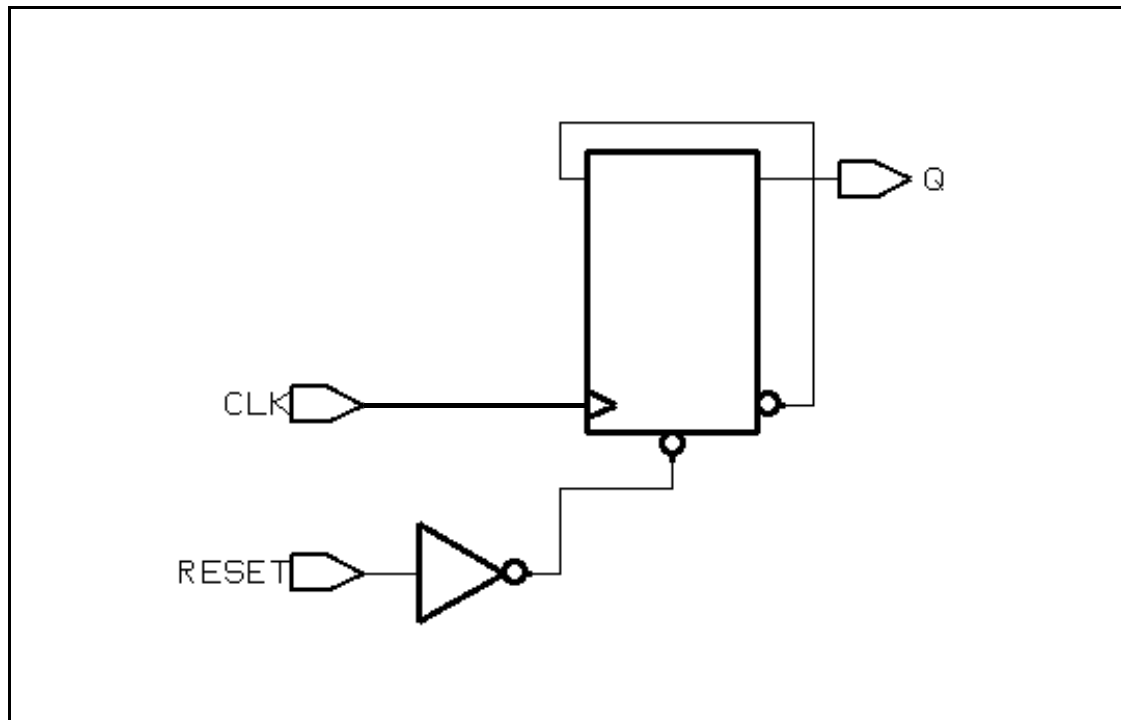
end rtl;
```

Example 7-49 Inference Report for a 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 7-21 Toggle Flip-Flop With Asynchronous Reset



Toggle Flip-Flop With Enable and Asynchronous Reset

Example 7-50 provides the VHDL 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 7-51. Figure 7-22 shows the inferred flip-flop.

Example 7-50 Toggle Flip-Flop With Enable and Asynchronous Reset

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity t_async_en_r is
  port(RESET, TOGGLE, CLK : in std_logic;
       Q : out std_logic );
end t_async_en_r;

architecture rtl of t_async_en_r is
  signal TMP_Q : std_logic;
begin

  infer: process (CLK, RESET) begin
    if (RESET = '1') then
      TMP_Q <= '0';
    elsif (CLK'event and CLK = '1') then
      if (TOGGLE = '1') then
        TMP_Q <= not (TMP_Q);
      end if;
    end if;
  end process infer;

  Q <= TMP_Q;

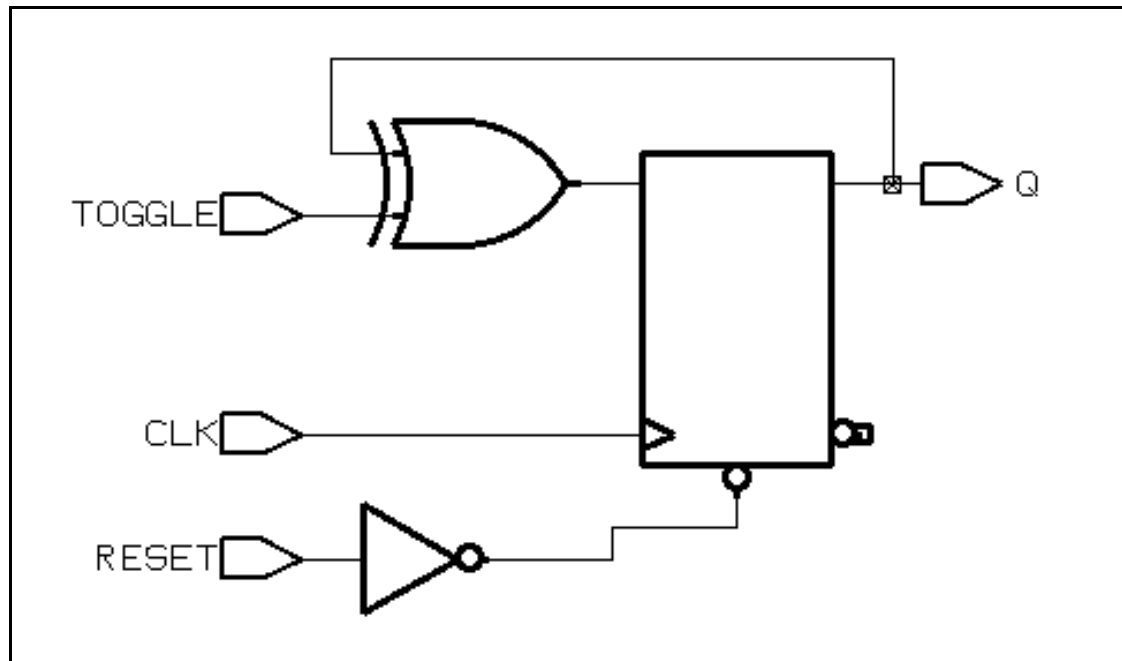
end rtl;
```

Example 7-51 Inference Report for 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 7-22 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

HDL descriptions should build only as many flip-flops as a design requires.

Circuit Description Inferring Too Many Flip-Flops

Example 7-52 shows a description that infers too many flip-flops. The inference report is shown in Example 7-53. Figure 7-23 shows the inferred flip-flops.

Example 7-52 Circuit With Six Inferred Flip-Flops

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity count is
    port (CLK, RESET : in std_logic;
          AND_BITS, OR_BITS,
          XOR_BITS : out std_logic );
end count;

architecture rtl of count is
begin

    process
        variable COUNT : std_logic_vector (2 downto 0);
    begin
        wait until (CLK'EVENT and CLK = '1');
        if (RESET = '1') then
            COUNT <= "000";
        else
            COUNT <= COUNT + 1;
        end if;
        AND_BITS <= COUNT(2) and COUNT(1) and COUNT(0);
        OR_BITS <= COUNT(2) or COUNT(1) or COUNT(0);
        XOR_BITS <= COUNT(2) xor COUNT(1) xor COUNT(0);
    end process;

end rtl;
```

Example 7-52 has only one process, which contains a wait statement and six output signals. FPGA Compiler II / *FPGA Express* infers six flip-flops, one for each output signal in the process:

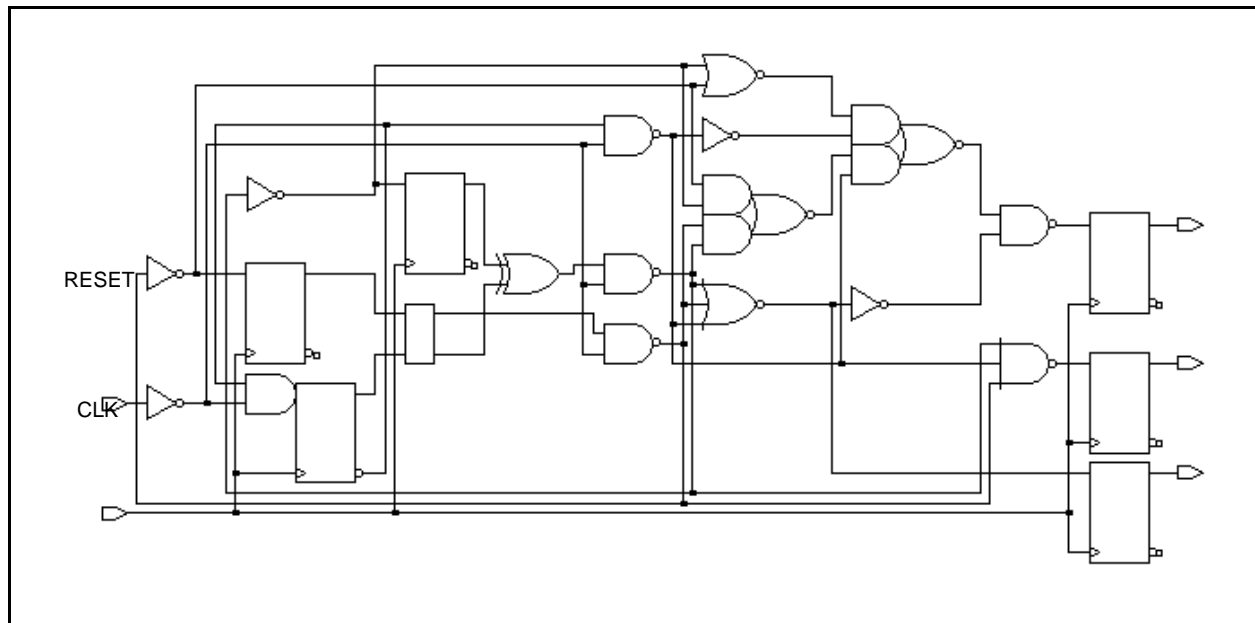
- COUNT(2:0) (three inferred flip-flops)
- AND_BITS (one inferred flip-flop)
- OR_BITS (one inferred flip-flop)
- XOR_BITS (one inferred flip-flop)

However, because the outputs AND_BITS, OR_BITS, and XOR_BITS depend solely on the value of variable COUNT, and variable COUNT is registered, these three outputs do not need to be registered. Therefore, assign AND_BITS, OR_BITS, and XOR_BITS within a process that does not have a wait statement (see the next section, “Circuit Description Inferring Correct Number of Flip-Flops” on page 7-54).

Example 7-53 Inference Report for Circuit With Six Inferred Flip-Flops

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
AND_BITS_reg	Flip-flop	1	-	-	N	N	N	N	N
COUNT_reg	Flip-flop	3	Y	N	N	N	N	N	N
OR_BITS_reg	Flip-flop	1	-	-	N	N	N	N	N
XOR_BITS_reg	Flip-flop	1	-	-	N	N	N	N	N

Figure 7-23 Circuit With Six Inferred Flip-Flops



Circuit Description Inferring Correct Number of Flip-Flops

To avoid inferring extra flip-flops, assign the output signals from within a process that does not have a wait statement.

Example 7-54 shows a description with two processes, one with a wait statement and one without. The registered (synchronous) assignments are in the first process, which contains the wait statement. The other (asynchronous) assignments are in the second process. Signals communicate between the two processes.

This description style lets you choose the signals that are registered and those that are not. The inference report is shown in Example 7-55. Figure 7-24 shows the resulting circuit.

Example 7-54 Circuit With Three Inferred Flip-Flops

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity count is
    port(CLK, RESET : in std_logic;
          AND_BITS, OR_BITS, XOR_BITS : out std_logic);
end count;

architecture rtl of count is
    signal COUNT : std_logic_vector (2 downto 0);
begin

    reg : process begin
        wait until (CLK'event and CLK = '1');
        if (RESET = '1') then
            COUNT <= "000";
        else
            COUNT <= COUNT + 1;
        end if;
    end process reg;
    combine : process(count) begin
        AND_BITS <= COUNT(2) and COUNT(1) and COUNT(0);
        OR_BITS <= COUNT(2) or COUNT(1) or COUNT(0);
        XOR_BITS <= COUNT(2) xor COUNT(1) xor COUNT(0);
    end process combine;

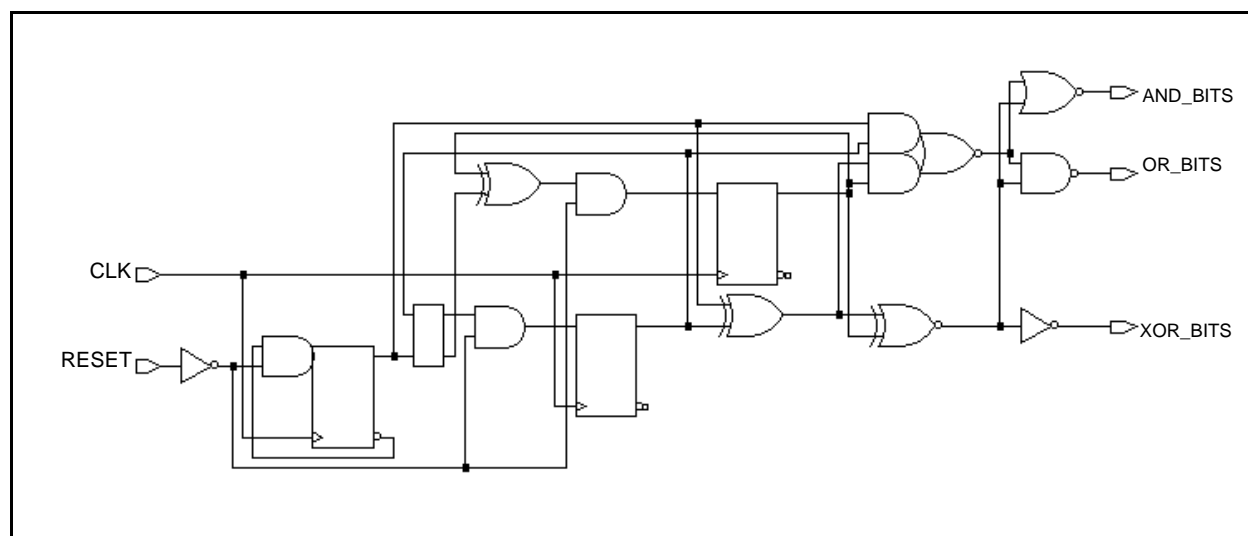
end rtl;
```

Example 7-55 Inference Report for Circuit With Three Inferred Flip-Flops

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
COUNT_reg	Flip-flop	3	Y	N	N	N	N	N	N

COUNT_reg (width 3)
set/reset/toggle: none

Figure 7-24 Circuit With Three Inferred Flip-Flops



This technique of separating combinational logic from registered or sequential logic in your design is useful when describing finite state machines. See these in Appendix A, "Examples:

- "Moore Machine" on page A-2
- "Mealy Machine" on page A-5
- "Count Zeros—Sequential Version" on page A-22
- "Soft Drink Machine—State Machine Version" on page A-24

Correlating Synthesis Results With Simulation Results

Using delay specifications with registered values can cause the simulation to behave differently from the logic synthesized by FPGA Compiler II / *FPGA Express*. For example, the description in Example 7-56 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 pre-synthesis simulation results).

Example 7-56 Delays in Registers

```
component flip_flop (D, CLK : in std_logic;
                    Q : out std_logic );
end component;

process (A, CLK);
    signal B: std_logic;
begin
    B <= A after 100ns;

    F1: flip_flop port map (A, CLK, C),

    F2: flip_flop port map (B, CLK, D);
end process;
```

In Example 7-56, 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 VHDL 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.

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

- An edge expression must be the only condition of an if or an elsif clause.

The following if statement is invalid because it has multiple conditions in the if clause:

```
if (edge and RST = '1')
```

- You can have only one edge expression in an if clause, and the if clause must not have an else clause.

The following if statement is invalid, because you cannot include an else clause when using an edge expression as the if or elsif condition:

```
if X > 5 then
    sequential_statement;
elsif edge then
    sequential_statement;
else
    sequential_statement;
end if;
```

- An edge expression cannot be part of another logical expression or be used as an argument.

The following function call is invalid, because you cannot use the edge expression as an argument:

```
any_function(edge);
```

Three-State Inference

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

Reporting Three-State Inference

Example 7-57 shows a three-state inference report.

Example 7-57 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 signal or variable. FPGA Compiler II / *FPGA Express* does not provide any means of controlling the inference.

Inferring Three-State Drivers

The following sections contain VHDL examples that infer the following types of three-state drivers:

- Simple three-state driver
- Three-state driver with registered enable
- Three-state driver without registered enable

Inferring a Simple Three-State Driver

The following section provides a template for a simple three-state driver. In addition, this section provides examples of how allocating high-impedance assignments to different processes affects three-state inference.

Example 7-58 provides the VHDL template for a simple three-state driver. FPGA Compiler II / *FPGA Express* generates the inference report shown in Example 7-59. Figure 7-25 shows the inferred three-state driver.

Example 7-58 Simple Three-State Driver

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
entity three_state is
port(IN1, ENABLE : in std_logic;
      OUT1 : out std_logic );
end;

architecture rtl of three_state is
begin

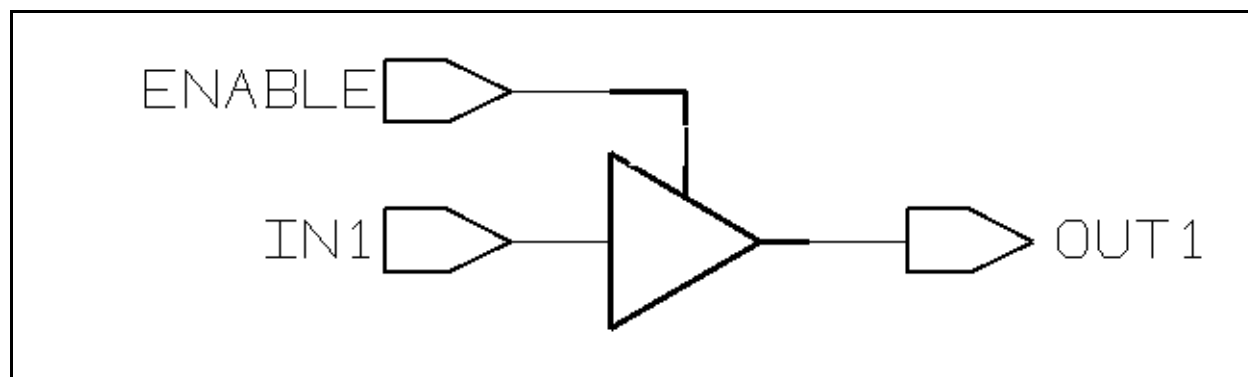
process (IN1, ENABLE) begin
  if (ENABLE = '1') then
    OUT1 <= IN1;
  else
    OUT1 <= 'Z';    -- assigns high-impedance state
  end if;
end process;

end rtl;
```

Example 7-59 Inference Report for Simple Three-State Driver

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

Figure 7-25 Schematic of Simple Three-State Driver



Inferring One Three-State Driver From a Single Process

Example 7-60 provides an example of placing all high-impedance assignments in a single process. In this case, the data is gated and FPGA Compiler II / FPGA Express infers a single three-state driver.

Example 7-61 shows the inference report. Figure 7-26 shows the three-state driver.

Example 7-60 Inferring One Three-State Driver From a Single Process

```
library IEEE;
use IEEE.std_logic_1164.all;

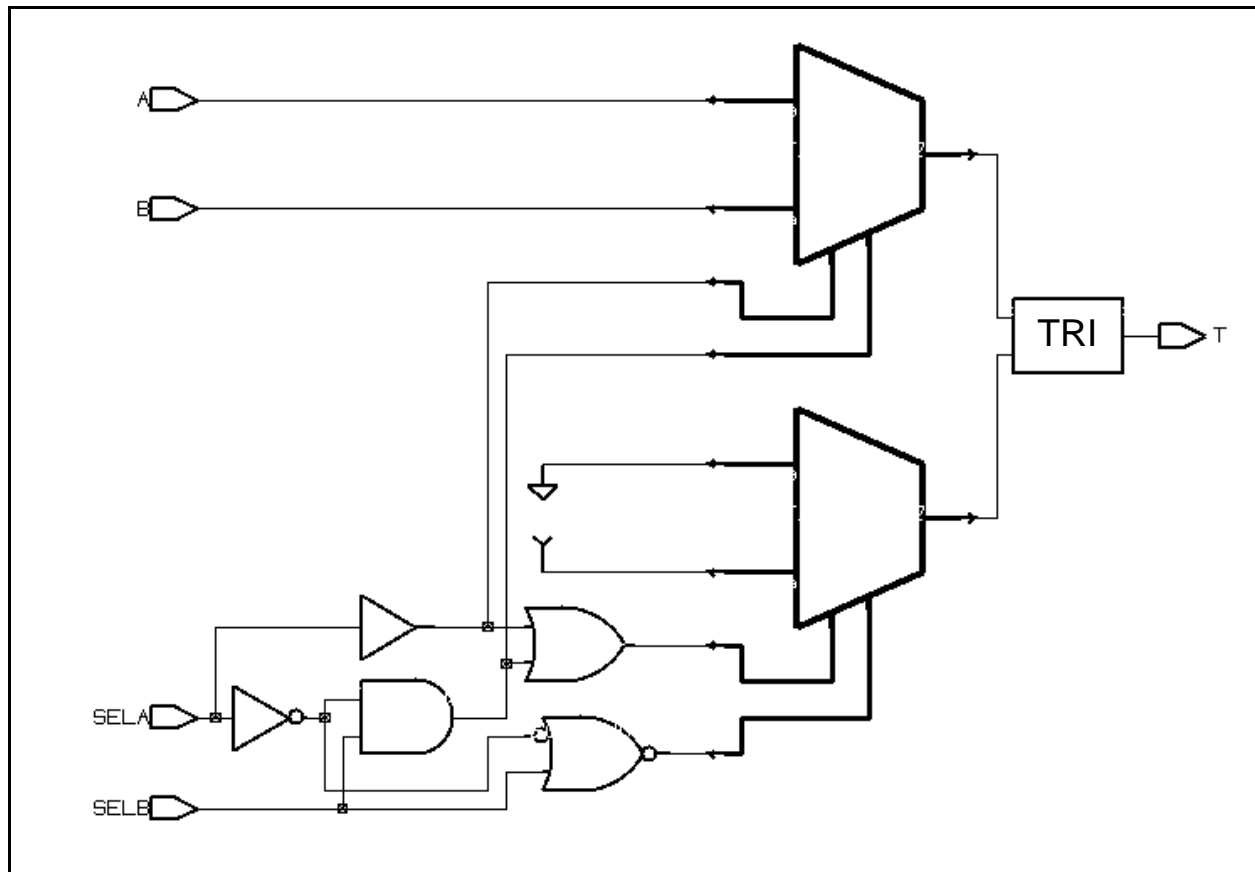
entity three_state is
  port ( A, B, SELA, SELB : in std_logic ;
        T : out std_logic );
end three_state;

architecture rtl of three_state is
begin
  infer : process (SELA, A, SELB, B) begin
    T <= 'Z';
    if (SELA = '1') then
      T <= A;
    elsif (SELB = '1') then
      T <= B;
    end if;
  end process infer;
end rtl;
```

Example 7-61 Single Process Inference Report

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

Figure 7-26 One Three-State Driver Inferred From a Single Process



Inferring Three-State Drivers From Separate Processes

Example 7-62 provides an example of placing each high-impedance assignment in a separate process. In this case, FPGA Compiler II / *FPGA Express* infers multiple three-state drivers.

Example 7-63 shows the inference report. Figure 7-27 shows the design.

Example 7-62 Inferring Two Three-State Drivers From Separate Processes

```
library IEEE;
use IEEE.std_logic_1164.all;

entity three_state is
  port ( A, B, SELA, SELB : in std_logic ;
        T : out std_logic );
end three_state;

architecture rtl of three_state is
begin
  infer1 : process (SELA, A) begin
    if (SELA = '1') then
      T <= A;
    else
      T <= 'Z';
    end if;
  end process infer1;

  infer2 : process (SELB, B) begin
    if (SELB = '1') then
      T <= B;
    else
      T <= 'Z';
    end if;
  end process infer2;

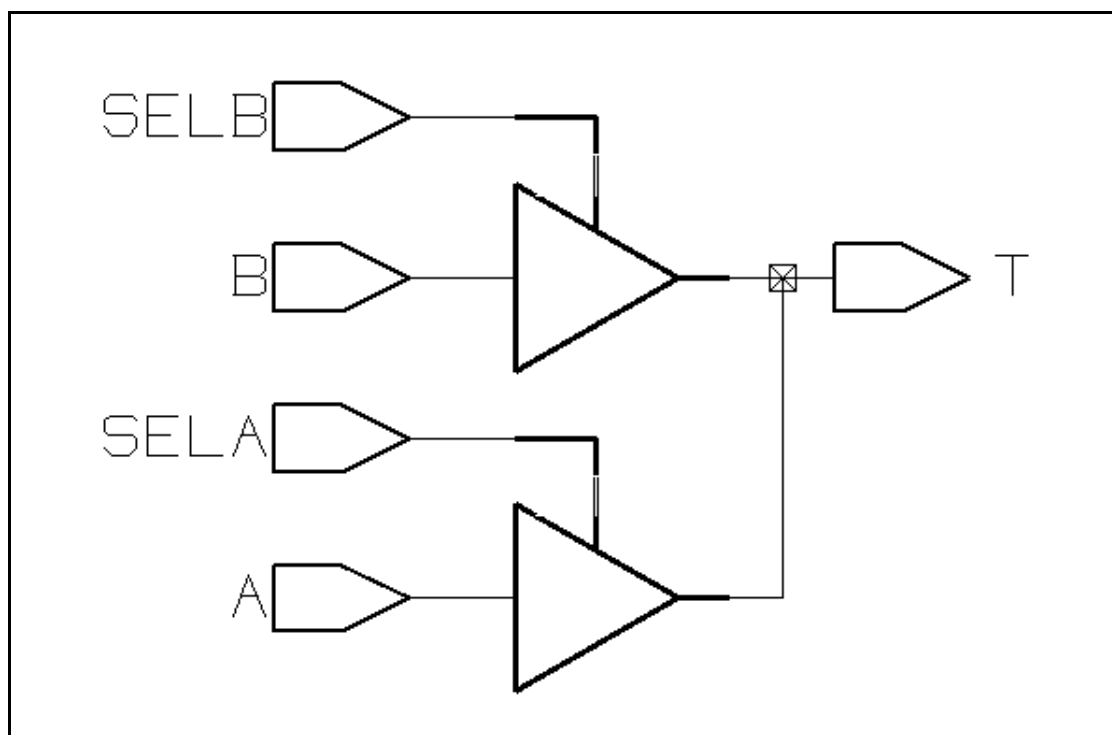
end rtl;
```

Example 7-63 Inference Report for Two Three-State Drivers From Separate Processes

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 7-27 Two Three-State Drivers Inferred From Separate Processes



Three-State Driver With Registered Enable

When a variable, such as `THREE_STATE` in Example 7-64, is assigned to a register and defined as a three-state gate within the same process, FPGA Compiler II / *FPGA Express* also registers the enable pin of the three-state gate.

Example 7-64 shows an example of this type of code, and Example 7-65 shows the inference report. Figure 7-28 shows the schematic generated by the code, a three-state gate with a register on its enable pin.

Example 7-64 Inferring a Three-State Driver With Registered Enable

```

library IEEE;
use IEEE.std_logic_1164.all;

entity three_state is
    port ( DATA, CLK, THREE_STATE : in std_logic ;
          OUT1 : out std_logic );
end three_state;

architecture rtl of three_state is
begin
    infer : process (THREE_STATE, CLK) begin
        if (THREE_STATE = '0') then
            OUT1 <= 'Z';
        elsif (CLK'event and CLK = '1') then
            OUT1 <= DATA;
        end if;
    end process infer;
end rtl;

```

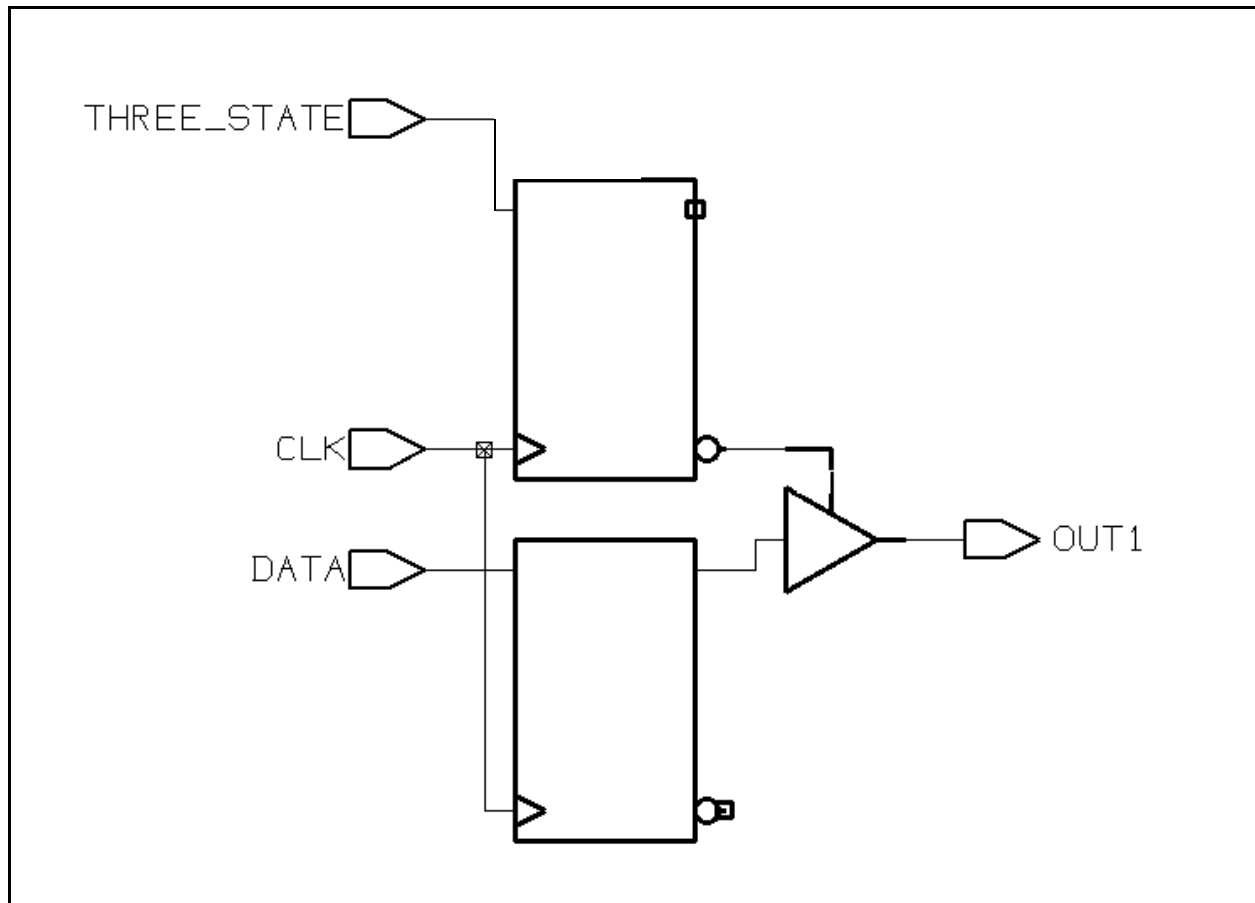
Example 7-65 Inference Report for Three-State Driver With Registered Enable

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
OUT1_reg	Flip-flop	1	-	-	N	N	N	N	N

Three-State Device Name	Type	MB
OUT1_tri	Three-State Buffer	N
OUT1_tri_enable_reg	Flip-Flop (width 1)	N

OUT1_reg
set/reset/toggle: none

Figure 7-28 Three-State Driver With Registered Enable



Three-State Driver Without Registered Enable

Example 7-66 uses two processes to instantiate a three-state gate, with a flip-flop on the input pin.

Example 7-67 shows the inference report. Figure 7-29 shows the schematic generated by the code.

Example 7-66 Three-State Driver Without Registered Enable

```
library IEEE;
use IEEE.std_logic_1164.all;

entity ff_3state2 is
    port ( DATA, CLK, THREE_STATE : in std_logic ;
          OUT1 : out std_logic );
end ff_3state2;

architecture rtl of ff_3state2 is
    signal TEMP : std_logic;
begin

    process (CLK) begin
        if (CLK'event and CLK = '1') then
            TEMP <= DATA;
        end if;
    end process;

    process (THREE_STATE, TEMP) begin
        if (THREE_STATE = '0') then
            OUT1 <= 'Z';
        else
            OUT1 <= TEMP;
        end if;
    end process;

end rtl;
```

Example 7-67 Inference Report for Three-State Driver Without Registered Enable

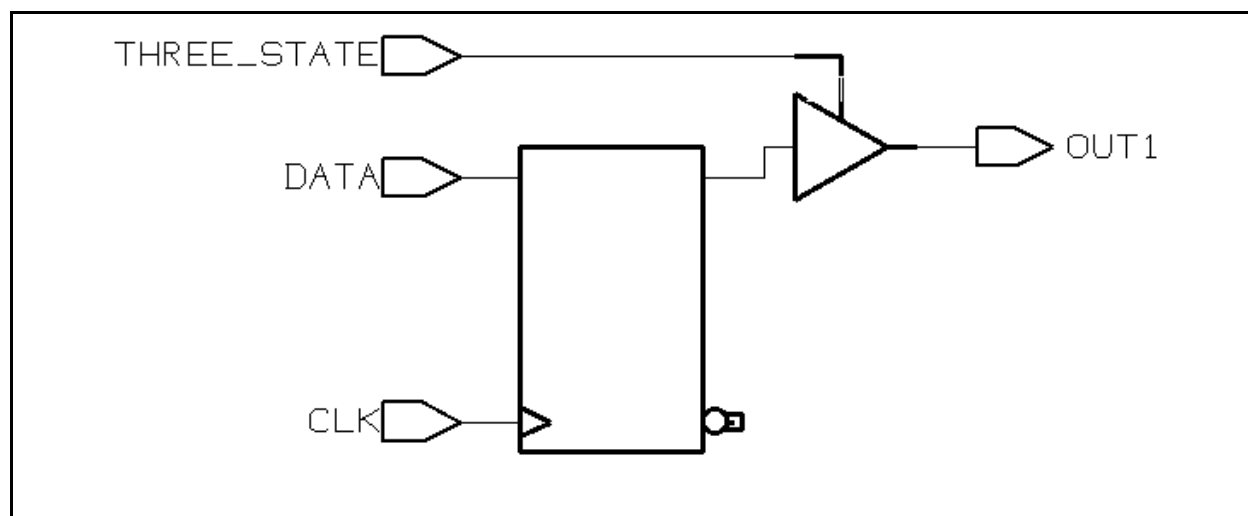
Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
TEMP_reg	Flip-flop	1	-	-	N	N	N	N	N

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

TEMP_reg
set/reset/toggle: none

Register and Three-State Inference

Figure 7-29 Three-State Driver Without Registered Enable



Understanding the Limitations of Three-State Inference

You can use the Z value as

- A signal assignment
- A variable assignment
- A function call argument
- A return value
- An aggregate definition

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 such expressions to false, and the pre- and post-synthesis simulation results might differ. For this reason, FPGA Compiler II / *FPGA Express* issues a warning when it synthesizes such comparisons.

Example 7-68 shows an incorrect use of the Z value. Example 7-69 shows a correct use of the Z value.

Example 7-68 Incorrect Use of the Z Value in an Expression

```
OUT_VAL <= ( 'Z' and IN_VAL );
```

Example 7-69 Correct Use of the Z Value in an Expression

```
if (IN_VAL = 'Z') then
```


8

Writing Circuit Descriptions

To understand FPGA Compiler II / FPGA *Express* and to write VHDL descriptions that produce efficient synthesized circuits, study the information presented in the following sections of this chapter:

- How Statements Are Mapped to Logic
- Design Structure
- Asynchronous Designs
- Don't Care Inference
- Synthesis Issues

Some general guidelines for writing efficient circuit descriptions are

- 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

VHDL descriptions are mapped to combinational logic by creation of blocks of logic. A statement or an operator in a VHDL function can represent a block of combinational logic or, in some cases, a latch or register.

The statements shown in Example 8-1 represent 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 8-1 Four Logic Blocks

```
if (B < 10) then
    Y = A + B;
else
    Y = A + 10;
end if;
```

The logic blocks created by FPGA Compiler II / *FPGA Express* are custom-built for their environment. 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 VHDL statements and operators.

Design Structure

A design's structure influences the size and complexity of the resulting synthesized circuit. These sections help you understand the concepts:

- Adding Structure
- Using Design Knowledge
- Optimizing Arithmetic Expressions
- Changing an Operator Bit-Width
- Using State Information
- Propagating Constants
- Sharing Complex Operators

Adding Structure

FPGA Compiler II / *FPGA Express* gives you 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.

Using Variables and Signals

You control design structure with your ordering of assignment statements and your use of variables. Each VHDL signal assignment, process, or component instantiation implies a piece of logic. Each variable or signal implies a wire. By using these constructs, you can connect entities in any configuration.

Example 8-2 and Example 8-3 show two possible descriptions of an adder's carry chain. Figure 8-1 illustrates the resulting design.

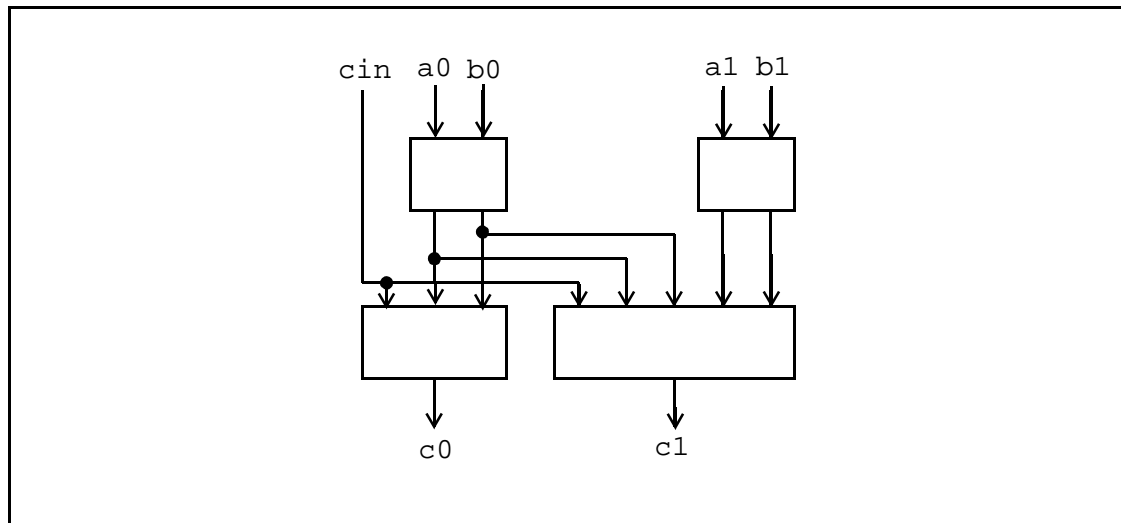
Example 8-2 Ripple Carry Chain

```
-- A is the addend
-- B is the augend
-- C is the carry
-- Cin is the carry in
C0 <= (A0 and B0) or
      ((A0 or B0) and Cin);
C1 <= (A1 and B1) or
      ((A1 or B1) and C0);
```

Example 8-3 Carry-Lookahead Chain

```
-- Ps are propagate
-- Gs are generate
p0 <= a0 or b0;
g0 <= a0 and b0;
p1 <= a1 or b1;
g1 <= a1 and b1;
c0 <= g0 or (p0 and cin);
c1 <= g1 or (p1 and g0) or
      (p1 and p0 and cin);
```

Figure 8-1 Ripple Carry and Carry-Lookahead Chain Design



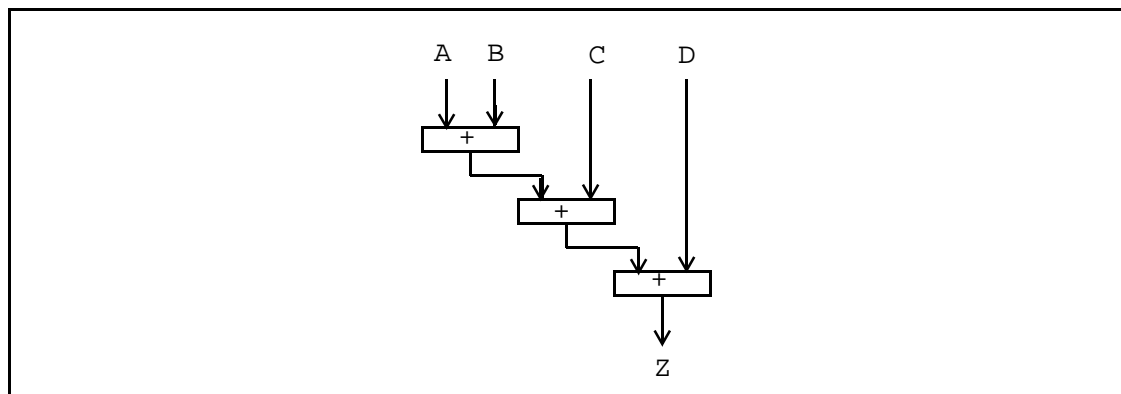
Using Parentheses

Another way to control the structure of a design is to use parentheses to define logic groupings. Example 8-4 describes a 4-input adder grouping. Figure 8-2 illustrates the resulting design.

Example 8-4 4-Input Adder

$Z \leq (A + B) + C + D;$

Figure 8-2 Diagram of 4-Input Adder

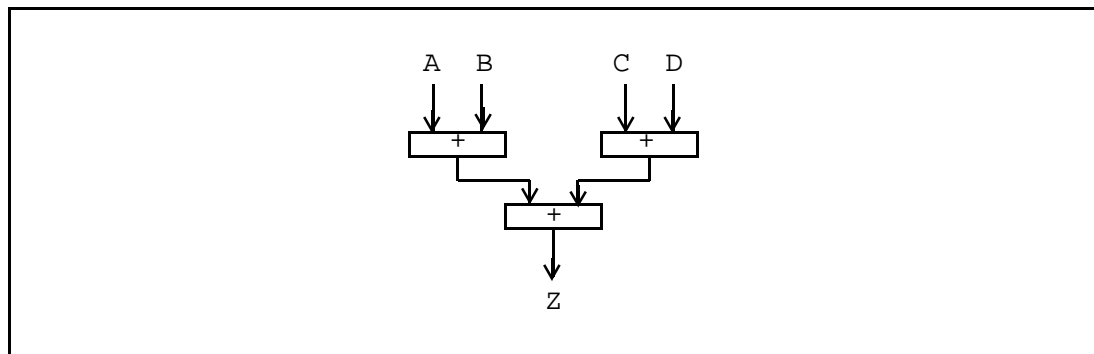


Example 8-5 describes a 4-input adder grouping that is structured with parentheses. Figure 8-3 illustrates the design.

Example 8-5 4-Input Adder Structured With Parentheses

$Z \leq (A + B) + (C + D);$

Figure 8-3 Diagram of 4-Input Adder With Parentheses



Using Design Knowledge

In many circumstances, you can improve the quality of synthesized circuits by describing your high-level knowledge of a circuit better. 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. Two 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 8-6.

Example 8-6 Simple Arithmetic Expression

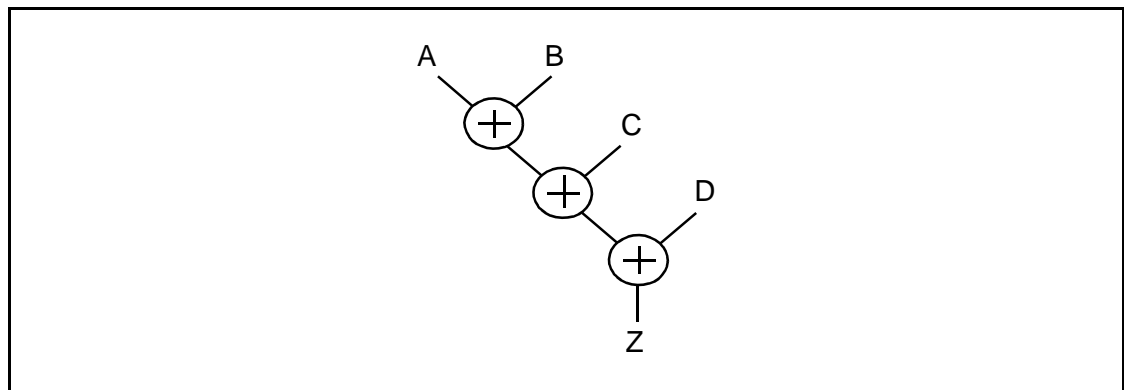
$Z \leq A + B + C + D;$

The parser performs each addition in order, as though parentheses were placed within the expression as follows:

$Z \leq ((A + B) + C) + D;$

The parser constructs the expression tree shown in Figure 8-4.

Figure 8-4 Default Expression Tree



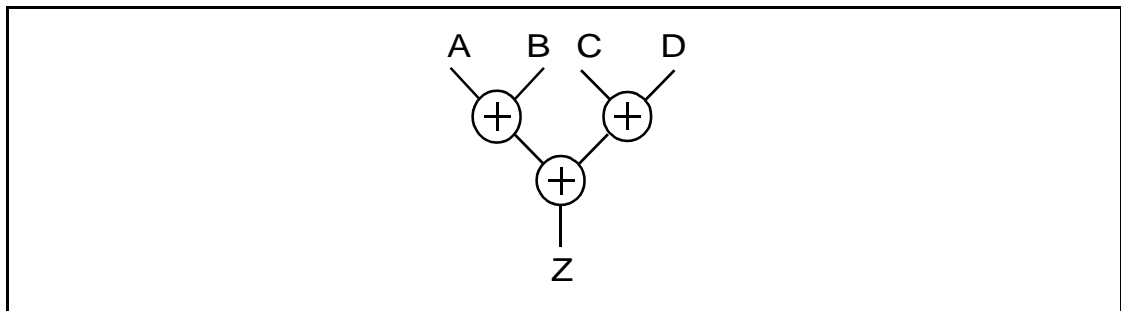
Considering Signal Arrival Times

To figure out the delay through an expression tree, FPGA Compiler II / *FPGA Express* considers the arrival times of each signal in the expression. If the arrival times of all the signals are the same, the length of the critical path of the expression in Example 8-6 equals three adder delays. The critical path delay can be reduced to two adder delays if you insert parentheses as follows:

```
Z <= (A + B) + (C + D);
```

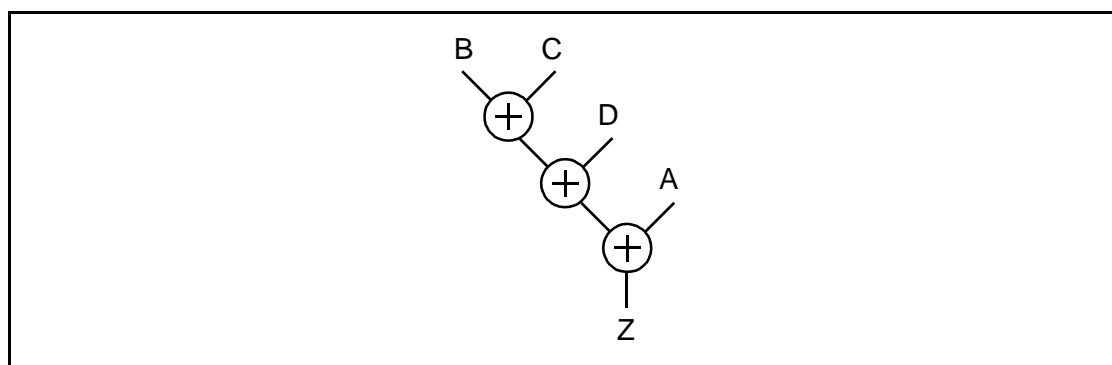
The parser constructs the subexpression tree shown in Figure 8-5:

Figure 8-5 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 8-6.

Figure 8-6 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 in a way that violates these groupings.

To see the effect of parentheses on the construction of an expression tree, consider Example 8-7.

Example 8-7 Parentheses in an Arithmetic Expression

$Q \leq ((A + (B + C)) + D + E) + F;$

The parentheses in the expression in Example 8-7 define the following subexpressions:

- 1 $(B + C)$
- 2 $(A + (B + C))$
- 3 $((A + (B + C)) + D + E)$

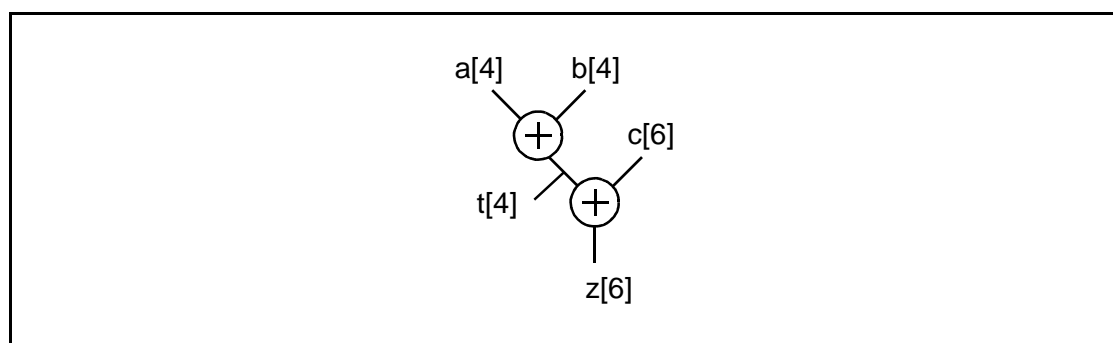
These subexpressions must be preserved in the expression tree. The default expression tree for Example 8-7 is shown in Figure 8-7.

Example 8-8 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
```

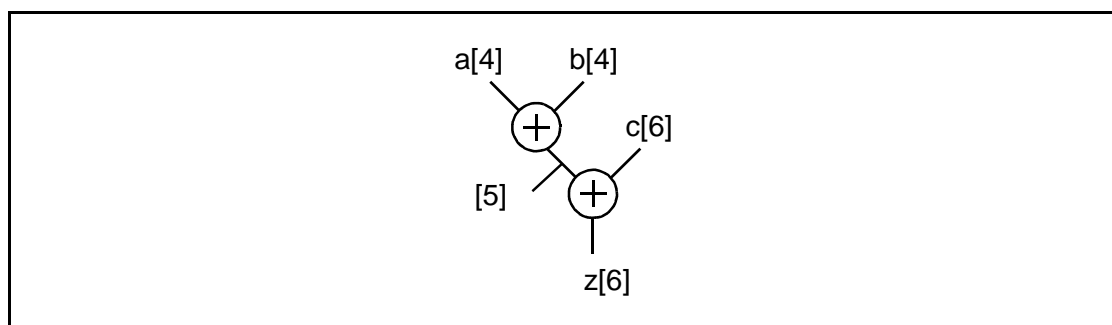
In Example 8-8, three variables ($a + b + c$) are added. A temporary variable, t , holds the intermediate result of $a + b$. If t is declared as a 4-bit variable, the overflow bits from the addition of $a + b$ are truncated. The parser determines the default structure of the expression tree, which is shown in Figure 8-8.

Figure 8-8 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 5 bits are needed to store the intermediate result of the addition, so no overflow condition exists. The results of the final addition can be different from those of 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 8-9.

Figure 8-9 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, and then use the temporary variable wherever you want to repeat the subexpression. Example 8-9 shows a group of simple additions that use the common subexpression $(a + b)$.

Example 8-9 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 the building of more adders. Consider Example 8-10, where $A + B$ is a common subexpression.

Example 8-10 Sharing Common Subexpressions—Increases Area

```
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 necessary to implement this section of code.

```
(A + B)
(C + D)
(E + F)
```

If the common subexpression is not shared, only two adders are necessary: 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 process and considers area costs and timing characteristics. To turn off the sharing of common subexpressions for the current design, use the constraint manager. The default is true.

Example 8-11 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.

Changing an Operator Bit-Width

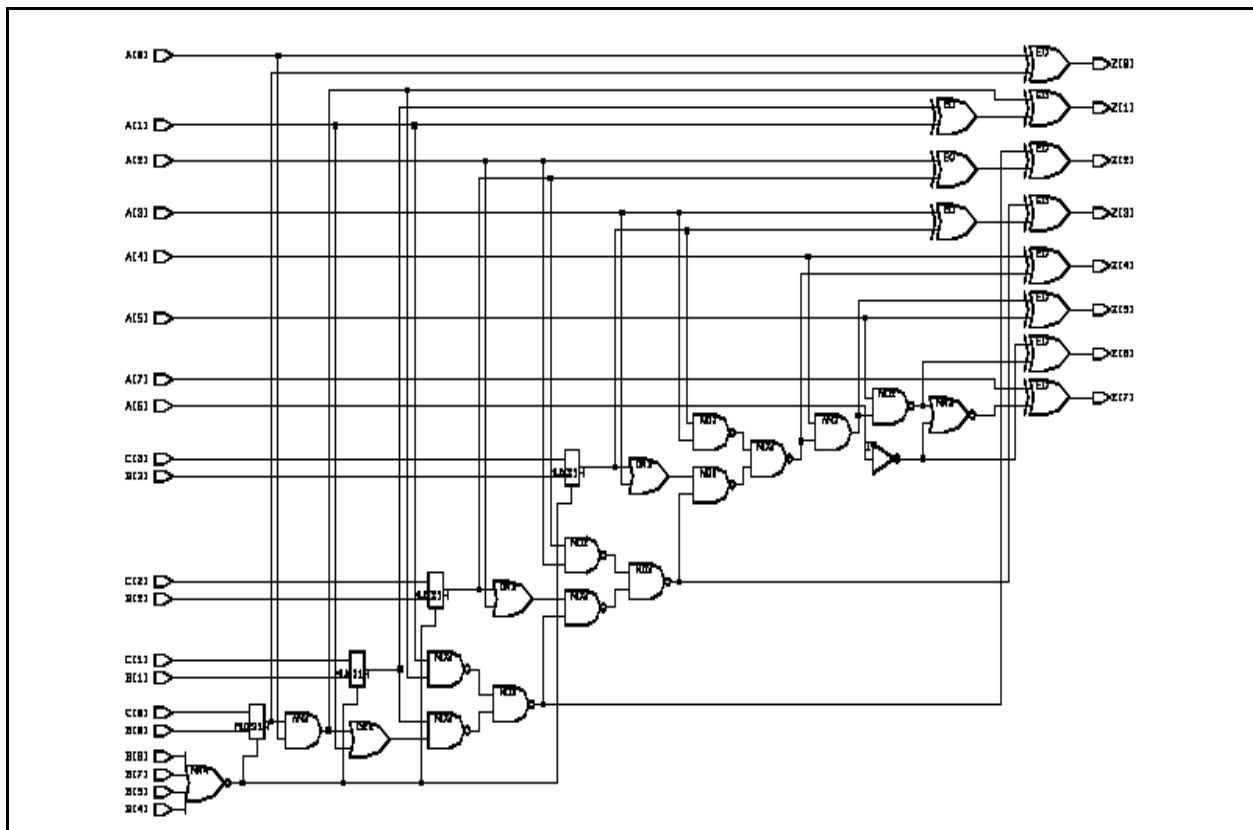
The adder in Example 8-12 sums the 8-bit value of A (a BYTE) with the 8-bit value of $TEMP$. $TEMP$'s value is either B , which is used only when it is less than 16, or C , which is a 4-bit value (a NIBBLE). Therefore, the upper 4 bits of $TEMP$ are always 0. FPGA Compiler II / *FPGA Express* cannot derive this fact, because $TEMP$ is declared with type BYTE.

You can simplify the synthesized circuit by changing the declared type of $TEMP$ to NIBBLE (a 4-bit value). With this modification, half adders, rather than full adders, are required to implement the top 4 bits of the adder circuit, which Figure 8-10 illustrates.

Example 8-12 Function With One Adder

```
function ADD_IT_16 (A, B: BYTE; C: NIBBLE) return BYTE is
  variable TEMP: BYTE;
begin
  if B < 16 then
    TEMP <= B;
  else
    TEMP <= C;
  end if;
  return A + TEMP;
end;
```

Figure 8-10 Function With One Adder Schematic



Example 8-13 shows how this change in TEMP's declaration can yield a significant savings in circuit area, which Figure 8-11 illustrates.

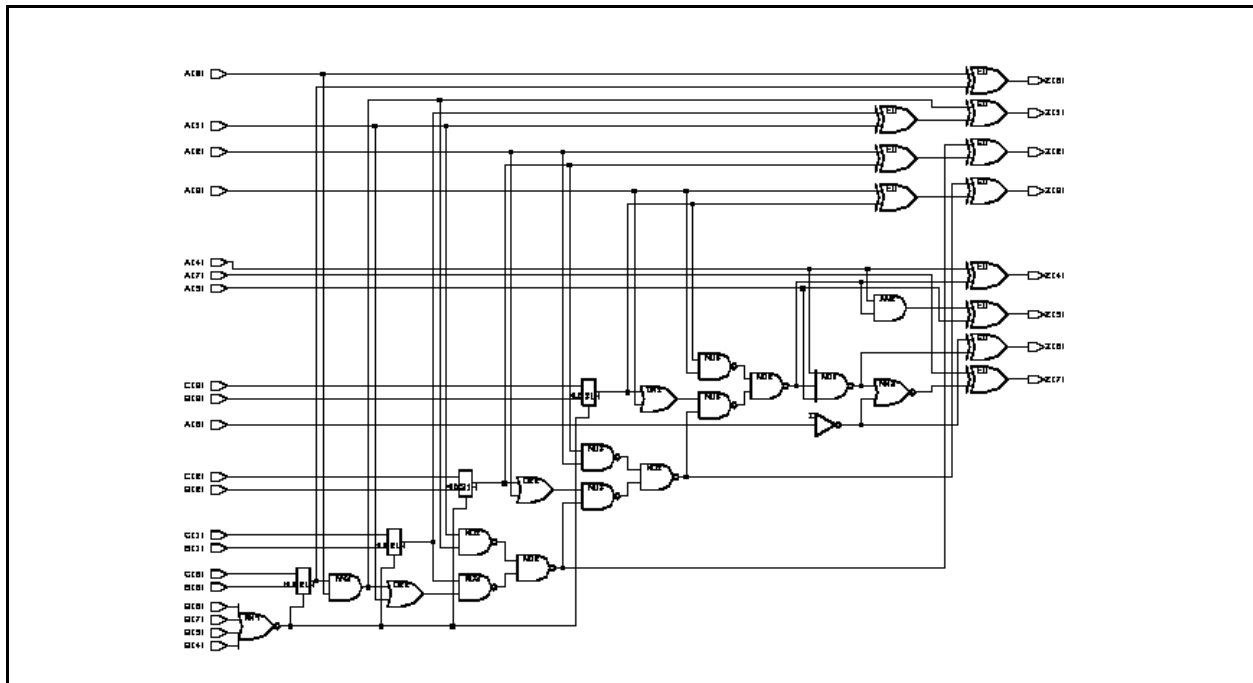
Example 8-13 Using Design Knowledge to Simplify an Adder

```

function ADD_IT_16 (A, B: BYTE; C: NIBBLE)
  return BYTE is
    variable TEMP: NIBBLE;    -- Now only 4 bits
begin
  if B < 16 then
    TEMP <= NIBBLE(B);        -- Cast BYTE to NIBBLE
  else
    TEMP <= C;
  end if;
  return A + TEMP;             -- Single adder
end;

```

Figure 8-11 Using TEMP Declaration to Save Circuit Area



Using State Information

You can also apply design knowledge in sequential designs. Often you can make strong assertions about the value of a signal in a particular state of a finite-state machine. You can describe this information to FPGA Compiler II / *FPGA Express*. Example 8-14 shows the VHDL description of a simple state machine that uses two processes. Figure 8-12 illustrates the design.

Example 8-14 A Simple State Machine

```
package STATES is
    type STATE_TYPE is (SET0, HOLD0, SET1);
end STATES;

use work.STATES.all;

entity MACHINE is
    port(X, CLOCK: in BIT;
         CURRENT_STATE: buffer STATE_TYPE;
         Z: buffer BIT);
end MACHINE;

architecture BEHAVIOR of MACHINE is
    signal NEXT_STATE: STATE_TYPE;
    signal PREVIOUS_Z: BIT;
begin

    -- Process to hold combinational logic.
    COMBIN: process(CURRENT_STATE, X, PREVIOUS_Z)
    begin
        case CURRENT_STATE is
            when SET0 =>
                Z <= '0';                -- Set Z to '0'
                NEXT_STATE <= HOLD0;

            when HOLD0 =>
                Z <= PREVIOUS_Z;          -- Hold value of Z
                if X = '0' then
```

```

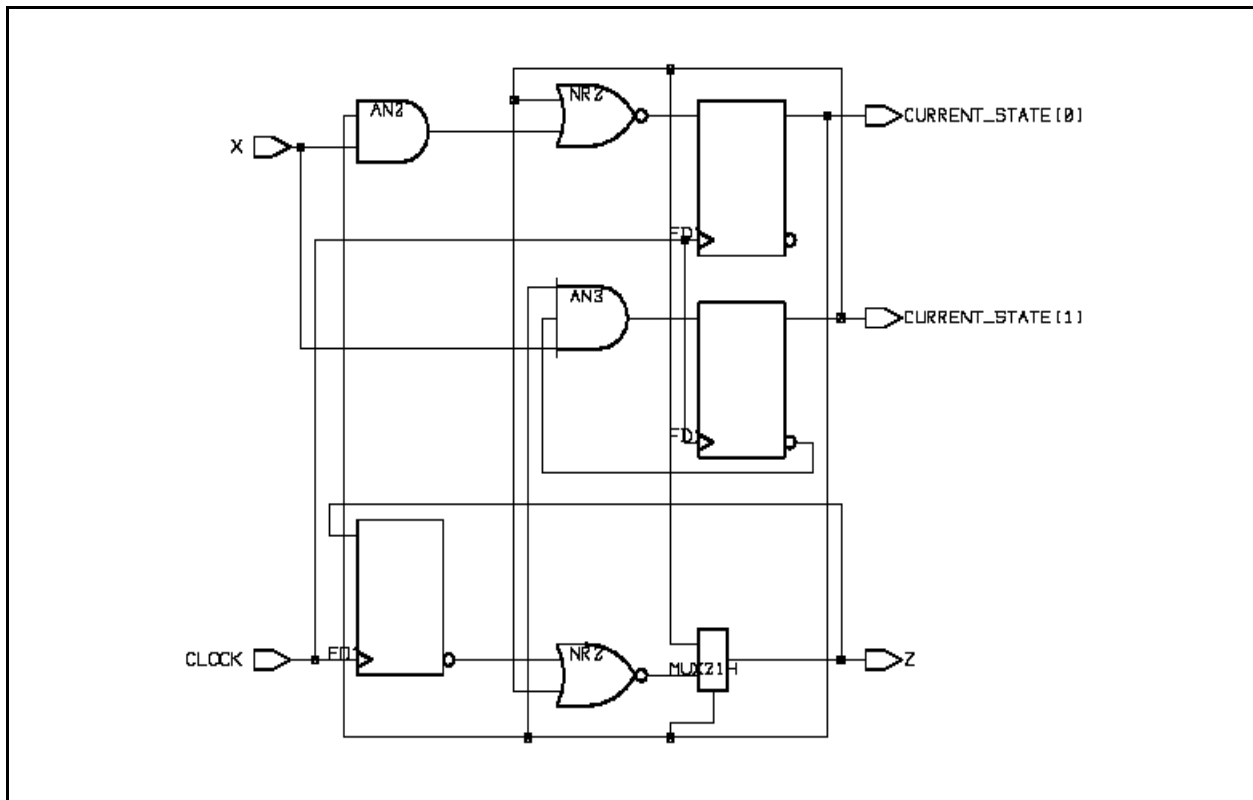
        NEXT_STATE <= HOLD0;
    else
        NEXT_STATE <= SET1;
    end if;

    when SET1 =>                                -- Set Z to '1'
        Z <= '1';
        NEXT_STATE <= SET0;
    end case;
end process COMBIN;

-- Process to hold synchronous elements (flip-flops).
SYNCH: process
begin
    wait until CLOCK'event and CLOCK = '1';
    CURRENT_STATE <= NEXT_STATE;
    PREVIOUS_Z <= Z;
end process SYNCH;
end BEHAVIOR;

```

Figure 8-12 Schematic of Simple State Machine With Two Processes



In the state HOLD0, output Z retains its value from the previous state. To accomplish this, you insert a flip-flop to hold PREVIOUS_Z. However, you can make some assertions about the value of Z. In state HOLD0, the value of Z is always 0. You can deduce this from the fact that state HOLD0 is entered only from state SET0, where Z is always assigned '0'.

Example 8-15 shows how you can change the VHDL description to use this assertion, resulting in a simpler circuit. Figure 8-13 illustrates the circuit.

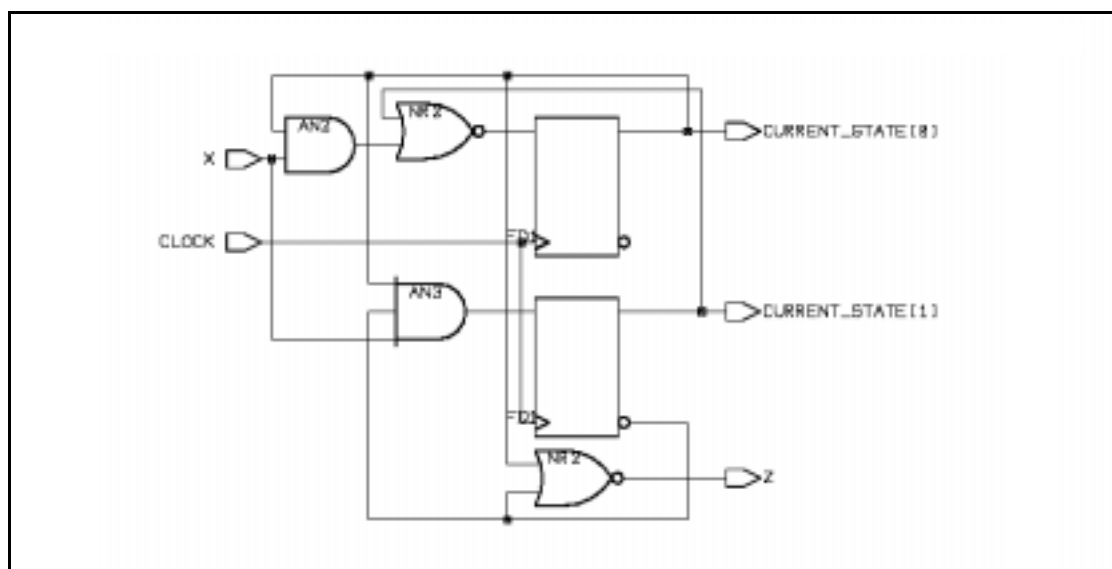
Example 8-15 A Better Implementation of a State Machine

```
package STATES is
    type STATE_TYPE is (SET0, HOLD0, SET1);
end STATES;
use work.STATES.all;

entity MACHINE is
    port(X, CLOCK: in BIT;
          CURRENT_STATE: buffer STATE_TYPE;
          Z: buffer BIT);
end MACHINE;

architecture BEHAVIOR of MACHINE is
    signal NEXT_STATE: STATE_TYPE;
begin
    -- Combinational logic.
    COMBIN: process(CURRENT_STATE, X)
    begin
        case CURRENT_STATE is
            when SET0 =>
                Z <= '0';                -- Set Z to '0'
                NEXT_STATE <= HOLD0;
            when HOLD0 =>
                Z <= '0';                -- Hold Z at '0'
                if X = '0' then
                    NEXT_STATE <= HOLD0;
                else
                    NEXT_STATE <= SET1;
                end if;
            when SET1 =>
                Z <= '1';                -- Set Z to '1'
                NEXT_STATE <= SET0;
        end case;
    end process COMBIN;
    -- Process to hold synchronous elements (flip-flops)
    SYNCH: process
    begin
        wait until CLOCK'event and CLOCK = '1';
        CURRENT_STATE <= NEXT_STATE;
    end process SYNCH;
end BEHAVIOR;
```

Figure 8-13 Schematic of an Improved State Machine



Propagating Constants

Constant propagation is the compile-time evaluation of expressions containing constants. FPGA Compiler II / *FPGA Express* uses constant propagation to reduce the amount of hardware required to implement operators. For example, a + operator with a constant 1 as one of its arguments causes an incrementer to be built, rather than a general adder. If both arguments of + or any other operator are constants, no hardware is constructed, because the expression's value is calculated by FPGA Compiler II / *FPGA Express* and inserted directly in the circuit.

Other operators that benefit from constant propagation include comparators and shifters. Shifting a vector by a constant amount requires no logic to implement; it requires only a reshuffling (rewiring) of bits.

Sharing Complex Operators

The efficiency of a synthesized design depends primarily on how you describe its component structure. The optimization of individual components, especially those made from random logic, produces similar results from two very different descriptions. Therefore, concentrate the majority of your design effort on the implied component hierarchy (as discussed in the preceding sections) rather than on the logical descriptions. Chapter 2, "Design Descriptions", discusses how to define a VHDL design hierarchy.

FPGA Compiler II / *FPGA Express* supports many shorthand VHDL expressions. There is no benefit to using a verbose syntax when a shorter description is adequate. Example 8-16 shows four equivalent groups of statements.

Example 8-16 Equivalent Statements

```
signal A, B, C: BIT_VECTOR(3 downto 0);  
.  
.  
.  
C <= A and B;  
-----  
C(3 downto 0) <= A(3 downto 0) and B(3 downto 0);  
-----  
C(3) <= A(3) and B(3);  
C(2) <= A(2) and B(2);  
C(1) <= A(1) and B(1);  
C(0) <= A(0) and B(0);  
-----  
for I in 3 downto 0 loop  
    C(I) <= A(I) and B(I);  
end loop;
```

Asynchronous Designs

In a synchronous design, all flip-flops use a single clock that is a primary input to the design and there are no combinational feedback paths. Synchronous designs perform the same function regardless of the clock rate if all signals can propagate through the design's combinational logic during the clock's cycle time.

FPGA Compiler II / *FPGA Express* treats all designs as synchronous. It can therefore change the timing behavior of the combinational logic if the maximum and minimum delay requirements are met.

FPGA Compiler II / *FPGA Express* always preserves the Boolean function computed by logic, assuming that the clock arrives after all signals have propagated. FPGA Compiler II / *FPGA Express*'s built-in timing verifier helps determine the slowest path (critical path) through the logic, which determines how fast the clock can run.

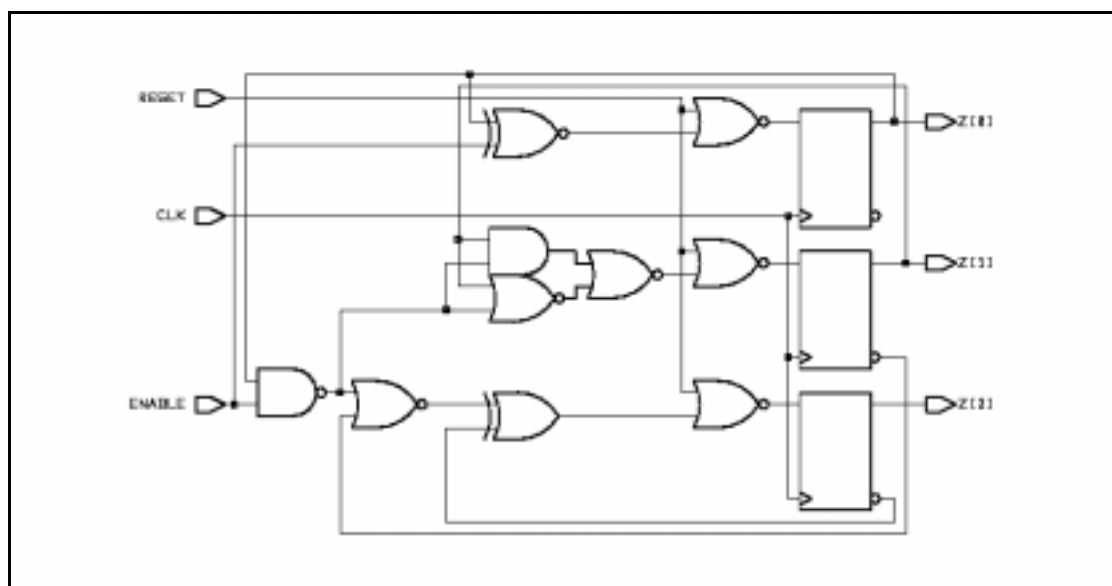
FPGA Compiler II / *FPGA Express* provides some support for asynchronous designs, but you must assume a greater responsibility for the accuracy of your circuits. Although fully synchronous circuits usually agree with their simulation models, asynchronous circuits might not. FPGA Compiler II / *FPGA Express* might not warn you when a design is not fully synchronous. Be aware of the possibility of asynchronous timing problems.

The most common way to produce asynchronous logic in VHDL is to use gated clocks on latches or flip-flops. Example 8-17 shows a fully synchronous design, a counter with synchronous ENABLE and RESET inputs. Because it is synchronous, this counter works if the clock speed is slower than the critical path. Figure 8-14 illustrates the design.

Example 8-17 Fully Synchronous Counter With Reset and Enable

```
entity COUNT is
  port(RESET, ENABLE, CLK: in      BIT;
        Z:                        buffer INTEGER range 0 to 7);
end;
architecture ARCH of COUNT is
begin
  process(RESET, ENABLE, CLK, Z)
  begin
    if (CLK'event and CLK = '1') then
      if (RESET = '1') then          -- occurs on clock edge
        Z <= 0;
      elsif (ENABLE = '1') then      -- occurs on clock edge
        if (Z = 7) then
          Z <= 0;
        else
          Z <= Z + 1;
        end if;
      end if;
    end if;
  end process;
end ARCH;
```

Figure 8-14 Schematic of Synchronous Counter With Reset and Enable



Example 8-18 shows an asynchronous version of the design in Example 8-17. The version in Example 8-18 uses two common asynchronous design techniques:

- The first technique, shown in Example 8-15, enables the counter by using an AND gate on the clock and enable signals.
- The second technique, shown in Figure 8-16, uses an asynchronous reset.

These techniques work only when the proper timing relationships exist between the reset signal (RESET) and the clock signal (CLK) and there are no glitches in these signals.

Example 8-18 Design With Gated Clock and Asynchronous Reset

```
entity COUNT is
  port(RESET, ENABLE, CLK: in      BIT;
        Z:                        buffer INTEGER range 0 to 7);
end;

architecture ARCH of COUNT is
  signal GATED_CLK: BIT;
begin
  GATED_CLK <= CLK and ENABLE; -- clock gated by ENABLE

  process(RESET, GATED_CLK, Z)
  begin
    if (RESET = '1') then          -- asynchronous reset
      Z <= 0;
    elsif (GATED_CLK'event and GATED_CLK = '1') then
      if (Z = 7) then
        Z <= 0;
      else
        Z <= Z + 1;
      end if;
    end if;
  end process;
end ARCH;
```

Figure 8-15 Design With AND Gate on Clock and Enable Signals

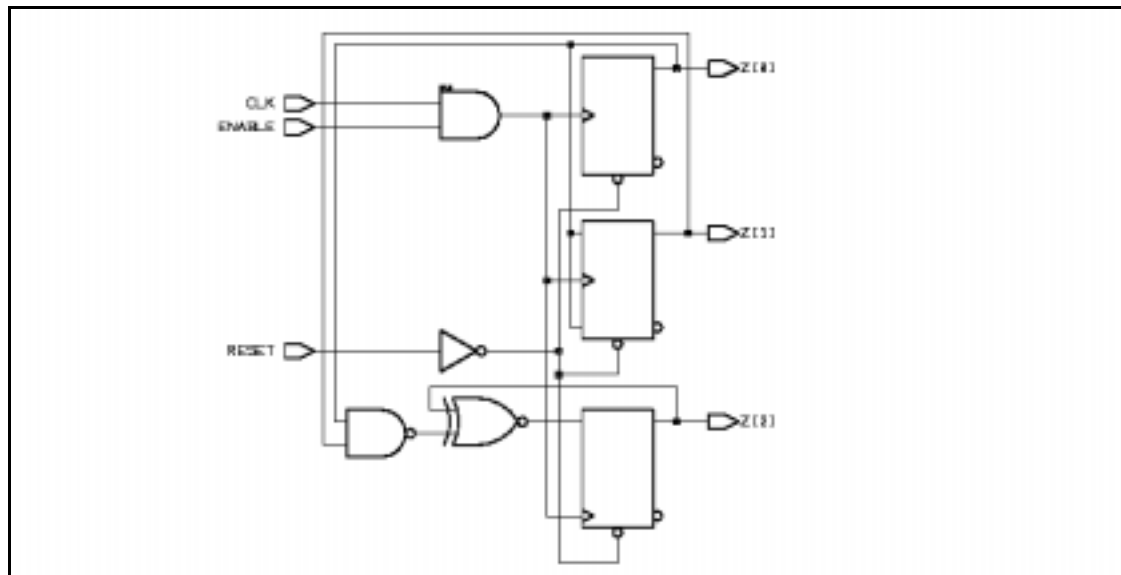
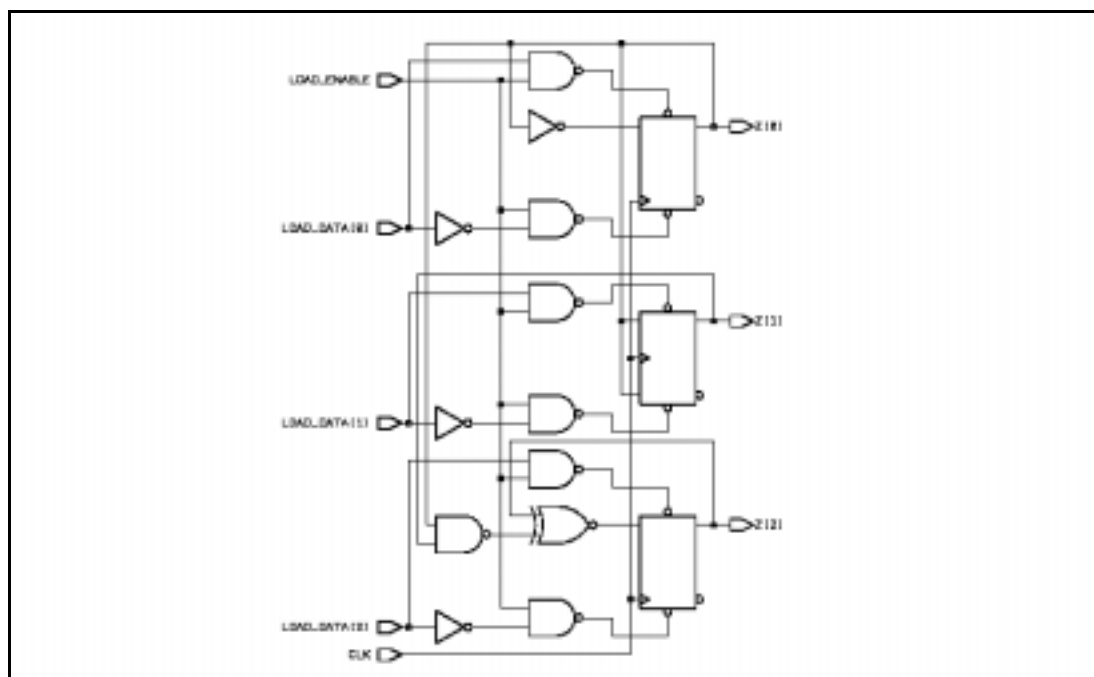


Figure 8-16 Design With Asynchronous Reset



Example 8-19 shows an asynchronous design that might not work, because FPGA Compiler II / *FPGA Express* does not guarantee that the combinational logic it builds has no hazards (glitches).

Example 8-19 Incorrect Design (Counter With Asynchronous Load)

```
entity COUNT is
  port (LOAD_ENABLE, CLK: in      BIT;
        LOAD_DATA:      in      INTEGER range 0 to 7;
        Z:              buffer INTEGER range 0 to 7);
end;

architecture ARCH of COUNT is

begin
  process (LOAD_ENABLE, LOAD_DATA, CLK, Z)
  begin
    if (LOAD_ENABLE = '1') then
      Z <= LOAD_DATA;
    elsif (CLK'event and CLK = '1') then
      if (Z = 7) then
        Z <= 0;
      else
        Z <= Z + 1;
      end if;
    end if;
  end process;
end ARCH;
```

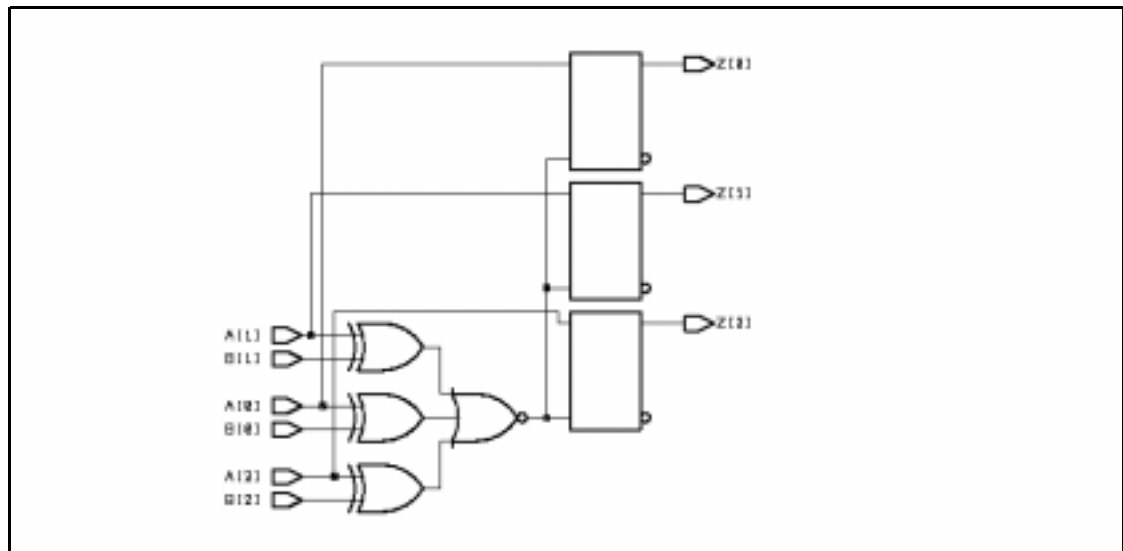
The design in Example 8-19 works only when the logic driving the preset and clear pins of the flip-flops that hold Z is faster than the clock speed. If you use this design style, you must simulate the synthesized circuit thoroughly. You also need to inspect the synthesized logic, because potential glitches might not appear in simulation. For a safer design, use a synchronous LOAD_ENABLE.

A design synthesized with complex logic driving the gate of a latch rarely works. Example 8-20 describes an asynchronous design that never works. Figure 8-17 shows the resulting schematic.

Example 8-20 Incorrect Asynchronous Design With Gated Clock

```
entity COMP is
  port(A, B: in      INTEGER range 0 to 7;
        Z:    buffer INTEGER range 0 to 7);
end;
architecture ARCH of COMP is
begin
  process(A, B)
  begin
    if (A = B) then
      Z <= A;
    end if;
  end process;
end ARCH;
```

Figure 8-17 Schematic of Incorrect Asynchronous Design With Gated Clock



In Example 8-20 and Figure 8-17, the comparator's output latches the value A onto the value Z. This design might work under behavioral simulation where the comparison happens instantly. However, the hardware comparator generates glitches that cause the latches to store new data when they should not.

Don't Care Inference

You can greatly reduce circuit area by using don't care inference. To use a don't care value in your design, create an enumerated type for the don't care value (the standard VHDL BIT type does not include don't care values).

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 process, in the default section of a case statement, or in the else section of an if statement.

Example 8-21 shows don't care encoding for a seven-segment LED decoder. Enumeration encoding 'D' represents the don't care state. Figure 8-18 illustrates the design.

Example 8-21 Using don't care Type for Seven-Segment LED Decoder

```
package P is
  type MULTI is ('0', '1', 'D', 'Z');
  attribute ENUM_ENCODING: STRING;
  attribute ENUM_ENCODING of MULTI : type is "0 1 D Z";
  type MULTI_VECTOR is array (INTEGER range <>) of MULTI;
end P;

use work.P.all;

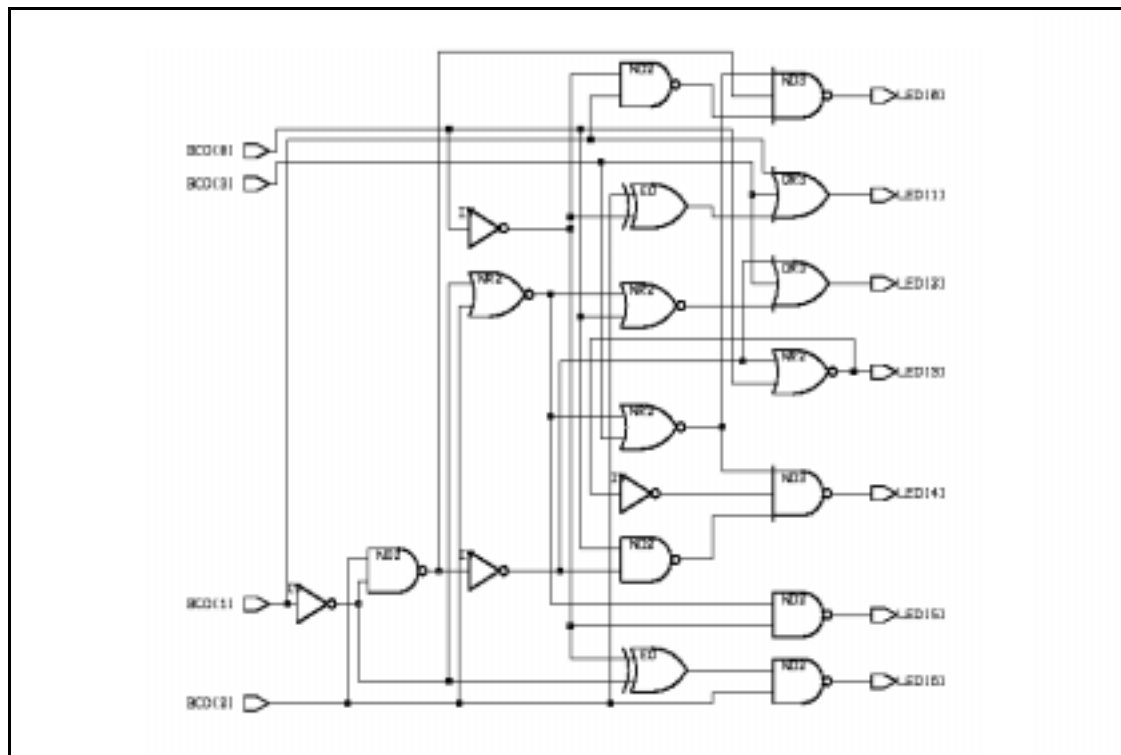
entity CONVERTER is
  port(BCD: in MULTI_VECTOR(3 downto 0);
        LED: out MULTI_VECTOR(6 downto 0));
  -- pragma dc_script_begin
  -- set_flatten true
  -- pragma dc_script_end
end CONVERTER;
```

```

architecture BEHAVIORAL of CONVERTER is
begin
  CONV: process(BCD)
  begin
    case BCD is
      when "0000" => LED <= "1111110";
      when "0001" => LED <= "1100000";
      when "0010" => LED <= "1011011";
      when "0011" => LED <= "1110011";
      when "0100" => LED <= "1100101";
      when "0101" => LED <= "0110111";
      when "0110" => LED <= "0111111";
      when "0111" => LED <= "1100010";
      when "1000" => LED <= "1111111";
      when "1001" => LED <= "1110111";
      when others => LED <= "DDDDDDD";
    end case;
  end process CONV;
end BEHAVIORAL;

```

Figure 8-18 Seven-Segment LED Decoder With Don't Care Type



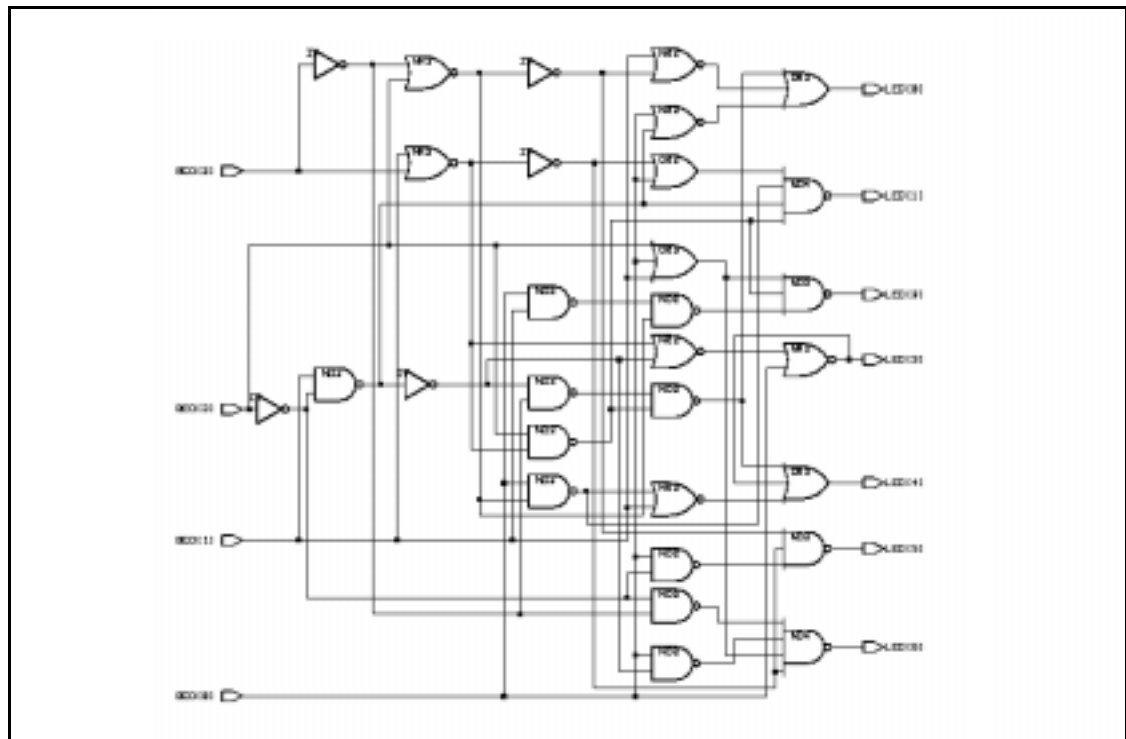
Example 8-22 shows the seven-segment decoder used in Example 8-21, but the default assignment to LED is 0 instead of don't care. Note the larger gate count in the circuit without don't care values. Figure 8-19 illustrates the design.

Example 8-22 Seven-Segment Decoder Without Don't Care Type

```
entity CONVERTER is
  port (BCD: in BIT_VECTOR(3 downto 0);
        LED: out BIT_VECTOR(6 downto 0));
  -- pragma dc_script_begin
  -- set_flatten true
  -- pragma dc_script_end
end CONVERTER;

architecture BEHAVIORAL of CONVERTER is
begin
  CONV: process(BCD)
  begin
    case BCD is
      when "0000" => LED <= "1111110";
      when "0001" => LED <= "1100000";
      when "0010" => LED <= "1011011";
      when "0011" => LED <= "1110011";
      when "0100" => LED <= "1100101";
      when "0101" => LED <= "0110111";
      when "0110" => LED <= "0111111";
      when "0111" => LED <= "1100010";
      when "1000" => LED <= "1111111";
      when "1001" => LED <= "1110111";
      when others => LED <= "0000000";
    end case;
  end process CONV;
end BEHAVIORAL;
```

Figure 8-19 Seven-Segment LED Decoder With 0 LED Default



Using don't care Default Values

You do not always want to assign a default value of don't care, although it can be beneficial in some cases, as the seven-segment decoder in Example 8-22 shows.

The reasons for not always defaulting to don't care are these:

- The potential for mismatches between simulation and synthesis is greater.
- Defaults for variables can hide mistakes in the VHDL code.

For example, if you assign a default don't care value to VAR and later assign a value to VAR, expecting VAR to be a don't care, you might have overlooked an intervening condition under which VAR is assigned.

Therefore, when you assign a value to a variable (or signal) containing a don't care value, make sure that the variable (or signal) is really a don't care type under those conditions.

Differences Between Simulation and Synthesis

Don't care types are treated differently in simulation than they are 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 1 or a 0. In synthesis, however, a don't care value becomes a 0 or a 1 (and the hardware built treats it as either a 0 or a 1).

Whenever a comparison is made to a variable whose value is don't care, simulation and synthesis can differ. The safest way to use don't care types is to

- Assign don't care values only to output ports
- Make sure the design never reads output ports

These guidelines guarantee that when you simulate in the scope of the design itself, the only difference between simulation and synthesis occurs when the simulator defines an output as a don't care.

Note:

If you use don't care values internally to a design, expressions compared with don't care ('D') are synthesized as though their values are not equal to 'D'.

For example,

```
if X = 'D' then  
...
```

is synthesized as

```
if FALSE then
```

If you use expressions comparing values with 'D', there might be a difference between pre- and post-synthesis simulation results. For this reason, FPGA Compiler II / FPGA *Express* issues a warning when it synthesizes such comparisons.

```
Warning: A partial don't-care value was read in routine test  
line 24 in file 'test.vhdl' This may cause simulation to  
disagree with synthesis. (HDL-171)
```

Synthesis Issues

Feedback paths and latches result from ambiguities in signal or variable assignments and language supersets, or the differences between a VHDL simulator view and the Synopsys use of VHDL.

Feedback Paths and Latches

Implied combinational feedback paths or latches in synthesized logic can occur when a signal or variable in a combinational process (one without a wait or if signal'event statement) is not fully specified in the VHDL description. A variable or signal is fully specified when it is assigned under all possible conditions. A variable or signal is not fully

specified when a condition exists under which the variable is not assigned.

Fully Specified Variables

Example 8-23 shows several variables. A, B, and C are fully specified; X is not.

Example 8-23 Fully Specified Variables

```
process (COND1)
  variable A, B, C, X : BIT;
begin
  A <= '0'      -- A is hereby fully specified
  C <= '0'      -- C is hereby fully specified

  if (COND1) then
    B <= '1';    -- B is assigned when COND1 is TRUE
    C <= '1';    -- C is already fully specified
    X <= '1';    -- X is assigned when COND1 is TRUE
  else
    B <= '0';    -- B is assigned when COND1 is FALSE
  end if;
  -- A is assigned regardless of COND1, so A is fully
  -- specified.

  -- B is assigned under all branches of if (COND1),
  -- that is, both when COND1 is TRUE and when
  -- COND1 is FALSE, so B is fully specified.

  -- C is assigned regardless of COND1, so C is fully
  -- specified. (The second assignment to C does
  -- not change this.)

  -- X is not assigned under all branches of
  -- if (COND1), namely, when COND1 is FALSE,
  -- so X is not fully specified.
end process;
```

The conditions of each if and else statement are considered independent in Example 8-23. A is considered not fully specified in the following fragment:

```
if (COND1) then
    A <= '1';
end if;

if (not COND1) then
    A <= '0';
end if;
```

A variable or signal that is not fully specified in a combinational process is considered conditionally specified. In this case a flow-through latch is implied. You can conditionally assign a variable, but you cannot read a conditionally specified variable. You can, however, both conditionally assign and read a signal.

If a fully specified variable is read before its assignment statements, combinational feedback might exist. For example, the following fragment synthesizes combinational feedback for VAL.

```
process(NEW, LOAD)
    variable VAL: BIT;
begin
    if (LOAD) then
        VAL <= NEW;
    else
        VAL <= VAL;
    end if;

    VAL_OUT <= VAL;
end process;
```

In a combinational process, you can ensure that a variable or signal is fully specified by providing an initial (default) assignment to the variable at the beginning of the process. This default assignment assures that the variable is always assigned a value, regardless of conditions. Subsequent assignment statements can override the default. A default assignment is made to variables A and C in Example 8-23.

Another way to ensure that you do not imply combinational feedback is to use a sequential process (one with a wait or if signal'event statement). In such a case, variables and signals are registered. The registers break the combinational feedback loop.

See Chapter 7, "Register and Three-State Inference", for more information about sequential processes and the conditions under which FPGA Compiler II / *FPGA Express* infers registers and latches.

Asynchronous Behavior

Some forms of asynchronous behavior are not supported. An example is a circuit description of a one-shot signal generator of the form

```
X <= A nand (not(not(not A)));
```

You might expect this circuit description to generate three inverters (an inverting delay line) and a NAND gate, but it is optimized to

```
X <= A nand (not A);
```

then

```
X <= 1;
```

Understanding Superset Issues and Error Checking

The Synopsys VHDL Analyzer is a full IEEE 1076 VHDL analyzer, described in the *VSS User Guide*.

When FPGA Compiler II / *FPGA Express* reads in a VHDL design, it first calls the Synopsys VHDL Analyzer to check the VHDL source for errors and then calls FPGA Compiler II / *FPGA Express* to translate the VHDL source to an intermediate form for synthesis. If an error is in the VHDL source, you get a VHDL Analyzer message and possibly a VHDL Compiler message.

VHDL Compiler allows globally static objects where only locally static objects are allowed, without issuing an error message. However, the Synopsys VSS Expert and VSS Professional tools detect and flag this error.

9

FPGA Compiler II / FPGA *Express* Directives

Synopsys has defined several methods of providing circuit design information directly in your VHDL source code.

Using FPGA Compiler II / FPGA *Express* directives, you can direct translation from VHDL to components with special VHDL comments. These synthetic comments turn translation on or off, specify one of several hard-wired resolution methods, and provide a means to map subprograms to hardware components.

To familiarize yourself with FPGA Compiler II / FPGA *Express* directives, consider the following topics presented in this chapter:

- Notation for FPGA Compiler II / FPGA Express Directives
- FPGA Compiler II / FPGA Express Directives

Notation for FPGA Compiler II / FPGA *Express* Directives

FPGA Compiler II / FPGA *Express* directives are special (synthetic) VHDL comments that affect the actions of FPGA Compiler II / FPGA *Express*. These comments are just a special case of regular VHDL comments, so they are ignored by other VHDL tools. Synthetic comments are used only to direct the actions of FPGA Compiler II / FPGA *Express*.

Synthetic comments begin just as regular comments do, with two hyphens (--). If the word following these characters is *pragma* or *synopsys*, FPGA Compiler II / FPGA *Express* treats the remaining comment text as a directive.

Note:

FPGA Compiler II / FPGA *Express* displays a syntax error if an unrecognized directive is encountered after -- *synopsys* or -- *pragma*.

FPGA Compiler II / FPGA *Express* Directives

The three types of directives are:

- Translation stop and start directives

```
-- pragma synthesis_off  
-- pragma synthesis_on
```

```
-- pragma translate_off      Use not recommended.  
-- pragma translate_on        Use not recommended.
```


- Resolution function directives

```
-- pragma resolution_method wired_and  
-- pragma resolution_method wired_or  
-- pragma resolution_method three_state
```

- Component implication directives

```
-- pragma map_to_entity entity_name  
-- pragma return_port_name port_name
```

Translation Stop and Start Pragma Directives

FPGA Compiler II / FPGA *Express* supports the `synthesis_off` and `synthesis_on` pragma directives.

Note:

It is recommended that you not use the following directives:

```
-- pragma translate_off  
-- pragma translate_on
```

The use of these directives in FPGA Compiler II / FPGA *Express* can lead to errors in your design.

`synthesis_off` and `synthesis_on` Directives

The `synthesis_off` and `synthesis_on` directives are the recommended mechanisms for hiding simulation-only constructs from synthesis. Any text between these directives is checked for syntax, but no corresponding hardware is synthesized.

Example 9-1 shows how you can use the directives to protect a simulation driver.

Example 9-1 Using synthesis_on and synthesis_off Directives

```
-- The following test driver for entity EXAMPLE
-- should not be translated:
-- pragma synthesis_off
-- Translation stops

entity DRIVER is
end DRIVER;
architecture VHDL of DRIVER is
    signal A, B : INTEGER range 0 to 255;
    signal SUM  : INTEGER range 0 to 511;

    component EXAMPLE
        port (A, B: in INTEGER range 0 to 255;
              SUM: out INTEGER range 0 to 511);
    end component;
begin
    U1: EXAMPLE port map(A, B, SUM);
    process
    begin
        for I in 0 to 255 loop
            for J in 0 to 255 loop
                A <= I;
                B <= J;
                wait for 10 ns;
                assert SUM = A + B;
            end loop;
        end loop;
    end process;
end VHDL;

-- pragma synthesis_on
-- Code from here on is translated

entity EXAMPLE is
    port (A, B: in INTEGER range 0 to 255;
          SUM: out INTEGER range 0 to 511);
end EXAMPLE;

architecture VHDL of EXAMPLE is
begin
    SUM <= A + B;
end VHDL;
```

Resolution Function Directives

Resolution function directives determine the resolution function associated with resolved signals (see “Resolution Functions” on page 2-40). FPGA Compiler II / FPGA *Express* does not support arbitrary resolution functions. It does support the following three methods:

```
-- pragma resolution_method wired_and
-- pragma resolution_method wired_or
-- pragma resolution_method three_state
```

Note:

Do not connect signals that use different resolution functions. FPGA Compiler II / FPGA *Express* supports only one resolution function per network.

Component Implication Directives

Component implication directives map VHDL subprograms onto existing components or VHDL entities. “Procedures and Functions as Design Components” on page 5-45 describes these directives:

```
-- pragma map_to_entity entity_name
-- pragma return_port_name port_name
```


A

Examples

This appendix presents examples that demonstrate basic concepts of Synopsys FPGA Compiler II / *FPGA Express*:

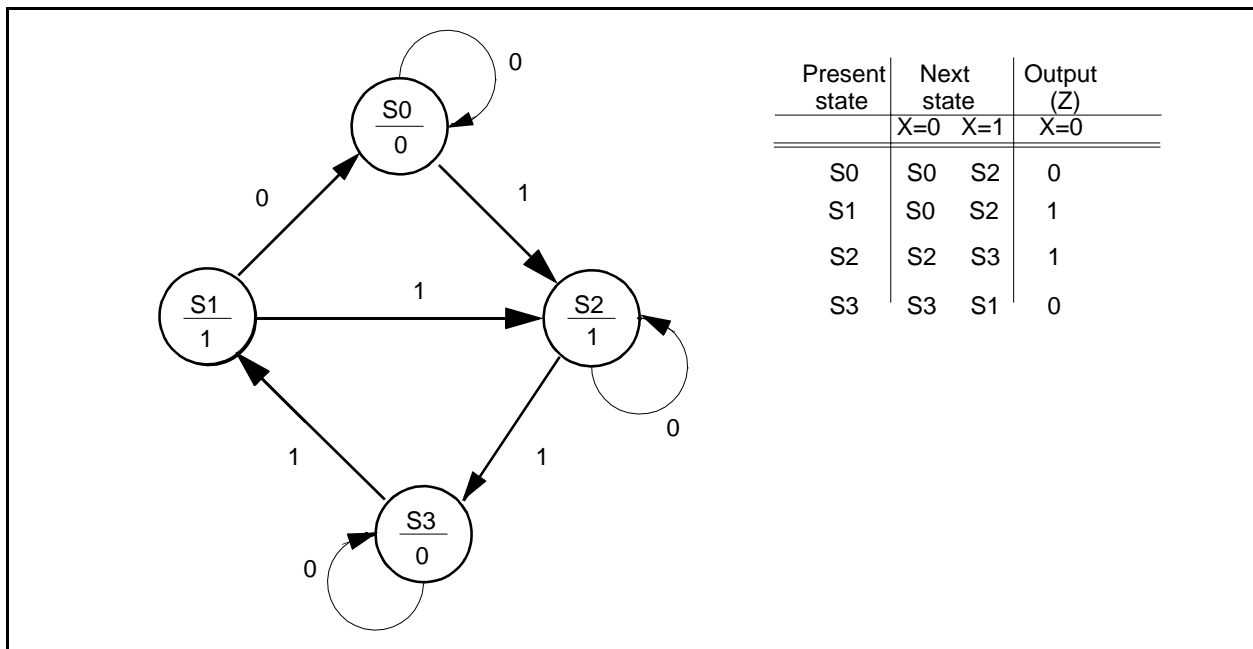
- Moore Machine
- Mealy Machine
- Read-Only Memory
- Waveform Generator
- Smart Waveform Generator
- Definable-Width Adder-Subtractor
- Count Zeros—Combinational Version
- Count Zeros—Sequential Version
- Soft Drink Machine—State Machine Version

- Soft Drink Machine—Count Nickels Version
- Carry-Lookahead Adder
- Serial-to-Parallel Converter—Counting Bits
- Serial-to-Parallel Converter—Shifting Bits
- Programmable Logic Arrays

Moore Machine

Figure A-1 is a diagram of a simple Moore finite state machine. It has one input (X), four internal states (S0 to S3), and one output (Z).

Figure A-1 Moore Machine Specification



The VHDL code implementing this finite state machine is shown in Example A-1, which includes a schematic of the synthesized circuit.

The machine description includes two processes. One process defines the synchronous elements of the design (state registers); the other process defines the combinational part of the design (state assignment case statement). For more details on using the two processes, see “Combinational Versus Sequential Processes” on page 5-55.

Example A-1 Implementation of a Moore Machine

```
entity MOORE is                                -- Moore machine
  port(X, CLOCK: in BIT;
        Z: out BIT);
end MOORE;

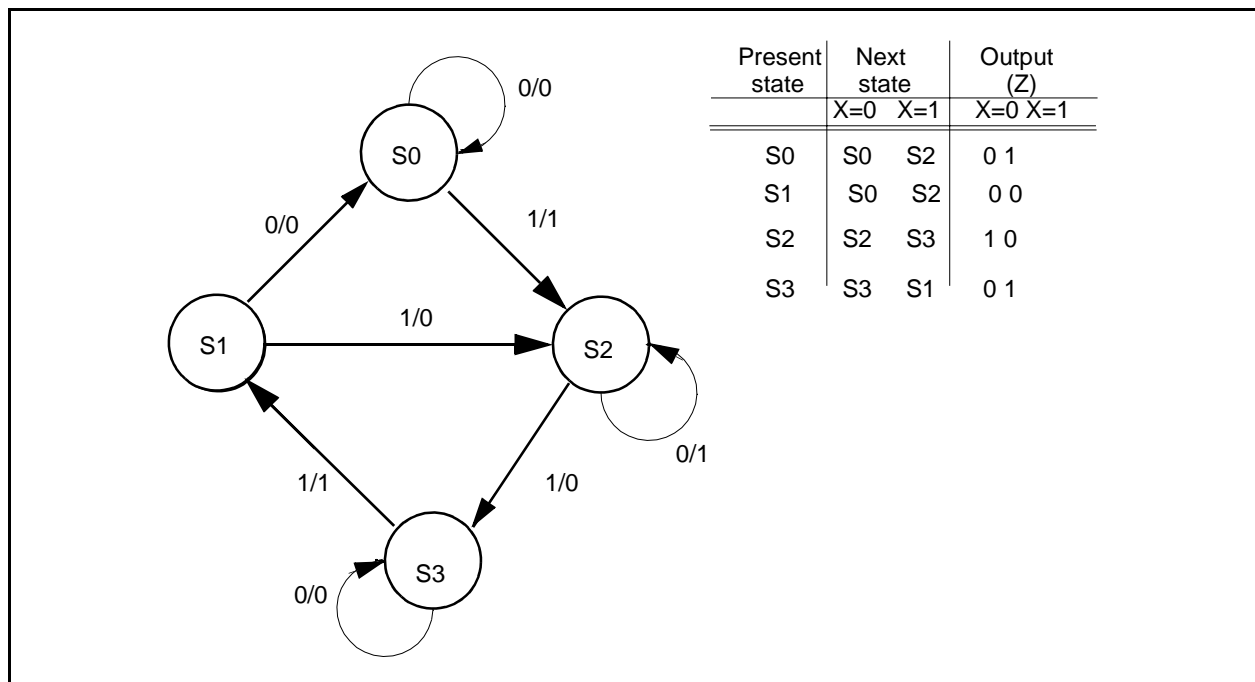
architecture BEHAVIOR of MOORE is
  type STATE_TYPE is (S0, S1, S2, S3);
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
begin

  -- Process to hold combinational logic
  COMBIN: process(CURRENT_STATE, X)
  begin
    case CURRENT_STATE is
      when S0 =>
        Z <= '0';
        if X = '0' then
          NEXT_STATE <= S0;
        else
          NEXT_STATE <= S2;
        end if;
      when S1 =>
        Z <= '1';
        if X = '0' then
          NEXT_STATE <= S0;
        else
          NEXT_STATE <= S2;
        end if;
      when S2 =>
        Z <= '1';
        if X = '0' then
```


Mealy Machine

Figure A-3 is a diagram of a simple Mealy finite state machine. The VHDL code for implementing this finite state machine is shown in Example A-2. The machine description includes two processes, as in the previous Moore machine example.

Figure A-3 Mealy Machine Specification



Example A-2 Implementation of a Mealy Machine

```
entity MEALY is                                -- Mealy machine
  port(X, CLOCK: in BIT;
        Z: out BIT);
end MEALY;

architecture BEHAVIOR of MEALY is
  type STATE_TYPE is (S0, S1, S2, S3);
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
```

```

begin

    -- Process to hold combinational logic.
    COMBIN: process(CURRENT_STATE, X)
    begin
        case CURRENT_STATE is
            when S0 =>
                if X = '0' then
                    Z <= '0';
                    NEXT_STATE <= S0;
                else
                    Z <= '1';
                    NEXT_STATE <= S2;
                end if;
            when S1 =>
                if X = '0' then
                    Z <= '0';
                    NEXT_STATE <= S0;
                else
                    Z <= '0';
                    NEXT_STATE <= S2;
                end if;
            when S2 =>
                if X = '0' then
                    Z <= '1';
                    NEXT_STATE <= S2;
                else
                    Z <= '0';
                    NEXT_STATE <= S3;
                end if;
            when S3 =>
                if X = '0' then
                    Z <= '0';
                    NEXT_STATE <= S3;
                else
                    Z <= '1';
                    NEXT_STATE <= S1;
                end if;
        end case;
    end process COMBIN;

    -- Process to hold synchronous elements (flip-flops)
    SYNCH: process

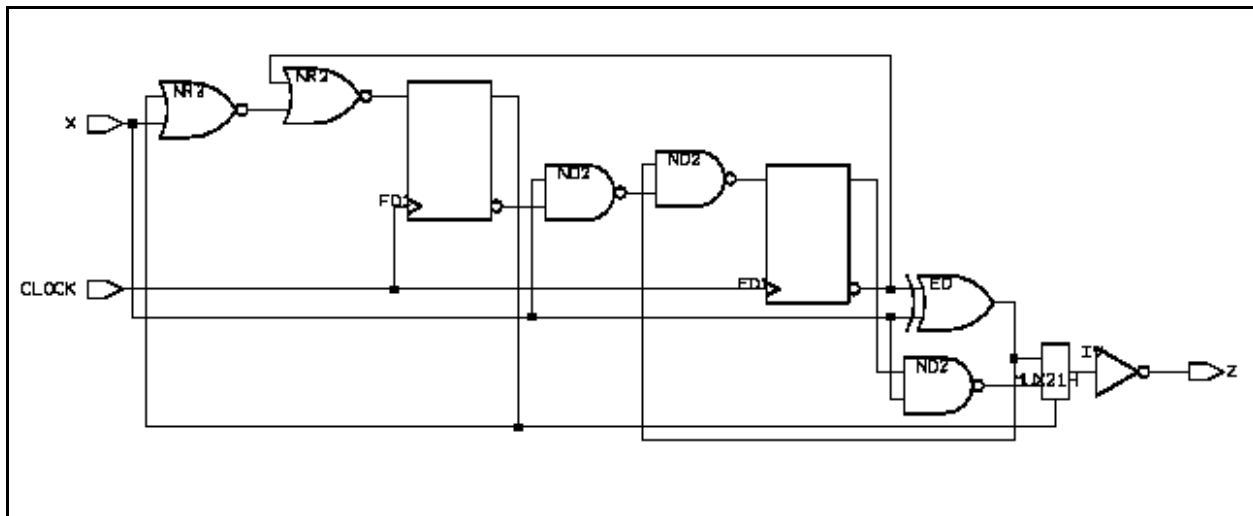
```

```

begin
    wait until CLOCK'event and CLOCK = '1';
    CURRENT_STATE <= NEXT_STATE;
end process SYNCH;
end BEHAVIOR;

```

Figure A-4 Mealy Machine Schematic



Read-Only Memory

Example A-3 shows how you can define a read-only memory in VHDL. The ROM is defined as an array constant, ROM. Each line of the constant array specification defines the contents of one ROM address. To read from the ROM, index into the array.

The number of ROM storage locations and bit-width is easy to change. The subtype ROM_RANGE specifies that the ROM contains storage locations 0 to 7. The constant ROM_WIDTH specifies that the ROM is 5 bits wide.

After you define a ROM constant, you can index into that constant many times to read many values from the ROM. If the ROM address is computable (see “Computable Operands” on page 4-16), no logic is built and the appropriate data value is inserted. If the ROM address is not computable, logic is built for each index into the value. In Example A-3, ADDR is not computable, so logic is synthesized to compute the value.

FPGA Compiler II / *FPGA Express* does not actually instantiate a typical array-logic ROM, such as those available from ASIC vendors. Instead, it creates the ROM from random logic gates (AND, OR, NOT, and so on). This type of implementation is preferable for small ROMs and for ROMs that are regular. For very large ROMs, consider using an array-logic implementation supplied by your ASIC vendor.

Example A-3 shows the VHDL source code and the synthesized circuit schematic.

Example A-3 Implementation of a ROM in Random Logic

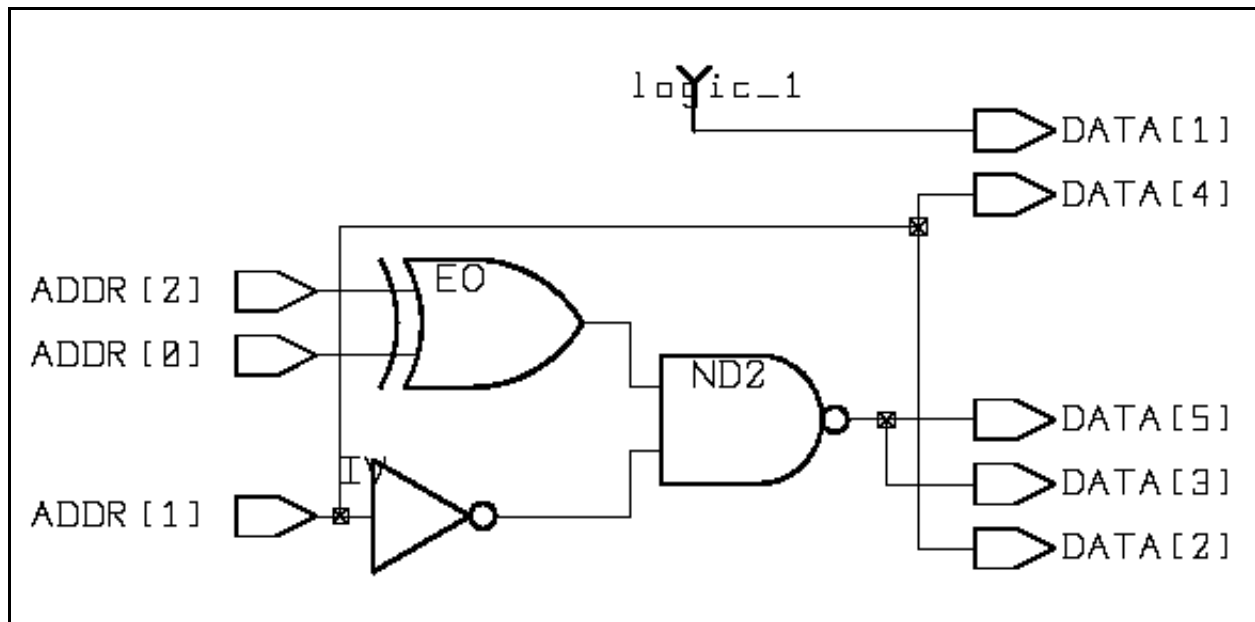
```
package ROMS is
  -- declare a 5x8 ROM called ROM
  constant ROM_WIDTH: INTEGER := 5;
  subtype ROM_WORD is BIT_VECTOR (1 to ROM_WIDTH);
  subtype ROM_RANGE is INTEGER range 0 to 7;
  type ROM_TABLE is array (0 to 7) of ROM_WORD;
  constant ROM: ROM_TABLE := ROM_TABLE'(
    ROM_WORD'("10101"),      -- ROM contents
    ROM_WORD'("10000"),
    ROM_WORD'("11111"),
    ROM_WORD'("11111"),
    ROM_WORD'("10000"),
    ROM_WORD'("10101"),
    ROM_WORD'("11111"),
    ROM_WORD'("11111"));
end ROMS;
use work.ROMS.all;      -- Entity that uses ROM
entity ROM_5x8 is
```

```

    port(ADDR: in ROM_RANGE;
          DATA: out ROM_WORD);
end ROM_5x8;
architecture BEHAVIOR of ROM_5x8 is
begin
    DATA <= ROM(ADDR);      -- Read from the ROM
end BEHAVIOR;

```

Figure A-5 ROM Schematic



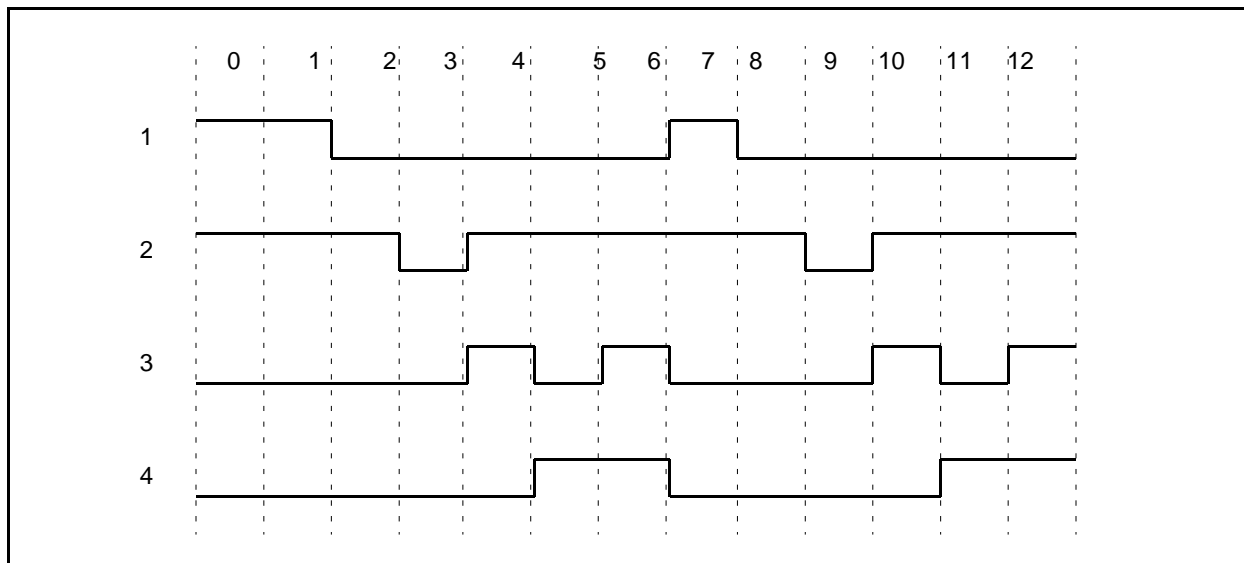
Waveform Generator

The waveform generator example shows how to use the previous ROM example to implement a waveform generator.

Assume that you want to produce the waveform output shown in Figure A-6.

1. First, declare a ROM wide enough to hold the output signals (4 bits) and deep enough to hold all time steps (0 to 12, for a total of 13).
2. Next, define the ROM so that each time step is represented by an entry in the ROM.
3. Finally, create a counter that cycles through the time steps (ROM addresses), generating the waveform at each time step.

Figure A-6 Waveform Example



Example A-4 shows an implementation for the waveform generator. It consists of a ROM, a counter, and some simple reset logic.

Example A-4 Implementation of a Waveform Generator

```
package ROMS is
  -- a 4x13 ROM called ROM that contains the waveform
  constant ROM_WIDTH: INTEGER := 4;
  subtype ROM_WORD is BIT_VECTOR (1 to ROM_WIDTH);
  subtype ROM_RANGE is INTEGER range 0 to 12;
  type ROM_TABLE is array (0 to 12) of ROM_WORD;
  constant ROM: ROM_TABLE := ROM_TABLE'(
    "1100", -- time step 0
    "1100", -- time step 1
    "0100", -- time step 2
    "0000", -- time step 3
    "0110", -- time step 4
    "0101", -- time step 5
    "0111", -- time step 6
    "1100", -- time step 7
    "0100", -- time step 8
    "0000", -- time step 9
    "0110", -- time step 10
    "0101", -- time step 11
    "0111"); -- time step 12
end ROMS;

use work.ROMS.all;
entity WAVEFORM is -- Waveform generator
  port(CLOCK: in BIT;
        RESET: in BOOLEAN;
        WAVES: out ROM_WORD);
end WAVEFORM;
```

```

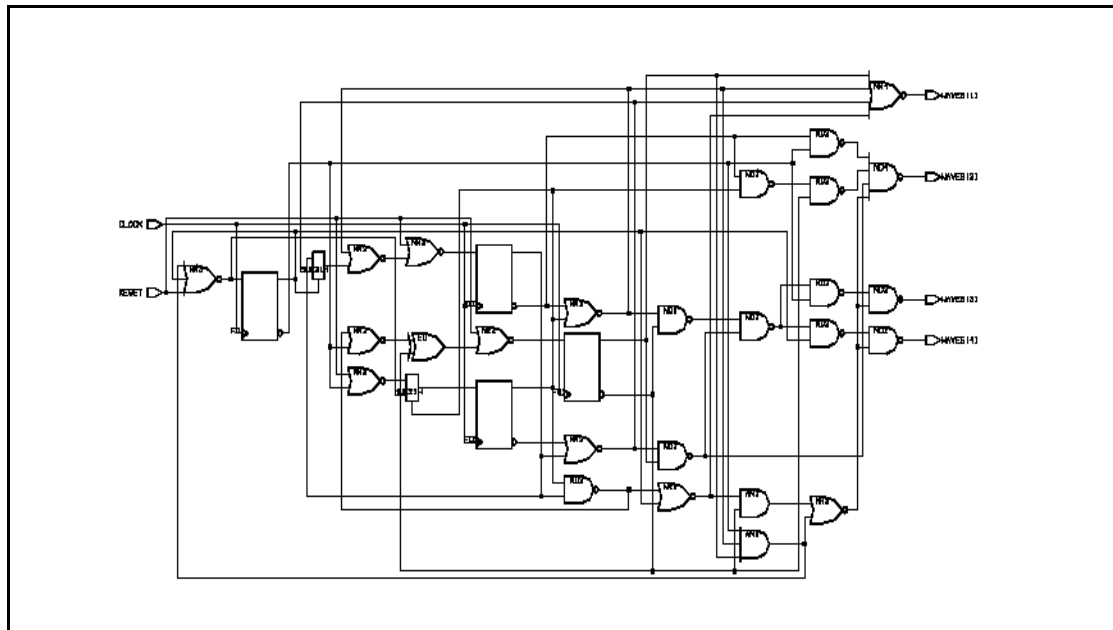
architecture BEHAVIOR of WAVEFORM is
    signal STEP: ROM_RANGE;
begin

    TIMESTEP_COUNTER: process    -- Time stepping process
    begin
        wait until CLOCK'event and CLOCK = '1';
        if RESET then            -- Detect reset
            STEP <= ROM_RANGE'low; -- Restart
        elsif STEP = ROM_RANGE'high then -- Finished?
            STEP <= ROM_RANGE'high; -- Hold at last value
        -- STEP <= ROM_RANGE'low; -- Continuous wave
        else
            STEP <= STEP + 1;      -- Continue stepping
        end if;
    end process TIMESTEP_COUNTER;

    WAVES <= ROM(STEP);
end BEHAVIOR;

```

Figure A-7 Waveform Generator Schematic



When the counter STEP reaches the end of the ROM, STEP stops, generates the last value, then waits until a reset. To make the sequence automatically repeat, remove the following statement:

```
STEP <= ROM_RANGE'high;  -- Hold at last value
```

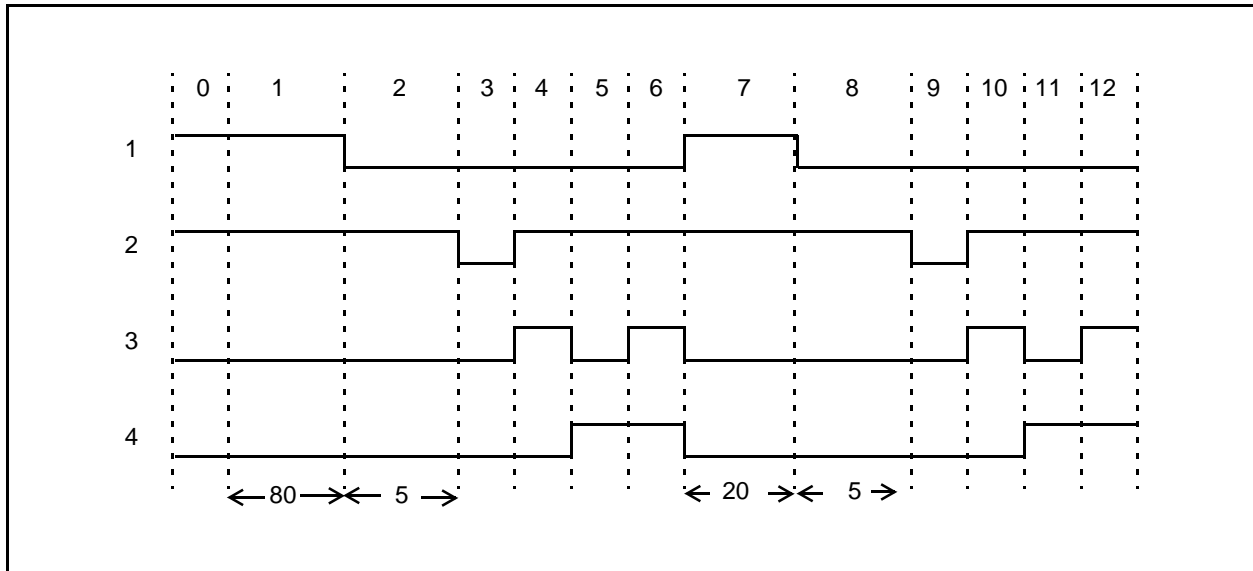
Use the following statement instead (commented out in Example A-4):

```
STEP <= ROM_RANGE'low;  -- Continuous wave
```

Smart Waveform Generator

The smart waveform generator in Figure A-8 is an extension of the waveform generator in Figure A-6 on page A-10. But this smart waveform generator is capable of holding the waveform at any time step for several clock cycles.

Figure A-8 Waveform for Smart Waveform Generator Example



The implementation of the smart waveform generator is shown in Example A-5. It is similar to the waveform generator in Example A-4 on page A-11, but has two additions. A new ROM, D_ROM, has been added to hold the length of each time step. A value of 1 specifies that the corresponding time step should be one clock cycle long; a value of 80 specifies that the time step should be 80 clock cycles long. The second addition to the previous waveform generator is a delay counter that counts the clock cycles between time steps.

In the architecture of this example, a selected signal assignment determines the value of the NEXT_STEP counter.

Example A-5 Implementation of a Smart Waveform Generator

```
package ROMS is

    -- a 4x13 ROM called W_ROM containing the waveform
    constant W_ROM_WIDTH: INTEGER := 4;
    subtype W_ROM_WORD is BIT_VECTOR (1 to W_ROM_WIDTH);
    subtype W_ROM_RANGE is INTEGER range 0 to 12;
    type W_ROM_TABLE is array (0 to 12) of W_ROM_WORD;
    constant W_ROM: W_ROM_TABLE := W_ROM_TABLE'(
        "1100",    -- time step 0
        "1100",    -- time step 1
        "0100",    -- time step 2
        "0000",    -- time step 3
        "0110",    -- time step 4
        "0101",    -- time step 5
        "0111",    -- time step 6
        "1100",    -- time step 7
        "0100",    -- time step 8
        "0000",    -- time step 9
        "0110",    -- time step 10
        "0101",    -- time step 11
        "0111");   -- time step 12

    -- a 7x13 ROM called D_ROM containing the delays
    subtype D_ROM_WORD is INTEGER range 0 to 100;
    subtype D_ROM_RANGE is INTEGER range 0 to 12;
```

```

    type D_ROM_TABLE is array (0 to 12) of D_ROM_WORD;
    constant D_ROM: D_ROM_TABLE := D_ROM_TABLE'(
        1,80,5,1,1,1,1,20,5,1,1,1,1);
end ROMS;

use work.ROMS.all;
entity WAVEFORM is          -- Smart Waveform Generator
    port(CLOCK: in BIT;
          RESET: in BOOLEAN;
          WAVES: out W_ROM_WORD);
end WAVEFORM;

architecture BEHAVIOR of WAVEFORM is
    signal STEP, NEXT_STEP: W_ROM_RANGE;
    signal DELAY: D_ROM_WORD;
begin

    -- Determine the value of the next time step
    NEXT_STEP <= W_ROM_RANGE'high when
        STEP = W_ROM_RANGE'high
    else
        STEP + 1;
    -- Keep track of which time step we are in
    TIMESTEP_COUNTER: process
    begin
        wait until CLOCK'event and CLOCK = '1';
        if RESET then          -- Detect reset
            STEP <= 0;          -- Restart waveform
        elsif DELAY = 1 then
            STEP <= NEXT_STEP;  -- Continue stepping
        else
            null;               -- Wait for DELAY to count down;
        end if;                -- do nothing here
    end process TIMESTEP_COUNTER;

    -- Count the delay between time steps
    DELAY_COUNTER: process
    begin
        wait until CLOCK'event and CLOCK = '1';
        if RESET then          -- Detect reset
            DELAY <= D_ROM(0);  -- Restart
        elsif DELAY = 1 then    -- Have we counted down?

```

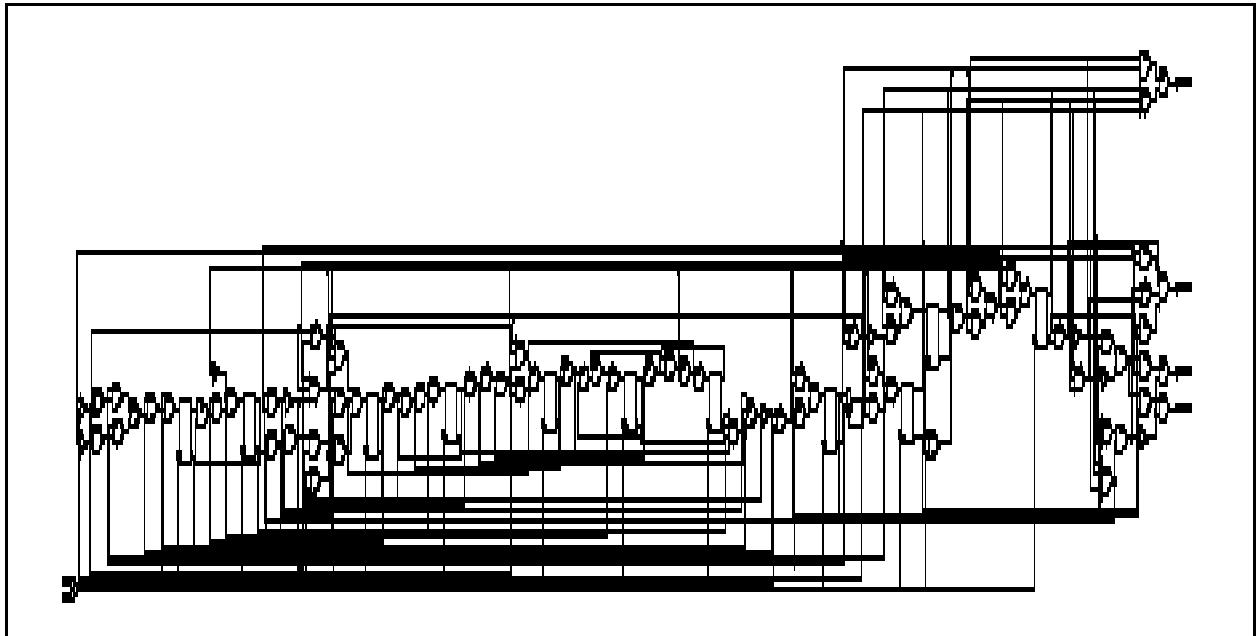
```

        DELAY <= D_ROM(NEXT_STEP);  -- Next delay value
    else
        DELAY <= DELAY - 1;  -- decrement DELAY counter
    end if;
end process DELAY_COUNTER;

WAVES <= W_ROM(STEP);  -- Output waveform value
end BEHAVIOR;

```

Figure A-9 Smart Waveform Generator Schematic



Definable-Width Adder-Subtractor

VHDL lets you create functions for use with array operands of any size. This example shows an adder-subtractor circuit that, when called, is adjusted to fit the size of its operands.

Example A-6 shows an adder-subtractor defined for two unconstrained arrays of bits (type BIT_VECTOR) in a package named MATH. When an unconstrained array type is used for an argument to a subprogram, the actual constraints of the array are taken from the actual parameter values in a subprogram call.

Example A-6 MATH Package for Example A-7

```
package MATH is
  function ADD_SUB(L, R: BIT_VECTOR; ADD: BOOLEAN)
    return BIT_VECTOR;
  -- Add or subtract two BIT_VECTORS of equal length
end MATH;

package body MATH is
  function ADD_SUB(L, R: BIT_VECTOR; ADD: BOOLEAN)
    return BIT_VECTOR is
    variable CARRY: BIT;
    variable A, B, SUM:
      BIT_VECTOR(L'length-1 downto 0);
  begin
    if ADD then
      -- Prepare for an "add" operation
      A := L;
      B := R;
      CARRY := '0';
    else
      -- Prepare for a "subtract" operation
      A := L;
      B := not R;
      CARRY := '1';
    end if;

    -- Create a ripple carry chain; sum up bits
    for i in 0 to A'left loop
      SUM(i) := A(i) xor B(i) xor CARRY;
      CARRY := (A(i) and B(i)) or
        (A(i) and CARRY) or
        (CARRY and B(i));
    end loop;
  end loop;
```

```

        return SUM;          -- Result
    end;
end MATH;

```

Within the function ADD_SUB, two temporary variables, A and B, are declared. These variables are declared to be the same length as L (and necessarily, R) but have their index constraints normalized to L'length-1 downto 0. After the arguments are normalized, you can create a ripple carry adder by using a for loop.

No explicit references to a fixed array length are in the function ADD_SUB. Instead, the VHDL array attributes 'left and 'length are used. These attributes allow the function to work on arrays of any length.

Example A-7 shows how to use the adder-subtractor defined in the MATH package. In this example, the vector arguments to functions ARG1 and ARG2 are declared as BIT_VECTOR(1 to 6). This declaration causes ADD_SUB to work with 6-bit arrays. A schematic of the synthesized circuit follows Example A-7.

Example A-7 Implementation of a 6-Bit Adder-Subtractor

```

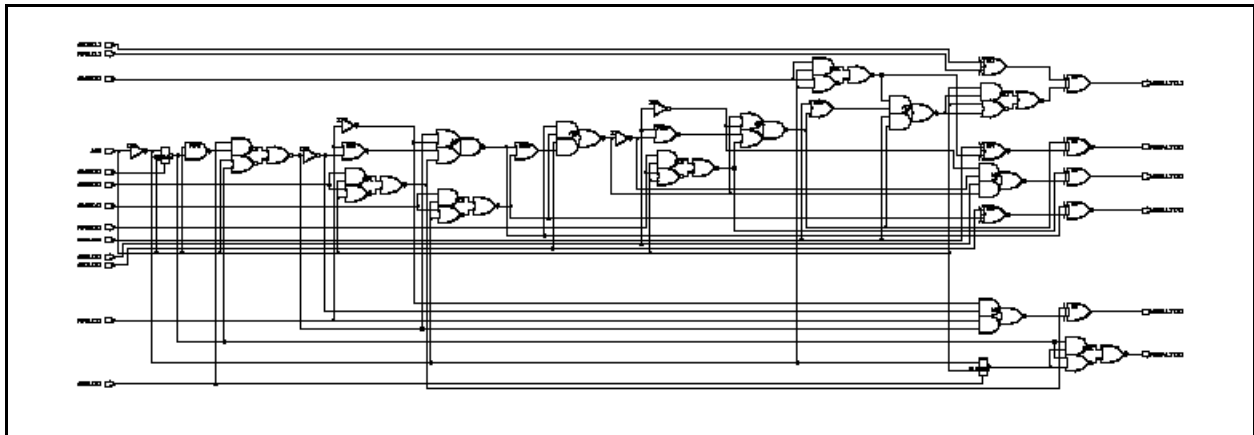
use work.MATH.all;

entity EXAMPLE is
    port(ARG1, ARG2: in BIT_VECTOR(1 to 6);
          ADD: in BOOLEAN;
          RESULT : out BIT_VECTOR(1 to 6));
end EXAMPLE;

architecture BEHAVIOR of EXAMPLE is
begin
    RESULT <= ADD_SUB(ARG1, ARG2, ADD);
end BEHAVIOR;

```

Figure A-10 6-Bit Adder-Subtractor Schematic



Count Zeros—Combinational Version

The count zeros—combinational example illustrates a design problem in which an 8-bit-wide value is given and the circuit determines two things:

- That no more than one sequence of zeros is in the value.
- The number of zeros in that sequence (if any). This computation must be completed in a single clock cycle.

The circuit produces two outputs: the number of zeros found and an error indication.

A valid input value can have at most one consecutive series of zeros. A value consisting entirely of ones is defined as a valid value. If a value is invalid, the zero counter resets to 0. For example, the value 00000000 is valid and has eight zeros; value 11000111 is valid and has three zeros; value 00111100 is invalid.

Example A-8 shows the VHDL description for the circuit. It consists of a single process with a for loop that iterates across each bit in the given value. At each iteration, a temporary INTEGER variable (TEMP_COUNT) counts the number of zeros encountered. Two temporary Boolean variables (SEEN_ZERO and SEEN_TRAILING), initially false, are set to true when the beginning and end of the first sequence of zeros is detected.

If a zero is detected after the end of the first sequence of zeros (after SEEN_TRAILING is true), the zero count is reset (to 0), ERROR is set to true, and the for loop is exited.

Example A-8 shows a combinational (parallel) approach to counting the zeros. The next example shows a sequential (serial) approach.

Example A-8 Count Zeros—Combinational

```
entity COUNT_COMB_VHDL is
  port(DATA: in BIT_VECTOR(7 downto 0);
        COUNT: out INTEGER range 0 to 8;
        ERROR: out BOOLEAN);
end COUNT_COMB_VHDL;

architecture BEHAVIOR of COUNT_COMB_VHDL is
begin
  process(DATA)
    variable TEMP_COUNT : INTEGER range 0 to 8;
    variable SEEN_ZERO, SEEN_TRAILING : BOOLEAN;
  begin
    ERROR <= FALSE;
    SEEN_ZERO <= FALSE;
    SEEN_TRAILING <= FALSE;
    TEMP_COUNT <= 0;
    for I in 0 to 7 loop
      if (SEEN_TRAILING and DATA(I) = '0') then
        TEMP_COUNT <= 0;
        ERROR <= TRUE;
        exit;
      elsif (SEEN_ZERO and DATA(I) = '1') then
```



```

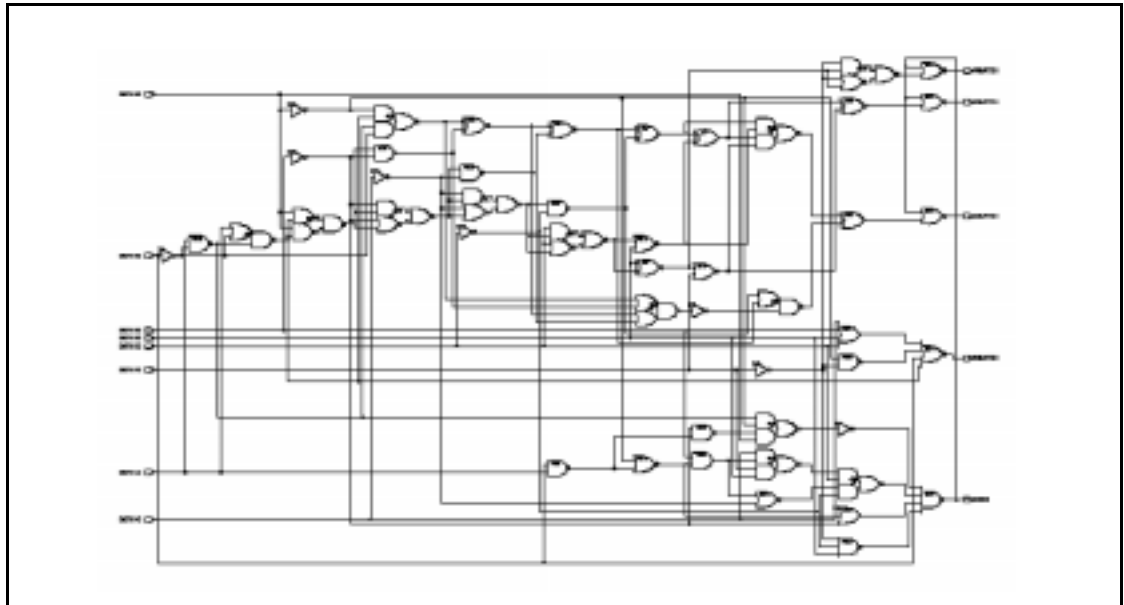
        SEEN_TRAILING <= TRUE;
    elsif (DATA(I) = '0') then
        SEEN_ZERO <= TRUE;
        TEMP_COUNT <= TEMP_COUNT + 1;
    end if;
end loop;

COUNT <= TEMP_COUNT;
end process;

end BEHAVIOR;

```

Figure A-11 Count Zeros—Combinational Schematic



Count Zeros—Sequential Version

The count zeros—sequential example shows a sequential (clocked) variant of the preceding design (Count Zeros—Combinational Version).

The circuit now accepts the 8-bit data value serially, 1 bit per clock cycle, by 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 bits

The circuit's three outputs are

- IS_LEGAL, which is true if the data was a valid value
- COUNT_READY, which is true at the first invalid bit or when all 8 bits have been processed
- COUNT, the number of zeros (if IS_LEGAL is true)

Note:

The output port COUNT is declared with mode BUFFER so that it can be read inside the process. OUT ports can only be written to, not read in.

Example A-9 Count Zeros—Sequential

```
entity COUNT_SEQ_VHDL is
  port(DATA, CLK: in BIT;
        RESET, READ: in BOOLEAN;
        COUNT: buffer INTEGER range 0 to 8;
        IS_LEGAL: out BOOLEAN;
        COUNT_READY: out BOOLEAN);
end COUNT_SEQ_VHDL;
architecture BEHAVIOR of COUNT_SEQ_VHDL is
```

```

begin
  process
    variable SEEN_ZERO, SEEN_TRAILING: BOOLEAN;
    variable BITS_SEEN: INTEGER range 0 to 7;
  begin
    wait until CLK'event and CLK = '1';

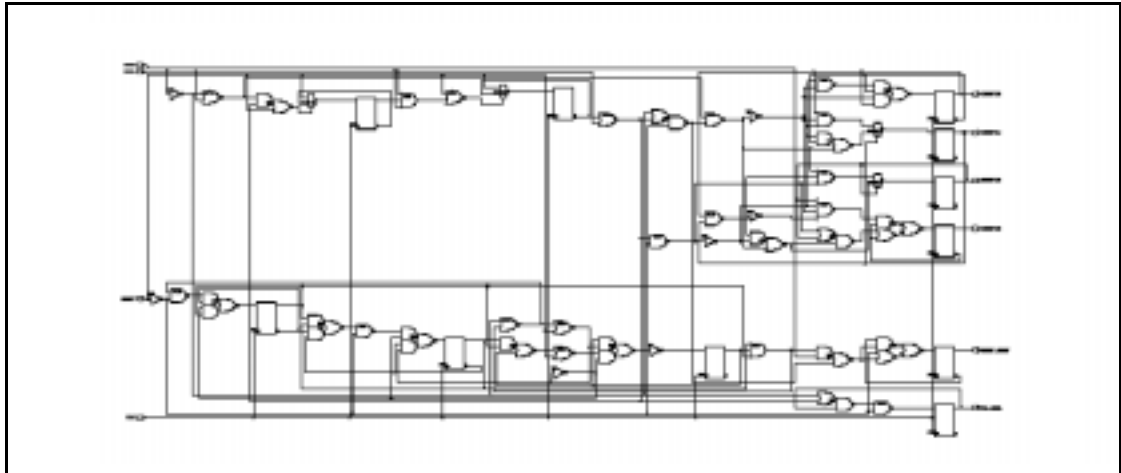
    if(RESET) then
      COUNT_READY <= FALSE;
      IS_LEGAL      <= TRUE;      -- signal assignment
      SEEN_ZERO     <= FALSE;    -- variable assignment
      SEEN_TRAILING <= FALSE;
      COUNT         <= 0;
      BITS_SEEN     <= 0;
    else
      if (READ) then
        if (SEEN_TRAILING and DATA = '0') then
          IS_LEGAL <= FALSE;
          COUNT <= 0;
          COUNT_READY <= TRUE;
        elsif (SEEN_ZERO and DATA = '1') then
          SEEN_TRAILING := TRUE;
        elsif (DATA = '0') then
          SEEN_ZERO <= TRUE;
          COUNT <= COUNT + 1;
        end if;

        if (BITS_SEEN = 7) then
          COUNT_READY <= TRUE;
        else
          BITS_SEEN <= BITS_SEEN + 1;
        end if;

      end if;      -- if (READ)
    end if;      -- if (RESET)
  end process;
end BEHAVIOR;

```

Figure A-12 Count Zeros—Sequential Schematic



Soft Drink Machine—State Machine Version

The soft drink machine—state machine example is a 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.

Here are the design parameters for Example A-10 and Example A-11:

- This example assumes that only one kind of soft drink is dispensed.
- This is a clocked design with CLK and RESET input signals.
- The price of the drink is 35 cents.
- The input signals from the coin input unit are NICKEL_IN (nickel deposited), DIME_IN (dime deposited), and QUARTER_IN (quarter deposited).

- The output signals to the change dispensing unit are NICKEL_OUT and DIME_OUT.
- The output signal to the drink dispensing unit is DISPENSE (dispense drink).
- The first VHDL description for this design uses a state machine description style. The second VHDL description is in Example A-11.

Example A-10 Soft Drink Machine—State Machine

```

library synopsys; use synopsys.attributes.all;

entity DRINK_STATE_VHDL is
  port(NICKEL_IN, DIME_IN, QUARTER_IN, RESET: BOOLEAN;
        CLK: BIT;
        NICKEL_OUT, DIME_OUT, DISPENSE: out BOOLEAN);
end DRINK_STATE_VHDL;

architecture BEHAVIOR of DRINK_STATE_VHDL is
  type STATE_TYPE is (IDLE, FIVE, TEN, FIFTEEN,
                      TWENTY, TWENTY_FIVE, THIRTY, OWE_DIME);
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
  attribute STATE_VECTOR : STRING;
  attribute STATE_VECTOR of BEHAVIOR : architecture is
    "CURRENT_STATE";

  attribute sync_set_reset of reset : signal is "true";
begin

  process(NICKEL_IN, DIME_IN, QUARTER_IN,
          CURRENT_STATE, RESET, CLK)
  begin
    -- Default assignments
    NEXT_STATE <= CURRENT_STATE;
    NICKEL_OUT <= FALSE;
    DIME_OUT <= FALSE;
    DISPENSE <= FALSE;

    -- Synchronous reset

```

```

if(RESET) then
    NEXT_STATE <= IDLE;
else

    -- State transitions and output logic
    case CURRENT_STATE is
        when IDLE =>
            if(NICKEL_IN) then
                NEXT_STATE <= FIVE;
            elsif(DIME_IN) then
                NEXT_STATE <= TEN;
            elsif(QUARTER_IN) then
                NEXT_STATE <= TWENTY_FIVE;
            end if;

        when FIVE =>
            if(NICKEL_IN) then
                NEXT_STATE <= TEN;
            elsif(DIME_IN) then
                NEXT_STATE <= FIFTEEN;
            elsif(QUARTER_IN) then
                NEXT_STATE <= THIRTY;
            end if;

        when TEN =>
            if(NICKEL_IN) then
                NEXT_STATE <= FIFTEEN;
            elsif(DIME_IN) then
                NEXT_STATE <= TWENTY;
            elsif(QUARTER_IN) then
                NEXT_STATE <= IDLE;
                DISPENSE <= TRUE;
            end if;

        when FIFTEEN =>
            if(NICKEL_IN) then
                NEXT_STATE <= TWENTY;
            elsif(DIME_IN) then
                NEXT_STATE <= TWENTY_FIVE;
            elsif(QUARTER_IN) then
                NEXT_STATE <= IDLE;
                DISPENSE <= TRUE;
                NICKEL_OUT <= TRUE;
            end if;
    end case;
end if;

```

```

when TWENTY =>
    if(NICKEL_IN) then
        NEXT_STATE <= TWENTY_FIVE;
    elsif(DIME_IN) then
        NEXT_STATE <= THIRTY;
    elsif(QUARTER_IN) then
        NEXT_STATE <= IDLE;
        DISPENSE <= TRUE;
        DIME_OUT <= TRUE;
    end if;

when TWENTY_FIVE =>
    if(NICKEL_IN) then
        NEXT_STATE <= THIRTY;
    elsif(DIME_IN) then
        NEXT_STATE <= IDLE;
        DISPENSE <= TRUE;
    elsif(QUARTER_IN) then
        NEXT_STATE <= IDLE;
        DISPENSE <= TRUE;
        DIME_OUT <= TRUE;
        NICKEL_OUT <= TRUE;
    end if;

when THIRTY =>
    if(NICKEL_IN) then
        NEXT_STATE <= IDLE;
        DISPENSE <= TRUE;
    elsif(DIME_IN) then
        NEXT_STATE <= IDLE;
        DISPENSE <= TRUE;
        NICKEL_OUT <= TRUE;
    elsif(QUARTER_IN) then
        NEXT_STATE <= OWE_DIME;
        DISPENSE <= TRUE;
        DIME_OUT <= TRUE;
    end if;

when OWE_DIME =>
    NEXT_STATE <= IDLE;
    DIME_OUT <= TRUE;

```

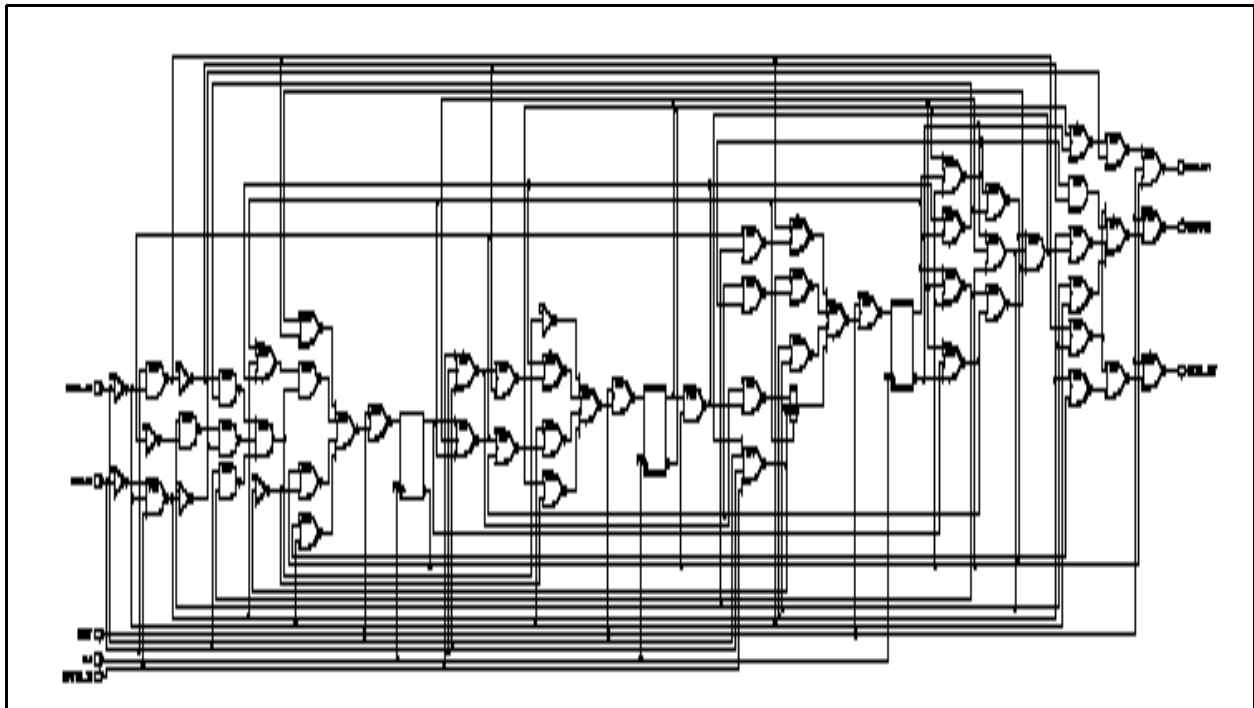
```

        end case;
    end if;
end process;

-- Synchronize state value with clock
-- This causes it to be stored in flip-flops
process
begin
    wait until CLK'event and CLK = '1';
    CURRENT_STATE <= NEXT_STATE;
end process;
end BEHAVIOR;

```

Figure A-13 Soft Drink Machine—State Machine Schematic



Soft Drink Machine—Count Nickels Version

The soft drink machine—count nickels example uses the same design parameters as the preceding Example A-10 (Soft Drink Machine—State Machine), with the same input and output signals. In this version, a counter counts the number of nickels deposited. This counter is incremented by 1 if the deposit is a nickel, by 2 if it is a dime, and by 5 if it is a quarter.

Example A-11 Soft Drink Machine—Count Nickels

```
entity DRINK_COUNT_VHDL is
  port(NICKEL_IN, DIME_IN, QUARTER_IN, RESET: BOOLEAN;
        CLK: BIT;
        NICKEL_OUT, DIME_OUT, DISPENSE: out BOOLEAN);
end DRINK_COUNT_VHDL;

architecture BEHAVIOR of DRINK_COUNT_VHDL is
  signal CURRENT_NICKEL_COUNT,
        NEXT_NICKEL_COUNT: INTEGER range 0 to 7;
  signal CURRENT_RETURN_CHANGE, NEXT_RETURN_CHANGE : BOOLEAN;
begin

  process(NICKEL_IN, DIME_IN, QUARTER_IN, RESET, CLK,
        CURRENT_NICKEL_COUNT, CURRENT_RETURN_CHANGE)
    variable TEMP_NICKEL_COUNT: INTEGER range 0 to 12;
  begin
    -- Default assignments
    NICKEL_OUT <= FALSE;
    DIME_OUT <= FALSE;
    DISPENSE <= FALSE;
    NEXT_NICKEL_COUNT <= 0;
    NEXT_RETURN_CHANGE <= FALSE;

    -- Synchronous reset
    if (not RESET) then
      TEMP_NICKEL_COUNT <= CURRENT_NICKEL_COUNT;

      -- Check whether money has come in
      if (NICKEL_IN) then
        -- NOTE: This design will be flattened, so
        --       these multiple adders will be optimized
        TEMP_NICKEL_COUNT <= TEMP_NICKEL_COUNT + 1;
```

```

elseif(DIME_IN) then
    TEMP_NICKEL_COUNT <= TEMP_NICKEL_COUNT + 2;
elseif(QUARTER_IN) then
    TEMP_NICKEL_COUNT <= TEMP_NICKEL_COUNT + 5;
end if;

-- Enough deposited so far?
if(TEMP_NICKEL_COUNT >= 7) then
    TEMP_NICKEL_COUNT <= TEMP_NICKEL_COUNT - 7;
    DISPENSE <= TRUE;
end if;

-- Return change
if(TEMP_NICKEL_COUNT >= 1 or
    CURRENT_RETURN_CHANGE) then
    if(TEMP_NICKEL_COUNT >= 2) then
        DIME_OUT <= TRUE;
        TEMP_NICKEL_COUNT <= TEMP_NICKEL_COUNT - 2;
        NEXT_RETURN_CHANGE <= TRUE;
    end if;
    if(TEMP_NICKEL_COUNT = 1) then
        NICKEL_OUT <= TRUE;
        TEMP_NICKEL_COUNT <= TEMP_NICKEL_COUNT - 1;
    end if;
end if;

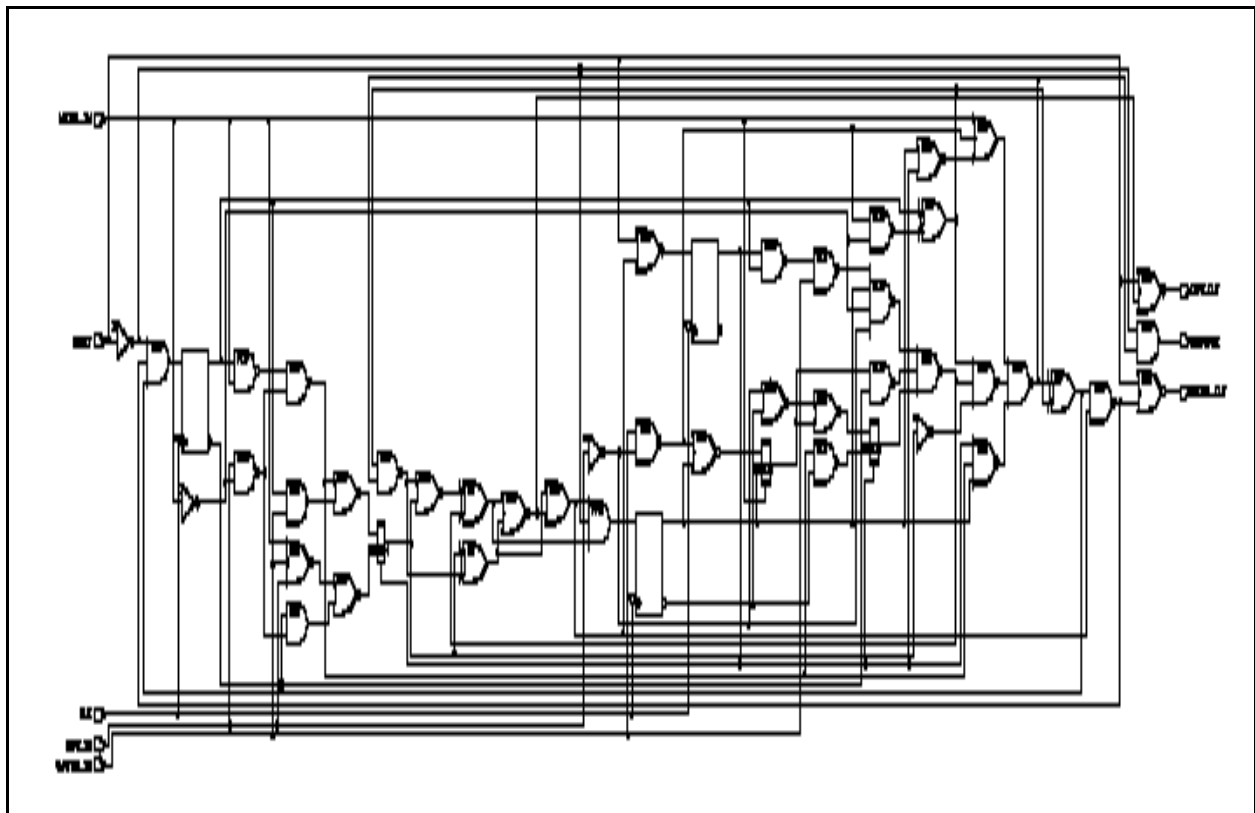
    NEXT_NICKEL_COUNT <= TEMP_NICKEL_COUNT;
end if;
end process;

-- Remember the return-change flag and
-- the nickel count for the next cycle
process
begin
    wait until CLK'event and CLK = '1';
    CURRENT_RETURN_CHANGE <= NEXT_RETURN_CHANGE;
    CURRENT_NICKEL_COUNT <= NEXT_NICKEL_COUNT;
end process;

end BEHAVIOR;

```

Figure A-14 Soft Drink Machine—Count Nickels Version Schematic



Carry-Lookahead Adder

This example uses concurrent procedure calls 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. Each of the eight slices computes propagate and generate values by using the PG procedure.

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. The logic computes the carry value for each bit position and makes the addition operation an XOR of the inputs and the carry values.

Carry Value Computations

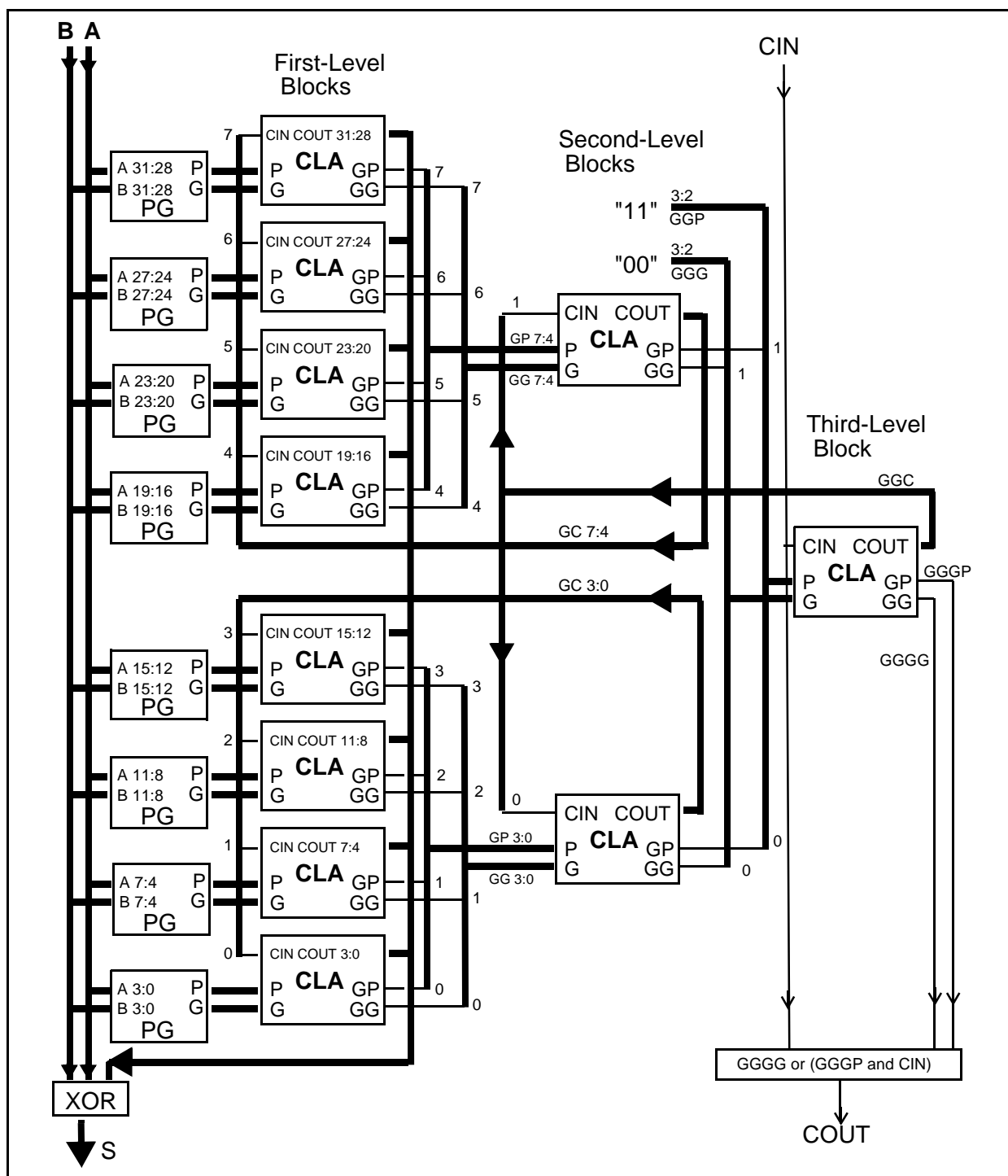
The carry values are computed by a three-level tree of 4-bit carry-lookahead blocks.

- The first level of the tree computes the 32 carry values and the eight 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 group. The first-level lookahead blocks read the group carry computed at the second level.

- The second-level 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. The second-level lookahead blocks also read group carry information computed at the third level to compute the carries for each of the third-level blocks.
- 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 is capable of processing four second-level blocks. But because there are only two second-level blocks, the high-order 2 bits of the computed carry are ignored; the high-order two bits of the generate input to the third-level are set to zero, "00"; and the propagate high-order bits are set to "11". These settings cause the unused portion to propagate carries but not to generate them. Figure A-15 shows the overall structure for the carry-lookahead adder.

Figure A-15 Carry-Lookahead Adder Block Diagram



Examples

The VHDL implementation of the design in Figure A-15 is accomplished with four procedures:

CLA

Names a 4-bit carry-lookahead block.

PG

Computes first-level propagate and generate information.

SUM

Computes the sum by adding the XOR values to the inputs with the carry values computed by CLA.

BITSLICE

Collects the first-level CLA blocks, the PG computations, and the SUM. This procedure performs all the work for a 4-bit value except for the second- and third-level lookaheads.

Example A-12 shows a VHDL description of the adder.

Example A-12 Carry-Lookahead Adder

```
package LOCAL is
  constant N:    INTEGER := 4;

  procedure BITSlice(
    A, B: in BIT_VECTOR(3 downto 0);
    CIN: in BIT;
    signal S: out BIT_VECTOR(3 downto 0);
    signal GP, GG: out BIT);
  procedure PG(
    A, B: in BIT_VECTOR(3 downto 0);
    P, G: out BIT_VECTOR(3 downto 0));
  function SUM(A, B, C: BIT_VECTOR(3 downto 0))
    return BIT_VECTOR;
  procedure CLA(
    P, G: in BIT_VECTOR(3 downto 0);
    CIN: in BIT;
    C: out BIT_VECTOR(3 downto 0);
    signal GP, GG: out BIT);
end LOCAL;
```

```

package body LOCAL is
-----
-- Compute sum and group outputs from a, b, cin
-----

procedure BITSlice(
    A, B: in BIT_VECTOR(3 downto 0);
    CIN: in BIT;
    signal S: out BIT_VECTOR(3 downto 0);
    signal GP, GG: out BIT) is

    variable P, G, C: BIT_VECTOR(3 downto 0);
begin
    PG(A, B, P, G);
    CLA(P, G, CIN, C, GP, GG);
    S <= SUM(A, B, C);
end;

-----
-- Compute propagate and generate from input bits
-----

procedure PG(A, B: in BIT_VECTOR(3 downto 0);
    P, G: out BIT_VECTOR(3 downto 0)) is

begin
    P <= A or B;
    G <= A and B;
end;

-----
-- Compute sum from the input bits and the carries
-----

function SUM(A, B, C: BIT_VECTOR(3 downto 0))
    return BIT_VECTOR is

begin
    return(A xor B xor C);
end;

-----
-- 4-bit carry-lookahead block
-----

procedure CLA(
    P, G: in BIT_VECTOR(3 downto 0);
    CIN: in BIT;
    C: out BIT_VECTOR(3 downto 0);

```



```

        signal GP, GG: out BIT) is
        variable TEMP_GP, TEMP_GG, LAST_C: BIT;
begin
    TEMP_GP <= P(0);
    TEMP_GG <= G(0);
    LAST_C <= CIN;
    C(0) <= CIN;

    for I in 1 to N-1 loop
        TEMP_GP <= TEMP_GP and P(I);
        TEMP_GG <= (TEMP_GG and P(I)) or G(I);
        LAST_C <= (LAST_C and P(I-1)) or G(I-1);
        C(I) <= LAST_C;
    end loop;

    GP <= TEMP_GP;
    GG <= TEMP_GG;
end;
end LOCAL;

use WORK.LOCAL.ALL;

-----
-- A 32-bit carry-lookahead adder
-----

entity ADDER is
    port(A, B: in BIT_VECTOR(31 downto 0);
          CIN: in BIT;
          S: out BIT_VECTOR(31 downto 0);
          COUT: out BIT);
end ADDER;
architecture BEHAVIOR of ADDER is

    signal GG,GP,GC: BIT_VECTOR(7 downto 0);
        -- First-level generate, propagate, carry
    signal GGG, GGP, GGC: BIT_VECTOR(3 downto 0);
        -- Second-level gen, prop, carry
    signal GGGG, GGGP: BIT;
        -- Third-level gen, prop

begin
    -- Compute Sum and 1st-level Generate and Propagate
    -- Use input data and the 1st-level Carries computed
    -- later.
    BITSlice(A( 3 downto 0),B( 3 downto 0),GC(0),
              S( 3 downto 0),GP(0), GG(0));
    BITSlice(A( 7 downto 4),B( 7 downto 4),GC(1),
              S( 7 downto 4),GP(1), GG(1));

```

```

BITSlice(A(11 downto 8),B(11 downto 8),GC(2),
         S(11 downto 8),GP(2), GG(2));
BITSlice(A(15 downto 12),B(15 downto 12),GC(3),
         S(15 downto 12),GP(3), GG(3));
BITSlice(A(19 downto 16),B(19 downto 16),GC(4),
         S(19 downto 16),GP(4), GG(4));
BITSlice(A(23 downto 20),B(23 downto 20),GC(5),
         S(23 downto 20),GP(5), GG(5));
BITSlice(A(27 downto 24),B(27 downto 24),GC(6),
         S(27 downto 24),GP(6), GG(6));
BITSlice(A(31 downto 28),B(31 downto 28),GC(7),
         S(31 downto 28),GP(7), GG(7));

-- Compute first-level Carries and second-level
-- generate and propagate.
-- Use first-level Generate, Propagate, and
-- second-level carry.
process(GP, GG, GGC)
    variable TEMP: BIT_VECTOR(3 downto 0);
begin
    CLA(GP(3 downto 0), GG(3 downto 0), GGC(0), TEMP,
        GGP(0), GGG(0));
    GC(3 downto 0) <= TEMP;
end process;

process(GP, GG, GGC)
    variable TEMP: BIT_VECTOR(3 downto 0);
begin
    CLA(GP(7 downto 4), GG(7 downto 4), GGC(1), TEMP,
        GGP(1), GGG(1));
    GC(7 downto 4) <= TEMP;
end process;

-- Compute second-level Carry and third-level
-- Generate and Propagate
-- Use second-level Generate, Propagate and Carry-in
-- (CIN)
process(GGP, GGG, CIN)
    variable TEMP: BIT_VECTOR(3 downto 0);
begin
    CLA(GGP, GGG, CIN, TEMP, GGGP, GGGG);
    GGC <= TEMP;
end process;

-- Assign unused bits of second-level Generate and
-- Propagate
GGP(3 downto 2) <= "11";
GGG(3 downto 2) <= "00";

```

```
-- Compute Carry-out (COUT)
-- Use third-level Generate and Propagate and
--   Carry-in (CIN).
COUT <= GGGG or (GGGP and CIN);
end BEHAVIOR;
```

Implementation

In the carry-lookahead adder implementation, procedures perform the computation of the design. The procedures can also be in the form of separate entities and used by component instantiation, producing a hierarchical design. FPGA Compiler II / *FPGA Express* does not collapse a hierarchy of entities, but it does collapse the procedure call hierarchy into one design.

The keyword `signal` is included before some of the interface parameter declarations. This keyword is required for the out formal parameters when the actual parameters must be signals.

The output parameter `C` from the CLA procedure is not declared as a signal; thus, it is not allowed in a concurrent procedure call. Only signals can be used in such calls. To overcome this problem, subprocesses are used, declaring a temporary variable `TEMP`. `TEMP` receives the value of the `C` parameter and assigns it to the appropriate signal (a generally useful technique).

Serial-to-Parallel Converter—Counting Bits

This example shows the design of a serial-to-parallel converter that reads a serial, bit-stream input and produces an 8-bit output.

The design reads the following inputs:

SERIAL_IN

The serial input data.

RESET

The input that, when it is '1', causes the converter to reset. All outputs are set to 0, and the converter is prepared to read the next serial word.

CLOCK

The value of RESET and SERIAL_IN, which is read on the positive transition of this clock. Outputs of the converter are also valid only on positive transitions.

The design produces the following outputs:

PARALLEL_OUT

The 8-bit value read from the SERIAL_IN port.

READ_ENABLE

The output that, when it is '1' on the positive transition of CLOCK, causes the data on PARALLEL_OUT to be read.

PARITY_ERROR

The output that, when it is '1' on the positive transition of CLOCK, indicates that a parity error has been detected on the SERIAL_IN port. When a parity error is detected, the converter halts until restarted by the RESET port.

Input Format

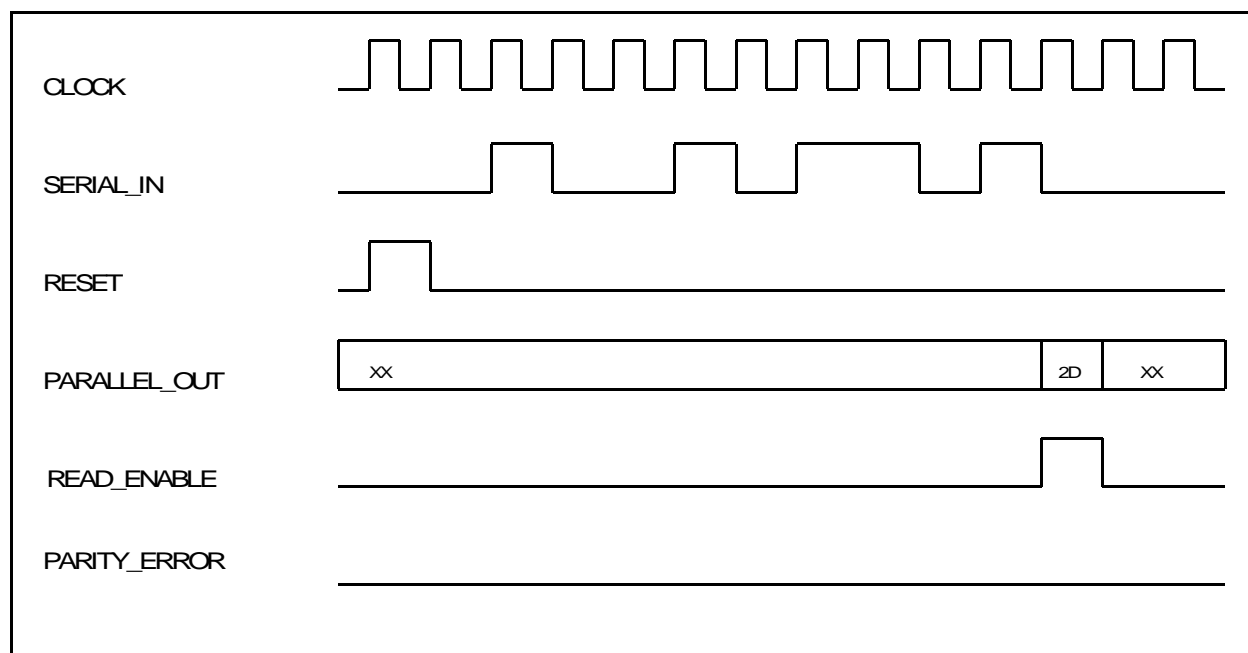
When no data is being transmitted to the serial port, keep it at a value of '0'. Each 8-bit value requires ten clock cycles to read it. On the eleventh clock cycle, the parallel output value can be read.

In the first cycle, a '1' is placed on the serial input. This assignment indicates that an 8-bit value follows. The next eight cycles transmit each bit of the value. The most significant bit is transmitted first. The tenth cycle transmits the parity of the 8-bit value. It must be '0' if an even number of '1' values are in the 8-bit data, and '1' otherwise. If the converter detects a parity error, it sets the PARITY_ERROR output to '1' and waits until the value is reset.

On the eleventh cycle, the READ_ENABLE output is set to '1' and the 8-bit value can be read from the PARALLEL_OUT port. If the SERIAL_IN port has a '1' on the eleventh cycle, another 8-bit value is read immediately; otherwise, the converter waits until SERIAL_IN goes to '1'.

Figure A-16 shows the timing of this design.

Figure A-16 Sample Waveform Through the Converter



Implementation Details

The implementation of the converter is as a four-state finite-state machine with synchronous reset. When a reset is detected, the converter enters a WAIT_FOR_START state. The description of each state follows

WAIT_FOR_START

Stay in this state until a '1' is detected on the serial input. When a '1' is detected, clear the PARALLEL_OUT registers and go to the READ_BITS state.

READ_BITS

If the value of the current_bit_position counter is 8, all 8 bits have been read. Check the computed parity with the transmitted parity. If it is correct, go to the ALLOW_READ state; otherwise, go to the PARITY_ERROR state.

If all 8 bits have not yet been read, set the appropriate bit in the PARALLEL_OUT buffer to the SERIAL_IN value, compute the parity of the bits read so far, and increment the current_bit_position.

ALLOW_READ

This is the state where the outside world reads the PARALLEL_OUT value. When that value is read, the design returns to the WAIT_FOR_START state.

PARITY_ERROR_DETECTED

In this state, the PARITY_ERROR output is set to '1' and nothing else is done.

This design has four values stored in registers:

CURRENT_STATE

Remembers the state as of the last clock edge.

CURRENT_BIT_POSITION

Remembers how many bits have been read so far.

CURRENT_PARITY

Keeps a running XOR of the bits read.

CURRENT_PARALLEL_OUT

Stores each parallel bit as it is found.

The design has two processes: the combinational NEXT_ST containing the combinational logic and the sequential SYNCH that is clocked.

NEXT_ST performs all the computations and state assignments. The NEXT_ST process starts by assigning default values to all the signals it drives. This assignment guarantees that all signals are driven under all conditions. Next, the RESET input is processed. If RESET is not active, a case statement determines the current state and its computations. State transitions are performed by assigning the next state's value you want to the NEXT_STATE signal.

The serial-to-parallel conversion itself is performed by these two statements in the NEXT_ST process:

```
NEXT_PARALLEL_OUT(CURRENT_BIT_POSITION) <= SERIAL_IN;  
NEXT_BIT_POSITION <= CURRENT_BIT_POSITION + 1;
```

The first statement assigns the current serial input bit to a particular bit of the parallel output. The second statement increments the next bit position to be assigned.

SYNCH registers and updates the stored values previously described. Each registered signal has two parts, NEXT_... and CURRENT_... :

NEXT_...

Signals hold values computed by the NEXT_ST process.

CURRENT_...

Signals hold the values driven by the SYNCH process. The CURRENT_... signals hold the values of the NEXT_... signals as of the last clock edge.

Example A-13 shows a VHDL description of the converter.

Example A-13 *Serial-to-Parallel Converter—Counting Bits*

```
-- Serial-to-Parallel Converter, counting bits

package TYPES is
    -- Declares types used in the rest of the design
    type STATE_TYPE is (WAIT_FOR_START,
                        READ_BITS,
                        PARITY_ERROR_DETECTED,
                        ALLOW_READ);
    constant PARALLEL_BIT_COUNT: INTEGER := 8;
    subtype PARALLEL_RANGE is INTEGER
        range 0 to (PARALLEL_BIT_COUNT-1);
    subtype PARALLEL_TYPE is BIT_VECTOR(PARALLEL_RANGE);
end TYPES;

use WORK.TYPES.ALL;      -- Use the TYPES package

entity SER_PAR is        -- Declare the interface
    port(SERIAL_IN, CLOCK, RESET: in BIT;
          PARALLEL_OUT: out PARALLEL_TYPE;
          PARITY_ERROR, READ_ENABLE: out BIT);
end SER_PAR;

architecture BEHAVIOR of SER_PAR is
    -- Signals for stored values
    signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
    signal CURRENT_PARITY, NEXT_PARITY: BIT;
    signal CURRENT_BIT_POSITION, NEXT_BIT_POSITION:
        INTEGER range PARALLEL_BIT_COUNT downto 0;
    signal CURRENT_PARALLEL_OUT, NEXT_PARALLEL_OUT:
        PARALLEL_TYPE;
begin
    NEXT_ST: process(SERIAL_IN, CURRENT_STATE, RESET,
                    CURRENT_BIT_POSITION, CURRENT_PARITY,
                    CURRENT_PARALLEL_OUT)
    -- This process computes all outputs, the next
    -- state, and the next value of all stored values
    begin
        PARITY_ERROR <= '0'; -- Default values for all
        READ_ENABLE <= '0'; -- outputs and stored values
        NEXT_STATE <= CURRENT_STATE;
        NEXT_BIT_POSITION <= 0;
        NEXT_PARITY <= '0';
        NEXT_PARALLEL_OUT <= CURRENT_PARALLEL_OUT;

        if (RESET = '1') then      -- Synchronous reset
            NEXT_STATE <= WAIT_FOR_START;
        else

```

```

case CURRENT_STATE is    -- State processing
when WAIT_FOR_START =>
    if (SERIAL_IN = '1') then
        NEXT_STATE <= READ_BITS;
        NEXT_PARALLEL_OUT <=
            PARALLEL_TYPE'(others=>'0');
    end if;
when READ_BITS =>
    if (CURRENT_BIT_POSITION =
        PARALLEL_BIT_COUNT) then
        if (CURRENT_PARITY = SERIAL_IN) then
            NEXT_STATE <= ALLOW_READ;
            READ_ENABLE <= '1';
        else
            NEXT_STATE <= PARITY_ERROR_DETECTED;
        end if;
    else
        NEXT_PARALLEL_OUT(CURRENT_BIT_POSITION) <=
            SERIAL_IN;
        NEXT_BIT_POSITION <=
            CURRENT_BIT_POSITION + 1;
        NEXT_PARITY <= CURRENT_PARITY xor
            SERIAL_IN;
    end if;
when PARITY_ERROR_DETECTED =>
    PARITY_ERROR <= '1';
when ALLOW_READ =>
    NEXT_STATE <= WAIT_FOR_START;
end case;
end if;
end process NEXT_ST;

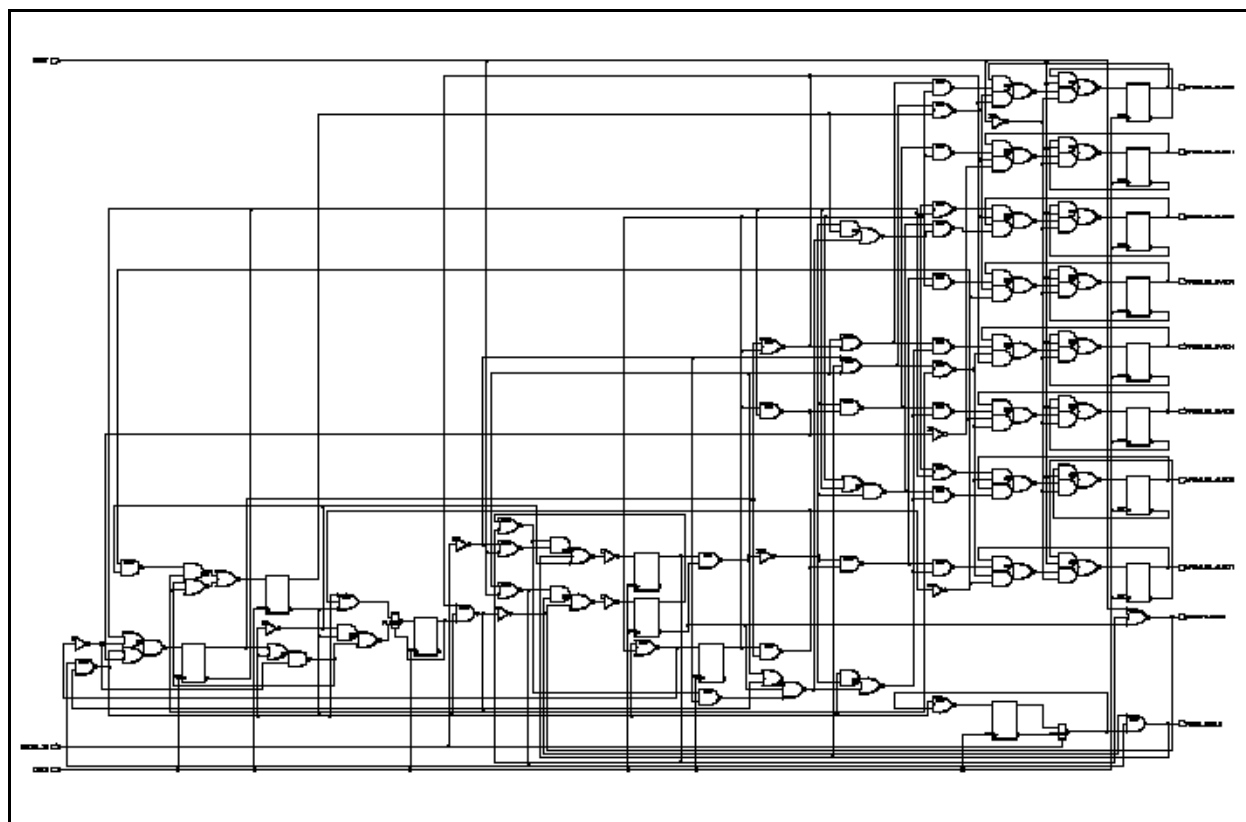
SYNCH: process
    -- This process remembers the stored values
    -- across clock cycles
begin
    wait until CLOCK'event and CLOCK = '1';
    CURRENT_STATE <= NEXT_STATE;
    CURRENT_BIT_POSITION <= NEXT_BIT_POSITION;
    CURRENT_PARITY <= NEXT_PARITY;
    CURRENT_PARALLEL_OUT <= NEXT_PARALLEL_OUT;
end process SYNCH;

PARALLEL_OUT <= CURRENT_PARALLEL_OUT;

end BEHAVIOR;

```

Figure A-17 Serial-to-Parallel Converter—Counting Bits Schematic



Serial-to-Parallel Converter—Shifting Bits

This example describes another implementation of the serial-to-parallel converter in the last example. This design performs the same function as the previous one but uses a different algorithm to do the conversion.

The previous implementation used a counter to indicate the bit of the output that was set when a new serial bit was read. In this implementation, the serial bits are shifted into place. Before the conversion occurs, a '1' is placed in the least-significant bit position. When that '1' is shifted out of the most significant position (position

0), the signal NEXT_HIGH_BIT is set to '1' and the conversion is complete.

Example A-14 shows the listing of the second implementation. The differences are highlighted in bold. The differences relate to the removal of the ..._BIT_POSITION signals, the addition of ..._HIGH_BIT signals, and the change in the way NEXT_PARALLEL_OUT is computed.

Example A-14 Serial-to-Parallel Converter—Shifting Bits

```
package TYPES is
    -- Declares types used in the rest of the design
    type STATE_TYPE is (WAIT_FOR_START,
                        READ_BITS,
                        PARITY_ERROR_DETECTED,
                        ALLOW_READ);
    constant PARALLEL_BIT_COUNT: INTEGER := 8;
    subtype PARALLEL_RANGE is INTEGER
        range 0 to (PARALLEL_BIT_COUNT-1);
    subtype PARALLEL_TYPE is BIT_VECTOR(PARALLEL_RANGE);
end TYPES;

use WORK.TYPES.ALL;      -- Use the TYPES package

entity SER_PAR is        -- Declare the interface
    port(SERIAL_IN, CLOCK, RESET: in BIT;
          PARALLEL_OUT: out PARALLEL_TYPE;
          PARITY_ERROR, READ_ENABLE: out BIT);
end SER_PAR;

architecture BEHAVIOR of SER_PAR is
    -- Signals for stored values
    signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;

    signal CURRENT_PARITY, NEXT_PARITY: BIT;
    signal CURRENT_HIGH_BIT, NEXT_HIGH_BIT: BIT;
    signal CURRENT_PARALLEL_OUT, NEXT_PARALLEL_OUT:
        PARALLEL_TYPE;
begin

    NEXT_ST: process(SERIAL_IN, CURRENT_STATE, RESET,
                    CURRENT_HIGH_BIT, CURRENT_PARITY,
                    CURRENT_PARALLEL_OUT)
        -- This process computes all outputs, the next
```

```

-- state, and the next value of all stored values
begin
  PARITY_ERROR <= '0'; -- Default values for all
  READ_ENABLE <= '0'; -- outputs and stored values
  NEXT_STATE <= CURRENT_STATE;
  NEXT_HIGH_BIT <= '0';
  NEXT_PARITY <= '0';
  NEXT_PARALLEL_OUT <= PARALLEL_TYPE('others=>'0');
  if(RESET = '1') then      -- Synchronous reset
    NEXT_STATE <= WAIT_FOR_START;
  else
    case CURRENT_STATE is -- State processing
      when WAIT_FOR_START =>
        if (SERIAL_IN = '1') then
          NEXT_STATE <= READ_BITS;
          NEXT_PARALLEL_OUT <=
            PARALLEL_TYPE('others=>'0');
        end if;
      when READ_BITS =>
        if (CURRENT_HIGH_BIT = '1') then
          if (CURRENT_PARITY = SERIAL_IN) then
            NEXT_STATE <= ALLOW_READ;
            READ_ENABLE <= '1';
          else
            NEXT_STATE <= PARITY_ERROR_DETECTED;
          end if;
        else
          NEXT_HIGH_BIT <= CURRENT_PARALLEL_OUT(0);
          NEXT_PARALLEL_OUT <=
            CURRENT_PARALLEL_OUT(
              1 to PARALLEL_BIT_COUNT-1) &
              SERIAL_IN;
          NEXT_PARITY <= CURRENT_PARITY xor
            SERIAL_IN;
        end if;
      when PARITY_ERROR_DETECTED =>
        PARITY_ERROR <= '1';
      when ALLOW_READ =>
        NEXT_STATE <= WAIT_FOR_START;
    end case;
  end if;
end process NEXT_ST;

SYNCH: process
  -- This process remembers the stored values
  -- across clock cycles
begin
  wait until CLOCK'event and CLOCK = '1';

```

```

CURRENT_STATE <= NEXT_STATE;
CURRENT_HIGH_BIT <= NEXT_HIGH_BIT;
CURRENT_PARITY <= NEXT_PARITY;
CURRENT_PARALLEL_OUT <= NEXT_PARALLEL_OUT;
end process SYNCH;

PARALLEL_OUT <= CURRENT_PARALLEL_OUT;

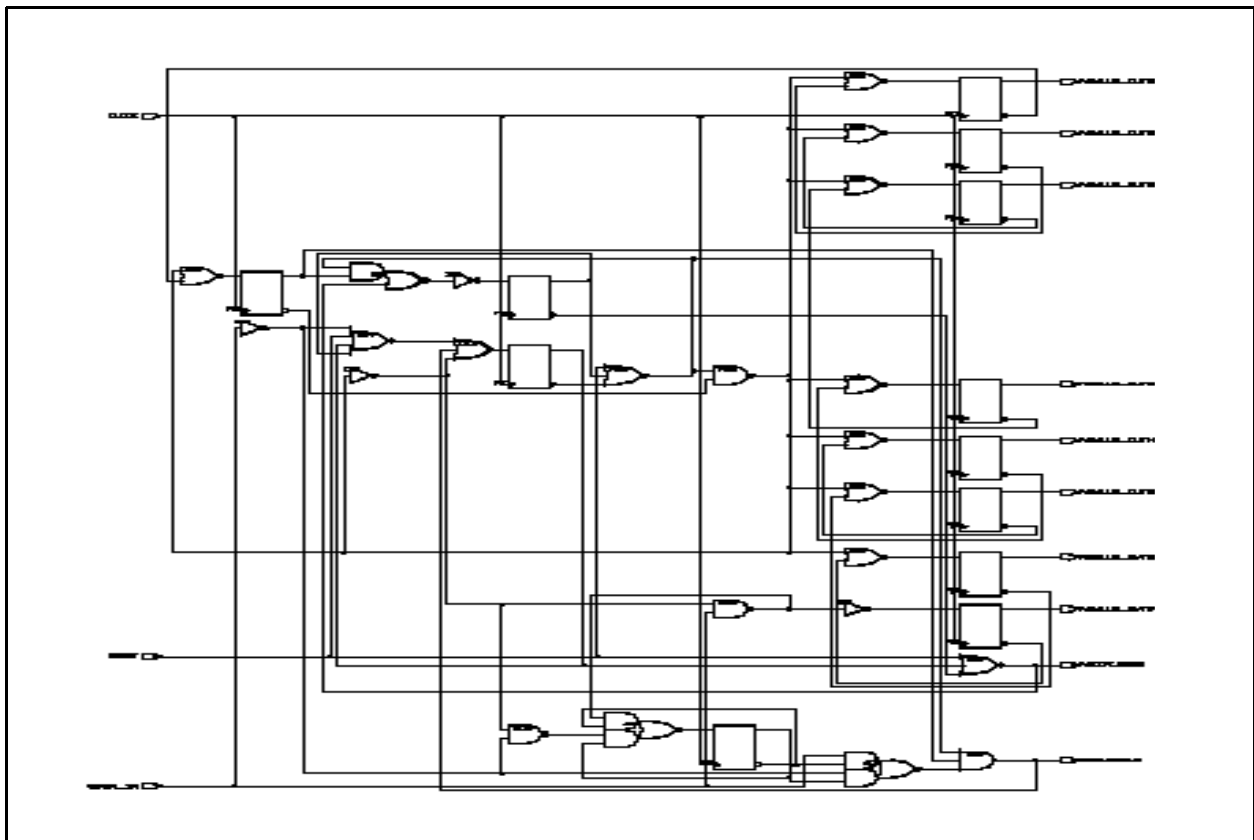
end BEHAVIOR;

```

Note:

The synthesized schematic for the shifter implementation is much simpler than that of the previous count implementation in Example A-13. It is simpler because the shifter algorithm is inherently easier to implement.

Figure A-18 Serial-to-Parallel Converter—Shifting Bits Schematic



With the count algorithm, each of the flip-flops holding the PARALLEL_OUT bits needed logic that decoded the value stored in the BIT_POSITION flip-flops to see when to route in the value of SERIAL_IN. Also, the BIT_POSITION flip-flops needed an incrementer to compute their next value.

In contrast, the shifter algorithm requires neither an incrementer nor flip-flops to hold BIT_POSITION. Additionally, the logic in front of most PARALLEL_OUT bits needs to read only the value of the previous flip-flop or '0'. The value depends on whether bits are currently being read. In the shifter algorithm, the SERIAL_IN port needs to be connected only to the least significant bit (number 7) of the PARALLEL_OUT flip-flops.

These two implementations illustrate the importance of designing efficient algorithms. Both work properly, but the shifter algorithm produces a faster, more area-efficient design.

Programmable Logic Arrays

This example shows a way to build programmable logic arrays (PLAs) in VHDL. The PLA function uses an input lookup vector as an index into a constant PLA table and then returns the output vector specified by the PLA.

The PLA table is an array of PLA rows, where each row is an array of PLA elements. Each element is either a one, a zero, a minus, or a space ('1', '0', '-', or ' '). The table is split between an input plane and an output plane. The input plane is specified by zeros, ones, and minuses. The output plane is specified by zeros and ones. The two planes' values are separated by a space.

In the PLA function, the output vector is first initialized to be all zeros. When the input vector matches an input plane in a row of the PLA table, the ones in the output plane are assigned to the corresponding bits in the output vector. A match is determined as follows:

- If a zero or one is in the input plane, the input vector must have the same value in the same position.
- If a minus is in the input plane, it matches any input vector value at that position.

The generic PLA table types and the PLA function are defined in a package named LOCAL. An entity PLA_VHDL that uses LOCAL needs only to specify its PLA table as a constant, then call the PLA function.

The PLA function does not explicitly depend on the size of the PLA. To change the size of the PLA, change the initialization of the TABLE constant and the initialization of the constants INPUT_COUNT, OUTPUT_COUNT, and ROW_COUNT. In Example A-15, these constants are initialized to a PLA equivalent to the ROM shown previously (Example A-3). Accordingly, the synthesized schematic is the same as that of the ROM, with one difference: in Example A-3, the DATA output port range is 1 to 5; in Example A-15, the OUT_VECTOR output port range is 4 down to 0.

Example A-15 Programmable Logic Array

```
package LOCAL is
    constant INPUT_COUNT: INTEGER := 3;
    constant OUTPUT_COUNT: INTEGER := 5;
    constant ROW_COUNT: INTEGER := 6;
    constant ROW_SIZE: INTEGER := INPUT_COUNT +
                                   OUTPUT_COUNT + 1;
    type PLA_ELEMENT is ('1', '0', '-', ' ');
    type PLA_VECTOR is
        array (INTEGER range <>) of PLA_ELEMENT;
    subtype PLA_ROW is
        PLA_VECTOR(ROW_SIZE - 1 downto 0);
    subtype PLA_OUTPUT is
        PLA_VECTOR(OUTPUT_COUNT - 1 downto 0);
    type PLA_TABLE is
        array(ROW_COUNT - 1 downto 0) of PLA_ROW;

    function PLA(IN_VECTOR: BIT_VECTOR;
                 TABLE: PLA_TABLE)
        return BIT_VECTOR;
end LOCAL;

package body LOCAL is

    function PLA(IN_VECTOR: BIT_VECTOR;
                 TABLE: PLA_TABLE)
        return BIT_VECTOR is
        subtype RESULT_TYPE is
            BIT_VECTOR(OUTPUT_COUNT - 1 downto 0);
        variable RESULT: RESULT_TYPE;
        variable ROW: PLA_ROW;
        variable MATCH: BOOLEAN;
        variable IN_POS: INTEGER;

    begin
        RESULT <= RESULT_TYPE'(others => BIT('0'));
        for I in TABLE'range loop
            ROW <= TABLE(I);
            MATCH <= TRUE;
            IN_POS <= IN_VECTOR'left;
```

```

-- Check for match in input plane
for J in ROW_SIZE - 1 downto OUTPUT_COUNT loop
    if(ROW(J) = PLA_ELEMENT'( '1' )) then
        MATCH <= MATCH and
            (IN_VECTOR(IN_POS) = BIT'( '1' ));
    elsif(ROW(J) = PLA_ELEMENT'( '0' )) then
        MATCH <= MATCH and
            (IN_VECTOR(IN_POS) = BIT'( '0' ));
    else
        null;      -- Must be minus ("don't care")
    end if;
    IN_POS <= IN_POS - 1;
end loop;

-- Set output plane
if(MATCH) then
    for J in RESULT'range loop
        if(ROW(J) = PLA_ELEMENT'( '1' )) then
            RESULT(J) <= BIT'( '1' );
        end if;
    end loop;
end if;
end loop;
return(RESULT);
end;
end LOCAL;

use WORK.LOCAL.all;
entity PLA_VHDL is
    port(IN_VECTOR: BIT_VECTOR(2 downto 0);
          OUT_VECTOR: out BIT_VECTOR(4 downto 0));
end PLA_VHDL;

architecture BEHAVIOR of PLA_VHDL is
    constant TABLE: PLA_TABLE := PLA_TABLE'(
        PLA_ROW'( " --- 1000" ),
        PLA_ROW'( "-1- 0100" ),
        PLA_ROW'( "0-0 00101" ),
        PLA_ROW'( "-1- 00101" ),
        PLA_ROW'( "1-1 00101" ),
        PLA_ROW'( "-1- 00010" ));

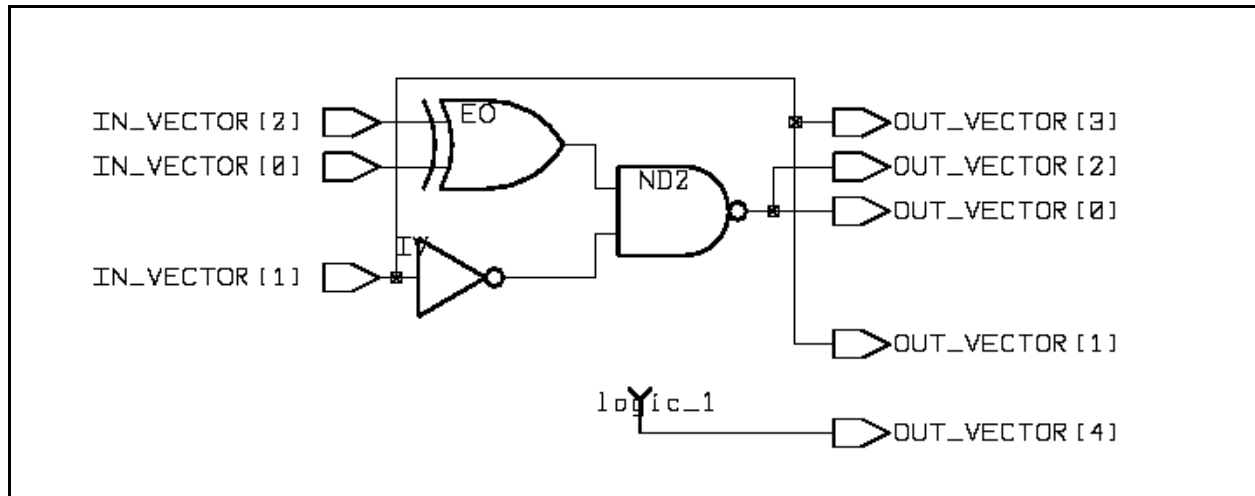
```

```

begin
    OUT_VECTOR <= PLA(IN_VECTOR, TABLE);
end BEHAVIOR;

```

Figure A-19 Programmable Logic Array Schematic



Examples

A-56

B

Synopsys Packages

The following Synopsys packages are included with this release:

- `std_logic_1164` Package

Defines a standard for designers to use in describing the interconnection data types used in VHDL modeling.

- `std_logic_arith` Package

Provides a set of arithmetic, conversion, and comparison functions for SIGNED, UNSIGNED, INTEGER, STD_ULOGIC, STD_LOGIC, and STD_LOGIC_VECTOR types.

- `numeric_std` Package

The `numeric_std` package is an alternative to the `std_logic_arith` package. It is the IEEE standard 1076.3-1997, and documentation about it is available from IEEE. For more information, see “`numeric_std` Package” on page B-20.

- **std_logic_misc Package**

Defines supplemental types, subtypes, constants, and functions for the std_logic_1164 package.

- **ATTRIBUTES Package**

Declares synthesis attributes and the resource sharing subtype and its attributes.

std_logic_1164 Package

The std_logic_1164 package defines the IEEE standard for designers to use in describing the interconnection data types used in VHDL modeling. The logic system defined in this package might be insufficient for modeling switched transistors, because such a requirement is out of the scope of this package. Furthermore, mathematics, primitives, and timing standards are considered orthogonal issues as they relate to this package and are, therefore, beyond its scope.

The std_logic_1164 package file has been updated with Synopsys synthesis directives.

To use this package in a VHDL source file, include the following lines at the beginning of the source file:

```
library IEEE;  
use IEEE.std_logic_1164.all;
```

When you analyze your VHDL source, FPGA Compiler II / *FPGA Express* automatically finds the IEEE library and the std_logic_1164 package. However, you must analyze those use packages that are not in the IEEE and Synopsys libraries before processing a source file that uses them.

std_logic_arith Package

Functions defined in the std_logic_arith package provide conversion to and from the predefined VHDL data type INTEGER, arithmetic, comparison, and BOOLEAN operations. This package lets you perform arithmetic operations and numeric comparisons on array data types. The package defines some arithmetic operators (+, -, *, ABS) and the relational operators (<, >, <=, >=, =, /=). (IEEE VHDL does not define arithmetic operators for arrays and defines the comparison operators in a manner inconsistent with an arithmetic interpretation of array values.)

The package also defines two major data types of its own: UNSIGNED and SIGNED (see “Data Types” on page B-6 for details). The std_logic_arith package is legal VHDL; you can use it for both synthesis and simulation.

You can configure the std_logic_arith package to work on any array of single-bit types. You encode single-bit types in 1 bit with the ENUM_ENCODING attribute.

You can make the vector type (for example, `std_logic_vector`) synonymous with either `SIGNED` or `UNSIGNED`. This way, if you plan to use mostly `UNSIGNED` numbers, you do not need to convert your vector type to call `UNSIGNED` functions. The disadvantage of making your vector type synonymous with either `UNSIGNED` or `SIGNED` is that it causes redefinition of the standard VHDL comparison functions (`=`, `/=`, `<`, `>`, `<=`, `>=`).

Table B-1 shows that the standard comparison functions for `BIT_VECTOR` do not match the `SIGNED` and `UNSIGNED` functions.

Table B-1 UNSIGNED, SIGNED, and BIT_VECTOR Comparison Functions

ARG1	op	ARG2	UNSIGNED	SIGNED	BIT_VECTOR
"000"	=	"000"	true	true	true
"00"	=	"000"	true	true	false
"100"	=	"0100"	true	false	false
"000"	<	"000"	false	false	false
"00"	<	"000"	false	false	true
"100"	<	"0100"	false	true	false

Using the Package

To use the `std_logic_arith` package in a VHDL source file, include the following lines at the beginning of the source file:

```
library IEEE;
use IEEE.std_logic_arith.all;
```

Modifying the Package

The `std_logic_arith` package is written in standard VHDL. You can modify or add to it. The appropriate hardware is then synthesized.

For example, to convert a vector of multivalued logic to an `INTEGER`, you can write the function shown in Example B-1. This `MVL_TO_INTEGER` function returns the integer value corresponding to the vector when the vector is interpreted as an unsigned (natural) number. If unknown values are in the vector, the return value is `-1`.

Example B-1 New Function Based on a `std_logic_arith` Package Function

```
library IEEE;
use IEEE.std_logic_1164.all;

function MVL_TO_INTEGER(ARG : MVL_VECTOR)
  return INTEGER is
  -- pragma built_in SYN_FEED_THRU
  variable uns: UNSIGNED (ARG'range);
begin
  for i in ARG'range loop
    case ARG(i) is
      when '0' | 'L' => uns(i) := '0';
      when '1' | 'H' => uns(i) := '1';
      when others    => return -1;
    end case;
  end loop;
  return CONV_INTEGER(uns);
end MVL_TO_INTEGER;
```

Note the use of the `CONV_INTEGER` function in Example B-1.

FPGA Compiler II / *FPGA Express* performs almost all synthesis directly from the VHDL descriptions. However, several functions are hard-wired for efficiency. They can be identified by the following comment in their declarations:

```
-- pragma built_in
```

This statement marks functions as special, causing the body of the function to be ignored. Modifying the body does not change the synthesized logic unless you remove the built_in comment. If you want new functionality, write it by using the built_in functions; this is more efficient than removing the built_in function and modifying the body of the function.

Data Types

The std_logic_arith package defines two data types: UNSIGNED and SIGNED.

```
type UNSIGNED is array (natural range <>) of std_logic;  
type SIGNED is array (natural range <>) of std_logic;
```

These data types are similar to the predefined VHDL type BIT_VECTOR, but the std_logic_arith package defines the interpretation of variables and signals of these types as numeric values.

UNSIGNED

The UNSIGNED data type represents an unsigned numeric value. FPGA Compiler II / *FPGA Express* interprets the number as a binary representation, with the farthest-left bit being most significant. For example, the decimal number 8 can be represented as

```
UNSIGNED' ("1000")
```

When you declare variables or signals of type UNSIGNED, a larger vector holds a larger number. A 4-bit variable holds values up to decimal 15, an 8-bit variable holds values up to 255, and so on. By definition, negative numbers cannot be represented in an UNSIGNED variable. Zero is the smallest value that can be represented.

Example B-2 illustrates some UNSIGNED declarations. The most significant bit is the farthest-left array bound, rather than the high or low range value.

Example B-2 UNSIGNED Declarations

```
variable VAR: UNSIGNED (1 to 10);
-- 11-bit number
-- VAR(VAR'left) = VAR(1) is the most significant bit

signal SIG: UNSIGNED (5 downto 0);
-- 6-bit number
-- SIG(SIG'left) = SIG(5) is the most significant bit
```

SIGNED

The SIGNED data type represents a signed numeric value. FPGA Compiler II / FPGA *Express* interprets the number as a 2's-complement binary representation, with the farthest-left bit as the sign bit. For example, you can represent decimal 5 and –5 as

```
SIGNED'("0101") -- represents +5
SIGNED'("1011") -- represents -5
```

When you declare SIGNED variables or signals, a larger vector holds a larger number. A 4-bit variable holds values from –8 to 7; an 8-bit variable holds values from –128 to 127. A SIGNED value cannot hold as large a value as an UNSIGNED value with the same bit-width.

Example B-3 shows some SIGNED declarations. The sign bit is the farthest-left bit, rather than the highest or lowest.

Example B-3 SIGNED Declarations

```
variable S_VAR: SIGNED (1 to 10);
-- 11-bit number
-- S_VAR(S_VAR'left) = S_VAR(1) is the sign bit

signal S_SIG: SIGNED (5 downto 0);
-- 6-bit number
-- S_SIG(S_SIG'left) = S_SIG(5) is the sign bit
```

Conversion Functions

The std_logic_arith package provides three sets of functions to convert values between its UNSIGNED and SIGNED types and the predefined type INTEGER. This package also provides the std_logic_vector. Example B-4 shows the declarations of these conversion functions, with BIT and BIT_VECTOR types.

Example B-4 Conversion Functions

```
subtype SMALL_INT is INTEGER range 0 to 1;
function CONV_INTEGER(ARG: INTEGER) return INTEGER;
function CONV_INTEGER(ARG: UNSIGNED) return INTEGER;
function CONV_INTEGER(ARG: SIGNED) return INTEGER;
function CONV_INTEGER(ARG: STD_ULOGIC) return SMALL_INT;

function CONV_UNSIGNED(ARG: INTEGER;
                      SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: UNSIGNED;
                      SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: SIGNED;
                      SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: STD_ULOGIC;
                      SIZE: INTEGER) return UNSIGNED;
```

```

function CONV_SIGNED(ARG: INTEGER;
                     SIZE: INTEGER)    return SIGNED;
function CONV_SIGNED(ARG: UNSIGNED;
                     SIZE: INTEGER)    return SIGNED;
function CONV_SIGNED(ARG: SIGNED;
                     SIZE: INTEGER)    return SIGNED;
function CONV_SIGNED(ARG: STD_ULOGIC;
                     SIZE: INTEGER)    return SIGNED;

function CONV_STD_LOGIC_VECTOR(ARG: INTEGER;
                               SIZE: INTEGER)    return STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: UNSIGNED;
                               SIZE: INTEGER)    return STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: SIGNED;
                               SIZE: INTEGER)    return STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: STD_ULOGIC;
                               SIZE: INTEGER)    return STD_LOGIC_VECTOR;

```

There are four versions of each conversion function. The VHDL operator overloading mechanism determines the correct version from the function call's argument types.

The CONV_INTEGER functions convert an argument of type INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC to an INTEGER return value. The CONV_UNSIGNED and CONV_SIGNED functions convert an argument of type INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC to an UNSIGNED or SIGNED return value whose bit width is SIZE.

The CONV_INTEGER functions have a limitation on the size of operands. VHDL defines INTEGER values as being between -2147483647 and 2147483647. This range corresponds to a 31-bit UNSIGNED value or a 32-bit SIGNED value. You cannot convert an argument outside this range to an INTEGER.

The CONV_UNSIGNED and CONV_SIGNED functions each require two operands. The first operand is the value converted. The second operand is an INTEGER that specifies the expected size of the converted result. For example, the following function call returns a 10-bit UNSIGNED value representing the value in sig.

```
ten_unsigned_bits := CONV_UNSIGNED(sig, 10);
```

If the value passed to CONV_UNSIGNED or CONV_SIGNED is smaller than the expected bit-width (such as representing the value 2 in a 24-bit number), the value is bit-extended appropriately. FPGA Compiler II / FPGA *Express* places zeros in the more significant (left) bits for an UNSIGNED return value, and it uses sign extension for a SIGNED return value.

You can use the conversion functions to extend a number's bit-width even if conversion is not required. For example,

```
CONV_SIGNED(SIGNED'("110"), 8) ⇒ "11111110"
```

An UNSIGNED or SIGNED return value is truncated when its bit-width is too small to hold the ARG value. For example,

```
CONV_SIGNED(UNSIGNED'("1101010"), 3) ⇒ "010"
```

Arithmetic Functions

The std_logic_arith package provides arithmetic functions for use with combinations of the Synopsys UNSIGNED and SIGNED data types and the predefined types STD_ULOGIC and INTEGER. These functions produce adders and subtractors.

There are two sets of arithmetic functions: binary functions having two arguments, such as A+B or A*B, and unary functions having one argument, such as –A. Example B-5 and Example B-6 show the declarations for these functions.

Example B-5 Binary Arithmetic Functions

```
function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED; R: SIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: SIGNED) return SIGNED;
function "+"(L: SIGNED; R: UNSIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: INTEGER) return UNSIGNED;
function "+"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED; R: INTEGER) return SIGNED;
function "+"(L: INTEGER; R: SIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
function "+"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED; R: STD_ULOGIC) return SIGNED;
function "+"(L: STD_ULOGIC; R: SIGNED) return SIGNED;

function "+"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "+"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "+"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "+"(L: STD_ULOGIC; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "+"(L: STD_ULOGIC; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
function "-"(L: SIGNED; R: SIGNED) return SIGNED;
function "-"(L: UNSIGNED; R: SIGNED) return SIGNED;
function "-"(L: SIGNED; R: UNSIGNED) return SIGNED;
function "-"(L: UNSIGNED; R: INTEGER) return UNSIGNED;
function "-"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
function "-"(L: SIGNED; R: INTEGER) return SIGNED;
function "-"(L: INTEGER; R: SIGNED) return SIGNED;
function "-"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
function "-"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
function "-"(L: SIGNED; R: STD_ULOGIC) return SIGNED;
function "-"(L: STD_ULOGIC; R: SIGNED) return SIGNED;

function "-"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
```

```

function "-"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "-"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "-"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "-"(L: STD_ULOGIC; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "-"(L: STD_ULOGIC; R: SIGNED) return STD_LOGIC_VECTOR;

function "*" (L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
function "*" (L: SIGNED; R: SIGNED) return SIGNED;
function "*" (L: SIGNED; R: UNSIGNED) return SIGNED;
function "*" (L: UNSIGNED; R: SIGNED) return SIGNED;

```

Example B-6 Unary Arithmetic Functions

```

function "+"(L: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED) return SIGNED;
function "-"(L: SIGNED) return SIGNED;
function "ABS"(L: SIGNED) return SIGNED;

```

The unary arithmetic functions in Example B-5 and Example B-6 determine the width of their return values, as follows:

1. When only one UNSIGNED or SIGNED argument is present, the width of the return value is the same as that argument's.
2. When both arguments are either UNSIGNED or SIGNED, the width of the return value is the larger of the two argument widths. An exception is that when an UNSIGNED number is added to or subtracted from a SIGNED number that is the same size or smaller, the return value is a SIGNED number 1 bit wider than the UNSIGNED argument. This size guarantees that the return value is large enough to hold any (positive) value of the UNSIGNED argument.

The number of bits returned by + and – is illustrated in Table B-2.


```

signal U4: UNSIGNED (3 downto 0);
signal U8: UNSIGNED (7 downto 0);
signal S4: SIGNED (3 downto 0);
signal S8: SIGNED (7 downto 0);

```

Table B-2 *Number of Bits Returned by + and –*

+ or -	U4	U8	S4	S8
U4	4	8	5	8
U8	8	8	9	9
S4	5	9	4	8
S8	8	9	8	8

In some circumstances, you might need to obtain a carry-out bit from the + or – operation. To do this, extend the larger operand by 1 bit. The high bit of the return value is the carry, as shown in Example B-7.

Example B-7 *Using the Carry-Out Bit*

```

process
    variable a, b, sum: UNSIGNED (7 downto 0);
    variable temp: UNSIGNED (8 downto 0);
    variable carry: BIT;
begin
    temp <= CONV_UNSIGNED(a,9) + b;
    sum <= temp(7 downto 0);
    carry <= temp(8);
end process;

```

Comparison Functions

The `std_logic_arith` package provides functions for comparing UNSIGNED and SIGNED data types with each other and with the predefined type INTEGER. FPGA Compiler II / FPGA Express compares the numeric values of the arguments, returning a BOOLEAN value. For example, the following evaluates true:

```
UNSIGNED'("001") > SIGNED'("111")
```

The `std_logic_arith` comparison functions are similar to the built-in VHDL comparison functions. The only difference is that the `std_logic_arith` functions accommodate signed numbers and varying bit-widths. The predefined VHDL comparison functions perform bitwise comparisons and do not have the correct semantics for comparing numeric values (see “Relational Operators” on page 4-5).

These functions produce comparators. The function declarations are listed in two groups: ordering functions ("`<`", "`<=`", "`>`", "`>=`"), shown in Example B-8, and equality functions ("`=`", "`/=`"), shown in Example B-9.

Example B-8 Ordering Functions

```
function "<"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "<="(L: INTEGER; R: SIGNED) return BOOLEAN;
```

Example B-9 Equality Functions

```
function "="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "="(L: SIGNED; R: SIGNED) return BOOLEAN;
function "="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "="(L: SIGNED; R: INTEGER) return BOOLEAN;
function "="(L: INTEGER; R: SIGNED) return BOOLEAN;

function "/="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "/="(L: SIGNED; R: SIGNED) return BOOLEAN;
function "/="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "/="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "/="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "/="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "/="(L: SIGNED; R: INTEGER) return BOOLEAN;
function "/="(L: INTEGER; R: SIGNED) return BOOLEAN;
```

Shift Functions

The `std_logic_arith` package provides functions for shifting the bits in SIGNED and UNSIGNED numbers. These functions produce shifters. Example B-10 shows the shift function declarations. For a list of shift and rotate operators, see “Operators” on page C-9.

Example B-10 Shift Functions

```
function SHL(ARG: UNSIGNED;  
             COUNT: UNSIGNED) return UNSIGNED;  
function SHL(ARG: SIGNED;  
             COUNT: UNSIGNED) return SIGNED;  
  
function SHR(ARG: UNSIGNED;  
             COUNT: UNSIGNED) return UNSIGNED;  
function SHR(ARG: SIGNED;  
             COUNT: UNSIGNED) return SIGNED;
```

The SHL function shifts the bits of its argument ARG left by COUNT bits. SHR shifts the bits of its argument ARG right by COUNT bits.

The SHL functions work the same for both UNSIGNED and SIGNED values of ARG, shifting in zero bits as necessary. The SHR functions treat UNSIGNED and SIGNED values differently. If ARG is an UNSIGNED number, vacated bits are filled with zeros; if ARG is a SIGNED number, the vacated bits are copied from the ARG sign bit.

Example B-11 shows some shift function calls and their return values.

Example B-11 Shift Operations

```
variable U1, U2: UNSIGNED (7 downto 0);
variable S1, S2: SIGNED   (7 downto 0);
variable COUNT:  UNSIGNED (1 downto 0);
. . .
U1 <= "01101011";
U2 <= "11101011";

S1 <= "01101011";
S2 <= "11101011";

COUNT <= CONV_UNSIGNED(ARG => 3, SIZE => 2);
. . .
SHL(U1, COUNT) = "01011000"
SHL(S1, COUNT) = "01011000"
SHL(U2, COUNT) = "01011000"
SHL(S2, COUNT) = "01011000"

SHR(U1, COUNT) = "00001101"
SHR(S1, COUNT) = "00001101"
SHR(U2, COUNT) = "00011101"
SHR(S2, COUNT) = "11111101"
```

Multiplication Using Shifts

You can use shift operations for simple multiplication and division of UNSIGNED numbers if you are multiplying or dividing by a power of 2.

For example, to divide the following UNSIGNED variable U by 4, use this syntax:

```
variable U: UNSIGNED (7 downto 0) := "11010101";
variable quarter_U: UNSIGNED (5 downto 0);

quarter_U <= SHR(U, "01");
```

ENUM_ENCODING Attribute

Place the synthesis attribute ENUM_ENCODING on your primary logic type (see “Enumeration Encoding” on page 3-4). This attribute allows FPGA Compiler II / *FPGA Express* to interpret your logic correctly.

pragma built_in

Label your primary logic functions with built_in pragmas. Pragmas allow FPGA Compiler II / *FPGA Express* to interpret your logic functions easily. When you use a built_in pragma, FPGA Compiler II / *FPGA Express* parses but ignores the body of the function. Instead, FPGA Compiler II / *FPGA Express* directly substitutes the appropriate logic for the function. You need not use built_in pragmas, but they can result in runtimes that are 10 times as fast.

Use a built_in pragma by placing a comment in the declaration part of a function. FPGA Compiler II / *FPGA Express* interprets a comment as a directive if the first word of the comment is pragma. Example B-12 shows the use of a built_in pragma.

Example B-12 Using a built_in pragma

```
function "XOR" (L, R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
  -- pragma built_in SYN_XOR
  begin
    if (L = '1') xor (R = '1') then
      return '1';
    else
      return '0';
    end if;
  end "XOR";
```

Two-Argument Logic Functions

Synopsys provides six built-in functions for performing two-argument logic functions:

- SYN_AND
- SYN_OR
- SYN_NAND
- SYN_NOR
- SYN_XOR
- SYN_XNOR

You can use these functions on single-bit arguments or equal-length arrays of single bits. Example B-13 shows a function that takes the logical AND of two equal-size arrays.

Example B-13 Built-In AND for Arrays

```
function "AND" (L, R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
  -- pragma built_in SYN_AND
  variable MY_L: STD_LOGIC_VECTOR (L'length-1 downto 0);
  variable MY_R: STD_LOGIC_VECTOR (L'length-1 downto 0);
  variable RESULT: STD_LOGIC_VECTOR (L'length-1 downto 0);
begin
  assert L'length = R'length;
  MY_L <= L;
  MY_R <= R;
  for i in RESULT'range loop
    if (MY_L(i) = '1') and (MY_R(i) = '1') then
      RESULT(i) <= '1';
    else
      RESULT(i) <= '0';
    end if;
  end loop;
  return RESULT;
end "AND";
```

One-Argument Logic Functions

Synopsys provides two built-in functions to perform one-argument logic functions:

- SYN_NOT
- SYN_BUF

You can use these functions on single-bit arguments or equal-length arrays of single bits. Example B-14 shows a function that takes the logical NOT of an array.

Example B-14 Built-In NOT for Arrays

```
function "NOT" (L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
-- pragma built_in SYN_NOT
  variable MY_L: STD_LOGIC_VECTOR (L'length-1 downto 0);
  variable RESULT: STD_LOGIC_VECTOR (L'length-1 downto 0);
begin
  MY_L <= L;
  for i in result'range loop
    if (MY_L(i) = '0' or MY_L(i) = 'L') then
      RESULT(i) <= '1';
    elsif (MY_L(i) = '1' or MY_L(i) = 'H') then
      RESULT(i) <= '0';
    else
      RESULT(i) <= 'X';
    end if;
  end loop;
  return RESULT;
end "NOT";
```

Type Conversion

The built-in function SYN_FEED_THRU performs fast type conversion between unrelated types. The synthesized logic from SYN_FEED_THRU wires the single input of a function to the return value. This connection can save CPU time required to process a complicated conversion function, as shown in Example B-15.

Example B-15 Use of SYN_FEED_THRU

```
type COLOR is (RED, GREEN, BLUE);
attribute ENUM_ENCODING : STRING;
attribute ENUM_ENCODING of COLOR : type is "01 10 11";
...

function COLOR_TO_BV (L: COLOR) return BIT_VECTOR is
-- pragma built_in SYN_FEED_THRU
begin
    case L is
        when RED    => return "01";
        when GREEN  => return "10";
        when BLUE   => return "11";
    end case;
end COLOR_TO_BV;
```

numeric_std Package

FPGA Compiler II / FPGA *Express* supports nearly all of numeric_std, the IEEE Standard VHDL Synthesis Package, which defines numeric types and arithmetic functions.

Caution!

The numeric_std package and the std_logic_arith package have overlapping operations. Use of these two packages simultaneously during analysis could cause type mismatches.

Understanding the Limitations of numeric_std package

The 1999.05 version of FPGA Compiler II / FPGA *Express* does not support the following numeric_std package components:

- divide, rem, or mod operators

If your design contains these operators, use the std_logic_arith package.

- TO_01 function as a simulation construct

Using the Package

Access numeric_std package with the following statement in your VHDL code:

```
library IEEE;  
use IEEE.numeric_std.all;
```

Synopsys packages are pre-analyzed and do not require further analyzing. To list the packages currently in memory, use the following command:

```
report_design_lib
```

Data Types

The numeric_std package defines the following two data types in the same way that the std_logic_arith package does:

- UNSIGNED

```
type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
```

See “UNSIGNED” on page B-6 for more information.

- SIGNED

```
type SIGNED is array (NATURAL range <>) of STD_LOGIC;
```

See “SIGNED” on page B-7 for more information.

Conversion Functions

The numeric_std package provides functions to convert values between its UNSIGNED and SIGNED types. Example B-16 shows the declarations of these conversion functions.

Example B-16 numeric_std Conversion Functions

```
function TO_INTEGER (ARG: UNSIGNED) return NATURAL;  
function TO_INTEGER (ARG: SIGNED) return INTEGER;  
function TO_UNSIGNED (ARG, SIZE: NATURAL) return UNSIGNED;  
function TO_SIGNED (ARG: INTEGER; SIZE: NATURAL) return SIGNED;
```

TO_INTEGER, TO_SIGNED, and TO_UNSIGNED are similar to CONV_INTEGER, CONV_SIGNED, and CONV_UNSIGNED in std_logic_arith (see “Conversion Functions” on page B-8).

Resize Function

The resize function numeric_std supports is shown in the declarations in Example B-17.

Example B-17 numeric_std Resize Function

```
function RESIZE (ARG: SIGNED; NEW_SIZE: NATURAL) return SIGNED;  
function RESIZE (ARG: UNSIGNED; NEW_SIZE: NATURAL) return UNSIGNED;
```

Arithmetic Functions

The `numeric_std` package provides arithmetic functions for use with combinations of Synopsys UNSIGNED and SIGNED data types and the predefined types STD_ULOGIC and INTEGER. These functions produce adders and subtractors.

There are two sets of arithmetic functions, which the `numeric_std` package defines in the same way that the `std_logic_arith` package does (see “Arithmetic Functions” on page B-10 for more information):

- Binary functions having two arguments, such as

$A+B$

$A*B$

Example B-18 shows the declarations for these functions.

- Unary functions having one argument, such as

$-A$

`abs A`

Example B-19 on page B-24 shows the declarations for these functions.

Example B-18 numeric_std Binary Arithmetic Functions

```
function "+" (L, R: UNSIGNED) return UNSIGNED;
function "+" (L, R: SIGNED) return SIGNED;
function "+" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
function "+" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
function "+" (L: INTEGER; R: SIGNED) return SIGNED;
function "+" (L: SIGNED; R: INTEGER) return SIGNED;
```

```

function "-" (L, R: UNSIGNED) return UNSIGNED;
function "-" (L, R: SIGNED) return SIGNED;
function "-" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
function "-" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
function "-" (L: SIGNED; R: INTEGER) return SIGNED;
function "-" (L: INTEGER; R: SIGNED) return SIGNED;

function "*" (L, R: UNSIGNED) return UNSIGNED;
function "*" (L, R: SIGNED) return SIGNED;
function "*" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
function "*" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
function "*" (L: SIGNED; R: INTEGER) return SIGNED;
function "*" (L: INTEGER; R: SIGNED) return SIGNED;

```

Example B-19 numeric_std Unary Arithmetic Functions

```

function "abs" (ARG: SIGNED) return SIGNED;
function "-" (ARG: SIGNED) return SIGNED;

```

Comparison Functions

The `numeric_std` package provides functions to compare UNSIGNED and SIGNED data types to each other and to the predefined type INTEGER. FPGA Compiler II / FPGA *Express* compares the numeric values of the arguments and returns a BOOLEAN value.

These functions produce comparators. The function declarations are listed in two groups:

- Ordering functions ("`<`", "`<=`", "`>`", "`>=`"), shown in Example B-20
- Equality functions ("`=`", "`/=`"), shown in Example B-21 on page B-25

Example B-20 numeric_std Ordering Functions

```
function ">" (L, R: UNSIGNED) return BOOLEAN;
function ">" (L, R: SIGNED) return BOOLEAN;
function ">" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
function ">" (L: INTEGER; R: SIGNED) return BOOLEAN;
function ">" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
function ">" (L: SIGNED; R: INTEGER) return BOOLEAN;

function "<" (L, R: UNSIGNED) return BOOLEAN;
function "<" (L, R: SIGNED) return BOOLEAN;
function "<" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
function "<" (L: INTEGER; R: SIGNED) return BOOLEAN;
function "<" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
function "<" (L: SIGNED; R: INTEGER) return BOOLEAN;

function "<=" (L, R: UNSIGNED) return BOOLEAN;
function "<=" (L, R: SIGNED) return BOOLEAN;
function "<=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
function "<=" (L: INTEGER; R: SIGNED) return BOOLEAN;
function "<=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
function "<=" (L: SIGNED; R: INTEGER) return BOOLEAN;

function ">=" (L, R: UNSIGNED) return BOOLEAN;
function ">=" (L, R: SIGNED) return BOOLEAN;
function ">=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
function ">=" (L: INTEGER; R: SIGNED) return BOOLEAN;
function ">=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
function ">=" (L: SIGNED; R: INTEGER) return BOOLEAN;
```

Example B-21 numeric_std Equality Functions

```
function "=" (L, R: UNSIGNED) return BOOLEAN;
function "=" (L, R: SIGNED) return BOOLEAN;
function "=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
function "=" (L: INTEGER; R: SIGNED) return BOOLEAN;
function "=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
function "=" (L: SIGNED; R: INTEGER) return BOOLEAN;

function "/=" (L, R: UNSIGNED) return BOOLEAN;
function "/=" (L, R: SIGNED) return BOOLEAN;
function "/=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
```

```
function "/"= (L: INTEGER; R: SIGNED) return BOOLEAN;
function "/"= (L: UNSIGNED; R: NATURAL) return BOOLEAN;
function "/"= (L: SIGNED; R: INTEGER) return BOOLEAN;
```

Defining Logical Operators Functions

The `numeric_std` package provides functions that define all of the logical operators: NOT, AND, OR, NAND, NOR, XOR, and XNOR. These functions work just like similar functions in `std_logic_1164`, except that they operate on SIGNED and UNSIGNED values rather than on STD_LOGIC_VECTOR values. Example B-22 shows these function declarations.

Example B-22 numeric_std Logical Operators Functions

```
function "not" (L: UNSIGNED) return UNSIGNED;
function "and" (L, R: UNSIGNED) return UNSIGNED;
function "or" (L, R: UNSIGNED) return UNSIGNED;
function "nand" (L, R: UNSIGNED) return UNSIGNED;
function "nor" (L, R: UNSIGNED) return UNSIGNED;
function "xor" (L, R: UNSIGNED) return UNSIGNED;
function "xnor" (L, R: UNSIGNED) return UNSIGNED;

function "not" (L: SIGNED) return SIGNED;
function "and" (L, R: SIGNED) return SIGNED;
function "or" (L, R: SIGNED) return SIGNED;
function "nand" (L, R: SIGNED) return SIGNED;
function "nor" (L, R: SIGNED) return SIGNED;
function "xor" (L, R: SIGNED) return SIGNED;
function "xnor" (L, R: SIGNED) return SIGNED;
```

Shift Functions

The `numeric_std` package provides functions for shifting the bits in UNSIGNED and SIGNED numbers. These functions produce shifters. Example B-23 shows the shift function declarations.

Example B-23 numeric_std Shift Functions

```
function SHIFT_LEFT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
function SHIFT_RIGHT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
function SHIFT_LEFT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;
function SHIFT_RIGHT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;

function ROTATE_LEFT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
function ROTATE_RIGHT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
function ROTATE_LEFT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;
function ROTATE_RIGHT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;
```

The SHIFT_LEFT function shifts the bits of its argument ARG left by COUNT bits. SHIFT_RIGHT shifts the bits of its argument ARG right by COUNT bits.

The SHIFT_LEFT functions work the same for both UNSIGNED and SIGNED values of ARG, shifting in zero bits as necessary. The SHIFT_RIGHT functions treat UNSIGNED and SIGNED values differently:

- If ARG is an UNSIGNED number, vacated bits are filled with zeros
- If ARG is a SIGNED number, the vacated bits are copied from the ARG sign bit

Example B-26 on page B-29 shows some shift functions calls and their return values.

Rotate Functions

ROTATE_LEFT and ROTATE_RIGHT are similar to the shift functions.

Example B-24 shows rotate function declarations.

Example B-24 numeric_std Rotate Functions

```
ROTATE_LEFT (U1, COUNT) = "01011011"  
ROTATE_LEFT (S1, COUNT) = "01011011"  
ROTATE_LEFT (U2, COUNT) = "01011111"  
ROTATE_LEFT (S2, COUNT) = "01011111"  
  
ROTATE_RIGHT (U1, COUNT) = "01101101"  
ROTATE_RIGHT (S1, COUNT) = "01101101"  
ROTATE_RIGHT (U2, COUNT) = "01111101"  
ROTATE_RIGHT (S2, COUNT) = "01111101"
```

Shift and Rotate Operators

The numeric_std package provides shift operators and rotate operators, which work in the same way that shift functions and rotate functions do. The shift operators are: sll, srl, sla, and sra.

Example B-25 shows some shift and rotate operator declarations.

Example B-26 on page B-29 includes some shift and rotate operators.

Example B-25 numeric_std Shift Operators

```
function "sll" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;  
function "sll" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;  
function "srl" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;  
function "srl" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;  
function "rol" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;  
function "rol" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;  
function "ror" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;  
function "ror" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;
```


Example B-26 Some numeric_std Shift Functions and Shift Operators

```
Variable U1, U2: UNSIGNED (7 downto 0);
Variable S1, S2: SIGNED (7 downto 0);
Variable COUNT: NATURAL;
...
U1 <= "01101011";
U2 <= "11101011";
S1 <= "01101011";
S2 <= "11101011";
COUNT <= 3;
...
SHIFT_LEFT (U1, COUNT) = "01011000"
SHIFT_LEFT (S1, COUNT) = "01011000"
SHIFT_LEFT (U2, COUNT) = "01011000"
SHIFT_LEFT (S2, COUNT) = "01011000"

SHIFT_RIGHT (U1, COUNT) = "00001101"
SHIFT_RIGHT (S1, COUNT) = "00001101"
SHIFT_RIGHT (U2, COUNT) = "00011101"
SHIFT_RIGHT (S2, COUNT) = "11111101"

U1 sll COUNT = "01011000"
S1 sll COUNT = "01011000"
U2 sll COUNT = "01011000"
S2 sll COUNT = "01011000"

U1 srl COUNT = "00001101"
S1 srl COUNT = "00001101"
U2 srl COUNT = "00011101"
S2 srl COUNT = "11111101"

U1 rol COUNT = "01011011"
S1 rol COUNT = "01011011"
U2 rol COUNT = "01011111"
S2 rol COUNT = "01011111"

U1 ror COUNT = "01101101"
S1 ror COUNT = "01101101"
U2 ror COUNT = "01111101"
S2 ror COUNT = "01111101"
```

std_logic_misc Package

The std_logic_misc package resides in the lib/packages/IEEE/src/std_logic_misc.vhd subdirectory of the FPGA Compiler II / FPGA *Express* directory. It declares the primary data types the Synopsys VSS tools support.

Boolean reduction functions take one argument (an array of bits) and return a single bit. For example, the AND reduction of "101" is "0", the logical AND of all three bits.

Several functions in the std_logic_misc package provide Boolean reduction operations for the predefined type STD_LOGIC_VECTOR. Example B-27 shows the declarations of these functions.

Example B-27 Boolean Reduction Functions

```
function AND_REDUCE  (ARG: STD_LOGIC_VECTOR) return UX01;
function NAND_REDUCE (ARG: STD_LOGIC_VECTOR) return UX01;
function OR_REDUCE   (ARG: STD_LOGIC_VECTOR) return UX01;
function NOR_REDUCE  (ARG: STD_LOGIC_VECTOR) return UX01;
function XOR_REDUCE  (ARG: STD_LOGIC_VECTOR) return UX01;
function XNOR_REDUCE (ARG: STD_LOGIC_VECTOR) return UX01;
function AND_REDUCE  (ARG: STD_ULONGIC_VECTOR) return UX01;
function NAND_REDUCE (ARG: STD_ULONGIC_VECTOR) return UX01;
function OR_REDUCE   (ARG: STD_ULONGIC_VECTOR) return UX01;
function NOR_REDUCE  (ARG: STD_ULONGIC_VECTOR) return UX01;
function XOR_REDUCE  (ARG: STD_ULONGIC_VECTOR) return UX01;
function XNOR_REDUCE (ARG: STD_ULONGIC_VECTOR) return UX01;
```

These functions combine the bits of the STD_LOGIC_VECTOR, as the name of the function indicates. For example, XOR_REDUCE returns the XOR of all bits in ARG. Example B-28 shows some reduction function calls and their return values.

Example B-28 Boolean Reduction Operations

AND_REDUCE("111") = '1'

AND_REDUCE("011") = '0'

OR_REDUCE("000") = '0'

OR_REDUCE("001") = '1'

XOR_REDUCE("100") = '1'

XOR_REDUCE("101") = '0'

NAND_REDUCE("111") = '0'

NAND_REDUCE("011") = '1'

NOR_REDUCE("000") = '1'

NOR_REDUCE("001") = '0'

XNOR_REDUCE("100") = '0'

XNOR_REDUCE("101") = '1'

ATTRIBUTES Package

The ATTRIBUTES package declares all the supported synthesis (and simulation) attributes. These include:

- FPGA Compiler II / FPGA *Express* constraints and attributes
- State vector attributes
- Resource sharing attributes
- General attributes for interpreting VHDL (described in Chapter 3, "Data Types")
- Attributes for use with the Synopsys VSS tools

Reference this package when you use synthesis attributes:

```
library SYNOPSIS;  
use SYNOPSIS.ATTRIBUTES.all;
```

C

VHDL Constructs

Many VHDL language constructs, although useful for simulation and other stages in the design process, are not relevant to synthesis. Because these constructs cannot be synthesized, FPGA Compiler II / FPGA *Express* does not support them.

This appendix provides a list of all VHDL language constructs, with the level of support for each, followed by a list of VHDL reserved words.

This appendix describes

- VHDL Construct Support
- VHDL Reserved Words

VHDL Construct Support

A construct can be fully supported, ignored, or unsupported. Ignored and unsupported constructs are defined as follows:

- Ignored means that the construct is allowed in the VHDL source but is ignored by FPGA Compiler II / *FPGA Express*.
- Unsupported means that the construct is not allowed in the VHDL source and that FPGA Compiler II / *FPGA Express* flags it as an error. If errors are in a VHDL description, the description is not translated (synthesized).

Constructs are listed in the following order:

- Design units
- Data types
- Declarations
- Specifications
- Names
- Operators
- Operands and expressions
- Sequential statements
- Concurrent statements
- Predefined language environment

Design Units

entity

The entity statement part is ignored. Generics are supported, but only of type INTEGER. Default values for ports are ignored.

architecture

Multiple architectures are allowed. Global signal interaction between architectures is unsupported.

configuration

Configuration declarations and block configurations are supported, but only to specify the top-level architecture for a top-level entity.

The use clauses, attribute specifications, component configurations, and nested block configurations are unsupported.

package

Packages are fully supported.

library

Libraries and separate compilation are supported.

subprogram

Default values for parameters are unsupported. Assigning to indexes and slices of unconstrained out parameters is unsupported, unless the actual parameter is an identifier.

Subprogram recursion is unsupported if the recursion is not bounded by a static value.

Resolution functions are supported for wired-logic and three-state functions only.

Subprograms can be declared only in packages and in the declaration part of an architecture.

Data Types

enumeration

Enumeration is fully supported.

integer

Infinite-precision arithmetic is unsupported.

Integer types are automatically converted to bit vectors whose width is as small as possible to accommodate all possible values of the type's range. The type's range can be either in unsigned binary for nonnegative ranges or in 2's-complement form for ranges that include negative numbers.

physical

Physical type declarations are ignored. The use of physical types is ignored in delay specifications.

floating

Floating-point type declarations are ignored. The use of floating-point types is unsupported except for floating-point constants used with Synopsys-defined attributes.

array

Array ranges and indexes other than integers are unsupported.

Multidimensional arrays are unsupported, but arrays of arrays are supported.

record

Record data types are fully supported.

access

Access type declarations are ignored, and the use of access types is unsupported.

file

File type declarations are ignored, and the use of file types is unsupported.

incomplete type declarations

Incomplete type declarations are unsupported.

Declarations

constant

Constant declarations are supported except for deferred constant declarations.

signal

Register and bus declarations are unsupported. Resolution functions are supported for wired and three-state functions only. Declarations other than from a globally static type are unsupported. Initial values are unsupported.

variable

Declarations other than from a globally static type are unsupported. Initial values are unsupported.

shared variable

Variable shared by different processes. Shared variables are fully supported.

file

File declarations are unsupported.

interface

Buffer and linkage are translated to out and inout, respectively.

alias

Alias declarations are supported, with the following exceptions:

- An alias declaration that lacks a subtype indication
- A nonobject alias—such as an alias that refers to a type.

component

Component declarations that list a name other than a valid entity name are unsupported.

attribute

Attribute declarations are fully supported, but the use of user-defined attributes is unsupported.

Specifications

attribute

Others and all are unsupported in attribute specifications. User-defined attributes can be specified, but the use of user-defined attributes is unsupported.

configuration

Configuration specifications are unsupported.

disconnection

Disconnection specifications are unsupported. Attribute declarations are fully supported, but the use of user-defined attributes is unsupported.

Names

simple

Simple names are fully supported.

selected

Selected (qualified) names outside a use clause are unsupported.
Overriding the scopes of identifiers is unsupported.

operator symbol

Operator symbols are fully supported.

indexed

Indexed names are fully supported, with one exception: Indexing an unconstrained out parameter in a procedure is unsupported.

slice

Slice names are fully supported, with one exception: Using a slice of an unconstrained out parameter in a procedure is unsupported unless the actual parameter is an identifier.

attribute

Only the following predefined attributes are supported: base, left, right, high, low, range, reverse_range, and length. The event and stable attributes are supported only as described with the wait and if statements (see “wait Statements” on page 5-50). User-defined attribute names are unsupported. The use of attributes with selected names (name.name'attribute) is unsupported.

Identifiers and Extended Identifiers

An identifier in VHDL is a user-defined name for any of these: constant, variable, function, signal, entity, port, subprogram, parameter, and instance.

Specifics of Identifiers

The characteristics of identifiers are:

- They can be composed of letters, digits, and the underscore character (_).
- Their first character cannot be a number, unless it is an extended identifier (see Example C-1).
- They can be of any length.
- They are case-insensitive.
- All of their characters are significant.

Specifics of Extended Identifiers

The characteristics of extended identifiers are:

- Any of the following can be defined as one:
 - Identifiers that contain special characters
 - Identifiers that begin with numbers
 - Identifiers that have the same name as a keyword

- They start with a backslash character (\), followed by a sequence of characters, followed by another backslash (\).
- They are case-sensitive.

Example C-1 shows some extended identifiers.

Example C-1 Sample Extended Identifiers

```
\a+b\           \3state\
\type\          \ (a&b) | c\
```

For more information about identifiers and extended identifiers, see “Identifiers” on page 4-23.

Operators

logical

Logical operators are fully supported.

relational

Relational operators are fully supported.

addition

Concatenation and arithmetic operators are fully supported.

signing

Signing operators are fully supported.

multiplying

The * (multiply) operator is fully supported. The / (division), mod, and rem operators are supported only when both operands are constant or when the right operand is a constant power of 2.

miscellaneous

The ****** operator is supported only when both operands are constant or when the left operand is 2. The **abs** operator is fully supported.

operator overloading

Operator overloading is fully supported.

short-circuit operation

The short-circuit behavior of operators is not supported.

Shift and Rotate Operators

You can define shift and rotate operators for any one-dimensional array type whose element type is either of the predefined types, **BIT** or **Boolean**. The right operand is always of type integer. The type of the result of a shift operator is the same as the type of the left operand. The shift and rotate operators are included in the list of VHDL reserved words in Table C-1 on page C-17. There is more information about the shift and rotate operators that **numeric_std** supports in “Shift and Rotate Operators” on page B-28. The shift operators are:

sll

Shift left logical

srl

Shift right logical

sla

Shift left arithmetic

sra

Shift right arithmetic

The rotate operators are

rol

Rotate left logical

ror

Rotate right logical

Example C-2 illustrates the use of shift and rotate operators.

Example C-2 Sample Showing Use of Shift and Rotate Operators

```
architecture arch of shft_op is
begin
    a <= "01101";
    q1 <= a sll 1;           -- q1 = "11010"
    q2 <= a srl 3;           -- q2 = "00001"
    q3 <= a rol 2;           -- q3 = "10101"
    q4 <= a ror 1;           -- q4 = "10110"
    q5 <= a sla 2;           -- q5 = "10100"
    q6 <= a sra 1;           -- q6 = "00110"
end;
```

xnor Operator

You can define the binary logical operator xnor for predefined types BIT and Boolean, as well as for any one-dimensional array type whose element type is BIT or Boolean. The operands must be the same type and length. The result also has the same type and length. The xnor operator is included in the list of VHDL reserved words in Table C-1 on page C-17.

Example C-3 Sample Showing Use of xnor Operator

```
a <= "10101";
b <= "11100";
c <= a xnor b;           -- c = "10110"
```

Operands and Expressions

based literal

Based literals are fully supported.

null literal

Null slices, null ranges, and null arrays are unsupported.

physical literal

Physical literals are ignored.

string

Strings are fully supported.

aggregate

The use of types as aggregate choices is supported. Record aggregates are supported.

function call

Function calls are supported, with one exception: Function conversions on input ports are not supported, because type conversions on formal ports in a connection specification (port map) are not supported.

qualified expression

Qualified expressions are fully supported.

type conversion

Type conversion is fully supported.

allocator

Allocators are unsupported.

static expression

Static expressions are fully supported.

universal expression

Floating-point expressions are unsupported, except in a Synopsys-recognized attribute definition. Infinite-precision expressions are not supported. Precision is limited to 32 bits; all intermediate results are converted to integer.

Sequential Statements

wait

The wait statement is unsupported unless it is in one of the following forms:

```
wait until                                clock = VALUE;  
wait until      clock'event  and clock = VALUE;  
wait until not clock'stable and clock = VALUE;
```

VALUE is '0', '1', or an enumeration literal whose encoding is 0 or 1. A wait statement in this form is interpreted to mean “wait until the falling (VALUE is '0') or rising (VALUE is '1') edge of the signal named clock.” You cannot use wait statements in subprograms.

assert

Assert statements are ignored.

report

Report statements are ignored.

statement label

Statement labels are ignored.

signal

Guarded signal assignment is unsupported. The transport and after signals are ignored. Multiple waveform elements in signal assignment statements are unsupported.

variable

Variable statements are fully supported.

procedure call

Type conversion on formal parameters is unsupported.

Assignment to single bits of vectored ports is unsupported.

if

The if statements are fully supported.

case

The case statements are fully supported.

loop

The for loops are supported, with two constraints: The loop index range must be globally static, and the loop body must not contain a wait statement. The while loops are supported, but the loop body must contain at least one wait statement. The loop statements with no iteration scheme (infinite loops) are supported, but the loop body must contain at least one wait statement.

next

Next statements are fully supported.

exit

Exit statements are fully supported.

return

Return statements are fully supported.

null

Null statements are fully supported.

Concurrent Statements

block

Guards on block statements are supported. Ports and generics in block statements are unsupported.

process

Sensitivity lists in process statements are ignored.

concurrent procedure call

Concurrent procedure call statements are fully supported.

concurrent assertion

Concurrent assertion statements are ignored.

concurrent signal assignment

The guarded keyword is supported. The transport keyword is ignored. Multiple waveforms are unsupported.

component instantiation

Type conversion on the formal port of a connection specification is unsupported.

generate

The generate statements are fully supported.

Predefined Language Environment

severity_level type

The severity_level type is unsupported.

time type

The time type is ignored if time variables and constants are used only in after clauses. In the following two code fragments, both the after clause and TD are ignored:

```
constant TD: time := 1.4 ns;  
X <= Y after TD;  
  
X <= Y after 1.4 ns;
```

now function

The now function is unsupported.

TEXTIO package

The TEXTIO package is unsupported.

predefined attributes

These predefined attributes are supported: base, left, right, high, low, range, reverse_range, ascending, and length. The event and stable attributes are supported only in the if and wait statements, as described in “wait Statements” on page 5-50.

VHDL Reserved Words

Table C-1 lists the words that are reserved for the VHDL language and cannot be used as identifiers:

Table C-1 VHDL Reserved Words

abs	exit	new	select
access		next	severity
after	file	nor	shared
alias	for	not	signal
all	function	null	sla
and			sll
architecture		of	sra
array	generate	on	srl
assert	generic	open	subtype
attribute	group	or	
	guarded	others	then
begin		out	to
block	if		transport
body	impure	package	type
buffer	in	port	
bus	inertial	postponed	unaffected
	inout	procedure	units
case	is	process	until
component		pure	use
configuration	label		
constant	library	range	variable
	linkage	record	
disconnect	literal	register	wait
downto	loop	reject	when
		rem	while
else	map	report	with
elsif	mod	return	
end		rol	xnor
entity	nand	ror	xor

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

- " B-25, B-25
- "&" (concatenation operator) 4-8
- "**" (exponentiation operator) 4-12
- "*" (multiplying operator) 4-11
- "*" function B-12, B-24
- "+" (adding operator) 4-8
- "+" (unary operator) 4-10
- "+" function B-11, B-24
- "/=" (relational operator) 4-6
- "/=" function B-14
- "/" (multiplying operator) 4-11
- "<=" function B-14
- "<" function B-14
- "=" (relational operator) 4-6
- "=" function B-14, B-26
- ">=" (relational operator) 4-6
- ">=" function B-14, B-25
- ">" (relational operator) 4-6
- ">" function B-14, B-25
- "-" (adding operator) 4-8
- "-" (unary operator) 4-10
- "-" function B-24
- "_" function B-11

A

- abs (absolute value operator) 4-12
- actual parameters
 - subprograms 2-26
- adder-subtractor (example) A-17
- adding operators 4-8
- aggregate target syntax 5-9
- aggregates C-12
- aggregates (array literals) 4-18
- aggregates, record 3-14
- algorithms
 - processes 2-19
- alias declarations
 - supported C-6
- and (logical operator) 4-3
- architecture 2-5
 - dataflow
 - two-input NAND gate 2-34
 - defined 2-5
 - overriding entity port names 2-8
 - RTL
 - two-input NAND gate 2-34
 - statement, entity 2-5
 - structural
 - two-input NAND gate 2-33
- arithmetic functions
 - numeric_std

- binary B-23
 - unary B-23
- arithmetic operators
 - adding 4-8
 - multiplying 4-11
 - negating 4-10
- arithmetic optimization
 - considering overflow from carry bits 8-10
 - introduction 8-6
- array attributes
 - RANGE
 - example 5-28
- array data type
 - attributes
 - high 3-12
 - index 3-12
 - left 3-12
 - length 3-12
 - low 3-12
 - predefined 3-12
 - range 3-12
 - reverse_range 3-12
 - right 3-12
 - using 3-12
- concatenating 4-9
- constrained
 - array_type_name 3-10
 - defining 3-10
 - illustration 3-10
 - index 3-10
 - syntax 3-10
- definition of 3-9
- index
 - constrained 3-9
- ordering 4-6
- unconstrained
 - advantages 3-11
 - array_type_name 3-11
 - defining 3-11
 - element_type_name 3-11
 - range_type_name 3-11
 - syntax 3-11

- array literals
 - as aggregates 4-18
 - as bit strings 4-28
- array_type_name 3-10, 3-11
- arrival time 8-8
- assert statement C-13
- assignment statement
 - aggregate target 5-9
 - field target 5-8
 - indexed name target 5-4
 - signal
 - syntax 5-12
 - simple name target 5-3
 - slice target 5-7
 - syntax 5-2
 - variable
 - syntax 5-11
- async_set_reset attribute 7-5
- async_set_reset_local attribute 7-5
- async_set_reset_local_all attribute 7-5
- asynchronous designs
 - optimization 8-37
 - using 8-23
- asynchronous processes 6-3
- attribute declarations C-6
- attributes
 - array 3-12
 - as operands 4-20
 - ENUM_ENCODING B-17
 - synthesis_off 7-8
 - synthesis_on 7-8
 - VHDL
 - ENUM_ENCODING 3-5
 - ENUM_ENCODING values 3-7
- ATTRIBUTES package B-2, B-31

B

- binary arithmetic functions
 - example B-11
 - numeric_std B-23

- binary bit string 4-28
- bit string literals 4-28
- BIT type 3-17
- bit vectors
 - as bit strings 4-28
- bit width (of operands) 4-15
- BIT_VECTOR type 3-17, B-4
- block 2-17
- block statement
 - block_declarative_item 6-10
 - edge-sensitive latch 6-13
 - guarded 6-10
 - guarded blocks 6-12
 - level-sensitive latch 6-12
 - nested blocks 6-11
- block statements
 - guards C-15
- block_declarative_item
 - entity architecture 2-6
 - in block statement 6-10
- body
 - subprogram 2-23
- Boolean reduction functions B-30
- BOOLEAN type 3-17
- buffer
 - port mode 2-4, 2-24, 2-25
- built_in directive
 - logic functions B-18
 - type conversion B-19
 - using B-17
- built_in pragma
 - example of using B-17
- bus resolution function 6-9
- bused clock
 - syntax 7-22

C

- carry-lookahead adder (example) A-32
- carry-out bit
 - example of using B-13

- case statement
 - invalid usages 5-21
 - syntax 5-17
- character literals 4-26
- character string literals 4-28
- CHARACTER type 3-17
- clock, bused 7-22
- combinational feedback
 - paths 8-35
- combinational logic 8-2
- combinational processes 5-55, 6-5
- common subexpressions
 - sharing 8-12
- comparison functions
 - numeric_std B-24
- compiler directives 5-45
- component
 - declaration
 - generic parameter 2-11
 - N-bit adder 2-11
 - port name and order 6-23
 - two-input AND gate, example 2-11
- implication
 - directives 5-46
 - example 5-47
 - latches and registers 5-55
 - three-state driver 7-59
- instantiation
 - defined 2-17
 - direct 6-25
 - port map 6-23
 - search order 2-13
 - statement 2-13, 6-22
 - mapping subprogram to 5-45
- component declarations C-6
- component implication
 - registers 7-1
- computable operands 4-16
- concatenation operator 4-9
- concurrent procedure call
 - equivalent process 6-14

- syntax 6-14
- concurrent signal assignment 6-17
 - conditional signal assignment 6-18
 - selected signal assignment 6-20
- concurrent statement
 - block 2-17
 - component instantiation 2-17
 - procedure call 2-17
 - signal assignment 2-18
 - supported C-15
- conditional signal assignment
 - equivalent process 6-19
 - syntax 6-18
- conditionally assigned variable 7-19
- conditionally specified signal 8-36
- constant declaration
 - defined 2-18
 - supported C-5
 - value 2-18
- constant propagation 8-21
- constants
 - record aggregates 3-14
- constrained data array 3-10
- constructs, VHDL
 - architecture 2-5
 - constant declaration 2-18
 - subtype declaration 2-32
 - type declaration 2-32
 - variable declaration 2-20, 2-31
- block
 - constant declaration 2-18
 - subtype declaration 2-32
 - type declaration 2-32
- component instantiation 2-13
- declaration
 - constant 2-18
 - signal 2-21
 - variable 2-20, 2-31
- entity
 - constant declaration 2-18
 - defined 2-2
 - subtype declaration 2-32
 - type declaration 2-32
- operator
 - overloading 2-30
- package
 - constant declaration 2-18
 - subtype declaration 2-32
 - type declaration 2-32
- process
 - constant declaration 2-18
 - defined 2-19
 - subtype declaration 2-32
 - type declaration 2-32
- signal
 - bus resolution function 6-9
 - resolution function 2-40
- subprogram
 - constant declaration 2-18
 - function 2-22, 2-24
 - overloading 2-29
 - procedure 2-22, 2-23
 - subtype declaration 2-32
 - type declaration 2-32
- subtype
 - declaration 2-32
 - defined 2-32
- variable
 - declaration 2-20, 2-31
- control unit (example)
 - counting A-29
 - state machine A-24
- CONV_INTEGER functions B-8
- CONV_SIGNED functions B-9
- CONV_UNSIGNED functions B-8
- conversion functions
 - arithmetic
 - binary B-11
 - for adders and subtracters B-10
 - unary B-12
- numeric_std
 - TO_INTEGER B-22
 - TO_SIGNED B-22

- TO_UNSIGNED B-22
- std_logic_arith package B-8
- count zeros (example)
 - combinational A-19
 - sequential A-22
- COUNTER3
 - description
 - structural design 2-15
- critical path 8-8

D

- data type
 - abstract
 - BOOLEAN 3-1
 - advantages 3-2
 - array
 - constrained 3-10
 - syntax 3-10
 - array attributes
 - high 3-12
 - index 3-12
 - left 3-12
 - length 3-12
 - low 3-12
 - range 3-12
 - reverse_range 3-12
 - right 3-12
 - BIT 3-18
 - BIT_VECTOR 3-19
 - BOOLEAN 3-18
 - CHARACTER 3-18
 - described 3-1
 - enumeration syntax 3-3
 - hardware-related BIT 3-1
 - integer
 - defined 3-19
 - syntax 3-8
 - new type defined
 - BYTE, example 3-2
 - predefined
 - STANDARD package 3-1

- record 3-13
- subtype
 - defined 3-3
 - syntax 2-32
- supported C-4
- SYNOPSIS
 - std_logic_signed 3-9
 - std_logic_unsigned 3-9
- data types
 - numeric_std
 - SIGNED B-22
 - UNSIGNED B-22
- dataflow architecture
 - NAND2 entity 2-34
- declaration
 - constant 2-18
 - example 2-18
 - incorrect use of port name example 2-9
 - signal
 - example 2-21
 - incorrect use of port name example 2-9
 - logical 4-4
 - subprogram
 - function syntax 2-24
 - procedure syntax 2-23
 - subtype 2-32
 - supported C-5
 - variable
 - defined 2-20, 2-31
 - example 2-20, 2-31
- definitions
 - register inference 7-1
- design architecture
 - concurrent statement 2-17
 - block 2-17
 - block_declarative_item 2-6
 - component instantiation 2-13, 2-17
 - procedure call 2-17
 - process 2-17, 2-19
 - signal assignment 2-18
- declaration section
 - component 2-10

- constant 2-10
- signal 2-10
- subprogram 2-10
- type 2-10
- design units
 - package 2-35, 2-36
 - subprogram 2-22
 - organization, illustrated 2-5
- Design Compiler
 - asynchronous designs 8-23
- design style
 - data type
 - enumeration 3-3
 - integer 3-8
 - data types 3-2
- design unit
 - package 2-35
 - supported C-3
- designs
 - efficiency 8-22
 - structure 8-3
- direct component instantiation 6-25
- directives
 - built_in
 - identifying B-6
 - using B-17
 - component implication 5-46
 - map_to_entity 5-45, 6-14
 - resolution_method 2-41
 - return_port_name 5-45
 - synthetic 9-2
 - translate_off, warning 9-3
 - translate_on, warning 9-3
 - using 2-42
- dont care inference
 - example 8-29
 - simulation versus synthesis 8-33
 - using 8-32

E

- edge expression 7-58

- element_type_name 3-11
- encoding
 - values
 - ENUM_ENCODING attribute 3-7
 - vectors
 - ENUM_ENCODING attribute 3-6
- entity
 - architecture
 - defined 2-2
 - syntax 2-5
 - three-bit counter 2-7
 - two-input NAND gate 2-34
 - composition 2-2
 - consistency
 - component instantiation 2-14
 - defined 2-2
 - generic specification
 - example 2-5
 - syntax 2-3
 - port specification
 - overriding port names 2-8
 - port modes 2-4, 2-24, 2-25
 - syntax 2-4
 - specification
 - NAND2 gate 2-3, 2-34
 - three-bit counter 2-7
- ENUM_ENCODING attribute 3-5, B-17
 - values 3-7
 - vectors 3-7
- enumerated types
 - ordering 4-6
- enumeration data type
 - encoding
 - ENUM_ENCODING attribute 3-5
 - ENUM_ENCODING value 3-7
 - example 3-5
 - literal value 3-4
- example
 - COLOR 3-4
 - encoding 3-5
 - MY_LOGIC 3-4
 - literal, overloaded 3-4

- syntax 3-3
- enumeration literals 4-27
- equality functions
 - example B-14
- equality operators 4-6
- escaped identifier. *See* extended identifier
- examples
 - adder-subtractor A-17
 - asynchronous design
 - incorrect 8-27
 - carry-lookahead adder A-32
 - case statement
 - enumerated type 5-18
 - combinational process 6-5, 6-6
 - component implication 5-47
 - control unit
 - counting A-29
 - state machine A-24
 - count zeros
 - combinational A-19
 - sequential A-22
 - dont care usage 8-29
 - enumeration encoding
 - dont care 8-29
 - for ... generate 6-28
 - function call 5-42
 - if statement 5-15
 - integer data type
 - definitions 3-8
 - Mealy finite state machine A-5
 - Moore finite state machine A-2
 - PLA A-51
 - ROM A-7
 - sequential processes 6-6
 - serial-to-parallel converter
 - counting bits A-40
 - shifting bits A-47
 - simulation driver 9-3
 - subprograms
 - component implication 5-47
 - declarations 5-36
 - function call 5-42

- synchronous design 8-23
- three-state component
 - registered input 7-67
- two-phase clocked design 7-20
- wait statement 5-56
 - in a loop 5-52
 - multiple waits 5-52
- waveform generator
 - complex A-13
 - simple A-10
- exit statement 5-33
- exponentiation operator 4-12
- expression tree 8-7
 - subexpressions in 8-9
- expressions
 - described 4-1
 - relational
 - true 4-7
 - supported C-12
 - use 4-1
 - using parentheses in 8-9
- expressions, VHDL, tick (') 4-29
- extended identifier C-8

F

- falling_edge 7-22, 7-28
- feedback paths 8-35
- field target syntax 5-8
- file declarations C-5
- flip-flop
 - definition 7-1
 - inference 7-22
- for ... generate statement
 - example 6-28
 - syntax 6-26
- for ... loop statement
 - and exit statement 5-33
 - arrays 5-28
 - label as identifier in 5-25, 6-27
 - syntax 5-23, 5-25

- formal parameters
 - subprograms 2-26
- fully specified
 - signal 8-35
 - variable 8-35
- function
 - call 4-22, 5-41
 - declaration
 - syntax 2-24
 - resolution
 - allowed 2-41
 - bus 6-9
 - creating 2-40
 - directives, using 2-41
 - example 2-42
 - marking 2-41
 - signal 2-40
 - syntax, declaration 2-40
 - syntax, subtype 2-41
 - syntax, type 2-40
 - value 2-22
- functions
 - description 5-38
 - implementations
 - mapped to component 5-47
 - mapped to gates 5-49
 - return statement 5-43

G

- generate statements
 - for ... generate 6-26
 - if ... generate 6-26
- generic
 - map
 - component instantiation 2-13
 - parameter
 - component declaration 2-11
 - two-input AND gate 2-11
 - specification
 - entity 2-3
 - entity syntax 2-3

- values
 - mapping 2-14
- guard
 - on block statement C-15
- guarded blocks
 - in block statement 6-12
- guarded keyword C-15

H

- hdlin_pragma_keyword variable 9-2
- hexadecimal bit string 4-28
- high attribute 3-12
- high impedance state 7-59

I

- identifier C-8
 - extended C-9
- identifiers
 - defined 4-23
 - enumeration literals 4-27
- if ... generate statement
 - syntax 6-31
- if statement
 - creating registers 7-23
- implying registers 7-1
- incompletely specified 8-35
- indexed name target 5-4
- indexed names
 - computability 4-25
 - using 4-24
- inequality operators 4-6
- inference report
 - example 7-3
- inferred registers
 - limitations 7-57
- instantiation
 - component
 - direct 6-25
- integer data type

- bits, accessing
 - std_logic_signed package 3-9
 - std_logic_unsigned package 3-9
- defining 3-8
- definitions
 - example 3-8
- encoding 3-8
- INTEGER type 3-17
- subrange 3-8

K

- keywords C-17

L

- language constructs, VHDL
 - concurrent statements
 - assertion C-15
 - block 2-17, C-15
 - component instantiation 2-13, 2-17, C-15
 - function 2-22, 2-24
 - generate C-15
 - procedure 2-22, 2-23
 - procedure call 2-17, C-15
 - process 2-17, 2-19, C-15
 - signal assignment 2-18, C-15
 - data types
 - access C-4
 - array 3-10, C-4
 - enumeration 3-3, C-4
 - file C-5
 - floating C-4
 - incomplete type declarations C-5
 - integer 3-8, C-4
 - physical C-4
 - record C-4
 - subtype 2-32
 - dataflow
 - entity, NAND2 2-34
 - declaration
 - constant 2-18

- signal 2-21
 - variable 2-20, 2-31
- declarations
 - alias C-6
 - attribute C-6
 - component C-6
 - constant C-5
 - file C-5
 - interface C-5
 - shared variable C-5
 - signal C-5
 - variable C-5
- design units
 - architecture 2-5, C-3
 - configuration 2-34, C-3
 - entity C-3
 - entity, NAND2 2-3
 - library C-3
 - package 2-35, 2-36, C-3
 - subprogram C-3
 - subprogram, overloading 2-29
- expressions
 - aggregate C-12
 - allocator C-12
 - based literal C-12
 - function call C-12
 - null literal C-12
 - physical literal C-12
 - static expression C-12
 - string C-12
 - type conversion C-12
 - universal expression C-13
- names
 - attribute C-7
 - indexed C-7
 - operator symbol C-7
 - selected C-7
 - simple C-7
 - slice C-7
- operands
 - aggregate C-12
 - allocator C-12

- based literal C-12
- function call C-12
- null literal C-12
- physical literal C-12
- static expression C-12
- string C-12
- type conversion C-12
- universal expression C-13
- operators
 - addition C-9
 - logical C-9
 - miscellaneous C-10
 - multiplying C-9
 - overloading 2-30, C-10
 - relational C-9
 - short-circuit operation C-10
 - signing C-9
- predefined language environment
 - now function C-16
 - predefined attributes C-16
 - severity_level type C-16
 - TEXTIO package C-16
 - time type C-16
- reserved words C-17
- sequential statements
 - assertion C-13
 - case C-14
 - exit C-14
 - if C-14
 - loop C-14
 - next C-14
 - null C-14
 - procedure call C-14
 - report C-13
 - return C-14
 - signal C-13
 - statement labels C-13
 - variable C-13
 - wait C-13
- specifications
 - attribute C-6
 - configuration C-6
 - disconnection C-6
- latch
 - definition 7-1
- latch inference
 - local variables 7-11
- latches
 - edge-sensitive
 - not in guarded block statement 6-13
 - level-sensitive
 - guarded block statement 6-12
- left attribute 3-12
- length attribute 3-12
- literal
 - enumeration
 - character, defined 3-3
 - identifier, defined 3-3
- literals
 - as operands 4-26
 - bit strings 4-28
 - character 4-26
 - character string 4-28
 - enumeration 4-27
 - numeric 4-26
 - string 4-28
- logic
 - combinational 8-2
- logical operators 4-3
- loop statement 5-22
 - syntax 5-23
- low attribute 3-12

M

- map_to_entity directive 5-45, 6-14
- mapping
 - generic values
 - example 2-14
 - instantiation 2-14
 - port connections
 - example 2-15
 - expressions 2-15

- Mealy finite state machine (example) A-5
- mod (multiplying operator) 4-11
- Moore finite state machine (example) A-2
- multiple driven signals 6-8
- multiplication using shifts B-16
- multiplying operators 4-11

N

- names C-7
 - attributes 4-20
 - field names 4-30
 - qualified 4-30
 - record names 4-30
 - slice names 4-32
- nand (logical operator) 4-3
- NAND2 entity
 - syntax
 - dataflow architecture 2-34
 - RTL architecture 2-34
 - specification 2-3
 - structural architecture 2-33
- NATURAL subtype 3-17
- N-bit adder
 - declaration
 - example 2-11
- nested blocks
 - in block statement 6-11
- netlist
 - defined 2-13
- next statement
 - in named loops 5-32
- noncomputable operands 4-16
- nor (logical operator) 4-3
- not (logical operator) 4-3
- null range 4-33
- null slice 4-33
- null statement 5-58
- numeric literals 4-26
- numeric_std package

- " B-25, B-25
- "*" function B-24
- "+" function B-24
- "/=" equality function B-26
- "=" equality function B-26
- ">=" ordering function B-25
- ">" ordering function B-25
- "-" function B-24
- accessing B-21
- arithmetic functions
 - binary B-23
 - binary example B-24
 - unary B-23
 - unary example B-24
- comparison functions
 - equality B-26
 - ordering B-25
- conversion functions
 - TO_INTEGER B-22
 - TO_UNSIGNED B-22
 - UNSIGNED B-22
- data types
 - SIGNED B-22
 - UNSIGNED B-22
- IEEE documentation B-1
- location B-21
- logical operators
 - AND B-26
 - NAND B-26
 - NOR B-26
 - NOT B-26
 - OR B-26
 - XNOR B-26
 - XOR B-26
- report_design_lib command B-21
- resize function B-23
- rotate functions B-28
- rotate operators B-28
- shift functions
 - ROTATE_LEFT B-27
 - ROTATE_RIGHT B-27
 - SHIFT_LEFT B-27

- SHIFT_RIGHT B-27
- shift operators B-28
- unsupported components B-21
- use with std_logic_arith package B-20

O

- octal bit string 4-28
- one_cold attribute 7-7
- one_hot attribute 7-7
- operands
 - aggregates 4-18
 - attributes 4-20
 - bit width 4-15
 - computable 4-16
 - defined 4-14
 - field 4-30
 - function call 4-22, 5-41
 - identifiers 4-23
 - in expressions
 - defined 4-1
 - grouping 4-5
 - integer
 - predefined operators 4-8
 - literal 4-26
 - character 4-26
 - enumeration 4-27
 - numeric 4-26
 - string 4-28
 - noncomputable 4-16
 - qualified expressions 4-29
 - record 4-30
 - slice names 4-32
 - supported C-12
 - type conversions 4-34
- operators
 - absolute value 4-12
 - adding 4-8
 - arithmetic
 - adding 4-8
 - multiplying 4-11
 - negation 4-10
 - array
 - catenation 4-9
 - relational 4-6
 - catenation 4-9
 - described 4-2
 - equality 4-6
 - exponentiation 4-12
 - in expressions 4-1
 - logical 4-3
 - multiplying
 - predefined 4-11
 - restrictions on use 4-11
 - ordering 4-6
 - and array types 4-6
 - and enumerated types 4-6
 - overloading 2-30
 - defined 2-30
 - examples 2-30
 - precedence 4-3
 - predefined 4-2
 - relational
 - described 4-5
 - std_logic_arith package 4-7
 - rotate C-11
 - numeric_std B-28
 - shift C-10
 - numeric_std B-28
 - sign 4-10
 - supported C-9
 - unary 4-10
 - xnor C-11
- optimization
 - arithmetic expressions 8-6
 - NAND2 gate 2-33
- or (logical operator) 4-3
- ordering
 - operators 4-6
- ordering functions
 - example B-14
- others (in aggregates) 4-20
- others (in case statement) 5-17
- overflow characteristics

- arithmetic optimization 8-10
- overloading
 - enumeration
 - literal 3-4
 - enumeration literals 4-27
 - operators 2-30
 - defined 2-30
 - resolving by qualification 4-30
 - subprograms 2-29
 - defined 2-29

P

- package
 - body syntax 2-39
 - component declaration in 2-35
 - constant declaration in 2-35
 - declaration
 - example 2-38
 - syntax 2-37
 - defined 2-35
 - numeric_std
 - IEEE documentation B-1
 - package_body_declarative_item 2-39
 - STANDARD 3-17
 - std_logic_arith 4-7
 - std_logic_signed 3-9
 - std_logic_unsigned 3-9
 - structure
 - body 2-36
 - declaration 2-36
 - subprogram in 2-35
 - TEXTIO 3-16
 - type declaration in 2-35
 - use statement syntax 2-36, 2-38
- package_body_declarative_item 2-39
- package_declarative_item 2-37
- package_name 2-37
- packages
 - Synopsys-supplied B-1
- parameters, subprogram
 - actual 2-26
 - formal 2-26
- PLA (example) A-51
- port
 - as signal 2-21
 - connections, mapping example 2-15
 - map 2-13
 - mode
 - buffer 2-4, 2-24
 - entity port specification 2-4, 2-24, 2-25
 - in 2-4, 2-24, 2-25
 - inout 2-4, 2-24
 - out 2-4, 2-24
 - name
 - consistency among entities 2-9, 2-12
 - incorrect use 2-9
 - type
 - consistency among components 2-12
- POSITIVE subtype 3-17
- pragma keyword comment
 - hdlin_pragma_keyword variable 9-2
- pragmas. *See* directives
- predefined attributes
 - array 3-12
 - supported C-7
- predefined attributes, supported C-16
- predefined language environment C-16
- predefined VHDL operators 4-3
- procedure
 - call (defined) 2-17
 - call syntax 5-39
 - subprogram declaration syntax 2-23
 - subprogram description 5-38
- process
 - as algorithm 2-19
 - declaration 2-19
 - defined 2-19
 - description 2-19
 - sequential statements in 2-19
- process statement 6-2, 6-10
- processes
 - asynchronous 6-3

- combinational
 - example 6-5
- combinational logic 5-55
- sensitivity lists 6-3
- sequential
 - example 6-6
- sequential logic 5-55
- synchronous 6-3
- wait statement 5-50

Q

- qualified expressions 4-29

R

- range attribute 3-12
- range_type_name 3-11
- record aggregates 3-14
- record data type 3-13
- record operands 4-30
- records
 - as aggregates C-12
- register
 - definition of 7-1
 - inference 7-1
- register inference
 - attribute
 - async_set_reset 7-5
 - async_set_reset_local 7-5
 - async_set_reset_local_all 7-5
 - one_cold 7-7
 - one_hot 7-7
 - sync_set_reset 7-6
 - sync_set_reset_local 7-6
 - sync_set_reset_local_all 7-6
- D latch 7-10
- definition 7-1
- edge expressions 7-22
- if statement 7-23
- if versus wait 7-23

- signal edge 7-22
- SR latch 7-8
- templates 7-3
- wait statement 7-22
- wait versus if 7-23
- relational operators 4-5
- rem (multiplying operator) 4-11
- report statement C-13
- reserved words C-17
- resize function
 - numeric_std B-23
- resolution function
 - allowed 2-41
 - creating 2-40
 - directive, using 2-41
 - directives
 - resolution_method three_state 2-41
 - resolution_method wired_and 2-41
 - resolution_method wired_or 2-41
 - example 2-42
 - marking 2-41
 - signal 2-40
 - syntax
 - declaration 2-40
 - subtype 2-41
 - type 2-40
- resolution functions
 - bus 6-9
- resolution_method
 - three_state directive 2-41
 - wired_and directive 2-41
 - wired_or directive 2-41
- resolved signal
 - creating 2-42
 - example 2-42
 - subtype declaration 2-40
 - syntax 2-41
 - using 2-42
- return statement 5-43
- return_port_name directive 5-45
- reverse_range attribute 3-12

- right attribute 3-12
- rising_edge 7-22, 7-26
- ROM (example) A-7
- rotate functions
 - numeric_std B-28
- rotate operators C-11
 - numeric_std B-28
- RTL Analyzer
 - architecture
 - NAND2 entity 2-34

S

- selected signal assignment
 - equivalent process 6-22
 - syntax 6-20
- sensitivity lists 6-3
- sequential processes 5-55, 6-6
- sequential statement
 - if, syntax 5-15
- sequential statements
 - supported C-13
- serial-to-parallel converter (example)
 - counting bits A-40
 - shifting bits A-47
- shared variable C-5
- sharing
 - common subexpressions
 - automatically determined 8-12
- shift functions
 - example B-15
 - numeric_std B-27
- shift operations
 - example B-16
- shift operators C-10
 - numeric_std B-28
- signal
 - as port 2-21
 - assignment 2-18
 - examples 5-11, 5-13
 - syntax 5-12
 - declaration 2-21
 - example 3-4
 - logical 4-4
 - in package 2-37
 - multiple drivers
 - bus 6-9
 - resolution function 2-40
 - resolved 2-40
- signals
 - concurrent signal assignment 6-17
 - conditional signal assignment 6-18
 - drivers 6-8
 - edge detection 7-22
 - registering 7-54
 - selected signal assignment 6-20
 - supported C-5
 - three-state 6-8
- SIGNED data type
 - defined B-6, B-7
 - std_logic_arith package B-4
- SIGNED data types
 - numeric_std package B-22
- simple name target 5-3
- simulation
 - dont care values 8-33
 - driver example 9-3
- slice names
 - limitations 4-33
 - syntax 4-32
- slice target syntax 5-7
- specifications C-6
- STANDARD package 3-17
- state machine (example)
 - controller A-24
 - Mealy A-5
 - Moore A-2
- statement
 - assignment
 - aggregate target, syntax 5-9
 - field target, syntax 5-8
 - indexed name target, syntax 5-4

- slice target, syntax 5-7
- case
 - enumerated type 5-18
 - invalid usages 5-21
 - syntax 5-17
- concurrent
 - block 2-17
 - component instantiation 2-17
 - procedure call 2-17
 - process 2-17, 2-19
 - signal assignment 2-18
- for ... loop
 - syntax 5-25
- loop
 - syntax 5-23
- loop syntax 5-22
- sequential
 - assignment, syntax 5-2
 - if 5-15
 - while ... loop syntax 5-24
- statement labels C-13
- std_logic_1164 package B-2
- std_logic_arith package B-2, B-3
- _REDUCE functions B-30
- "*" function B-12
- "+" function B-11
- "/=" function B-14
- "<=" function B-14
- "<" function B-14
- "=" function B-14
- ">=" function B-14
- ">" function B-14
- "—" function B-11
- arithmetic functions B-10
- Boolean reduction functions B-30
- built_in functions B-6
- comparison functions B-13
- CONV_INTEGER functions B-8
- CONV_SIGNED functions B-9
- CONV_UNSIGNED functions B-8
- conversion functions B-10
- data types B-6
 - modifying the package B-5
 - ordering functions B-14
 - shift function B-15
 - using the package B-4
- std_logic_misc package B-30
- std_logic_signed package 3-9
- std_logic_unsigned package 3-9
- string literals 4-28
 - bit 4-28
 - character 4-28
- STRING type 3-17
- structural architecture
 - NAND2 entity 2-33
- structural design
 - component
 - instantiation statement 2-13
 - description
 - COUNTER3 2-15
- subexpressions in expression tree 8-9
- subprogram
 - body
 - calls, examples 2-26
 - examples 2-29
 - function syntax 2-28
 - procedure syntax 2-26
 - declaration
 - examples 2-25
 - function syntax 2-24
 - overloading 2-29
 - procedure, syntax 2-23
 - syntax 2-28
 - overloading
 - defined 2-29
 - examples 2-29
 - parameter 2-26
 - profile 2-29
 - sequential statement 2-22
- subprograms
 - calling 5-37
 - defined 5-35
 - defining 5-36
 - mapping to components

- example 5-47
 - matching entity 5-45
- procedure versus function 5-38
- subrange
 - integer data type 3-8
- subtype data type
 - declaration 2-32
 - defining 3-21
- SYN_FEED_THRU
 - example of using B-20
- sync_set_reset attribute 7-6
- sync_set_reset_local attribute 7-6
- sync_set_reset_local_all attribute 7-6
- synchronous
 - designs 8-23
 - example 8-23
 - processes 6-3
- synopsys keyword comment
 - hdlin_pragma_keyword variable 9-2
- Synopsys packages B-1
 - std_logic_misc package B-30
- Synopsys-defined package
 - std_logic_arith 3-21
 - std_logic_signed
 - integers 3-9
 - overload for arithmetic 3-9
 - std_logic_unsigned
 - integers 3-9
 - overload for arithmetic 3-9
- syntax
 - array data type
 - constrained 3-10
 - unconstrained 3-11
 - assignment statement
 - aggregate target 5-9
 - field target 5-8
 - indexed name target 5-4
 - signal 5-2, 5-12
 - simple name target 5-3
 - slice target 5-7
 - variable 5-2, 5-11
 - bused clock 7-22
 - case statement 5-17
 - clock, bused 7-22
 - component
 - declaration statement 2-10
 - instantiation statement 2-13
 - constant declaration 2-18
 - enumeration data type 3-3
 - for ... loop statement 5-25
 - generic_declaration 2-3
 - if statement 5-15
 - integer data type 3-8
 - loop statement 5-22, 5-23
 - NAND2
 - dataflow architecture 2-34
 - RTL architecture 2-34
 - specification 2-3
 - structural architecture 2-33
 - operator
 - overloading 2-30
 - package body 2-39
 - resolution function
 - declaration 2-40
 - subtype 2-41
 - type 2-40
 - signal declaration 2-21
 - subprogram
 - overloading 2-29
 - subprogram declaration
 - body, examples 2-29
 - body, function syntax 2-28
 - function 2-24
 - procedure 2-23
 - procedure body 2-27
 - subtype 2-32
 - type
 - declaration 2-30
 - use statement, package 2-36
 - variable declaration 2-20, 2-31
 - while ... loop statement 5-24
- synthetic comments
 - hdlin_pragma_keyword variable 9-2

synthetic comments. *See* directives

T

target

- signal assignment syntax 5-3
- variable assignment syntax 5-2

TEXTIO package 3-16

three-bit counter

- circuit description
- entity architecture 2-7
- entity specification 2-7

three-state

- gate 7-65
 - registered enable 7-67
 - without registered enable 7-68
- inference 7-59
- registered drivers 7-65, 7-67
- registered input 7-67
- signals 6-8

tick (') in VHDL expressions 4-29

time type C-16

TO_INTEGER function

- conversion
- numeric_std B-22

TO_SIGNED function

- conversion
- numeric_std B-22

TO_UNSIGNED function

- conversion
- numeric_std B-22

translate_off directive, warning 9-3

translate_on directive, warning 9-3

transport keyword C-15

two-input AND gate

- component declaration example 2-11

two-input NAND gate

- dataflow architecture syntax 2-34
- RTL architecture syntax 2-34
- specification syntax 2-3
- structural architecture 2-33

two-input N-bit comparator

- example 2-5

two-phase design 7-20

type

- conversion
- syntax 4-34

types

- as aggregates C-12

U

unary arithmetic functions

- example B-12
- numeric_std B-23

unary operators

- sign 4-10

unconstrained arrays

- example using A-17

unconstrained data array 3-9, 3-10

UNSIGNED data type

- defined B-6
- std_logic_arith package B-4

UNSIGNED data types

- numeric_std package B-22

use statement 2-36

V

variable

- assignment
 - examples 5-11, 5-13
 - syntax 5-2, 5-11
- declaration 2-20, 2-31
 - defined 2-20, 2-31
 - example 3-4
 - initializing 2-20, 2-31

variables

- conditionally-assigned 7-19
- hdlin_pragma_keyword 9-2

vectors

- encoding
 - ENUM_ENCODING attribute 3-6

VHDL

- aggregates 4-18
- architectures 2-5, 6-1
- array data type 3-9
- BIT data type 3-18
- BIT_VECTOR data type 3-19
- BOOLEAN data type 3-18
- case statement 5-17
- CHARACTER data type 3-18
- component
 - implication 5-46
 - instantiation 2-13
- concurrent procedure call 6-14
- concurrent statement
 - block 6-1
 - process 6-1, 6-2, 6-10
- concurrent statements
 - supported C-15
- configuration 2-34
- data type supported
 - enumeration C-4
- data type unsupported
 - integer C-4
- data type, supported
 - enumeration 3-3
 - integer 3-8
- data type, unsupported
 - access (pointer) types 3-20
 - file (disk file) types 3-20
 - floating-point 3-20
 - physical 3-20
- declarations C-5
- design units C-3
- directives 9-2
- enumeration data type 3-3
- errors in descriptions 8-38
- exit statement 5-33
- expressions, supported C-12
- for ... loop statement 5-23, 5-25, 6-27
- functions 2-22
- generate statement 6-26
- identifiers 4-23

- integer data type 3-8, 3-19
- keywords C-17
- literals 4-26
- names C-7
- NATURAL subtype 3-19
- null statement 5-58
- operands
 - categories 4-14
 - supported C-12
- operators
 - precedence 4-3
 - predefined 4-2
 - supported C-9
- package
 - composition 2-35
 - use statement syntax 2-36
- port modes 2-4, 2-24, 2-25
- POSITIVE subtype 3-19
- predefined attributes, supported C-7
- predefined data types 3-16
- predefined language environment C-16
- procedures 2-22
- process statement 6-2, 6-10
- qualified expressions 4-29
- record data type 3-13
- reserved words C-17
- return statement 5-43
- sensitivity lists 6-3
- sequential statements, supported C-13
- shorthand expressions 8-22
- specifications C-6
- STANDARD package 3-17
- STRING type 3-19
- subprograms 2-22, 5-35
- subtype data type 3-3, 3-21
- TEXTIO package 3-16
- three-state components 7-59
- type conversion 4-34
- wait statement 5-50

VHDL Analyzer

- in synthesis process 8-38

VHDL assertions 7-8

- VHDL Compiler
 - asynchronous designs 8-23
 - attributes
 - supported C-7
 - Synopsys C-7
 - ATTRIBUTES package 3-5
 - component
 - consistency 2-12
 - implication 5-46
 - instantiation, entities 2-14
 - design efficiency 8-22
 - design structure 8-3
 - directives 9-2
 - dont care information 8-29
 - entities
 - consistency 2-12
 - enumeration encoding 3-5
 - integer encoding 3-8
 - operators
 - supported C-9
 - port names
 - consistency 2-12
 - sensitivity lists 6-3

- source directives 9-1, 9-2
- syntax checking 8-38
- wait statement
 - limitations 5-54
 - usages 5-50

W

- wait statement 5-50
 - creating registers 7-22
 - example 5-56
 - multiple waits 5-52
 - while loop 5-52
- waveform generator (example)
 - complex A-13
 - simple A-10
- while ... loop statement syntax 5-24

X

- xnor (logical operator) 4-3
- xnor operator C-11
- xor (logical operator) 4-3