CoCentric™ SystemC Compiler
Behavioral Modeling Guide

Version 2000.11-SCC1, March 2001

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

SYNOPSYS®

Copyright Notice and Proprietary Information
Copyright © 2000 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and 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, AMPS, Arcadia, CMOS-CBA, COSSAP, Cyclone, DelayMill, DesignPower, DesignSource, DesignWare, dont_use, EPIC, ExpressModel, Formality, in-Sync, Logic Automation, Logic Modeling, Memory Architect, ModelAccess, ModelTools, PathBlazer, PathMill, PowerArc, PowerMill, PrimeTime, RailMill, Silicon Architects, SmartLicense, SmartModel, SmartModels, SNUG, SOLV-IT!, SolvNET, Stream Driven Simulator, Synopsys Eagle Design Automation, Synopsys Eagle*i*, Synthetic Designs, TestBench Manager, and TimeMill are registered trademarks of Synopsys, Inc.

Trademarks
ACE, BCView, Behavioral Compiler, BOA, BRT, CBA, CBAII, CBA Design System, CBA-Frame, Cedar, CoCentric, DAVIS, DC Expert, DC Expert *Plus*, DC Professional, DC Ultra, DC Ultra Plus, Design Advisor, Design Analyzer, Design Compiler, DesignTime, Direct RTL, Direct Silicon Access, dont_touch, dont_touch_network, DW8051, DWPCI, ECL Compiler, ECO Compiler, Floorplan Manager, FoundryModel, FPGA Compiler, FPGA Compiler II, FPGA *Express*, Frame Compiler, General Purpose Post-Processor, GPP, HDL Advisor, HDL Compiler, Integrator, Interactive Waveform Viewer, Liberty, Library Compiler, Logic Model, MAX, ModelSource, Module Compiler, MS-3200, MS-3400, Nanometer Design Experts, Nanometer IC Design, Nanometer Ready, Odyssey, PowerCODE, PowerGate, Power Compiler, ProFPGA, ProMA, Protocol Compiler, RMM, RoadRunner, RTL Analyzer, Schematic Compiler, Scirocco, Shadow Debugger, SmartModel Library, Source-Level Design, SWIFT, Synopsys EagleV, Test Compiler, Test Compiler Plus, Test Manager, TestGen, TestSim, TetraMAX, TimeTracker, Timing Annotator, Trace-On-Demand, VCS, VCS Express, VCSi, VERA, VHDL Compiler, VHDL System Simulator, Visualyze, VMC, and VSS are trademarks of Synopsys, Inc.

Service Marks
TAP-in 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.

Document Order Number: 37581-000 JB
CoCentric™ SystemC Compiler Behavioral Modeling Guide, v2000.11-SCC1

Contents

Preface

What's New in This Release xxiv

About This Guide xxvi

Customer Support xxx

1. Introduction

Defining Levels of Abstraction in System Design 1-3

Architectural Level 1-4

Untimed Functional Model 1-5

Timed Functional Model 1-5

Functional Coding Style 1-5

Behavioral Model 1-9

Behavioral Coding Style 1-10

Refining From Functional to Behavioral Model 1-10

Register Transfer Level Model 1-14

RTL Coding Style 1-15

Refining into RTL 1-15

Choosing the Right Abstraction for Synthesis 1-19

Identifying Attributes Suitable for Behavioral Synthesis. 1-19
Identifying Attributes Suitable for RTL Synthesis. 1-21
Comparison of Behavioral and RTL Synthesis 1-22

2. Refining for Behavioral Synthesis

Refinement Overview 2-3

Creating and Refining the Structure From a C/C++ Model 2-6
Define I/O Ports 2-6
Specify Internal Structure 2-6
Specify the Internal Communication 2-7
Specify the Detailed Architecture. 2-8
Atomic and Hierarchical Blocks 2-9
Modules 2-12
Module Header File 2-13
Module Ports 2-14
Internal Signals. 2-16
Reading and Writing Ports 2-17
Internal Data Variables. 2-18
Processes 2-20
Types of Processes 2-21
Creating a Process in a Module 2-22
Member Functions. 2-23
Module Constructor 2-24
Module Implementation File. 2-26
Using an Infinite Loop. 2-26

Refining the Structure From a High-Level SystemC Model. 2-28

Creating and Refining Processes 2-28
Converting to a Synthesizable Subset 2-30
Excluding Simulation-Specific Code 2-31
SystemC and C++ Synthesizable Subset 2-32
Nonsynthesizable Subset of SystemC 2-33
Nonsynthesizable C/C++ Constructs 2-34
Refining Data. 2-37
Synthesizable Data Types 2-37
Nonsynthesizable Data Types 2-38
Recommended Types for Synthesis. 2-39
Using SystemC Types 2-41
Bit and Bit Vector Data Type Operators 2-41
Fixed and Arbitrary Precision Data Type Operators. 2-42
Using Enumerated Types 2-43
Using Aggregate Data Types 2-43
Using C++ Types 2-43
Recommendations About Data Types 2-46
Refining Control. 2-47
Advanced Refinement Techniques 2-48
Refinement Recommendations 2-49
3. Behavioral Coding Guidelines
Using Clocked Thread Processes 3-2
Characteristics of the Clocked Thread Process. 3-2
Using the wait Statement 3-3
Using the wait_until Statement. 3-3

Controlling a Clocked Thread Process 3-4
Simple Clocked Thread Example. 3-4

Using Inputs and Outputs 3-6
Registered Outputs 3-6
Inputs and Outputs Within Cycles 3-6
Specifying I/O Read and Write. 3-7
Specifying I/O Cycles. 3-7
I/O Scheduling Modes 3-8
Cycle-Fixed Scheduling Mode 3-8
Superstate-Fixed Schedule Mode 3-8
Comparing I/O Scheduling Modes 3-9

Behavioral Coding Style Rules 3-10
Definition of Coding Rule Terms 3-10
General Coding Rules 3-11
Cycle-Fixed Mode Coding Rules 3-12
Superstate-Fixed Mode Coding Rules. 3-12
General Coding Rules Examples. 3-13
General Coding Rule 1. 3-13
General Coding Rule 2. 3-14
General Coding Rule 3. 3-15
General Coding Rule 4. 3-16
General Coding Rule 5. 3-20
Cycle-Fixed Mode Coding Rules Examples. 3-23
Cycle-Fixed Coding Rule 1. 3-23
Cycle-Fixed Coding Rule 2. 3-26
Cycle-Fixed Coding Rule 3. 3-28
Superstate-Fixed Mode Coding Rules Examples 3-29

Superstate-Fixed Coding Rule 1 3-29
Superstate-Fixed Coding Rule 2 3-30
Finding the Cause of Timing-Dependent Coding Errors 3-31
Using Conditional Statements 3-32
Using Loops 3-33
Understanding How Loops Are Scheduled 3-33
Labeling a Loop 3-34
Using while Loops 3-35
Using an Infinite while Loop 3-35
Using do...while Loops 3-36
Using for Loops 3-36
Rolled Versus Unrolled Loops 3-36
Rolled for Loops 3-37
Unrolling for Loops 3-37
Comparing Rolled and Unrolled Loops 3-39
Selectively Unrolling Loop Iterations 3-40
Ensuring a Statically Determinable Exit Condition 3-41
Consecutive Loops 3-42
Pipelining Loop Rules 3-45
Using Resets 3-46
Describing a Global Reset 3-46
Specifying the Reset Behavior 3-46
Specifying a Reset Implementation 3-48
Using Variables and Signals 3-49
Initializing Variables 3-49
Using Signals and Wait Statements 3-50

Using Variables and Wait Statements 3-52
Using Variables for Register Allocation Efficiency 3-53
Determining the Lifetime of Variables 3-54

4. Using Functions and DesignWare Components

Using Member Functions 4-2

Using Nonmember Functions 4-4

Using Preserved Functions 4-4
When to Preserve Functions 4-5
Preserved Function Restrictions 4-5
Creating Preserved Functions 4-6
Nonmember Preserved Functions 4-9
Using Reference Parameters in Preserved Functions. 4-10

Using DesignWare Components 4-11
Using map_to_operator 4-11
Guidelines for Using map_to_operator 4-12

5. Using Arrays, Register Files, and Memories

Using Arrays 5-2
Declaring Arrays 5-2
Reading From and Writing to Variable Arrays 5-3
Reading From and Writing to Signal Arrays. 5-4
Accessing Slices of an Array Location. 5-5

Array Implementations 5-7

Mapping Arrays to Register Files 5-8

Mapping All Arrays to Register Files 5-9
Mapping Specific Arrays to Register Files 5-9
Mapping Arrays to Memories 5-11
Local Memory 5-11
Multiple Arrays Accessing One Memory 5-13
Exploring Alternative Memory Types 5-14
Accessing Register Files and Memories Efficiently 5-15
Accessing Memory. 5-16
Allowing for Vendor Memory Timing 5-17
Eliminating Redundant Memory Accesses 5-18
Accessing Bit Slices of Memory Data 5-19

6. Using Handshaking in the Circuit and Testbench

Using Handshake Protocols 6-3
Using One-Way Handshake Protocols 6-4
One-Way Handshake Initiated From Behavioral Block 6-4
One-Way Handshake Initiated From Testbench 6-12
Constraining the Width of Handshake Strobes 6-19
Using Two-Way Handshake Protocols 6-21
Two-Way Handshake Initiated From Behavioral Block 6-21
Two-Way Handshake Initiated From Testbench 6-29
Fast Handshaking 6-36
Using if…else. 6-37
Using wait_until 6-39
Using a Pipeline Handshake Protocol. 6-40

Appendix A. Compiler Directives

Synthesis Compiler Directives A-2

line_label A-3

map_to_operator A-3

return_port_name A-4

preserve_function A-4

inout_param A-5

resource A-6

synthesis_off and synthesis_on A-7

translate_off and translate_on A-7

unroll A-8

C/C++ Compiler Directives A-9

C Line Label A-9

C Conditional Compilation A-9

Appendix B. First-In-First-Out Example

FIFO Description B-2

Architectural Model B-2

Behavioral Model B-6

Ports and Signals B-6

Behavioral Description B-8

Behavioral Testbench B-11

RTL Model B-15

RTL Description B-16

RTL Testbench B-20

Appendix C. Memory Controller Example

Memory Controller Description C-2
Commands C-2
Ports C-3
Communication Protocol C-4

Functional Simulation Model C-5

Refined Behavioral Model C-9
Data Types C-9
Communication Protocol C-9
Clock Placement C-10
Behavioral Model C-10

Appendix D. Fast Fourier Transform Example

FFT Description D-2
FFT Computation D-2
Refining From Functional to Behavioral D-3
Data Read Two-Way Handshake D-3
Data Write Two-Way Handshake D-3

FFT Functional Model D-4

FFT Behavioral Model D-9

FFT Testbench D-17

Appendix E. Inverse Quantization Example

IQ Description E-2
IQ Data Flow E-3

IQ Block Diagram. E-4
IQ Behavioral Model E-4

Appendix F. Expressions and Operations

Using Expressions. F-2
Operator Precedence F-3

Index

Figures

Figure 1-1 System Design Levels of Abstraction 1-3
Figure 1-2 Architectural Model 1-4
Figure 1-3 Behavioral Model 1-9
Figure 1-4 RTL Model 1-14
Figure 2-1 Refinement Stages and Activities 2-4
Figure 2-2 MPEG Decoder Functional Structure 2-7
Figure 2-3 MPEG Decoder Top-Level Architecture 2-8
Figure 2-4 MPEG Decoder Detailed Architecture 2-9
Figure 2-5 Module 2-12
Figure 2-6 Module Ports 2-14
Figure 2-7 Processes and Signals 2-16
Figure 3-1 Simple Multiplier I/O Protocol 3-7
Figure 3-2 Rolled and Unrolled for Loops 3-39
Figure 3-3 Loop Latency and Initiation Interval 3-45
Figure 3-4 Loop Exit 3-45
Figure 3-5 Comparing Signal Use and Data Flow 3-51

Figure 3-6 Variable Use and Data Flow 3-52
Figure 5-1 Register File Architecture 5-8
Figure 5-2 Multiple Array Address Space Mapping 5-13
Figure 5-3 Memory Access Time Specification 5-17
Figure 5-4 Bit Slice Accesses 5-20
Figure 6-1 One-Way Handshake Protocol 6-5
Figure 6-2 Testbench-Initiated One-Way Handshake 6-12
Figure 6-3 Constraining Input Handshake Signals. 6-19
Figure 6-4 Constraining Output Handshake Signals 6-20
Figure 6-5 Two-Way Handshake Protocol 6-22
Figure 6-6 Two-Way Handshake Protocol 6-29
Figure 6-7 Timing Diagram of while Loop 6-37
Figure 6-8 Timing Diagram Using if…else 6-38
Figure 6-9 Timing Diagram Using wait_until 6-39
Figure 6-10 Incorrect Loop Pipeline With Handshake 6-41
Figure 6-11 Correct Loop Pipeline With Extended Initiation Interval 6-42
Figure 6-12 Correct Loop Pipeline Without Handshake Signal De-assertion 6-44
Figure C-1 Behavioral Input Data Flow. C-10
Figure D-1 FFT Ports and Data Types D-2
Figure E-1 IQ Blocks E-2
Figure E-2 IQ Data Flow. E-3
Figure E-3 IQ Block Diagram E-4

Tables

Table 2-1 Nonsynthesizable SystemC Classes 2-33
Table 2-2 Nonsynthesizable C/C++ Constructs 2-34
Table 2-3 Synthesizable Data Types 2-39
Table 2-4 SystemC Bit and Bit Vector Data Type Operators 2-41
Table 2-5 SystemC Integer Data Type Operators. 2-42
Table A-1 SystemC Compiler Compiler Directives A-2
Table F-1 Operator Precedence F-4

Examples

Example 1-1 FIFO Functional Model 1-6
Example 1-2 FIFO Behavioral Coding 1-12
Example 1-3 RTL Coding 1-16
Example 2-1 Using read() and write() Methods 2-18
Example 2-2 Creating a Clocked Thread Process in a Module 2-22
Example 2-3 Module Constructor 2-25
Example 2-4 Module Behavior 2-27
Example 2-5 Basic Reset Action and Main Loop 2-29
Example 2-6 Excluding Simulation-Only Code 2-31
Example 2-7 Aggregate Data Type 2-43
Example 2-8 Implicit Bit Size Restriction 2-44
Example 2-9 Unknown Variable Bit Size 2-44
Example 2-10 Incorrectly Using a Data Member as a Variable 2-45
Example 2-11 Correct Use of Local Variables 2-45
Example 3-1 Infinite Loops 3-4
Example 3-2 Simple Clocked Thread Multiplier 3-5

Example 3-3 Error in Use of General Coding Rule 1 3-13
Example 3-4 Correct General Coding Rule 1. 3-13
Example 3-5 Error in Use of General Coding Rule 2 3-14
Example 3-6 Correct General Coding Rule 2. 3-14
Example 3-7 Error in Use of General Coding Rule 3 3-15
Example 3-8 Correct General Coding Rule 3. 3-15
Example 3-9 Error in Use of General Coding Rule 4, If Conditional 3-17
Example 3-10 Correct General Coding Rule 4, If Conditional 3-17
Example 3-11 Error in Use of General Coding Rule 4, If Conditional With Implied Else 3-18
Example 3-12 Correct General Coding Rule 4, If Conditional 3-18
Example 3-13 Error in Use of General Coding Rule 4, Switch Conditional 3-19
Example 3-14 Correct General Coding Rule 4, Switch Conditional . . 3-19
Example 3-15 Error in Use of General Coding Rule 5 3-21
Example 3-16 Correct General Coding Rule 5. 3-22
Example 3-17 Error in Use of Cycle-Fixed Mode Coding Rule 1, for Loop. 3-23
Example 3-18 Correct Cycle-Fixed Mode Coding Rule 1, for Loop . . 3-23
Example 3-19 Error in Use of Cycle-Fixed Mode Coding Rule 1, while Loop. 3-24
Example 3-20 Correct Cycle-Fixed Mode Coding Rule 1, while loop 3-24

Example 3-21 Error in Use of Cycle-Fixed Mode Coding Rule 1, do-while Loop 3-25
Example 3-22 Correct Cycle-Fixed Mode Coding Rule 1, do-while loop. 3-25
Example 3-23 Error in Use of Cycle-Fixed Mode Coding Rule 2 3-26
Example 3-24 Correct Cycle-Fixed Mode Coding Rule 2. 3-27
Example 3-25 Error in Use of Cycle-Fixed Mode Coding Rule 2, Write 3-27
Example 3-26 Correct Cycle-Fixed Mode Coding Rule 2, Write 3-28
Example 3-27 Error in Use of Cycle-Fixed Mode Coding Rule 3 3-29
Example 3-28 Correct Cycle-Fixed Mode Coding Rule 3. 3-29
Example 3-29 Error in Use of Superstate-Fixed Mode Coding Rule 1 3-30
Example 3-30 Correct Superstate-Fixed Mode Coding Rule 1 3-30
Example 3-31 Error in Use of Superstate-Fixed Mode Coding Rule 2 3-31
Example 3-32 Correct Superstate-Fixed Mode Coding Rule 2 3-31
Example 3-33 Operations That Are Not Mutually Exclusive. 3-32
Example 3-34 Mutually Exclusive Operations 3-32
Example 3-35 Labeling a Loop 3-34
Example 3-36 Structure of a while Loop 3-35
Example 3-37 Infinite while Loop 3-35
Example 3-38 Structure of do...while Loop 3-36
Example 3-39 for Loop. 3-36

Example 3-40 Unrolled for Loop Compiler Directive 3-37
Example 3-41 Unrolled for Loop and Its Execution 3-38
Example 3-42 When to Use unroll 3-40
Example 3-43 Selective Unrolling of a for Loop 3-41
Example 3-44 for Loop Without Static Exit Condition 3-42
Example 3-45 Consecutive Loops With Overhead 3-43
Example 3-46 Collapsed Consecutive Loops 3-44
Example 3-47 Global Reset Watching 3-47
Example 4-1 Member Function 4-3
Example 4-2 Creating Preserved Functions 4-7
Example 4-3 Nonmember Preserved Function Declaration 4-9
Example 4-4 Preserved Function With Reference Parameter 4-10
Example 4-5 Using DesignWare Parts 4-11
Example 5-1 Data Member Array 5-2
Example 5-2 Array Local to a Process 5-2
Example 5-3 Reading From a Variable Array 5-3
Example 5-4 Writing to a Variable Array 5-3
Example 5-5 Reading From a Signal Array 5-4
Example 5-6 Writing to a Signal Array 5-4
Example 5-7 Multiple Accesses to Slices in the Same Array 5-5
Example 5-8 Multiple Array Accesses Using a Variable 5-6
Example 5-9 Accessing Slices of a Signal Array Location 5-6
Example 5-10 Mapping Specific Arrays to Register Files 5-10
Example 5-11 Declaring Local Memory Resources 5-12

Example 5-12 Multiple Arrays Accessing One Memory 5-13
Example 5-13 Changing Memory Types 5-14
Example 5-14 Incorrect Memory Read Timing for Cycle-Fixed 5-16
Example 5-15 Correct Memory Read Timing for Cycle-Fixed 5-16
Example 5-16 Redundant Memory Read. 5-18
Example 5-17 Array Location Assigned to Temporary Variable 5-18
Example 6-1 One-Way Handshake Protocol Behavioral Block 6-6
Example 6-2 Behavioral Block Responding to One-Way Handshake 6-13
Example 6-3 Two-Way Handshake Protocol From GCD Block 6-23
Example 6-4 Two-Way Handshake Protocol From Testbench 6-30
Example 6-5 Two-Way Handshake Using a while Loop. 6-36
Example 6-6 Fast Two-Way Handshake Using while Loop 6-37
Example 6-7 Fast Two-Way Handshake Using wait_until 6-39
Example 6-8 Incorrect Loop Pipeline With Handshake 6-40
Example 6-9 Correct Handshake in a Pipelined Loop 6-43
Example B-1 Architectural Simulation Model B-3
Example B-2 Behavioral Header File B-8
Example B-3 Behavioral Implementation File. B-9
Example B-4 Behavioral Synthesis to Gates Script B-10
Example B-5 Behavioral Testbench Header File B-11
Example B-6 Behavioral Testbench Implementation File B-11
Example B-7 Behavioral Top-Level Simulation File B-14
Example B-8 RTL Header File B-16

Example B-9 RTL Implementation File B-17
Example B-10 RTL Top-Level Simulation File B-20
Example C-1 Memory Controller Header File C-6
Example C-2 Memory Controller Implementation File C-7
Example C-3 Token Header File C-8
Example C-4 Memory Controller Command Types C-8
Example C-5 Behavioral Header File C-11
Example C-6 Behavioral Implementation File C-12
Example C-7 Behavioral Synthesis to Gates Script C-13
Example D-1 FFT Functional Header File D-4
Example D-2 FFT Functional Description File D-5
Example D-3 FFT Header File D-9
Example D-4 FFT Implementation File D-10
Example D-5 Behavioral Synthesis to Gates Script D-16
Example D-6 FFT Testbench Source D-18
Example D-7 FFT Testbench Sink D-20
Example D-8 FFT Testbench Top-Level Model D-21
Example E-1 IQ Header File E-5
Example E-2 IQ Implementation File E-7
Example E-3 Behavioral Synthesis to Gates Script E-16

Preface

This preface includes the following sections:

- What’s New in This Release
- About This Guide
- Customer Support

What's New in This Release

This section describes the new features, enhancements, and changes included in SystemC Compiler version 2000.11-SCC1. Unless otherwise noted, you can find additional information about these changes later in this book.

New Features

SystemC Compiler version 2000.11-SCC1 includes the following new features:

- The `write_rtl` command generates either a synthesizable RTL model or an RTL model optimized for simulation. This command provides a single interface to generate RTL models that replaces setting several dc_shell variables and using the `write` command.
- Using either the `write_rtl` or `write` command, you can write an RTL SystemC model optimized for simulation.

For information about these commands, see the *CoCentric™ SystemC Compiler Behavioral User Guide*.

Enhancements

SystemC Compiler version 2000.11-SCC1 includes the following enhancements:

- Synthesizable RTL models now contain operators such as +, which are used instead of instantiations of Synopsys DesignWare components like DW01_add. Substitutions are made when

possible. This eliminates the dependency on Synopsys-specific components for synthesizable RTL models, unless the behavioral description specifies them.

- The memory wrapper generation tool now allows you to specify a memory write latency in addition to a read latency.

  You can now customize the address and data bus waveforms. In previous versions of the memory wrapper generation tool, address and data bus waveforms were fixed to the first cycle.

  For information about this enhancement, see the *CoCentric™ SystemC Compiler Behavioral User Guide*.

Known Limitations and Resolved STARs

Information about known problems and limitations, as well as about resolved Synopsys Technical Action Requests (STARs), is available in the *CoCentric SystemC Compiler Release Note* in SolvNET.

To see the *CoCentric SystemC Compiler Release Note*,

1. Go to the Synopsys Web page at http://www.synopsys.com and click SolvNET.
2. If prompted, enter your name and password. If you do not have a SOLV-IT! user name and password, you can obtain them at http://www.synopsys.com/registration.
3. Click Release Notes, then open the *CoCentric SystemC Compiler Release Note*.

About This Guide

The *CoCentric™ SystemC Compiler Behavioral Modeling Guide* describes system-level design terminology and explains how to develop or refine a SystemC model for behavioral synthesis with SystemC Compiler.

For information about SystemC, see the Open SystemC Community web site at http://www.systemc.org.

Audience

The *CoCentric™ SystemC Compiler Behavioral Modeling Guide* is for system and hardware designers and electronic engineers who are familiar with the SystemC Class Library and the C or C++ language and development environment.

Familiarity with one or more of the following Synopsys tools is advantageous but not required:

- Synopsys Behavioral Compiler
- Synopsys Design Compiler
- Synopsys Scirocco VHDL Simulator
- Synopsys Verilog Compiled Simulator (VCS)

Related Publications

In addition to the *CoCentric™ SystemC Compiler Behavioral Modeling Guide*, see the following manuals:

- The *CoCentric™ SystemC Compiler Behavioral User Guide*, which provides information about synthesize a refined SystemC behavioral module into an RTL or a gate-level netlist.
- The *CoCentric™ SystemC Compiler RTL User and Modeling Guide*, which provides information about how to synthesize a SystemC RTL module. It also describes the coding guidelines and how to develop a SystemC RTL module for synthesis.
- The *SystemC HDL Cosimulation User Guide*, which provides information about cosimulating a system with mixed SystemC and HDL modules
- The *CoCentric SystemC Compiler Quick Reference*, which provides a list of commands with their options and a list of variables.
- The *SystemC User's Manual* available from the Open SystemC Community web site at http://www.systemc.org.

For additional information about SystemC Compiler and other Synopsys products, see

- Synopsys Online Documentation (SOLD), which is included with the software
- Documentation on the Web, which is available through SolvNET on the Synopsys Web page at http://www.synopsys.com
- The Synopsys Print Shop, from which you can order printed copies of Synopsys documents, at http://docs.synopsys.com

You can also refer to the documentation for the following related Synopsys products:

- Design Compiler
- Scirocco VHDL Simulator
- Verilog Compiled Simulator

Conventions

The following conventions are used in Synopsys documentation.

<table>
<tr><th>Convention</th><th>Description</th></tr>
<tr><td>Courier</td><td>Indicates command syntax.</td></tr>
<tr><td>Courier italic</td><td>Indicates a user-defined value in Synopsys syntax, such as object_name. (A user-defined value that is not Synopsys syntax, such as a user-defined value in a Verilog or VHDL statement, is indicated by regular text font italic.)</td></tr>
<tr><td>Courier bold</td><td>Indicates user input—text you type verbatim—in Synopsys syntax and examples. (User input that is not Synopsys syntax, such as a user name or password you enter in a GUI, is indicated by regular text font bold.)</td></tr>
<tr><td>[ ]</td><td>Denotes optional parameters, such as
pin1 [pin2 ... pinN]</td></tr>
<tr><td>|</td><td>Indicates a choice among alternatives, such as
low | medium | high
(This example indicates that you can enter one of three possible values for an option:
low, medium, or high.)</td></tr>
<tr><td>_</td><td>Connects terms that are read as a single term by the system, such as
set_annotated_delay</td></tr>
<tr><td>Control-c</td><td>Indicates a keyboard combination, such as holding down the Control key and pressing c.</td></tr>
<tr><td>\</td><td>Indicates a continuation of a command line.</td></tr>
<tr><td>/</td><td>Indicates levels of directory structure.</td></tr>
<tr><td>Edit > Copy</td><td>Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.</td></tr>
</table>

Customer Support

Customer support is available through SOLV-IT! and through contacting the Synopsys Technical Support Center.

Accessing SOLV-IT!

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

To access SOLV-IT!,

1. Go to the SolvNET Web page at http://solvnet.synopsys.com.
2. If prompted, enter your user name and password.

   If you do not have a SOLV-IT! user name and password, you can obtain them at http://www.synopsys.com/registration.

If you need help using SOLV-IT!, click SolvNET Help in the column on the left side of the SolvNET Web page.

Contacting the Synopsys Technical Support Center

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

- Open a call to your local support center from the Web by going to http://solvnet.synopsys.com (SOLV-IT! user name and password required), then clicking "Enter a Call."
- Send an e-mail message to support_center@synopsys.com.
- Telephone your local support center.
  - Call (800) 245-8005 from within the continental United States.
  - Call (650) 584-4200 from Canada.
  - Find other local support center telephone numbers at http://www.synopsys.com/support/support_ctr.

Training

For SystemC and SystemC Compiler training and private workshops, contact the Synopsys Customer Education Center in one of the following ways:

- Go to the Synopsys Web page at http://www.synopsys.com/services/education.
- Telephone (800) 793-3448.

1

Introduction

CoCentric SystemC Compiler synthesizes a SystemC behavioral hardware module into an RTL description or a gate-level netlist. (A future release of SystemC Compiler will provide synthesis of RTL descriptions.)

After synthesis, you can use other Synopsys tools for verification, test insertion, power optimization, and physical design.

This modeling guide defines system design terminology and explains how to develop and refine SystemC behavioral models for synthesis with SystemC Compiler. Before reading this modeling guide, read the *CoCentric SystemC Compiler User Guide* to learn about behavioral synthesis concepts and how to run the tool. This modeling guide assumes you are knowledgeable about the SystemC Class Library, available from the Open SystemC Community at http://www.systemc.org.

Synthesizable behavioral design examples are available in Appendix B, “First-In-First-Out Example,” Appendix C, “Memory Controller Example,” Appendix D, “Fast Fourier Transform Example,” and Appendix E, “Inverse Quantization Example” for an MPEG-2 decoder. These examples show you various coding styles and design techniques used with SystemC Compiler.

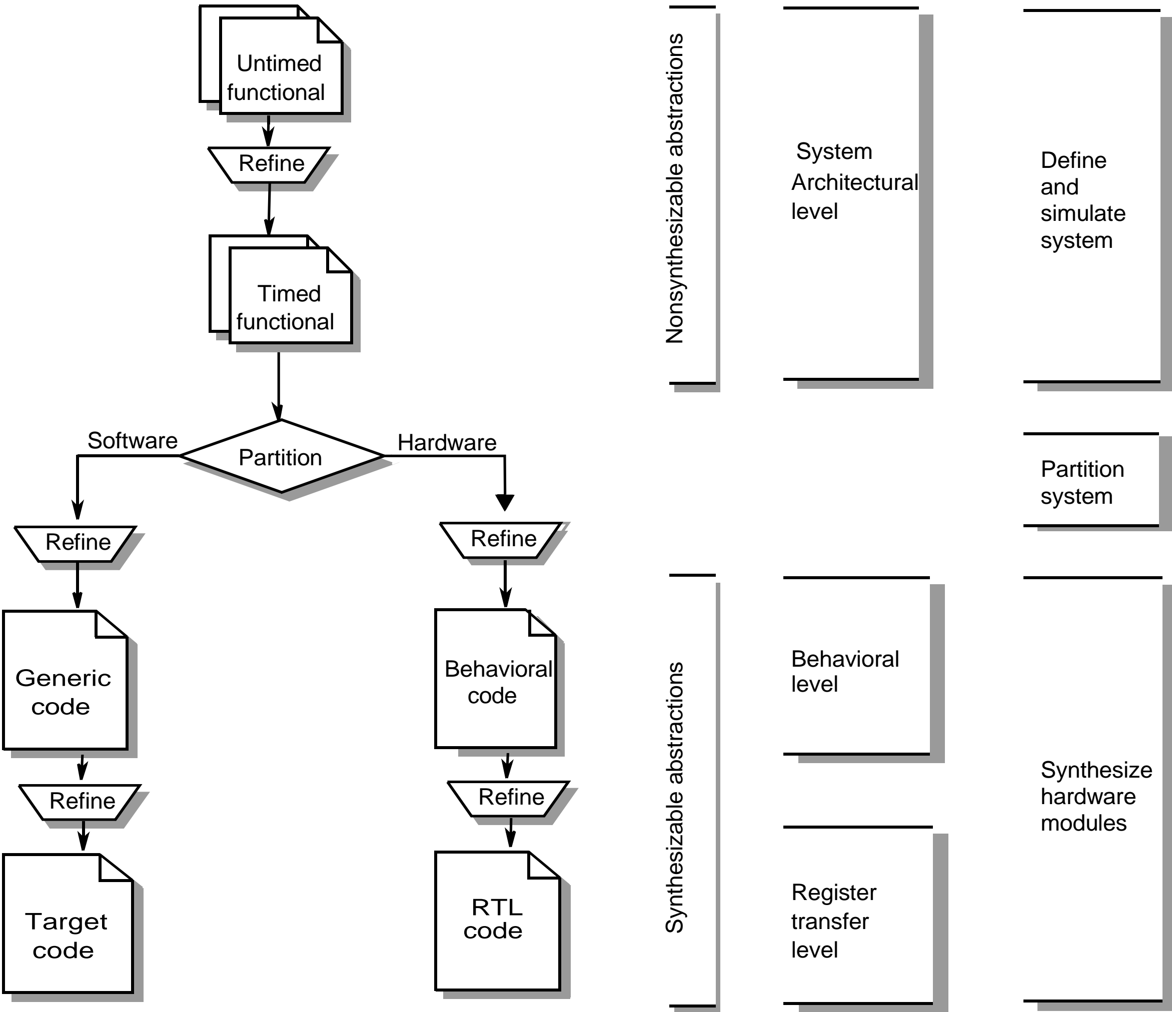
This chapter contains the following sections:

- Defining Levels of Abstraction in System Design
- Choosing the Right Abstraction for Synthesis

Defining Levels of Abstraction in System Design

Figure 1-1 shows the traditional levels of abstraction in system design: system architectural level, behavioral level, and RTL. This section describes the traditional levels of abstraction, their purpose, characteristics, and coding style.

Figure 1-1 System Design Levels of Abstraction

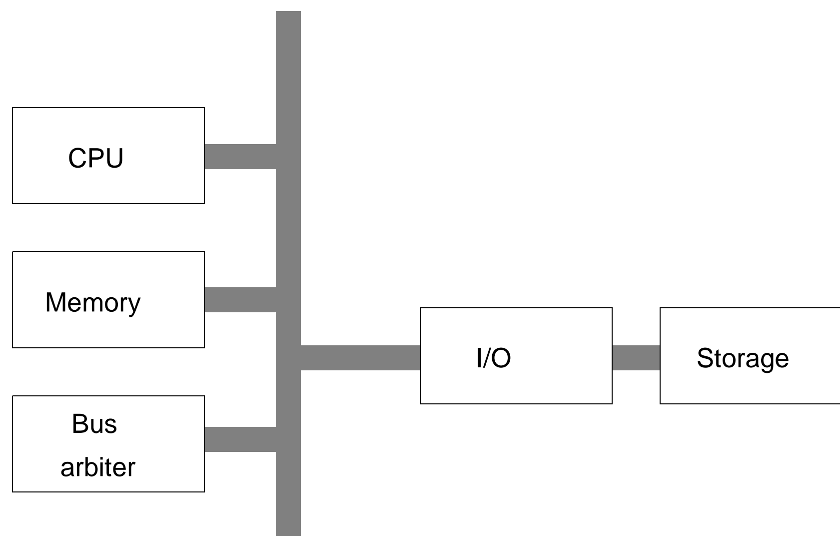


Architectural Level

In a typical top-down design flow, you start with a purely functional model of your system. This functional model is a software program that describes the system functionality so that it can be validated. This functional model is then mapped into a system architectural model. In addition to the system functionality, the system architectural model describes its architecture (buses, memory, processors, peripherals, and so forth).

A system architectural model, illustrated in Figure 1-2, is algorithmic in nature. It may be an untimed or timed model. The model is an accurate description of the system behavior, although the description is abstract. The interfaces between modules are transaction oriented and event driven, rather than cycle accurate.

Figure 1-2   Architectural Model



Untimed Functional Model

An untimed functional model is an executable specification of the system. The system is described as a set of processes, communicating through abstract communication links. The system may be described in a sequential form, concurrent form, or a combination of both. Time is expressed as causality.

Timed Functional Model

A timed functional model is a performance model at a high level of abstraction. The processes and communication links in the untimed functional model are assigned execution times, specified in clock cycles or actual time.

Functional Coding Style

A functional model uses a coding style that is abstract, concise, easy to write, and functionally accurate. You can use the SystemC classes and data types, or you can code the functionality by using only the C/C++ language.

Example 1-1 shows a functional model of a simple first-in-first-out (FIFO) circular buffer. (The complete description and set of files for the available in Appendix B, “First-In-First-Out Example.”)

Example 1-1 FIFO Functional Model

```
/*
 fifo.cc executable specification.

 This model works for a FIFO
 with a size that is a power of 2.
 */


#include "systemc.h"

#define BUFSIZE 4
#define LOGBUFSIZE 2

struct circ_buf {
  int buffer[BUFSIZE];        // The FIFO buffer
  sc_uint<LOGBUFSIZE> headp; // Pointer to head of FIFO
  sc_uint<LOGBUFSIZE> tailp; // Pointer to tail of FIFO
  int num_in_buf;             // Number of buffer elements

  // Routine to initialize the FIFO
  void init() {
     num_in_buf = 0;
     headp = 0;
     tailp = 0;
  }

  // Constructor
  circ_buf() {
    init();
  }

  void status();         // Status of the FIFO
  int read();            // To read from the FIFO
  void write(int data); // To write to the FIFO
  bool is_full();        // To determine if FIFO is full
  bool is_empty();       // To determine if FIFO is empty
};

int
circ_buf::read() {
  if (num_in_buf) {
    num_in_buf--;
    return (buffer[headp++]);
  }
    // Otherwise ignore read request
}
```

```
void
circ_buf::write(int data) {
    if (num_in_buf < BUFSIZE) {
        buffer[tailp++] = data;
         num_in_buf++;
    }
    // Otherwise ignore write request
}

bool
circ_buf::is_full() {
    return (num_in_buf == BUFSIZE);
}

bool
circ_buf::is_empty() {
    return (num_in_buf == 0);
}

void
circ_buf::status() {
    cout << "FIFO is ";
    if(is_empty()) cout << "empty\n" ;
    else if (is_full()) cout << "full\n" ;
    else cout << "neither full nor empty\n";
}

int
main()
{
    circ_buf fifo;  // instantiate buffer

    // This is the testbench for the FIFO

    fifo.status();

    cout << "FIFO write 1\n"; fifo.write(1);
    cout << "FIFO write 2\n"; fifo.write(2);
    cout << "FIFO write 3\n"; fifo.write(3);
    fifo.status();
    cout << "FIFO write 4\n"; fifo.write(4);
    fifo.status();

    cout << "FIFO read " << fifo.read() << endl;
    fifo.status();
    cout << "FIFO read " << fifo.read() << endl;
    cout << "FIFO read " << fifo.read() << endl;
    cout << "FIFO read " << fifo.read() << endl;
```

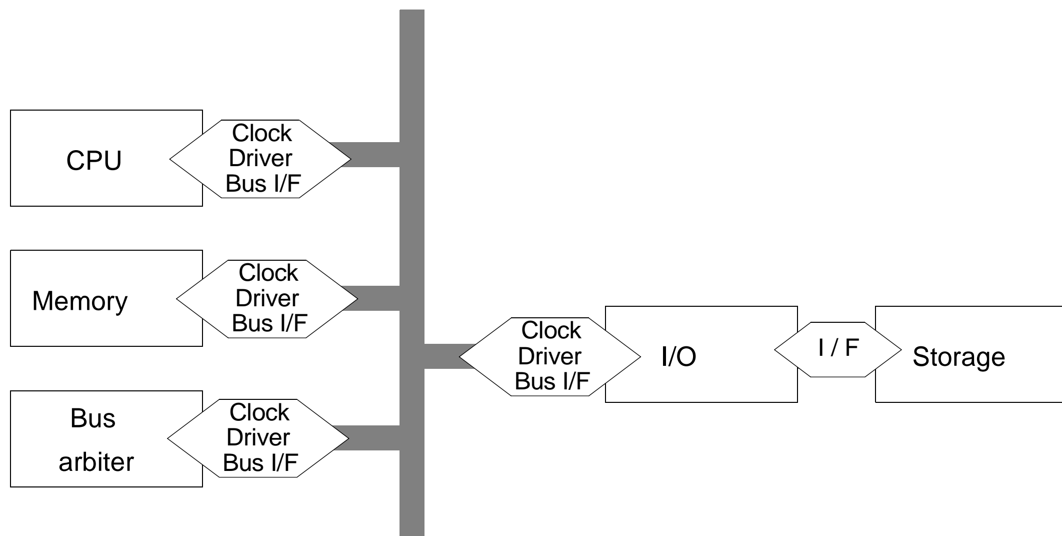
```
    fifo.status();

    cout << "FIFO write 1\n"; fifo.write(1);
    cout << "FIFO write 2\n"; fifo.write(2);
    cout << "FIFO write 3\n"; fifo.write(3);
    fifo.status();
    cout << "FIFO read " << fifo.read() << endl;
    cout << "FIFO read " << fifo.read() << endl;
    fifo.status();

    cout << "FIFO write 4\n"; fifo.write(4);
    cout << "FIFO write 5\n"; fifo.write(5);
    fifo.status();
    cout << "FIFO write 6\n"; fifo.write(6);
    fifo.status();

    cout << "FIFO read " << fifo.read() << endl;
    fifo.status();
    cout << "FIFO read " << fifo.read() << endl;
    cout << "FIFO read " << fifo.read() << endl;
    fifo.status();
    cout << "FIFO read " << fifo.read() << endl;
    fifo.status();

    return 0;
}
```

Behavioral Model

A behavioral model of a block in a system is an algorithmic description of the block's behavior. Unlike a pure software program, however, the I/O behavior of the block is described in a cycle-accurate fashion. Therefore, wait statements are inserted into the algorithmic description to clearly delineate clock-cycle boundaries and when I/O happens. Unlike register-transfer-level (RTL) descriptions, the behavior of the block is still described algorithmically rather than in terms of a finite state machine (FSM) and a data path. Therefore, behavioral descriptions are more compact, and easier to understand, and because of the higher level of abstraction, they simulate faster than RTL.

Figure 1-3 shows a block diagram for a behavioral model.

Figure 1-3 Behavioral Model



Behavioral Coding Style

The general characteristics of the behavioral coding style for synthesis are the following:

- The behavior is described like an algorithm (a software program), and functions can be used to manage complexity.
- Although the initial model may have float or integer data types, you need to refine these types to synthesizable types, described in “Recommended Types for Synthesis” on page 2-39.
- You specify the I/O protocol of the design by defining in which clock cycle the I/O happens. Note that only the I/O, not the operations described in the algorithm, is bound to clock cycles.
- It uses the synthesizable subset of the SystemC language, described in “SystemC and C++ Synthesizable Subset” on page 2-32

Refining From Functional to Behavioral Model

To refine a functional model into a behavioral model,

- Restrict constructs to the synthesizable subset of C++. See the “Nonsynthesizable Subset of SystemC” on page 2-33.
- Refine ports from abstract data types to synthesizable data types, and refine all other data types to synthesizable data types, which are described in “Synthesizable Data Types” on page 2-37.
- Define a clock port for the module.
- Specify the I/O interface by adding wait statements to your description and put signal and port read and write operations with the correct wait statements, described in Chapter 3, “Behavioral Coding Guidelines."

- If required, manage complexity by using functions.

Chapter 2, “Refining for Behavioral Synthesis” describes the refinement activities in more detail.

Example 1-2 shows a behavioral description of the FIFO that was refined from the algorithmic description in Example 1-1 on page 1-6. The design description and complete set of files for the FIFO are available in Appendix B, “First-In-First-Out Example."

Example 1-2 FIFO Behavioral Coding

```
/* fifo_bhv.h header file */

#define BUFSIZE 4
#define LOGBUFSIZE 2
#define LOGBUFSIZEPLUSONE 3

SC_MODULE(circ_buf) {
    sc_in_clk clk;           // The clock
    sc_in<bool> read_fifo;   // Indicate read from FIFO
    sc_in<bool> write_fifo;  // Indicate write to FIFO
    sc_in<int> data_in;      // Data written to FIFO
    sc_in<bool> reset;       // Reset the FIFO

    sc_out<int> data_out;    // Data read from the FIFO
    sc_out<bool> full;       // Indicate FIFO is full
    sc_out<bool> empty;      // Indicate FIFO is empty

    int buffer[BUFSIZE];        // FIFO buffer
    sc_uint<LOGBUFSIZE> headp; // Pointer to FIFO head
    sc_uint<LOGBUFSIZE> tailp; // Pointer to FIFO tail
    // Counter for number of elements
    sc_uint<LOGBUFSIZEPLUSONE> num_in_buf;

    void read_write(); // FIFO process

    SC_CTOR(circ_buf) {
            SC_CTHREAD(read_write, clk.pos());
            watching(reset.delayed() == true);
    }
};


  /***********************************/
  /* fifo_bhv.cc implementation file */

#include “systemc.h”
#include “fifo_bhv.h”

void
circ_buf::read_write() {
    // Reset operations
    headp = 0;
    tailp = 0;
    num_in_buf = 0;
    full = false;
    empty = true;
    data_out = 0;
```

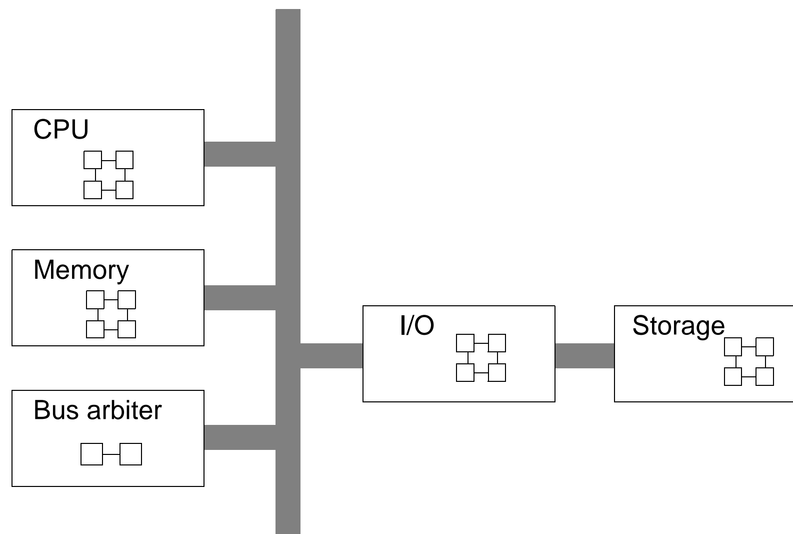
```
    wait();

    // Main loop
    while (true) {
        if (read_fifo.read()) {

            // Check if FIFO is not empty
            if (num_in_buf != 0) {
                num_in_buf--;
                data_out = buffer[headp++];
                full = false;
                if (num_in_buf == 0) empty = true;
            }
            // Ignore read request otherwise
            wait();
        }
        else if (write_fifo.read()) {

            // Check if FIFO is not full
            if (num_in_buf != BUFSIZE) {
                buffer[tailp++] = data_in;
                num_in_buf++;
                empty = false;
                if (num_in_buf == BUFSIZE) full = true;
            }
            // Ignore write request otherwise
            wait();
        }
        else {
           wait();
        }
    }
}
```

Register Transfer Level Model

An RTL model describes registers in your design and the combinational logic between the registers. As such, the functionality of your system is specified as an FSM and a data path. Because register updates are tied to a clock, the model is cycle accurate, both at the interfaces and also internally. Internal cycle accuracy means the clock cycle in which each operation is performed is specified. This is different from a behavioral model that is cycle accurate at the interface, and the operation execution is not cycle accurate

Figure 1-4 shows a block diagram for a cycle-accurate model.

Figure 1-4 RTL Model



RTL Coding Style

The general characteristics of the RTL coding style for synthesis are the following:

- Implements the design as combinational logic between registers. The finite state machine and the data path are explicitly specified.
- Uses only the synthesizable data types, described in "Synthesizable Data Types" on page 2-37.
- Uses the synthesizable subset of the C++ language, described in "SystemC and C++ Synthesizable Subset" on page 2-32.

Refining into RTL

To refine a behavioral model into a RTL model

- Separate the control logic and data path.
- Determine the data-path architecture.
- Define an explicit FSM for the control logic.

Example 1-3 shows the RTL version of the FIFO behavioral model in Example 1-2. The RTL coding style has separate processes for the FSM control and data path. Notice that the RTL version of the FIFO is much longer and more detailed than the equivalent behavioral version, and it is harder to follow than the behavioral description. For details about refining a functional or behavioral model into an RTL model, see the *SystemC Compiler RTL Modeling Guide*.

Example 1-3   RTL Coding

```
/* fifo_rtl.h header file */

#define BUFSIZE 4
#define LOGBUFSIZE 2
#define LOGBUFSIZEPLUSONE 3


SC_MODULE(circ_buf) {
    // Same I/O as behavioral
    sc_in<bool> clk;
    sc_in<bool> read_fifo;
    sc_in<bool> write_fifo;
    sc_in<int> data_in;
    sc_in<bool> reset;
    sc_out<int> data_out;
    sc_out<bool> full;
    sc_out<bool> empty;

    // Internal signals
    sc_signal<int> buf0, buf0_next;
    sc_signal<int> buf1, buf1_next;
    sc_signal<int> buf2, buf2_next;
    sc_signal<int> buf3, buf3_next;
    sc_signal<sc_uint<LOGBUFSIZEPLUSONE> >
              num_in_buf, num_in_buf_next;
    sc_signal<bool> full_next, empty_next;
    sc_signal<int> data_out_next;

    // Declare processes
    void ns_logic();   // Next-state logic
    void update_regs();// Update all registers
    void gen_full();   // Generate a full signal
    void gen_empty();  // Generate an empty signal

    // Constructor
    SC_CTOR(circ_buf) {

         SC_METHOD(ns_logic);
         sensitive << read_fifo << write_fifo
           << data_in << num_in_buf;

         SC_METHOD(update_regs);
         sensitive_pos << clk;

         SC_METHOD(gen_full);
         sensitive << num_in_buf_next;
```

```
        SC_METHOD(gen_empty);
        sensitive << num_in_buf_next;
    }
};
/**********************************/
/* fifo_rtl.cc implementation file */

#include "systemc.h"
#include "fifo_rtl.h"

void circ_buf::gen_full(){
    if (num_in_buf_next.read() == BUFSIZE)
       full_next = 1;
    else
       full_next = 0;
}

void circ_buf::gen_empty(){
    if (num_in_buf_next.read() == 0)
       empty_next = 1;
    else
       empty_next = 0;
}

void circ_buf::update_regs(){
    if (reset.read() == 1) {
       full = 0;
       empty = 1;
       num_in_buf = 0;
       buf0 = 0;
       buf1 = 0;
       buf2 = 0;
       buf3 = 0;
       data_out = 0;
    }
    else {
       full = full_next;
       empty = empty_next;
       num_in_buf = num_in_buf_next;
       buf0 = buf0_next;
       buf1 = buf1_next;
       buf2 = buf2_next;
       buf3 = buf3_next;
       data_out = data_out_next;
    }
}

void circ_buf::ns_logic(){
```

```
    // Default assignments
    buf0_next = buf0;
    buf1_next = buf1;
    buf2_next = buf2;
    buf3_next = buf3;
    num_in_buf_next = num_in_buf;
    data_out_next = 0;

    if (read_fifo.read() == 1) {
       if (num_in_buf.read() != 0) {
         data_out_next = buf0;
         buf0_next = buf1;
         buf1_next = buf2;
         buf2_next = buf3;
         num_in_buf_next = num_in_buf.read() - 1;
       }
    }
    else if (write_fifo.read() == 1) {
       switch(int(num_in_buf.read())) {
         case 0:
            buf0_next = data_in.read();
            num_in_buf_next = num_in_buf.read() + 1;
            break;
         case 1:
            buf1_next = data_in.read();
            num_in_buf_next = num_in_buf.read() + 1;
            break;
         case 2:
            buf2_next = data_in.read();
            num_in_buf_next = num_in_buf.read() + 1;
            break;
         case 3:
            buf3_next = data_in.read();
            num_in_buf_next = num_in_buf.read() + 1;
         default:
            // ignore the write command
            break;
       }
    }
}
```

Choosing the Right Abstraction for Synthesis

You can implement a hardware module by using behavioral-level synthesis or RTL synthesis. Behavioral descriptions are smaller, make it easier to capture complex algorithms, are faster to simulate, accommodate late specification changes, and are more intuitive to write and understand (and therefore maintain) than RTL descriptions.

At this level of abstraction, the model's architecture refers to its hardware implementation, which is not yet specified.

Behavioral synthesis, however, is not suitable for all modules of a design. Evaluate each design module by module, and consider each module's attributes, described in the following sections, to determine whether behavioral or RTL synthesis is applicable.

Identifying Attributes Suitable for Behavioral Synthesis

Look for the following design attributes when identifying a hardware module that is suitable for behavioral synthesis with SystemC Compiler:

- It is easier to conceive the design as an algorithm than as an FSM and a data path – for example, an FFT, filter, IQ, or DSP.
- The design has a complex control flow – for example, a network processor.
- The design has memory accesses, and you need to synthesize access to synchronous memory.

Applications that are suitable for behavioral modeling and synthesis are

- Digital communication applications such as cable modems, cellular phones, cordless phones, two-way pagers, wireless LANs, satellite DSPs, and XDSL modems
- Image and video processing applications such as digital cameras, printers, set-top boxes, 3-D graphic devices, and video capture devices
- Networking applications such as ATM switches, fast networking switches, and packet routers
- Digital signal processing applications such as filters, codecs, IQ, IDCT, and channel equalizers
- Computers applications such as cache controllers, hardware accelerators, and fixed-point arithmetic units

Identifying Attributes Suitable for RTL Synthesis

Some designs are more appropriate for RTL synthesis than for behavioral synthesis. The following design attributes indicate that the design is suitable for RTL synthesis:

- The design is asynchronous.
- It is easier to conceive the design as an FSM and a data path than as an algorithm – for example, a microprocessor.
- The design is very high performance, and the designer, therefore, needs complete control over the architecture.
- The design contains complex memory such as SDRAM or RAMBUS.

Comparison of Behavioral and RTL Synthesis

The following are benefits of behavioral synthesis compared to RTL synthesis.

A behavioral description

- Promotes communication of design intent
- Is usually smaller than RTL code
- Promotes greater design reuse, because the design is technology and architecture independent
- Accommodates late design specification changes, because the code is architecture independent
- Cuts implementation time significantly, increasing designer productivity
- Increases verification speed and decreases verification time
- Promotes exploration of alternative architectures
- Automatically creates the control FSM and data path
- Pipelines critical parts of the design such as loops
- Shares operators and registers
- Automatically synthesizes memory accesses

This modeling guide tells you how to develop descriptions for behavioral synthesis with SystemC Compiler.

2

Refining for Behavioral Synthesis

This chapter explains how to refine a high-level SystemC model or a purely C/C++ model into a behavioral model that can be synthesized with SystemC Compiler. The SystemC and C/C++ language elements that are important for synthesis are also described.

This chapter contains the following sections:

- Refinement Overview
- Creating and Refining the Structure From a C/C++ Model
- Refining the Structure From a High-Level SystemC Model
- Creating and Refining Processes
- Converting to a Synthesizable Subset
- Refining Data
- Refining Control

- Advanced Refinement Techniques
- Refinement Recommendations

Refinement Overview

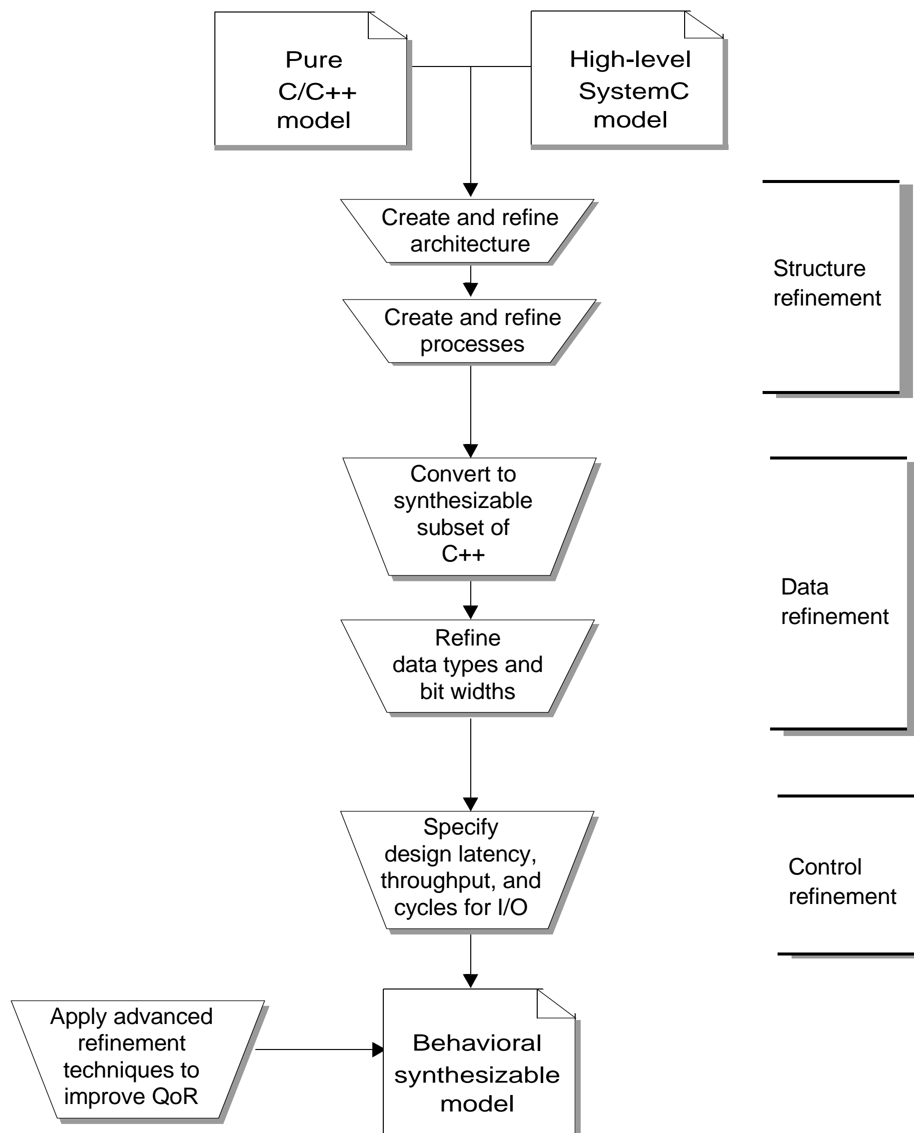
Assuming that you have decided on the architecture for your system and have identified the functionality you want to synthesize with SystemC Compiler, you need to refine the functional model for synthesis. For information about deciding on an system architecture and other design methodology tradeoffs, refer to books and other information sources about design methodology.

Starting with either a purely C/C++ model or a high-level SystemC model, the stages for refining the high-level model into a behavioral model for synthesis with SystemC Compiler are

- Structure refinement
- Data refinement
- Control refinement

Figure 2-1 shows the three major stages for refining the model and the activities in each stage.

Figure 2-1   Refinement Stages and Activities



For structure refinement, you create the architecture of your design, which is the hierarchical structure, and the communication your design uses.

For data refinement, you restrict the model to use only the synthesizable subset of C++ and you choose appropriate data types and bit-widths.

For control refinement, you specify the latency of the design and the cycles in which I/O happens. You also need to ensure that your model adheres to the coding rules required for synthesis.

After synthesis, you can use advanced refinement techniques such as preserved functions (“Using Preserved Functions” on page 4-4) and loop recoding (“Using Loops” on page 3-33) to further refine your design and achieve a higher quality of results (QoR).

You typically perform the refinement activities in the order shown in Figure 2-1. You do not need to complete each stage before going on to the next stage. You may want to partially complete stages and iterate over the entire set of stages several times to develop a synthesizable model.

Creating and Refining the Structure From a C/C++ Model

A pure C/C++ model of your hardware describes only what the hardware is intended to do. When you start with a C/C++ model, the goal of the first refinement stage is to create the hardware structure. To synthesize the hardware, you need to

- Define I/O ports for the hardware module
- Specify the internal structure as blocks
- Specify the internal communication between the blocks
- Define the clock and reset signals, described in Chapter 3, “Behavioral Coding Guidelines”

Define I/O Ports

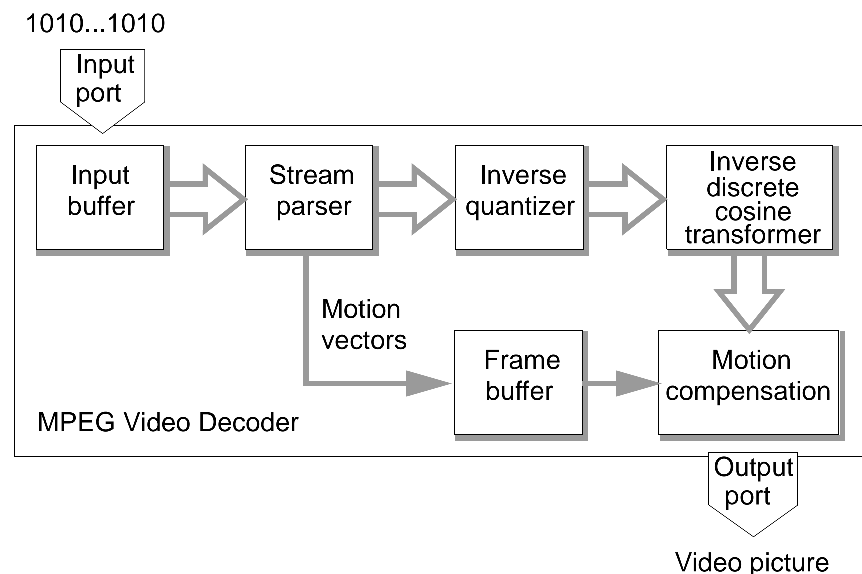
To define I/O ports for the hardware, you need to determine input ports for reading data into the module and output ports for writing data out from the module. Ports are communication resources, and they can be shared. You can define any number of ports, dedicate a port for each I/O, or share ports for I/O based on the requirements of your design.

Specify Internal Structure

Next, you need to specify the internal structure of your design as blocks. Structuring the hardware depends on the way you conceptualize your design and how you intend to synthesize the design with SystemC Compiler. For example, consider an MPEG decoder. You can conceptualize an MPEG decoder to consist of an input port that accepts an MPEG stream, an output port that produces

a decoded MPEG stream, an inverse quantizer (IQ) block, an inverse discrete cosine transformer (IDCT) block, an MPEG stream parser (SP) block, a motion compensation (MC) unit block, input and output buffers (IBs and OBs), and a controller (CT) that controls all the other blocks. Figure 2-2 shows an initial structure of an MPEG design with these blocks.

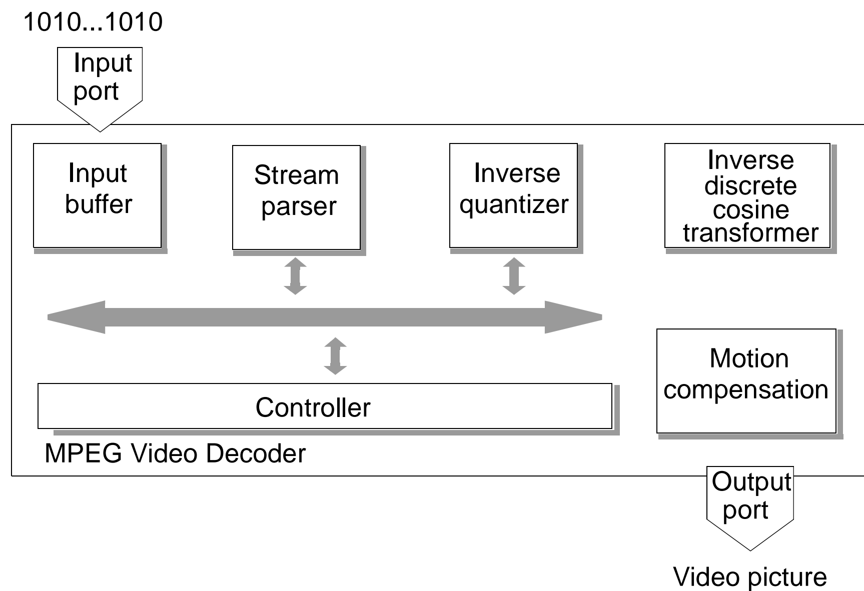
Figure 2-2 MPEG Decoder Functional Structure



Specify the Internal Communication

After you determine the blocks of your design, you need to decide how these blocks communicate with one another. You can use dedicated communication resources between blocks, or you can use a shared communication resource such as a bus. For the blocks in your design, you need to decide what ports they use and what communication resources are used to connect them. For the MPEG example, assume that a bus was chosen. The blocks and the communication between them determines the top-level architecture of your design, as shown in Figure 2-3.

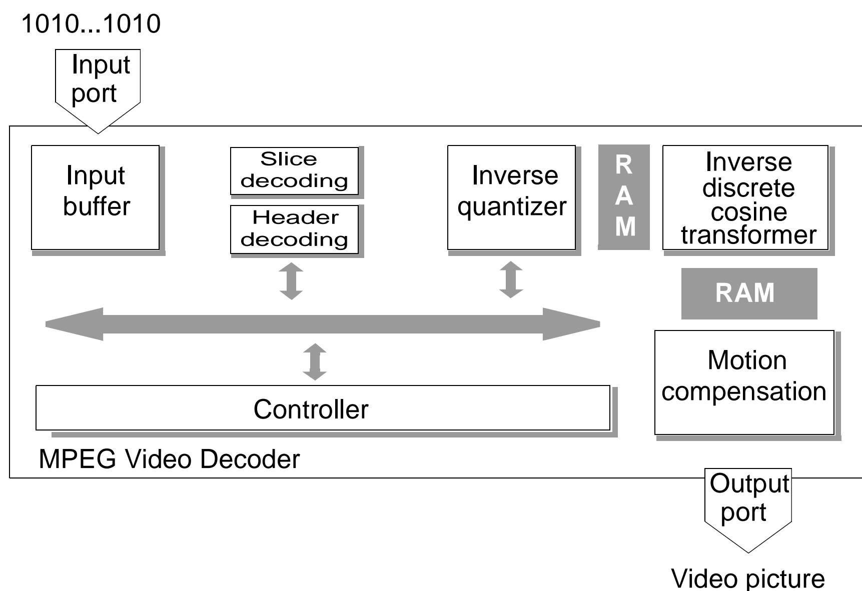
Figure 2-3 MPEG Decoder Top-Level Architecture



Specify the Detailed Architecture

After the top-level architecture is complete, you can take a closer look at each block of the design to decide if you want to apply the same principles of structure refinement to all the blocks. For example, you might want to decompose the MPEG stream parser block into a slice decoding (SD) block and a header decoding (HD) block. You might also want to insert buffers (B) between the IQ and IDCT and between the IDCT and MC. This further refines your architecture and creates hierarchy in your design, as shown in Figure 2-4.

Figure 2-4 MPEG Decoder Detailed Architecture



Continue this type of structure refinement until you are satisfied with the final architecture. In your final architecture, all blocks, their ports, and all communication resources are defined. Though it is not required to have the communication protocols defined at this stage, it is highly recommend that you define the bus protocols and protocols used on dedicated links. You will use the protocol later, during the control refinement stage, to specify when I/O happens.

Atomic and Hierarchical Blocks

In your final architecture, blocks can be atomic, which means they do not contain other blocks. You can also have blocks that are hierarchical, which means they contain other blocks. In the MPEG example, the MPEG decoder and the SP blocks are hierarchical and the other blocks are atomic.

You need to create a SystemC module for each atomic block and a SystemC signal for each communication resource, which is described in “Modules” on page 2-12. A SystemC module can contain only SystemC behavioral or RTL processes. (This document describes only behavioral processes. For information about RTL processes, see the *CoCentric™ SystemC Compiler RTL User and Modeling Guide*.)

For each atomic block, you

- Create a SystemC module
- Define input and output ports
- Define the clock and reset ports (the clock port is mandatory, and the reset port is highly recommended)
- Create a behavioral clocked thread process
- Declare reset watching

For each hierarchical block, you create a SystemC module in which you can define processes as well as instantiations of other modules.

SystemC Compiler synthesizes processes in a module, and their interconnection is inferred from the module instantiation. To synthesize hierarchical modules, RTL synthesis is required. How to create hierarchical modules and integrated behavioral and RTL modules is described in the *CoCentric™ SystemC Compiler RTL User and Modeling Guide*.

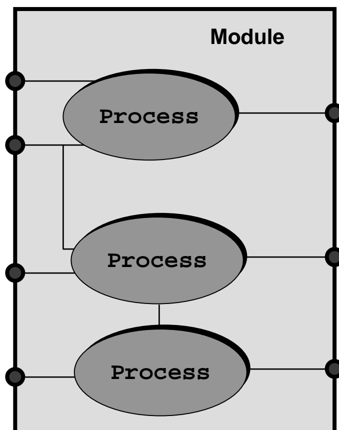
When creating the hardware structure, adhere to the following guidelines:

- A hierarchical module contains instances of other modules and the interconnections between the instances. The hierarchical module can contain RTL processes, but it cannot contain behavioral processes.
- Each atomic module for behavioral synthesis can contain only clocked thread processes.
- For behavioral synthesis, if SystemC Compiler runtime is excessive, you can break your module into smaller modules.

Modules

The basic building block in SystemC is a module. A SystemC module is a container in which processes and other modules are instantiated. Figure 2-5 shows a typical module with several processes. The processes within a module are concurrent.

Figure 2-5 Module



Note:

For synthesis with SystemC Compiler version 2000.05-SCC1.0, a module cannot contain instances of other modules.

As a recommended coding practice, describe a module by using two separate files, a separate header file (*module_name*.h) and an implementation file (*module_name*.cpp or *module_name*.cc).

Module Header File

Each module header file contains the module declaration, which includes

- Port declarations
- Internal signal variable declarations
- Internal data variable declarations
- Process declarations
- Member function declarations
- Constructor of the module

Module Syntax

Declare a module by using the syntax shown in bold in the following example:

```
SC_MODULE (module_name) {
     //Module port declarations
     //Signal variable declarations
     //Data variable declarations
     //Clocked thread process declarations
     //Member function declarations

     //Module constructor
     SC_CTOR (module_name) {
              //Register processes
              //Declare sensitivity list
              //Define global watching
     }
};
```

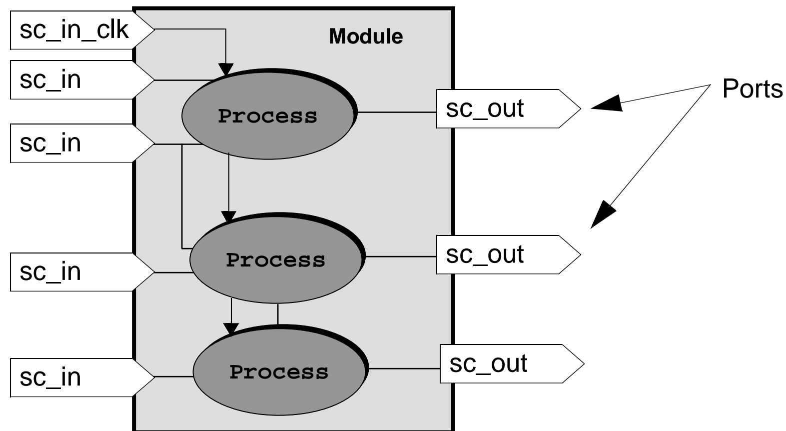
Note:

SC_MODULE and SC_CTOR are C++ macros defined in the System Class library.

Module Ports

Each module has any number of input and output ports (Figure 2-6), which determine the direction of data into or out of the module.

Figure 2-6 Module Ports



A port is a data member of SC_MODULE. You can declare any number of sc_in or sc_out ports. For a module with a behavioral (SC_THREAD) process, you must declare one sc_in_clk port.

Note:

SystemC sc_inout ports are not used for behavioral synthesis.

Port Syntax

Declare ports by using the syntax shown in bold in the following example:

```
SC_MODULE (module_name) {
     //Module port declarations
     sc_in<port_data_type> port_name;
     sc_out<port_data_type> port_name;
     sc_in_clk port_name; // Mandatory
    sc_in<bool> reset;   // Highly recommended

     //Signal variable declarations
     //Data variable declarations
     //Clocked thread processes
     //Member function declarations

     //Module constructor
     SC_CTOR (module_name) {
              //Register processes
              //Declare sensitivity list
              //Define global watching
     }
};
```

Port Data Types

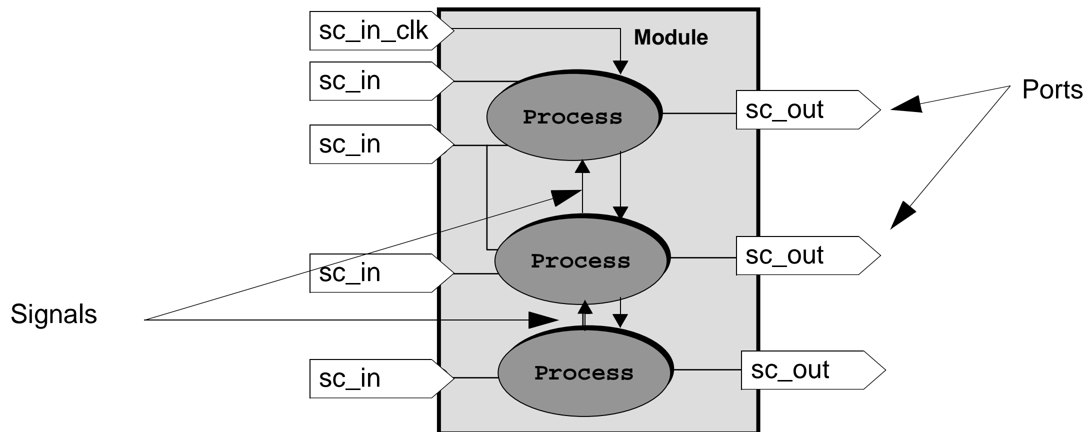
Ports connect to signals and, like signals, have a data type associated with them. For synthesis, declare each port as one of the synthesizable data types, described in “Synthesizable Data Types” on page 2-37.

An sc_in_clk is a special port that connects to the clock signal to trigger one or more SC_CTHREAD processes. Each SC_CTHREAD process requires one sc_in_clk port. You can use the same clock port for all processes in a module, or you can declare a separate sc_in_clk port for each SC_CTHREAD process.

Internal Signals

Modules use ports to communicate with other modules. Internal signals are used by processes to communicate with other processes within the same module, as shown in Figure 2-7.

Figure 2-7 Processes and Signals



Signal Syntax

Declare signals by using the syntax shown in bold in the following example:

```
SC_MODULE (module_name) {
     //Module port declarations
     sc_in<port_type> port_name;
     sc_out<port_type> port_name;
     sc_in_clk port_name;

     //Internal signal variable declarations
     sc_signal<signal_type> signal_name;

     //Data variable declarations
     //Clocked thread processes
     //Member function declarations

     //Module constructor
     SC_CTOR (module_name) {
              //Register processes
              //Declare sensitivity list
              //Define global watching
     }
};
```

Signal Data Types

A signal's bit-width is determined by its corresponding data type. Specify the data type as any of the synthesizable SystemC or C++ data types listed in "Synthesizable Data Types" on page 2-37. Signals and the ports they connect must have compatible data types.

Reading and Writing Ports

When you read a port, it is recommended to use the read() and write() methods to distinguish ports from variable assignments. Example 2-1 shows in bold how to use these methods rather than simple assignments.

Example 2-1   Using read() and write() Methods

```
//...
wait();
address = into.read();     // read from into port
wait();                     // wait one clock
data_tmp = memory[address]; // get data from memory
outof.write(data_tmp);     // write to outof port
wait();
//...
```

You need to read or write all bits of a port. You cannot read or write the individual bits, regardless of its type. To select a bit on a port, read the port data into a temporary variable and select a bit in the temporary variable.

Reading and Writing Signals

You can read or write a signal using either the read() and write() methods or by assignment. You cannot read or write the individual bits, regardless of its type. To select a bit on a signal, read the signal data into a temporary variable and select a bit in the temporary variable.

Internal Data Variables

Inside a module, you can define data variables of any synthesizable SystemC or C++ type. These variables are typically used for internal storage in the module. Do not use them for interprocess communication, because it can lead to nondeterminism (order mismatch) during simulation and can cause possible mismatches between the results of synthesis and simulation.

Declare internal data variables by using the syntax shown in bold in the following example:

```
SC_MODULE (module_name) {
     //Module port declarations
     sc_in<port_type> port_name;
     sc_out<port_type> port_name;
     sc_in_clk port_name;

     //Internal signal variable declarations
     sc_signal<signal_type> signal_name;

     //Data variable declarations
     int count_val;          //Internal counter
     sc_int<8> mem[1024];    //Array of sc_int

     //Clocked thread processes
     //Member function declaration

     //Module constructor
     SC_CTOR (module_name) {
               //Register processes
               //Declare sensitivity list
               //Define global watching
     }
};
```

Processes

Electronic systems are inherently parallel, but programming languages such as C and C++ execute sequentially. SystemC provides processes for describing the parallel behavior of hardware systems. This means processes execute concurrently, rather than sequentially like C++ functions. The code within a process, however, executes sequentially.

Processes use signals to communicate with each other. One process can cause another process to execute by assigning a new value to a signal that interconnects them. Do not use data variables for communication between processes to avoid causing nondeterminism (order dependency) during simulation.

Defining a process is similar to defining a C++ function. A process is declared as a member function of a module and registered as a process in the module's constructor. You can declare and instantiate more than one process in a module, but processes cannot contain other processes or modules.

A process is registered inside the module's constructor. Registering a process makes it recognizable by SystemC Compiler as a process rather than as an ordinary member function. You can register multiple different processes, but it is an error to register more than one instance of the same process.

A process can read from and write to ports and internal signals.

Types of Processes

SystemC provides three process types – SC_CTHREAD, SC_METHOD, and SC_THREAD – that execute whenever their sensitive inputs change. A process has a sensitivity list that identifies which inputs trigger the code within the process to execute when the value on one of its sensitive inputs change.

For simulation, you can use any of the process types. For synthesis, you can use only the SC_CTHREAD and SC_METHOD processes. The SC_THREAD process is used mainly for testbenches, although the SC_CTHREAD and SC_METHOD processes can also be used for testbenches.

Clocked Thread Process

The SC_CTHREAD clocked thread process is sensitive to one edge of one clock. Use a clocked thread process to describe functionality for behavioral synthesis with SystemC Compiler.

The SC_CTHREAD process models the behavior of a sequential logic circuit with nonregistered inputs and registered outputs. A registered output comes directly from a register (flip-flop) in the synthesized circuit.

Method Process

The SC_METHOD process is sensitive to a set of signals and executes when one of its sensitive inputs change. Use a method process to describe a hierarchical behavioral design or register-transfer-level hardware. (For information about RTL modeling, see the *CoCentric™ SystemC Compiler RTL User and Modeling Guide*.)

Note:

Although you can create an RTL model in SystemC, SystemC Compiler version 2000.05-SCC1.0 does not synthesize RTL models. RTL synthesis is planned for a future release of SystemC Compiler.

Creating a Process in a Module

SystemC processes are declared in the module body and registered as processes inside the constructor of the module, as shown in bold in Example 2-2.

You must declare a process with a return type of void and no arguments, which is also shown in bold in Example 2-2.

Example 2-2 Creating a Clocked Thread Process in a Module

```
// cmult.h header file
SC_MODULE(cmult) {
     // Declare ports
     sc_in<sc_int<8> > data_in;
     sc_in_clk clk;
     sc_out<sc_int<16> > real_out;
     sc_out<sc_int<16> > imaginary_out;

     // Declare internal variables and signals

     // Declare processes in the module
     void entry();

     // Constructor
     SC_CTOR (cmult) {

          // Register processes and define
          // the active clock edge
          SC_CTHREAD(entry, clk.pos());
     }
};
```

To register a function as a process, use the SC_CTHREAD macro that is defined in the SystemC Class library. The SC_CTHREAD macro takes two arguments:

1. The name of the process
2. The edge of the clock that triggers the process, which is also called the active edge

Member Functions

In a module, you can declare other member functions that are not processes. They are not registered as processes in the module's constructor. These functions can be called from within a process. Member functions can contain any synthesizable C++ or SystemC statement allowed in the SC_CTHREAD process.

Appendix E, "Inverse Quantization Example," shows an example that uses numerous member functions.

A member function that is not a process can return any data type, but a member function that is a process can return only a void type.

See "Using Member Functions" on page 4-2 for further information.

Module Constructor

For each module, you need to create a constructor, which is used to

- Register processes
- Define a sensitivity list for each SC_METHOD process
- Define an optional global reset

For synthesis, other statements are not allowed in the constructor.

Note:

> A single global reset is supported for synthesis, which is explained in “Describing a Global Reset” on page 3-46. Multiple global resets are not allowed.

Example 2-3 shows the header file for a complex number multiplier with a global reset. In this example, the constructor registers an SC_CTHREAD process and defines a global reset, which is shown in bold.

Example 2-3 Module Constructor

```
// cmult_hs.h header file
SC_MODULE(cmult_hs) {
     // Declare ports
     sc_in<bool> reset;
     sc_in<sc_bv<8> > data_in;
     sc_in_clk clk;
     sc_out<sc_int<16> > real_out;
     sc_out<sc_int<16> > imaginary_out;

     // Declare internal variables and signals

     // Declare processes in the module
     void entry();

     // Constructor
     SC_CTOR (cmult_hs) {
          // Register processes and
          // define active clock edge
          SC_CTHREAD(entry, clk.pos());

          // Watching for global reset
          watching(reset.delayed() == true);
     }
};
```

Module Implementation File

As a recommended coding practice, write the module's behavior in a separate implementation file. Name the file with either a .cpp or .cc file extension, for example *my_module*.cpp.

Using an Infinite Loop

When using a clocked thread process, enclose the module behavior within an infinite loop (while (true)) in the module's implementation file. This ensures that the process runs continuously, like hardware. In addition, each clocked thread process must have at least one wait statement, which is explained further in Chapter 3, "Behavioral Coding Guidelines."

Example 2-4 shows the implementation file for the complex number multiplier header file shown in Example 2-2 on page 2-22. The required infinite loop is shown in bold.

Example 2-4 Module Behavior

```
// cmult.cc implementation file

#include "systemc.h"
#include "cmult.h"

void cmult :: entry() {
   sc_int<8> a, b, c, d;
   while (true) {
     // Read four data values from input port
     a = data_in.read();
     wait();
     b = data_in.read();
     wait();
     c = data_in.read();
     wait();
     d = data_in.read();
     wait();
     //Calculate and write output ports
     real_out.write(a * c - b * d);
     imaginary_out.write(a * d + b * d);
     wait();
   }
}
```

Refining the Structure From a High-Level SystemC Model

When you start from a high-level SystemC model, your model might or might not already have the structure required for your hardware. If your model does not have the structure you need, follow the steps beginning at “Creating and Refining the Structure From a C/C++ Model” on page 2-6 to create the hardware structure.

A high-level SystemC model, unlike a pure C/C++ model, may contain abstract ports. Abstract ports are types that are not readily translated to hardware. For each abstract port, you need to define a signal port to replace each terminal of the abstract port. You also need to replace all accesses to the abstract ports or terminals with accesses to the newly defined signal ports. For more information about abstract ports, see the *SystemC User’s Guide*.

Creating and Refining Processes

After you create the detailed architecture of your hardware and decompose the functionality into hierarchical and atomic modules, you need to create behavioral and RTL processes inside the modules. (For information about RTL process creation, see the *CoCentric™ SystemC Compiler RTL User and Modeling Guide*.)

The description of defining a module and declaring a clocked thread process for behavioral synthesis begins at “Modules” on page 2-12. To implement the behavior, define the process body. The process body, which consists of two distinct sections, the reset action and the main functionality, as shown in Example 2-5.

Example 2-5 Basic Reset Action and Main Loop

```
/**** my_module.h header file ****/

#include "systemc.h"

SC_MODULE(SOME_MODULE) {
   // Ports

   // data_t is a struct defined elsewhere
   sc_in<data_t> in_data;      // Input port
   sc_out<data_t> out_data;    // Output port
   sc_in_clk clk;              // Mandatory
   sc_in<bool> reset;          // Highly recommended

    // Process
   void work();

    // Constructor
   SC_CTOR(SOME_MODULE) {

     SC_CTHREAD(work, clk.pos()); // Declare process
     watching(reset.delayed() == 1); // Specify reset
   }
};

/**** my_module.cpp implementation file ****/

// Process

void SOME_MODULE::work() {

   // Reset actions
   //...
   wait(); // Required wait

   // Main loop
   while (true) {

     // Main functionality
     //...
   }
}
```

In the reset action section, you specify the reset behavior of the circuit. When the reset signal is asserted, the reset code is executed. Writing reset functionality is explained in “Using Resets” on page 3-46.

To ensure that a process executes infinitely, enclose its functionality in an infinite while loop, which is designated as the main loop in Example 2-5. Put the functionality of your process inside this main loop.

If you started with a C/C++ model of your hardware or a high-level SystemC model, the design functionality of your hardware is already described as a software algorithm. In that case, you need little additional refinement of the behavioral code to implement the design in hardware, using behavioral synthesis.

Converting to a Synthesizable Subset

As the next stage in refinement, you need to convert all nonsynthesizable code into synthesizable code. This is required only for functionality that is to be synthesized.

Although you can use any SystemC class or C++ construct for simulation and other stages of the design process, many C and C++ language constructs and SystemC classes are not relevant for synthesis. Because these constructs cannot be synthesized into hardware, SystemC Compiler does not support them, and it displays an error message if it encounters any of these constructs in your code. You can comment out code that is needed only for simulation, such as print statements for debugging.

Excluding Simulation-Specific Code

SystemC Compiler provides compiler directives you can use in your code

- To include synthesis-specific directives
- To exclude or comment out simulation-specific code so it does not interfere with synthesis

You can isolate synthesis-specific or simulation-specific code with a compiler directive, either the C language #ifdef or a comment starting with the word `synopsys` or `snps` and `synthesis_off`. Example 2-6 shows compiler directives in bold that exclude simulation code from synthesis.

Example 2-6 Excluding Simulation-Only Code

```
//C directive
#ifdef SIM
...//Simulation-only code
#endif

//SystemC Compiler directive
/* synopsys synthesis_off */
... //Simulation-only code
/* snps synthesis_on */
```

“Synthesis Compiler Directives” in Appendix A provides a list of the SystemC Compiler directives.

SystemC and C++ Synthesizable Subset

The synthesizable subsets of SystemC and C++ are provided in the sections that follow. Wherever possible, a recommended corrective action is indicated for converting nonsynthesizable constructs into synthesizable constructs. For many nonsynthesizable constructs, there is no obvious recommendation to convert them into synthesizable constructs or there are numerous ways to convert them. In such cases, a recommended corrective action is not indicated. Familiarize yourself with the synthesizable subset and use the synthesizable subset as much as possible in your pure C/C++ or high-level SystemC models to minimize the effort of data refinement for synthesis.

You can use any SystemC or C++ construct for a testbench. You do not need to restrict your code to the synthesizable subset in the testbench.

Nonsynthesizable Subset of SystemC

SystemC Compiler does not support the SystemC constructs listed in Table 2-1 for behavioral synthesis.

Table 2-1 Nonsynthesizable SystemC Classes

Category	Construct	Comment	Corrective action
Thread process	SC_THREAD	Used for modeling a testbench, but not supported for synthesis.	Change to SC_CTHREAD.
Method process	SC_METHOD	Used for simulation and modeling at the RT level, but not supported for synthesis in SystemC Compiler version 2000.05-SCC1.0. RTL synthesis is planned for a later release.	
Channels	sc_channel	Used only in initial stages of modeling system functionality.	Replace with sc_signal.
Clock generators	sc_start()	Used for simulation.	Comment out.
Bidirectional port	sc_inout	Bidirectional ports are not allowed.	Change to separate sc_in and sc_out ports.
Local watching	W_BEGIN, W_END, W_DO, W_ESCAPE	Local watching is not supported.	
Multiple global resets	Multiple watching( )	One global reset is supported for synthesis. Multiple resets are not supported.	Combine multiple resets into a single reset, using an AND operator.
Tracing	sc_trace, sc_create* trace_file	Creates waveforms of signals, channels, and variables for simulation.	Comment out for synthesis.

Nonsynthesizable C/C++ Constructs

SystemC Compiler does not support the C and C++ constructs listed in Table 2-2 for behavioral synthesis.

Table 2-2 Nonsynthesizable C/C++ Constructs

Category	Construct	Comment	Corrective action
Local class declaration		Not allowed.	Replace.
Nested class declaration		Not allowed.	Replace.
Derived class		Only SystemC modules and processes are supported.	Replace.
Dynamic storage allocation	malloc(), free(), new, new[], delete[]	malloc(), free(), new, new[], delete, and delete[] are not supported.	Use static memory allocation.
Exception handling	try, catch, throw	Not allowed.	Comment out.
Recursive function call		Not allowed.	Replace with iteration.
Function overloading		Not allowed (except the classes overloaded by SystemC).	Replace with unique function calls.
C++ built-in functions		The math library, I/O library, file I/O, and similar built-in C++ functions are not allowed.	Comment out.
Virtual function		Not allowed.	Replace with a nonvirtual function.
Inheritance		Not allowed.	Replace.
Multiple inheritance		Not allowed.	Replace.

Table 2-2 Nonsynthesizable C/C++ Constructs (continued)

Category	Construct	Comment	Corrective action
Member access control specifiers	public, protected, private, friend	Allowed in code, but are ignored for synthesis. All member access is public.	Change to public access, or ignore the compiler warnings.
Accessing struct members with the (->) operator	-> operator	Not allowed.	Replace with access using the period (.) operator.
Static member		Not allowed.	Replace with nonstatic or member variable.
Dereference operator	* and & operators	Not allowed.	Replace dereferencing with direct access to the variable or array.
Operator overloading		Not allowed (except the classes overloaded by SystemC).	Replace overloading with unique function calls.
Operator, sizeof	sizeof	Not allowed.	Determine size statically for use in synthesis.
Pointer	*	Pointers are allowed only in hierarchical modules, which are not supported in SystemC Compiler version 2000.05-SCC1.0. A *char is treated as a string, not as a pointer to memory.	Replace all pointers with access to array elements or individual elements.
Pointer type conversions		Not allowed.	Do not use pointers. Use explicit variable reference.
this pointer	this	Not allowed.	Replace.

Table 2-2 Nonsynthesizable C/C++ Constructs (continued)

Category	Construct	Comment	Corrective action
Reference, C++	&	Allowed only for passing parameters to functions.	Replace in all other cases.
Reference conversion		Reference conversion is supported for implicit conversion of signals only.	Replace in all other cases.
User-defined template class		Only SystemC templates classes such as sc_int<> are supported.	Replace.
Type casting at runtime		Not allowed.	Replace.
Type identification at runtime		Not allowed.	Replace.
Explicit user-defined type conversion		The C++ built-in types and SystemC types are supported for explicit conversion.	Replace in all other cases.
Unconditional branching	goto	Not allowed.	Write structured code with breaks and continues.
Unions		Not allowed.	Replace with structs.
Global variable		Not supported for synthesis.	Replace with local variables.
Member variable		Member variables accessed by two or more SC_THREAD processes are not supported. However, access to member variable by only one process is supported.	Use signals instead of variables for communication between processes.
Volatile variable		Not allowed.	Use only nonvolatile variables.

Refining Data

A pure C/C++ model or a high-level SystemC model typically uses native C++ types or aggregates (structures) of such types. Native C++ types such as int, char, bool, and long have fixed, platform-dependent widths, which are often not the correct width for efficient hardware. For example, you might need only a 6-bit integer for a particular operation, instead of the native C++ 32-bit integer. In addition, C++ does not support four-valued logic vectors, operations such as concatenation, and other features that are needed to efficiently describe hardware operations.

SystemC provides a set of limited precision and arbitrary precision data types that allows you to create integers, bit vectors, and logic vectors of any length. SystemC also supports all common operations on these data types.

During the data refinement step, you need to evaluate all variable declarations, formal parameters, and return types of all functions to determine the appropriate data type as well as the appropriate widths of each data type. The following sections recommend the appropriate data type to use and when. Selecting the data widths is a design decision, and it is typically a tradeoff between the cost of hardware and the required precision. This decision is, therefore, left to you.

Synthesizable Data Types

C++ is a strongly typed language. Every constant, port, signal, variable, function return type, and parameter is declared as a data type, such as bool or sc_bit, and can hold or return a value of that type. Therefore, it is important that you use the correct data types in expressions.

Nonsynthesizable Data Types

All SystemC and C++ data types can be used for behavioral synthesis, except the following types:

- Floating-point types such as float and double
- Fixed-point types sc_fixed, sc_ufixed, sc_fix, and sc_ufix
- Access types such as pointers
- File types such as FILE
- I/O streams such as stdout and cout, which are ignored by SystemC Compiler
- SystemC sc_logic and sc_lv are used for RTL synthesis only, not for behavioral synthesis

Recommended Types for Synthesis

For the best synthesis, use appropriate data types and bit-widths so SystemC Compiler does not build unnecessary hardware. Use the SystemC data types listed in Table 2-3 in place of the equivalent C++ native type to restrict bit size for synthesis – for example, change an int type to an sc_int<*n*> type, where *n* specifies the number of bits.

Table 2-3 Synthesizable Data Types

SystemC and C++ type	Description
sc_bit	A single-bit true or false value
sc_bv<*n*>	An arbitrary-length bit vector
sc_logic	A single-bit 0, 1, X, or Z for RTL synthesis only
sc_lv<*n*>	An arbitrary-length logic vector for RTL synthesis only
sc_int<*n*>	Fixed-precision integers restricted in size up to 64 bits
sc_uint<*n*>	Fixed-precision integers restricted in size up to 64 bits, unsigned
sc_bigint<*n*>	Arbitrary-precision integers recommended for sizes over 64 bits
sc_biguint<*n*>	Arbitrary-precision integers recommended for sizes over 64 bits, unsigned
bool	A single-bit true or false value
int	A signed integer, typically 32 or 64 bits, depending on the platform
unsigned int	An unsigned integer, typically 32 or 64 bits, depending on the platform
long	A signed integer, typically 32 or 64 bits, or longer, depending on the platform

Table 2-3 Synthesizable Data Types (continued)

SystemC and C++ type	Description
unsigned long	An unsigned integer, typically 32 or 64 bits, or longer, depending on the platform
char	A signed integer to represent individual characters and small integers, typically -128 through 127
unsigned char	An unsigned integer to represent individual characters and small integers, typically 0 through 255
short	A signed integer, typically 32 bits, depending on the platform
unsigned short	An unsigned integer, typically 32 bits, depending on the platform
struct	A user-define aggregate of synthesizable data types
enum	A user-defined enumerated data type associated with an integer constant

Using SystemC Types

The following sections describe the operations that are supported by the SystemC data types.

Bit and Bit Vector Data Type Operators

Table 2-4 provides a list of operators available for the SystemC sc_bit and sc_bv data types. In Table 2-5, Yes indicates that the operator is available for the specified data type.

Table 2-4 SystemC Bit and Bit Vector Data Type Operators

Operators	sc_bit	sc_bv
Bitwise & (and), | (or), ^ (xor), and ~ (not)	Yes	Yes
Bitwise << (shift left) and >> (shift right)	No	Yes
Assignment =, &=, |=, and ^=	Yes	Yes
Equality ==, !=	Yes	Yes
Bit selection [x]	No	Yes
Part selection range (x-y)	No	Yes
Concatenation (x,y)	No	Yes
Reduction: and_reduce( ), or_reduce( ), and xor_reduce( )	No	Yes

Fixed and Arbitrary Precision Data Type Operators

Table 2-5 provides a list of operators available for the SystemC sc_int and sc_uint fixed precision and sc_bigint and sc_biguint arbitrary precision integer data types. In Table 2-5, Yes indicates that the operator is available for the specified data types.

Table 2-5 SystemC Integer Data Type Operators

<table>
<tr><th>Operators</th><th>sc_int, sc_uint,
sc_bigint, sc_biguinit</th></tr>
<tr><td>Bitwise & (and), | (or), ^ (xor), and ~ (not)</td><td>Yes</td></tr>
<tr><td>Bitwise << (shift left) and >> (shift right)</td><td>Yes</td></tr>
<tr><td>Assignment =, &=, |=, ^=, +=, -=, *=, /=, and %=</td><td>Yes</td></tr>
<tr><td>Equality ==, !=</td><td>Yes</td></tr>
<tr><td>Relational <, <=, >, and >=</td><td>Yes</td></tr>
<tr><td>Autoincrement ++ and autodecrement --</td><td>Yes</td></tr>
<tr><td>Bit selection [x]</td><td>Yes</td></tr>
<tr><td>Part selection range (x-y)</td><td>Yes</td></tr>
<tr><td>Concatenation (x,y)</td><td>Yes</td></tr>
<tr><td>Reduction and_reduce( ), or_reduce( ), and xor_reduce( )</td><td>Yes</td></tr>
</table>

Using Enumerated Types

SystemC Compiler interprets an enumerated (enum) data type as a numerical value, where the first element is equal to zero. Appendix C, “Memory Controller Example,” shows an example of using an enumerated data type.

Using Aggregate Data Types

To group data types into a convenient aggregate type, define them as a struct type similar to Example 2-7. You need to use all synthesizable data types in a struct in order for the struct to be synthesizable. SystemC Compiler splits the struct type into individual elements for synthesis.

Example 2-7 Aggregate Data Type

```
struct package {
    sc_int<8> command;
    sc_int<8> address;
    sc_int<12> data;
};
```

Appendix C, “Memory Controller Example,” shows an example of using an aggregate data type.

Using C++ Types

The native C++ data types, such as bool, char, int, long, short, unsigned char, unsigned int, unsigned long, and unsigned short have a platform-specific size. SystemC Compiler synthesizes variables of these types to have the width dictated by your platform.

for Loop Counter

In some situations, SystemC Compiler can determine that fewer bits are required in hardware than is specified by the data type, which produces a higher-quality result after synthesis. For example, if a unique integer variable is declared as a for loop counter, SystemC Compiler can determine the number of bits and build only the required hardware. Example 2-8 shows a unique loop counter variable in bold. SystemC Compiler can determine that 3 bits are required, and it would build a 3-bit incrementer for variable *i*.

Example 2-8 Implicit Bit Size Restriction

```
for (int i=0; i < 7; i++){
        ... //loop code
}
```

If a variable is declared outside of the for loop, SystemC Compiler cannot determine the bit size, because the intended use of the variable is not known at the time of declaration. In this situation, SystemC Compiler builds hardware for the platform-specific bit size. Example 2-9 shows code (in bold) where the loop counter variable is declared outside the loop. In such a situation, SystemC Compiler infers that a variable of 32 or 64 bits is required, depending on the platform. Therefore, it is strongly recommended that you use the coding style shown in Example 2-8 instead of the style in Example 2-9.

Example 2-9 Unknown Variable Bit Size

```
int i;
...
for (i=0; i < 7; i++){
      ... //loop code
}
```

Data Members of a Module

It is strongly recommended that you do not use data members for storage. Use variables local to the process for all storage requirements in a process. Example 2-10 shows a data member *x* that is used by the process *entry*. Rewrite this code in the style shown in Example 2-11. This prevents inadvertent use of the data member variable for interprocess communication.

Example 2-10   Incorrectly Using a Data Member as a Variable

```
SC_MODULE module_name {
     int x; // Data member
     ...
};
/***************************/
module::entry() {
  ...
if (x == 0){
     for (int i=0; i < 7; i++) {
          ... //loop code
     }
}
```

Example 2-11   Correct Use of Local Variables

```
/***************************/
/* Implementation file     */
module::entry() {
...
int x;   // Local variable declaration
...
if (x == 0){
    ...
    for (int i=0; i < 7; i++) {
       ... //loop code
    }
}
```

Recommendations About Data Types

For a single-bit variable, use the native C++ type bool or the SystemC type sc_bit.

For variables less than 64 bits wide, use sc_int or sc_uint data types. Use sc_uint for all logic and unsigned arithmetic operations. Use sc_int for signed arithmetic operations as well as for logic operations.

For variables larger than 64 bits, use sc_bigint or sc_biguint if you want to do arithmetic operations with these variables. If you want to do logic operations, use sc_bv instead.

Use sc_logic or sc_lv only when you need to model three-state signals or buses. When you use these data types, avoid comparison with X and Z values, because such comparisons are not synthesizable.

Use native C++ integer types for loop counters or when you need a variable of the size defined by the native C++ type. For example, on most platforms, a char is 8 bits wide, a short is 16 bits wide, and an int and a long are each 32 bits wide.

Use the C++ enum for all enumerated types – for example, state variables. Use the C++ struct for all aggregate types.

Refining Control

To refine control, you specify I/O behavior and latency.

At this point in the refinement process, your code still looks like the original software algorithm. To refine control, you need to insert wait statements in your code to clearly specify the relative ordering of I/O operations, the cycles in which I/O happens, and the latency of your design.

The placement of wait statements is governed by the coding style rules for the I/O scheduling mode you plan to use. Chapter 3, "Behavioral Coding Guidelines," describes the coding style rules that govern the placement of wait statements in your code.

Follow the coding style rules in order to insert the minimal number of wait statements to schedule your design with SystemC Compiler. For cycle-fixed mode, the total number of waits in the longest path through the code dictates the latency of the design. For superstate-fixed mode, the scheduler determines the latency of the design. Therefore, the latency of your design is specified indirectly through the number of wait statements in the code.

Note that when you specify the I/O behavior of one module, you are constraining the I/O behavior of the modules that interact with it. Similarly, the I/O behavior of your module may be constrained by the I/O behavior of the modules that interact with it. You respect these constraints by properly placing wait statements in your code. Examples of different communication protocols are provided in Chapter 6, "Using Handshaking in the Circuit and Testbench," and several coding examples appear in Appendix B through E.

Advanced Refinement Techniques

If you are not satisfied with the QoR obtained from synthesis after your first pass through refinement, you can use the following techniques to refine your design further to improve the QoR:

- Use preserved functions. See “Using Preserved Functions” on page 4-4.
- Collapse consecutive loops into a single loop. See “Consecutive Loops” on page 3-42.
- Rewrite loops to selectively unroll them. See “Selectively Unrolling Loop Iterations” on page 3-40.
- Use fast handshaking. See “Fast Handshaking” on page 6-36.

Refinement Recommendations

We recommend the following practices during refinement:

- After each step in refinement, reverify your design to ensure that you did not introduce errors during that step.
- Although it is recommended that you thoroughly refine at each refinement stage, it is not necessary. For example, during data refinement, you can refine one data type at a time and evaluate the impact on synthesizability and QOR using SystemC Compiler. Similarly, you may want to replace one non-synthesizable construct with a synthesizable construct, and reverify the design before replacing the next non-synthesizable construct.
- Thoroughly refine the control at one time. Control refinement affects the I/O behavior of the block and the blocks that interact with it. It is easier to fix the I/O timing of all hardware blocks during structure refinement and use that I/O timing during control refinement.

3

Behavioral Coding Guidelines

This chapter describes the behavioral coding style guidelines you can use to ensure successful synthesis with SystemC Compiler.

This chapter contains the following sections:

- Using Clocked Thread Processes
- Using Inputs and Outputs
- Behavioral Coding Style Rules
- Using Conditional Statements
- Using Loops
- Using Resets
- Using Variables and Signals

Using Clocked Thread Processes

A clocked thread process, SC_CTHREAD, is the basic unit for behavioral synthesis with SystemC Compiler. (For general information about processes, see “Processes” on page 2-20.)

Each process is synthesized independently.

Characteristics of the Clocked Thread Process

An SC_CTHREAD process uses wait statements in the SystemC code to synchronize reading from and writing to signals and ports in the process. The SC_CTHREAD process is associated with a single clock and is sensitive to either the clock’s positive or negative edge, which is called the *active* edge. The clock and its active edge are defined in the module’s constructor.

When the SC_CTHREAD process is invoked, it executes statements in the process until either a wait(*n*) or a wait_until(*cond*) statement is encountered. The process is then suspended until the next active edge, or it is suspended until the next active edge where the condition of wait_until is satisfied. All variables that are local to the process are saved when the process is suspended, which means that the process state is implicitly saved. When the process restarts, execution continues at the statement that follows the wait or wait_until statement.

The clock referenced by wait and implicitly referenced by wait_until is the clock specified as an sc_in_clk port, which is defined as the active edge in the module’s constructor.

Using the wait Statement

Each SC_CTHREAD process must have at least one wait statement.

The wait(*n*) statement suspends process execution for *n* active edges of the clock. The default value of *n* is 1.

The outputs of an SC_CTHREAD process are modeled as being registered. When an SC_CTHREAD process writes to an output signal, the value appears after the next active edge of the clock.

Using the wait_until Statement

The wait_until(*cond*) statement suspends the process until the next active edge. If the *cond* expression is false at the active edge, the process remains suspended and the expression is tested at the next active edge. When the *cond* expression is true, the process execution resumes at the statement immediately following the wait_until statement.

The wait_until argument is an expression for testing the value of a port or signal; the port or signal must be type bool or sc_bit. The expression is evaluated at the next active edge of the clock. This is called a delay-evaluated expression, and it must use the delayed method of the signal. For example,

```
wait_until(data_ready.delayed() == 1);
```

You can define complex expressions by using equal, not equal, and, and or (==, !==, &&, ||) operators. For example,

```
wait_until(data_ready.delayed() == 1 &&
          enable.delayed() == 0);
```

For further details about using the wait and wait_until statements, see the *SystemC User's Guide*.

Controlling a Clocked Thread Process

Hardware typically executes continuously. To model this, place the behavior of the hardware inside an infinite loop within the clocked thread process, as shown in bold in Example 3-1. This ensures that the behavior executes continuously. You can use the following types of infinite loops:

Example 3-1 Infinite Loops

```
while (true) {
  // loop operations
}

do {
  // loop operations
} while (true);

for (;;) {
  // loop operations
}
```

Simple Clocked Thread Example

Example 3-2 shows a complete clocked thread example of a complex number multiplier design. This example uses the port assignment methods, *port*.read( ) and *port*.write( ), to differentiate port reads and writes from variable reads and writes. The port read and write methods are shown in bold.

Example 3-2 Simple Clocked Thread Multiplier

```
// cmult.h header file
SC_MODULE(cmult) {
     // Declare ports
     sc_in<sc_int<8> > data_in;
     sc_in_clk clk;
     sc_out<sc_int<16> > real_out;
     sc_out<sc_int<16> > imaginary_out;

     // Declare internal variables and signals

     // Declare processes in the module
     void entry();

     // Constructor
     SC_CTOR (cmult) {

          // Register processes and define
          // the active clock edge
          SC_CTHREAD(entry, clk.pos());
     }
};
/******************************/
// cmult.cc implementation file

#include "systemc.h"
#include "cmult.h"

void cmult :: entry() {
   sc_int<8> a, b, c, d;
   while (true) {
     // Read four data values from input port
     a = data_in.read();
     wait();
     b = data_in.read();
     wait();
     c = data_in.read();
     wait();
     d = data_in.read();
     wait();
     //Calculate and write output ports
```

```
        real_out.write(a * c - b * d);
        imaginary_out.write(a * d + b * d);
        wait();
    }
}
```

Using Inputs and Outputs

SystemC Compiler creates I/O when you write to or read from a port or signal. For describing behavioral coding guidelines, this manual treats ports and signals as identical.

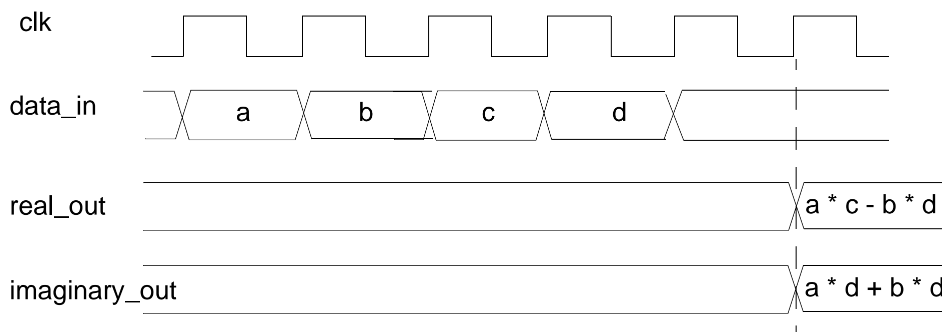
Registered Outputs

SystemC Compiler registers outputs, which means that outputs come from registers rather than directly from combinational logic. SystemC Compiler does not register inputs and does not support bidirectional I/Os for behavioral synthesis.

Inputs and Outputs Within Cycles

Input signals are read at the beginning of a clock cycle. Because outputs are registered, outputs appear at the beginning of the next cycle, as shown in Figure 3-1.

Figure 3-1   Simple Multiplier I/O Protocol



Specifying I/O Read and Write

In a behavioral description, the placement of port read and write operations and wait statements in the code defines the I/O access rate, the sequencing of I/O operations, the I/O groupings, and the relationships between I/O operations. The description of the complex number multiplier in Example 3-2 on page 3-5 implies the I/O protocol in Figure 3-1.

Specifying I/O Cycles

The wait statements in your behavioral SystemC code constrain I/O during scheduling. In the I/O protocol specification in Example 3-2 on page 3-5, each read operation is followed by a wait statement. Therefore, the read operations must occur in the described sequence and are separated by at least one clock cycle. The wait statement after the last read operation means the write operations occur at least one clock cycle after all read operations are complete. Because there is no wait statement between the write operations, both write operations must occur in the same clock cycle.

I/O Scheduling Modes

The effect of the wait statements in your code depends on the I/O scheduling mode, which defines how I/O operations are scheduled (or fixed) in specific clock cycles and dictates how other operations are scheduled around the I/O operations.

You can specify an I/O scheduling mode to be either cycle-fixed or superstate-fixed for the SystemC Compiler `schedule` command. For a description of how to select scheduling modes with the `schedule` command, see Chapter 3, “Scheduling,” in the *CoCentric SystemC Compiler User Guide*.

Cycle-Fixed Scheduling Mode

In cycle-fixed scheduling mode, the I/O behavior of your synthesized design matches your behavioral description cycle by cycle. I/O is fully scheduled based on the wait statements in the behavioral description.

In cycle-fixed scheduling mode, you may need to add wait statements to your code to allow SystemC Compiler to properly construct the FSM and schedule the design.

A testbench monitors I/O at the clock boundaries, and handshake is not required. Verification before and after synthesis is straight forward.

Superstate-Fixed Schedule Mode

In the superstate-fixed scheduling mode, SystemC Compiler can add clock cycles as needed to schedule the design. The region between any two consecutive wait statements in the behavioral code is a

superstate. In the behavioral code, the superstate is one cycle long. In the synthesized design, however, SystemC Compiler can add more clock cycles to the superstate.

In the behavioral code, the I/O reads and writes between the two consecutive wait statements belong to the superstate. I/O writes always take place in the last cycle of the superstate. I/O reads that belong to a superstate can happen in any clock cycle of that superstate.

I/O constraints are not implicit from the behavioral description. For verifying the design before and after synthesis, a testbench with handshake is required. See Chapter 6, “Using Handshaking in the Circuit and Testbench.”

Comparing I/O Scheduling Modes

Using the superstate-fixed I/O scheduling mode allows SystemC Compiler greater flexibility to determine an optimum design. Using cycle-fixed scheduling mode requires that you fully define the I/O schedule with wait statements, which is more difficult than writing the equivalent schedulable description for the superstate-fixed scheduling mode. The superstate-fixed I/O scheduling mode allows you to use SystemC Compiler commands and constraints to quickly perform tradeoff analysis of clock period, latency, and resources without modifying your source code. Latency tradeoff analysis for cycle-fixed scheduling descriptions is not possible.

It is recommended that you use superstate-fixed scheduling mode because the majority of designs are well suited for superstate-fixed scheduling. The verification methodology for a design synthesized with superstate-fixed scheduling mode that uses handshake is described in Chapter 6, “Using Handshaking in the Circuit and Testbench.”

Behavioral Coding Style Rules

Wait statements define the boundaries of clock cycles or a superstate, depending on the scheduling mode. The behavioral coding style rules for placing wait statements in your code are summarized in the following three sections. Examples of using these coding rules begin on page 3-13.

Definition of Coding Rule Terms

Terms used in the coding rules have the following meanings:

- Conditional loop

  A conditional loop is a for, while, or do-while loop that is executed only if the condition evaluates to true.

- Loop iteration condition

  A loop iteration condition (if…else, switch…case, or the ?: operator) is evaluated within a loop. If the condition evaluates to true, the next iteration of the loop is executed. If the condition evaluates to false, the loop is exited.

- Loop continue

  A loop continue means the loop continues with execution of another iteration of the loop.

General Coding Rules

The general coding rules are the following:

1. Place at least one wait statement in every loop, except an unrolled for loop.
2. Place at least one wait statement between successive writes to the same output.
3. Place at least one wait statement after the reset action and before the main infinite loop. Do not include either a conditional branch or a rolled loop in the reset behavior description.
4. If one branch of a conditional (if…else, switch…case, or the ?: operator) has at least one wait statement, then place at least one wait statement in each of the other branches, including the default branch. You can have a different number of wait statements in each branch.
5. Place at least one wait statement immediately after each loop to exit the level of scheduling hierarchy created by the loop.

Cycle-Fixed Mode Coding Rules

In addition to the general coding rules, the cycle-fixed scheduling mode rules are the following:

1. Place at least one wait statement immediately before a conditional loop (for, while, or do-while).
2. With *n* representing the number of cycles required to evaluate a loop iteration condition,

   a. Inside the loop, place *n* wait statements immediately after the loop conditional is evaluated

   b. Outside the loop, place *n* wait statements immediately after the loop exit

   The value of *n* must be at least one. Do not place I/O read or write statements between the *n* wait statements.
3. With *n* representing the number of cycles to perform computations and memory access between reading from an input and writing to an output, place *n* wait statements between reading from and writing to the ports.

Superstate-Fixed Mode Coding Rules

In addition to the general coding rules, the superstate-fixed scheduling mode rules are the following:

1. Place at least one wait statement after the last write inside a loop, and before a loop continue or exit.
2. Place at least one wait statement after the last write before a loop.

General Coding Rules Examples

The general coding rules apply when you are using the `schedule` command with an I/O mode (`-io_mode` option) of either `cycle_fixed` or `superstate_fixed` modes.

Following are the general coding rules and an example of each.

General Coding Rule 1

Place at least one wait statement in every loop, except an unrolled for loop.

Example 3-3 shows a loop without a wait statement, which causes an HLS-52 error in cycle-fixed scheduling mode. To correct this error, insert a wait statement, as shown in bold in Example 3-4.

Example 3-3 Error in Use of General Coding Rule 1

```
for (int i = 0; i>4; i++) {
      e = (a * c * d + i);
      // Error: no wait in loop.
}
```

Example 3-4 Correct General Coding Rule 1

```
for (int i = 0; i>4; i++) {
      e = (a * c * d + 1);
      wait();
}
```

General Coding Rule 2

Place at least one wait statement between successive writes to the same output.

Example 3-5 shows a loop without a wait statement between two successive writes. In either cycle-fixed or superstate-fixed scheduling mode, SystemC Compiler removes the first signal write, because it is eclipsed by the second write. It issues an SCC-142 warning during execution of the `compile_systemc` command. Therefore, when you violate this coding rule, no error is issued when the `bc_check_design` or `schedule` command is executed. However, the first write to the output is not executed. To correct this situation, insert a wait statement, as shown in bold in Example 3-6.

Example 3-5 Error in Use of General Coding Rule 2

```
for (int i = 0; i>4; i++) {
  real_out.write(e + i);
  // Error: no wait between successive writes.
  real_out.write(e + i +1);
  wait();
}
```

Example 3-6 Correct General Coding Rule 2

```
e = (a * c * d);
for (int i = 0; i>4; i++) {
  real_out.write(e + i);
  wait();
  real_out.write(e + i + i);
  wait();
}
```

General Coding Rule 3

Place at least one wait statement after the reset action and before the main infinite loop. Do not include either a conditional branch or a rolled loop in the reset behavior description. For more information about resets, see “Using Resets” on page 3-46.”

Example 3-7 shows reset statements without a wait statement before the infinite loop that implements the process behavior. This causes an HLS-354 error. To correct this error, insert a wait statement, as shown in bold in Example 3-8.

Example 3-7 Error in Use of General Coding Rule 3

```
//Initialize and reset if reset asserts
ready_for_data.write(false);
output_data_ready.write(false);
real_out.write(0);
imaginary_out.write(0);
//Error: need wait after reset.

while (true) {
//Implement process behavior
}
```

Example 3-8 Correct General Coding Rule 3

```
//Initialize and reset if reset asserts
ready_for_data.write(false);
output_data_ready.write(false);
real_out.write(0);
imaginary_out.write(0);
wait();

while (true) {
// Implement process behavior
}
```

General Coding Rule 4

If one branch of a conditional (if…else, switch…case, or the ?: operator) has at least one wait statement, then place at least one wait statement in each branch (including the default branch). You can have a different number of wait statements in the branches.

Example 3-9 shows if…else conditional branching. The if branch has a wait statement, and the else…if and else branches do not have wait statements. This causes an HLS-233 error in cycle-fixed mode, and an HLS-47 error in superstate-fixed mode. To correct this error, insert a wait statement as shown in bold in Example 3-10.

Notice that Example 3-10 has two wait statements in the if branch and only one wait statement in each of the other branches. This is valid code because the number of wait statements does not need to be the same in each conditional branch.

Example 3-9 Error in Use of General Coding Rule 4, If Conditional

```
if (a < b) {
        e = (a * c * d);
        wait();
    }
    else if (a = b) {
        e = (b * c * d);
        //no wait();
    }
    else {
        e = (c * d);
        //no wait();
    }
```

Example 3-10 Correct General Coding Rule 4, If Conditional

```
if (a < b) {
        e = (a * c * d);
        wait();
        wait();
    }
    else if (a = b) {
        e = (b * c * d);
        wait();
    }
    else {
        e = (c * d);
        wait();
    }
```

Example 3-11 shows if…else conditional branching with an implicit else branch. The if branch and the else…if branches have wait statements. An if…else conditional statement implies an else branch by default. The implicit else branch causes an HLS-233 error in cycle-fixed mode, and an HLS-47 error in superstate-fixed mode. To correct this error, insert an else branch with a wait statement as shown in bold in Example 3-12.

Example 3-11 Error in Use of General Coding Rule 4, If Conditional With Implied Else

```
if (a < b) {
        e = (a * c * d);
        wait();
    }
    else if (a = b) {
        e = (b * c * d);
        wait();
    }
```

Example 3-12 Correct General Coding Rule 4, If Conditional

```
if (a < b) {
        e = (a * c * d);
        wait();
        wait();
    }
    else if (a = b) {
        e = (b * c * d);
        wait();
    }
    else {
        e = (c * d);
        wait();
    }
```

Example 3-13 shows switch…case conditional branching. The case 1 branch and the default branch are missing wait statements. This causes an HLS-233 error in cycle-fixed mode and a HLS-43 error in superstate-fixed mode. To correct this error, insert wait statements, as shown in bold in Example 3-14.

Example 3-13 Error in Use of General Coding Rule 4, Switch Conditional

```
switch (sel) {
  case 0: real_out.write(a); wait(); break;
  case 1: real_out.write(b); break;//no wait
  case 2: real_out.write(c); wait(3); break;
  case 3: real_out.write(d); wait(); break;
  default: real_out.write(a + b); // no wait
          break;
}
```

Example 3-14 Correct General Coding Rule 4, Switch Conditional

```
switch (sel) {
  case 0: real_out.write(a); wait(); break;
  case 1: real_out.write(b); wait(); break;
  case 2: real_out.write(c); wait(3); break;
  case 3: real_out.write(d); wait();break;
  default: real_out.write(a + b); wait();
             break;
  }
```

General Coding Rule 5

Place at least one wait statement immediately after each loop to exit the level of scheduling hierarchy the loop creates.

Each loop, except an unrolled for loop, is a level of scheduling hierarchy. Example 3-15 shows a for loop with a nested while loop. Both loops are missing a wait statement immediately after the loop body, which causes an HLS-52 error in cycle-fixed mode. To correct this error, insert wait statements, as shown in bold in Example 3-16.

In superstate-fixed mode, SystemC Compiler adds clock cycles to exit the loop hierarchy. Therefore, not placing a wait statement immediately after the loop does not cause an error. To avoid a mismatch with post-synthesis simulation, you need to add a wait statement after the loop.

You can exit from a local loop by using a conditional break statement. Example 3-15 shows a conditional if statement with a break to exit from the do...while loop.

Example 3-15 Error in Use of General Coding Rule 5

```
...
for (int i = 0; i>4; i++) {
    e = (a - 2);
    wait();
    while (i == 0) {
       e = (b - 2);
       wait();
       do {
          wait();
          e = (b - 2);
          if (i == 0) break;
       } while (i == 0);
       wait();
     }
     // no wait
     real_out.write(e);
     wait();
 }
 //no wait();
 real_out.write(e);
 wait();
 ...
```

Example 3-16 Correct General Coding Rule 5

```
for (int i = 0; i>4; i++) {
   e = (a - 2);
   wait();
   while (i == 0) {
      e = (b - 2);
      wait();
      do {
        wait();
        e = (b - 2);
        if (i == 0) break;
      } while (i == 0);
      wait();
   }
   wait();
   e = (a - 2);
   wait();
}
wait();
...
```

Cycle-Fixed Mode Coding Rules Examples

Following are the cycle-fixed coding rules and an example of each.

Cycle-Fixed Coding Rule 1

Place at least one wait statement before a conditional (for, while, or do-while) loop, except the main infinite loop.

Example 3-17 shows a for loop without a wait statement before the loop, which causes an HLS-52 error. To correct this error, insert a wait statement, as shown in bold in Example 3-18.

Example 3-17 Error in Use of Cycle-Fixed Mode Coding Rule 1, for Loop

```
e = (a * c * d);
// no wait();
for (int i = 0; i>4; i++) {
  e = (a * c * d + i);
  wait();
}
```

Example 3-18 Correct Cycle-Fixed Mode Coding Rule 1, for Loop

```
e = (a * c * d);
wait();
for (int i = 0; i>4; i++) {
  e = (a * c * d + 1);
  wait();
}
```

Example 3-19 shows a while loop without a wait statement before the loop, which causes an HLS-52 error. To correct this error, insert a wait statement, as shown in bold in Example 3-20.

Example 3-19   Error in Use of Cycle-Fixed Mode Coding Rule 1, while Loop

```
e = (a * c * d);
// no wait();
while (e == 0) {
  e = (a * c * d + i);
  wait();
}
```

Example 3-20   Correct Cycle-Fixed Mode Coding Rule 1, while loop

```
e = (a * c * d);
wait();
while (e == 0) {
  e = (a * c * d + 1);
  wait();
}
```

Example 3-21 shows a do-while loop without a wait statement before the loop, which causes an HLS-52 error. To correct this error, insert a wait statement as shown in bold in Example 3-22.

Example 3-21 Error in Use of Cycle-Fixed Mode Coding Rule 1, do-while Loop

```
e = (a * c * d);
// no wait();
do {
  e = (a * c * d + 1);
  wait();
} while (e == 0);
```

Example 3-22 Correct Cycle-Fixed Mode Coding Rule 1, do-while loop

```
e = (a * c * d);
wait();
do {
  e = (a * c * d + 1);
  wait();
} while (e == 0);
```

Cycle-Fixed Coding Rule 2

With *n* representing the number of cycles required to evaluate a loop iteration condition,

1. Inside the loop, place *n* wait statements immediately after the loop conditional is evaluated.
2. Outside the loop, place *n* wait statements immediately after the loop exit.

   The value of *n* must be at least one. Do not place I/O read or write statements between the *n* wait statements in either case.

Example 3-23 shows a while loop with a loop iteration condition that takes several clock cycles to evaluate. For this example, the conditional evaluation takes seven clock cycles. To correct this error, insert wait statements, as shown in bold in Example 3-24.

You can determine *n* number of cycles from the report created by the `bc_time_design` command. To calculate *n*, divide the time required for loop iteration and computation by the available clock period (clock period minus any margin).

Example 3-23 causes an HLS-52 error during `schedule` command execution. See “Finding the Cause of Timing-Dependent Coding Errors” on page 3-31.

Example 3-23 Error in Use of Cycle-Fixed Mode Coding Rule 2

```
while (e == !(a * b * c * d * a * d)) {
  wait(3);  // insufficient wait statements
            // for condition evaluation.
  e = (a * b * c * d * a * d);
}
wait(3);   // insufficient wait statements
           // after loop
```

Example 3-24 Correct Cycle-Fixed Mode Coding Rule 2

```
while (e == !(a * b * c * d * a * d)) {
  wait(7);
  e = (a * b * c * d * a * d);
}
wait(7);
```

Example 3-25 shows a while loop with a loop iteration condition that takes seven clock cycles to evaluate. I/O statements are not allowed during the clock cycles required for condition evaluation. The write statement in this example is a coding error. To correct this error, insert wait statements, as shown in bold in Example 3-26.

Example 3-25 causes an HLS-52 error during `schedule` command execution. Because the coding rule violation is directly related to operator timing, the `bc_check_design` command cannot catch this coding rule violation. See "Finding the Cause of Timing-Dependent Coding Errors" on page 3-31.

Example 3-25 Error in Use of Cycle-Fixed Mode Coding Rule 2, Write

```
while (e == !(a * b * c * d * a * d)) {
  wait(3);  // insufficient wait statements
            // for condition evaluation.
  e = (a * b);
  wait(2);
  real_out.write(e); // No I/O allowed here
  wait();
}
wait(3);   // insufficient wait statements
           // after loop
```

Example 3-26 Correct Cycle-Fixed Mode Coding Rule 2, Write

```
while (e == !(a * b * c * d * a * d)) {
  wait(7);
  e = (a * b);
  wait(2);
  real_out.write(e);
  wait();
}
wait(7);
```

Cycle-Fixed Coding Rule 3

With *n* representing the number of cycles to perform computations and memory access between reading from an input and writing to an output, place *n* wait statements between reading from and writing to the ports.

You can determine *n* number of cycles from the report created by the `bc_time_design` command. To calculate *n*, divide the time required for loop iteration and computation by the available clock period (clock period minus any margin).

Example 3-27 shows a computation that takes several clock cycles, but the code allows only one clock cycle. To correct this error, insert the appropriate number of wait statements to allow the computation to complete before writing the output as shown in bold in Example 3-28.

Example 3-27 causes an HLS-52 error during `schedule` command execution. See “Finding the Cause of Timing-Dependent Coding Errors” on page 3-31.

Example 3-27 Error in Use of Cycle-Fixed Mode Coding Rule 3

```
c = data_in.read();
d = data_in.read();
e = (c * d * c * d * (d + d));
wait(); // insufficient wait statements
        // for computation.
real_out.write(e);
wait();
```

Example 3-28 Correct Cycle-Fixed Mode Coding Rule 3

```
c = data_in.read();
d = data_in.read();
e = (c * d * c * d * (d + d));
wait(8);  // wait for computation
real_out.write(e);
wait();
```

Superstate-Fixed Mode Coding Rules Examples

Following are the superstate-fixed coding rules and an example of each.

Superstate-Fixed Coding Rule 1

Place at least one wait statement after the last write inside a loop and before a loop continue or exit.

Example 3-29 shows a for loop without a wait statement after a write to an output and before the loop continue or break, which causes an HLS-46 error. To correct this error, move the wait statement from after the if statement to before the if statement, as shown in bold in Example 3-30.

Example 3-29 Error in Use of Superstate-Fixed Mode Coding Rule 1

```
for (int i = 0; i>4; i++) {
  e = (a * c * d + i);
  real_out.write(e);
      // no wait after last write
      // and before continue or exit.
  if (i == data_in.read()) break;
  wait();
}
```

Example 3-30 Correct Superstate-Fixed Mode Coding Rule 1

```
for (int i = 0; i>4; i++) {
  e = (a * c * d + i);
  real_out.write(e);
  wait(); // move the wait before the if
  if (i == data_in.read()) break;
}
```

Superstate-Fixed Coding Rule 2

Place at least one wait statement after the last write before a loop.

Example 3-31 shows a write to an output without a wait statement before the loop. This causes an HLS-44 error. To correct this error, add a wait statement before the loop statement, as shown in bold in Example 3-32.

Example 3-31 Error in Use of Superstate-Fixed Mode Coding Rule 2

```
e = (a * c * d + 1);
real_out.write(e);
// Error, no wait before loop

for (int i = 0; i>4; i++) {
  real_out.write(e);
  wait();
  c = data_in.read();
  e = (a * c * d);
}
```

Example 3-32 Correct Superstate-Fixed Mode Coding Rule 2

```
 e = (a * c * d);
 real_out.write(e);
wait(); // Add wait before loop.

for (int i = 0; i>4; i++) {
  real_out.write(e + i);
  wait();
  c = data_in.read();
  e = (a * c * d);
}
```

Finding the Cause of Timing-Dependent Coding Errors

Use the SystemC Compiler `bc_check_design` command to detect coding errors that are not dependent on operator timing. You can use the `bc_check_design` command prior to using the `bc_time_design` or `schedule` commands. The `schedule` command checks for all errors, including those dependent on operator timing. For details about these commands, see the *CoCentric SystemC Compiler User Guide*.

Using Conditional Statements

Use conditional statements (if...else, switch...case, and the ?: operator) in your code to specify your control flow.

SystemC Compiler uses the structure of conditional blocks to determine mutually exclusive operations. Mutually exclusive operations can share hardware, which reduces design costs.

In Example 3-33, SystemC Compiler does not consider the operations A + B and A – B as mutually exclusive because they appear in different if statements.

Example 3-33 Operations That Are Not Mutually Exclusive

```
if (A < 0) {
     out = A + B;
}
if (A >= 0) {
     out = A - B;
}
```

You can combine both operations into an else...if statement, as shown in Example 3-34, so SystemC Compiler considers the operations mutually exclusive.

Example 3-34 Mutually Exclusive Operations

```
if (A < 0) {
     out = A + B;
}
else if (A => 0) {
     out = A - B;
}
```

Using Loops

Loops repeat a sequence of operations. SystemC Compiler synthesizes hardware based on while loops, do...while loops, and for loops.

Understanding How Loops Are Scheduled

If a design contains nested loops, SystemC Compiler schedules the innermost loop first, then successively schedules the next (outward) loop until all the loops are scheduled.

After scheduling an inner loop to a number of cycles, SystemC Compiler treats the loop as though it is fixed, which means the inner-loop operations must remain scheduled relative to each other. The latency of the inner loop reported by SystemC Compiler equals the longest path through all the inner loop's iterations.

Similarly, the latency reported by SystemC Compiler of an outer loop equals the longest path through the loop, including the latency of any inner loops.

Timing constraints on the outer loops do not affect scheduling of the inner loops, but timing constraints on the inner loops affect scheduling of the outer loops. Therefore, specify timing constraints on inner loops rather than on outer loops.

When a design contains successive loops at the same level, SystemC Compiler preserves the source code ordering of these loops, even if there are no data dependencies between them. This means that the first loop in the source code will be fully executed in hardware before the loop that follows it is entered.

Labeling a Loop

To simplify setting constraints on loops, give each loop a label. If you do not assign labels, SystemC Compiler assigns a default name to each loop. Example 3-35 shows in bold how to label a loop with either a C language line label or a `synopsys` compiler directive. If both are applied to a line of code, SystemC Compiler uses the C line label for scheduling constraints and in generated reports.

Example 3-35 Labeling a Loop

```
my_module1 :: entry {
    // C style line label
    reset_loop1: while (true) {
      ...
      wait();
      ...
      wait();
    }
}

my_module2 :: entry {
        // Synopsys compiler directive
    while (true) { //snps line_label reset_loop2
      ...
      wait();
      ...
      wait();
    }
}
```

In reports generated by SystemC Compiler commands, the label is reflected in report hierarchy as

```
my_module1
   entry
      reset_loop1
```

Using while Loops

A while loop has a conditional exit that can be dynamic, which means it is data dependent and not determinable at compile time. A while loop is always rolled and has its own level of hierarchy. Example 3-36 shows the structure of a while loop.

Example 3-36   Structure of a while Loop

```
//SystemC

while (cond) {
  //operations
}
```

Using an Infinite while Loop

An infinite while loop is one with a condition that is always true, as shown in Example 3-37. SystemC Compiler requires that you enclose a clocked thread process body within an infinite while loop.

Example 3-37   Infinite while Loop

```
while (true) {
  //operations of clocked thread
}
```

Using do...while Loops

You can use the do...while loop construct in situations where you want to guarantee that the loop is executed once before evaluation of the while condition test. Example 3-38 shows the structure of the do…while loop.

Example 3-38 Structure of do...while Loop

```
do {
  //operations
} while(cond);
```

Using for Loops

A for loop executes a certain number of iterations by using an iteration counter, as shown in Example 3-39.

Example 3-39 for Loop

```
for (int i = 0; i <= 7; i++) {
  //operations
}
```

Rolled Versus Unrolled Loops

SystemC Compiler keeps all loops rolled by default. An unrolled loop replicates the code body for each loop iteration. If you unroll a loop, it is no longer considered a loop during synthesis.

Rolled for Loops

SystemC Compiler keeps for loops rolled by default. This strategy means

- Shorter elaboration times
- Serial execution of each iteration
- Less area
- Shorter scheduling times

Rolled for loops have their own level of hierarchy for scheduling. This means that the scheduled loop is treated as a subdesign of the entire design.

Unrolling for Loops

You can force a for loop to unroll by using the `synopsys unroll` compiler directive. Place the `synopsys unroll` compiler directive as a comment in the first line in the body of the for loop, as shown in bold in Example 3-40.

Example 3-40 Unrolled for Loop Compiler Directive

```
...
for (int i=0; i < 8; i++) {
  // synopsys unroll
  .. // loop operations
}
...
```

The advantages to unrolling are the following:

- The iterations are performed in parallel to reduce latency, if there are no data dependencies.
- Loop overhead is eliminated, reducing loop counting hardware.
- Execution of consecutive loop iterations has the potential to overlap.
- Execution of operations before and after the loop can overlap with loop execution.
- Constants can propagate to the operations in the loop body.

Data dependencies between loop iterations allow overlapping of their schedules.

Unrolling for loops can sometimes improve the quality of scheduling by reducing latency, but it can also produce longer SystemC Compiler runtimes and larger designs.

Example 3-41 shows an unrolled for loop.

Example 3-41 Unrolled for Loop and Its Execution

```
//SystemC

x4_times: for (i = 0; i <= 3; i++) {
     /* synopsys unroll */
     a[i] = b[i] + c[i];
}
```

When the for loop in Example 3-41 is unrolled, the loop appears as

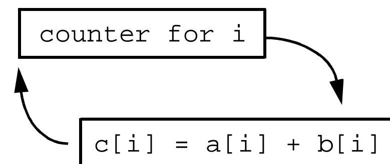
```
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
a[2] = b[2] + c[2];
a[3] = b[3] + c[3];
```

Comparing Rolled and Unrolled Loops

Figure 3-2 shows a comparison of rolled and unrolled loops.

Figure 3-2 Rolled and Unrolled for Loops

```
rolled_loop:
  for (int i=0; i<=7; i++) {
       c[i] = a[i] + b[i];
       wait();
  } // end rolled_loop
```




```
unrolled_loop:
  for (int i=0; i<=7; i++) {
    /* synopsys unroll */
    c[i] = a[i] + b[i];
    wait();
  } // end unrolled_loop
```

```
c[0] = a[0] + b[0]
c[1] = a[1] + b[1]
c[2] = a[2] + b[2]
c[3] = a[3] + b[3]
c[4] = a[4] + b[4]
c[5] = a[5] + b[5]
c[6] = a[6] + b[6]
c[7] = a[7] + b[7]
```

When to Use Unroll

When the cycle budget for implementing the entire for loop is less than the number of iterations multiplied by the latency of the loop, use the `unroll` directive shown in Example 3-42.

Example 3-42 When to Use unroll

```
for (i = 0; i <= 5; i++) {
    /* synopsys unroll */
    // Loop body takes 2 cycles
    a[i] = (b[i] * c[i]) + d[i];
    ...
} // end of unrolled loop
```

If the cycle budget is greater than 10, the for loop in Example 3-42 should not be unrolled, because the number of iterations times the latency of the loop is 10. Keeping the loop rolled gives SystemC Compiler extra cycles to schedule operations, which can reduce design costs while meeting the latency specifications.

Selectively Unrolling Loop Iterations

Keeping a for loop rolled simplifies the scheduling process, but unrolling the loop allows exploration of parallelism between operations in different loop iterations. Selectively unrolling loop iterations helps balance the advantages of rolling and unrolling. Example 3-43 shows a for loop and how you can convert it to a group of nested for loops.

Example 3-43 Selective Unrolling of a for Loop

```
//Rolled for loop

for (k=0; k<=7; k++) {
     // loop_operations that are dependent on k
}

//The same for loop converted to a group
//of nested for loops, selectively unrolled.
loop1: for (i=0; i<=1; i++) {
     loop2: for (j=0; j<=3; j++) {
          /* synopsys unroll */
          k = 4 * i + j;
          //loop_operations that are dependent on k
     }
}
```

To unroll the inner loop while keeping the outer loop rolled, place an `unroll` directive on the inner loop. This change results in scheduling two loop iterations rather than the eight in the original code. Each iteration of the outer loop contains four iterations of the inner. Use this methodology to explore the most efficient implementation while retaining design simplicity.

Ensuring a Statically Determinable Exit Condition

SystemC Compiler requires that unrolled for loops have a statically determinable exit condition at compile time. Example 3-44 is an example of code that does not have a statically determinable exit condition.

Example 3-44   for Loop Without Static Exit Condition

```
if (x) count = 12;
else count = in_port.read();
for (i = 0; i <= count; i++) {
     /* synopsys unroll */
     //operations
     wait();
}
```

The code in Example 3-44 does not have a statically determinable exit condition because the value of count depends on the value of an input, which cannot be determined at compile time. In this situation, SystemC Compiler ignores the `unroll` directive, keeps the loop rolled, and issues a warning that it cannot unroll the loop.

Consecutive Loops

Each loop is a level of scheduling hierarchy. According to general coding rule 5, you need to place a wait statement immediately after each loop to exit the level of scheduling hierarchy the loop creates. When your design contains consecutive loops, there is an overhead of one clock cycle latency to exit the loop hierarchy, as shown in bold in Example 3-45.

Example 3-45 Consecutive Loops With Overhead

```
...
for(int i = 0; i>4; i++) {
        e = (a - 2);
        wait();
        while (i == 0) {
          e = data_in.read();
          wait();
        }
        wait(); // Extra cycle
        while (i == 1) {
          e = (b + 3);
          wait();
        }
        wait();
        real_out.write(e);
        wait();
    }
    wait();
...
```

If your design has consecutive loops, you can improve the latency by modifying your code to collapse consecutive loops, as shown in Example 3-46.

Example 3-46 Collapsed Consecutive Loops

```
...
// Collapse the consecutive loops
    for (int i = 0; i>4; i++) {
        wait();
        e = (a - 2);
        wait();
        while (i == 0 || i == 1) {
          wait();
          if (i == 0){
             e = data_in.read();
          }
          else if (i == 1){
             e = (b + 3);
          }
          else e = (a - 2);
        }
        wait();
        real_out.write(e);
        wait();
     }
   wait();
   }
}
...
```

Pipelining Loop Rules

Loops can be automatically pipelined. For information about using the `loop_pipeline` command, see the *CoCentric SystemC Compiler User Guide*. The pipeline rules are the following:

- Only rolled loops can be pipelined.
- Pipelined loops cannot contain other loops except unrolled for loops.
- Pipelined loops cannot contain a wait_until statement.
- Loop latency must be an integer multiple of the initiation interval, as illustrated in Figure 3-3.
- Loop exits can occur only within the initiation interval, as illustrated in Figure 3-4.

Figure 3-3 Loop Latency and Initiation Interval

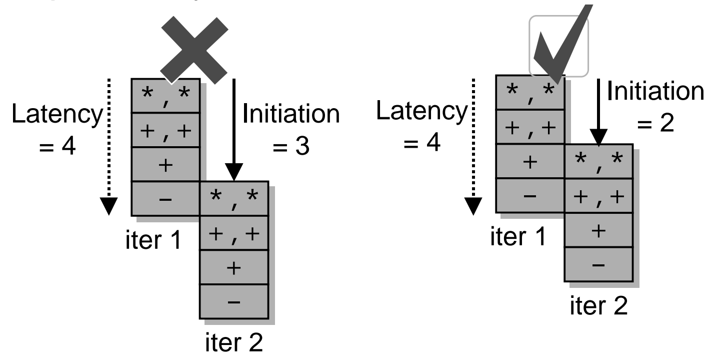
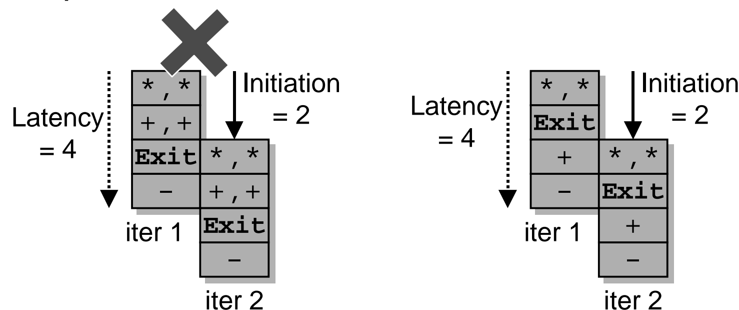


Figure 3-4 Loop Exit



Using Resets

A reset for SystemC Compiler is a global signal that resets the FSM registers and other registers in the design to a known state. Describe the global reset behavior explicitly in the SystemC code so you can simulate the reset behavior at the behavioral level.

Describing a Global Reset

You can define only one global reset signal per process for synthesis. To define global reset behavior, you need to specify an input signal to watch, as shown in Example 3-47 in bold. Notice that the reset port is an sc_in port of type bool. Use the delayed method in the constructor so reset assertion is checked at every active clock edge in the process.

Specifying the Reset Behavior

Specify the reset behavior before the infinite while loop, as shown in Example 3-47 in bold. In the reset behavior, define the appropriate constant values for ports, internal signals, and variables needed to reset the design.

The reset behavior must not contain

- Conditional branches such as an if…else or switch…case statements, or the ?: operator
- Rolled loops such as a for, while, or do-while loop
- Operations that require more than the mandatory wait statement

A wait statement, also shown in bold in Example 3-47, is required before the infinite while loop that contains the main process body. For details about the reset coding rule and an example of it, see “General Coding Rule 3” on page 3-15.

Place initialization or operations that require one or more wait statements at the beginning of the main behavioral process body rather than making it part of the reset behavior.

Example 3-47 Global Reset Watching

```
//Interface file for module
SC_MODULE(example) {
     sc_in<bool> reset;
     sc_in_clk clk;
     sc_out<bool> out_valid;
     sc_out<sc_uint<8> > out1, out2;
     //other ports
     ...
     void entry();
     //constructor
       SC_CTOR(example) {
          SC_CTHREAD(entry, clk.pos());
          // Declare global watched signal here
          watching(reset.delayed() == true);
     }
};

/*******************************/
//Implementation file for module
#include "systemc.h"
#include "example.h"
void example::entry()
{
     //Code to handle reset
     out_valid.write(true);
     out1.write(0);
     out2.write("11111111");
     wait(); //wait required before while loop
```

```
     //Infinite while loop with process behavior
     while (true) {
          //process behavior
     }
}
```

Specifying a Reset Implementation

You can define only a synchronous reset. It is possible to force asynchronous reset behavior in the gate-level description by specifying a specific implementation, using the `set_behavioral_reset` command during synthesis. The command can also be used to set other properties of the reset behavior of the design.

For a discussion of reset implementation, see the *CoCentric SystemC Compiler User Guide* or the man page for the `set_behavioral_reset` command.

Using Variables and Signals

Storing data values that are internal to a SystemC process as signals or variables can significantly affect coding flexibility and the quality of results (QOR). Store intermediate results in variables. SystemC Compiler can use one register to store multiple variables if the lifetimes of the variables do not overlap. Register sharing reduces design costs.

Use variables instead of signals whenever possible to store intermediate results, because

- SystemC Compiler can move variables (non-I/O operations) anyplace in the schedule if doing so does not violate data and control dependencies
- SystemC Compiler allocates a dedicated register for each signal used in the process; variables can share registers if the variable lifetimes do not overlap
- Signal reads and writes are constrained by wait statements (depending on the I/O scheduling mode); variable reads and writes are not constrained by wait statements

Initializing Variables

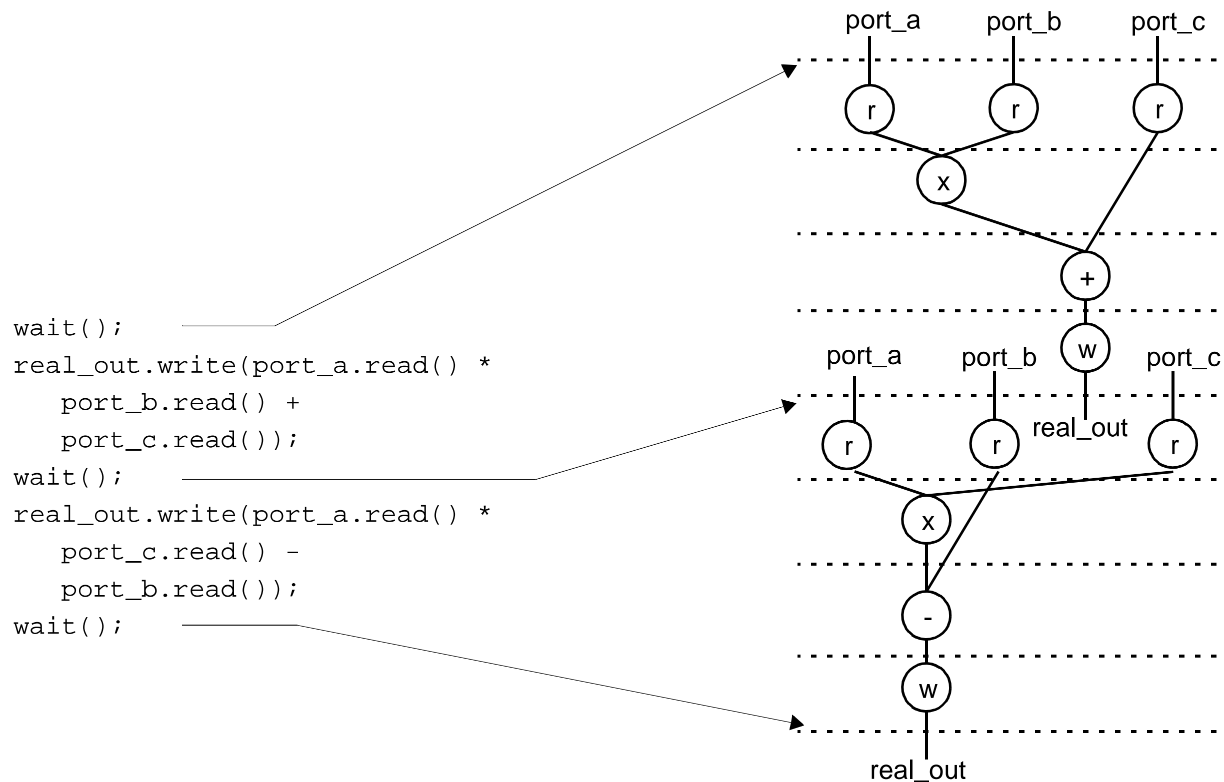
SystemC Compiler supports initialization of ports, signals, or variables only during global reset, as shown in Example 3-47 on page 3-47. Use a global reset to define initial values, to ensure that pre-synthesis and post-synthesis simulation results match.

Using Signals and Wait Statements

Figure 3-5 shows a segment of code that reads data from signals, performs a calculation, and writes to the output port. Rather than storing the read data in variables, the code reads again from the signals to perform the second calculation. Seven or more clock cycles are required to execute this segment of code. The operations happen sequentially because the read operations create data dependencies.

Figure 3-5 illustrates the design's data flow graph, where circles represent operations and lines represent dependent data relationships. A circle containing an r represents a port read operation, a w is a port write, x is a multiply, and so forth. The dashed lines represent clock cycles.

Figure 3-5 Comparing Signal Use and Data Flow



SystemC Compiler implements the following functionality for the code snippet in Figure 3-5:

1. Read port_a, port_b, and port_c.
2. Compute the result of (port_a * port_b + port_c).
3. Write the result to real_out at the end of the clock cycle.
4. Read port_a, port_b, and port_c again in the next clock cycle.
5. Compute the result of (port_a * port_c - port_b).
6. Write the result to real_out in the next clock cycle.

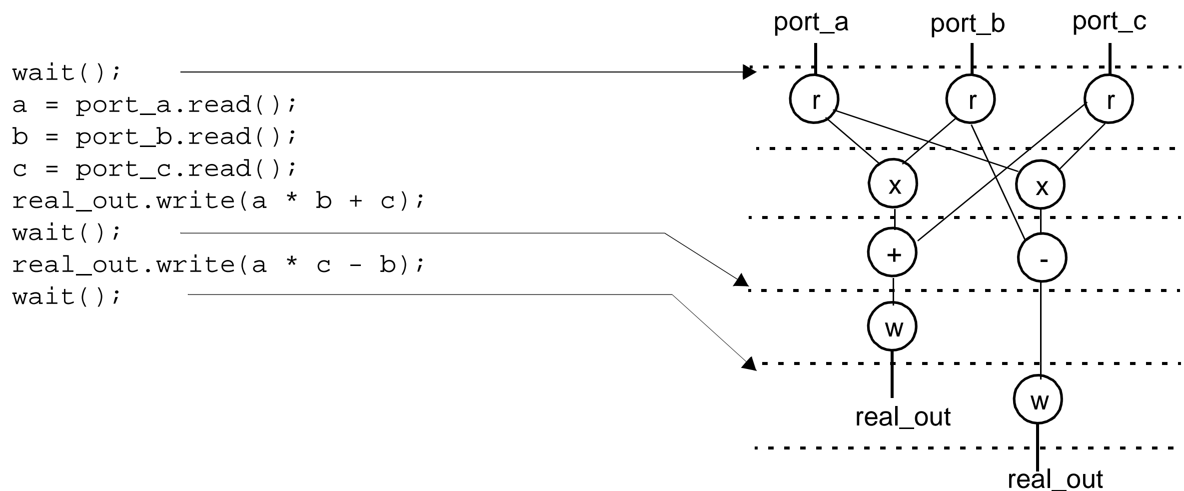
Note that this design schedules in `superstate_fixed` mode but fails to schedule in `cycle_fixed` mode unless the operations between a read and write can be computed in one clock cycle. See “Cycle-Fixed Mode Coding Rules” on page 3-12.

Using Variables and Wait Statements

Figure 3-6 shows a data flow graph and a modified version of the code segment in Figure 3-5. In this version, variables store intermediate results.

The ports are read and assigned to variables. Because the operations based on variables can happen as soon as the variable data becomes available, data availability rather than the wait statements constrain operations. When you use variables, SystemC Compiler processes operations in parallel rather than sequentially.

Figure 3-6   Variable Use and Data Flow



SystemC Compiler implements the following functionality for the code in Figure 3-6:

1. Read port_a, port_b, and port_c and assigns their values to variables a, b, and c.
2. Compute the result of (a * b + c) and write the result to the real_out port.
3. Compute the result of (a * c - b) and write the result to the real_out port.

Note that the functionality of Figure 3-5 is different from the functionality of Figure 3-6 if the values of port_a, port_b, and port_c change in successive cycles. As a general recommendation, read ports, store their values in variables, and read the ports when you need new data.

Using Variables for Register Allocation Efficiency

The efficiency of register allocation depends on how the SystemC design description uses variables. SystemC Compiler can map a variable to many registers or many variables to a single register.

Registers store multiple variables if the lifetimes of the variables do not overlap. A register can store different variables, and the same variable can be stored in different registers at different times. SystemC Compiler minimizes the number of registers needed for the design.

Determining the Lifetime of Variables

SystemC Compiler automatically determines the lifetime of variables. The way you write your design description, however, affects variable lifetime.

The lifetime of a variable starts with the cycle it is first assigned to and ends at the end of the cycle when it is last used. The last use of a variable is the latest reference to its value. Multiple assignments to the same variable are equivalent to single assignments to different variables with different lifetimes.

4

Using Functions and DesignWare Components

Use functions to increase code readability and to reduce scheduling complexity. Using functions to encapsulate combinational and sequential behavior allows you to reuse these structures in multiple places in your code. Functions are also useful for grouping bit manipulating logic or timing-critical operations.

This chapter contains the following sections:

- Using Member Functions
- Using Nonmember Functions
- Using Preserved Functions
- Using DesignWare Components

Using Member Functions

In C++, a function defined inside a class is called a member function. These member functions have access to all the class data variables, and they provide a powerful means for describing functionality.

Example 4-1 illustrates declaring, defining, and calling a member function (in bold). The semantics of C++ allow you to define a member function before or after the function is called.

You can define and use any number of member functions. A member function can contain wait statements, but you are not required to have a wait statement in a function. Appendix E, “Inverse Quantization Example,” shows an example that uses numerous member functions to ensure that the complex functionality is easy to understand.

Example 4-1 Member Function

```
//member_example.h file
SC_MODULE(member_example) {
  ...
  //Member function declaration
  sc_int<19> mac_func(sc_int<8> x,
             sc_int<8> y,
             sc_int<8> z);

  SC_CTOR(member_example) {
    SC_CTHREAD(entry, clk.pos();
  }
}

/*************************/
//member_example.cpp file
#include <systemc.h>
#include <member_example.h>

//Member function definition
sc_int<19>member_example::mac_func (
    sc_int<8> x,
    sc_int<9> y,
    sc_int<19> z) {
  sc_int<19> temp;
  temp = x*y;
  temp += z;
  return temp;
}

void member_example::entry() {
  sc_int<19> tmp_out;
  sc_int<8> val1, val2, val3;
  ...
  //Calling the member function
  tmp_out = mac_func(val1, val2, val3);
  real_out.write(tmp_out);
  wait();
  ...
}
```

Using Nonmember Functions

You declare a nonmember function outside of a class. Nonmember functions are standard C functions that you can use for any purpose. Nonmember functions can contain wait statements, but they do not need a wait statement.

Note that nonmember functions do not have access to the data members of a class.

Using Preserved Functions

Preserved functions allow you to create complex components. By default, SystemC Compiler creates inline code for functions and removes the level of hierarchy the functions might represent. You can direct SystemC Compiler to preserve a function instead of inlining it.

For each preserved function, SystemC Compiler creates a level of hierarchy during elaboration. During synthesis, the level of hierarchy is compiled into a component that is treated exactly the same way as any other combinational component, such as an adder or a multiplier. Only functions that describe purely combinational RTL designs can be preserved.

When to Preserve Functions

Use a preserved function when you want to do the following:

- Preserve a complex function as an operator
- Group components that belong in the same cycle into one operation so SystemC Compiler treats the encapsulated function as a single operator
- Incorporate custom netlists into your design (for example, preexisting combinational and pipelined parts)
- Precompile parts and enable more accurate timing estimation
- Use the preserved function as a resource that can be shared

Preserved Function Restrictions

The following sequential constructs are not allowed in preserved functions:

- Sequential DesignWare parts, such as memories and pipelined parts, although the preserved function itself can be pipelined
- Wait statements
- Signal reads and writes
- Rolled loops
- Preserved functions (no nesting of preserved functions)

Creating Preserved Functions

To preserve a function, annotate it with the `preserve_function` compiler directive, as shown in bold in Example 4-2. This example also shows the declaration, definition, and the call to the preserved member function in bold. Note that the `preserve_function` directive must be the first line in the function body.

A preserved function may be either a member function or a nonmember function. If it is a member function, define the function in the implementation file.

Example 4-2 Creating Preserved Functions

```
// cmult_hs.h header file
SC_MODULE(cmult_hs) {
    // Declare ports
    ...
    // Declare processes in the module
    void entry();

        // Declare member functions
    sc_int<19> my_prefunc (sc_int<8> aa,
               sc_int<8> bb, sc_int<8> cc);
    ...// Constructor
};
/*********************************/
// cmult_func.cc implementation file

#include "systemc.h"
#include "cmult_func.h"

void cmult_hs :: entry() {
   sc_int<8> a, b, c, d;
   sc_int<19> e;
   sc_int<8> val1, val2, val3;

   //Initialize and reset if reset asserts
    ...
    while (true) {
       ...
       e = my_prefunc(val1, val2, val3);
       real_out.write(e);
       wait();
   } //end while
} // end entry

// Definition of preserved function.
sc_int<19>  cmult_hs::my_prefunc (
   sc_int<8> aa,
   sc_int<8> bb,
   sc_int<8> cc) {
   /* snps preserve_function */
     sc_int<19> temp;
```

```
        temp = aa * bb;
        temp += cc;
        return temp;
}
```

SystemC Compiler automatically synthesizes preserved functions into components, using a default compile strategy. You can implement finer control of the compile strategy by using the `compile_preserved_functions` command prior to using the `bc_time_design` command.

You can direct the `compile_preserved_functions` command to save the synthesized components as .db files. Then you can use the `read_preserved_function_netlist` command to read in the previously synthesized component as a preserved function. This means you do not have to resynthesize the preserved functions every time you use SystemC Compiler.

For information about using the `compile_preserved_functions` and `read_preserved_function_netlist` commands, see the *CoCentric SystemC Compiler User Guide*.

Nonmember Preserved Functions

You can define a nonmember function as a preserved function. Define the nonmember function in the same file as the module that uses it. Place the `preserve_function` compiler directive in the first line in the block of code that defines the nonmember function, as shown in bold in Example 4-3.

Example 4-3 Nonmember Preserved Function Declaration

```
//my_module.h header file
SC_MODULE(my_module) {
  ...
  ...
  SC_CTOR(my_module) {
    SC_CTHREAD(entry,clk.pos());
  }
}

/**********************/
// my_module.cc implementation file
#include "my_module.h"

// Define my_func
int my_func(int y, int& x) {
  /* synopsys preserve_function */
  x = x + y;
  return x;
}

void my_module::entry() {
  int a, b, c;
  ...
  c = my_func(a , b);
  ...
}
```

Using Reference Parameters in Preserved Functions

SystemC Compiler maps nonconstant C++ reference parameters to the output ports of the design corresponding to a preserved function. If the preserved function contains a read from a reference parameter, SystemC Compiler assumes that you are trying to read an output port and issues an error message unless you use the `inout_param` compiler directive, shown in bold in Example 4-4. Notice that the `inout_param` is placed immediately after the reference parameter and is inside the parentheses. The `preserve_function` directive is the first line in the function body.

Example 4-4 Preserved Function With Reference Parameter

```
void my_func (int y, int& x /*snps inout_param */) {
  /* snps preserve_function */
  x = x + y;
}
```

When you use the `inout_param` compiler directive, SystemC Compiler creates an input port x and an output port x' for the x reference parameter so it can perform the read and write.

Using DesignWare Components

The map_to_operator compiler directive performs an action similar to the preserve_function compiler directive, except that it enables use of standard DesignWare components.

Using map_to_operator

Example 4-5 shows code in bold that uses a DesignWare component. The map_to_operator and return_port_name compiler directives must be the first line in the function body.

Example 4-5 Using DesignWare Parts

```
//Code fragment

sc_int<16> my_mult (const sc_int<8> A,
                    const sc_int<8> B) {

   // snps map_to_operator MULT2_TC_OP
   // snps return_port_name Z
   // Function code block
   ...
   return (A*B);
}
```

After you execute the SystemC Compiler compile_systemc command, this function is replaced by the DesignWare component MULT2_TC_OP, provided it exists in a synthetic library.

See the *DesignWare Developer Guide* for information on using DesignWare components.

Guidelines for Using map_to_operator

Functions with the `map_to_operator` compiler directive require special consideration. The following guidelines apply:

- In the declaration of the function's prototype, specify the `map_to_operator` compiler directive in the first line of the function body; for example,

```
int xyz(int a, intb) {
    /* snps map_to_operator XYZ_OP */;
    ...
}
```

- Declare input parameters as either pass-by-value or constant-qualified references. For example, if a and b are inputs,

```
int xyz(int a, const int& b) {
    /* snps map_to_operator XYZ_OP */;
    ...
}
```

- Declare output parameters as nonconstant-qualified references. For example, if a and b are inputs and c is an output,

```
 void abc(int a, const int& b, int& c) {
    /* snps map_to_operator ABC_OP */;
    ...
}
```

  If you have a C++ simulation model, you need to ensure that your code only writes to output c and does not read from it.

- Ensure that the parameter names for inputs and outputs exactly match the DesignWare operator port names, which is required by the linker of the Design Compiler tool. DesignWare operator port names are case-sensitive. SystemC Compiler issues an error if the names do not match.

- The name of the return port must exactly match an output port of the DesignWare component, which is Z by default. You can override the name by using the `return_port_name` compiler directive; for example,

  ```
  int xyz(int a, const int& b) {
     /* snps map_to_operator XYZ_OP */
     /* snps return_port_name C */
     ...
  }
  ```

  If the DesignWare synthetic operator does not have a port Z, you need to include the `return_port_name` directive to specify its name.

- If you use reference parameters, you need to ensure that you are not using an alias by mistake. You create an alias when you pass the same object by reference to different parameters. For example, this problem occurs in the following:

  ```
  //Definition
  void abc(int a, const int& b, int& c) {
     /* snps map_to_operator ABC_OP */
     ...
  }

  void xyz () {
     //function call that causes alias
     abc(x, y, y);
     ...
  }
  ```

  In above example, parameters b and c are bound to the same y variable, causing an error. Another more subtle alias can result from the following function call:

  ```
  abc(x, a[i], a[j]);
  ```

In the above function call, a potential alias occurs, based on the value of i and j. In such a situation, you can use a temporary variable to avoid the problem; for example,

```
abc(x, a[i], temp);
a[j] = temp;
```

5

Using Arrays, Register Files, and Memories

This chapter describes how to use arrays, including how to map arrays to register files and memories. It also provides coding guidelines for efficiently accessing register files and memories.

This chapter contains the following sections:

- Using Arrays
- Array Implementations
- Mapping Arrays to Register Files
- Mapping Arrays to Memories
- Accessing Register Files and Memories Efficiently

Using Arrays

SystemC Compiler supports single-dimension arrays and multidimensional arrays. Variable indexing into arrays creates decoding logic in hardware, and sharing of array index operations creates multiplexing hardware. Array accesses can have an impact on SystemC Compiler runtimes, area estimates, and timing estimates.

Declaring Arrays

You can declare an array of variables or signals as a data member, which allows all processes in a module to access the array. Example 5-1 shows a single-dimension data member array declaration in bold.

Example 5-1 Data Member Array

```
SC_MODULE (my_module) {
   ...
   int arr1[64];
   SC_CTOR(my_module) {
      SC_CTHREAD(process1, clk.pos());
      SC_CTHREAD(process2, clk.pos());
      ...
      }
}
```

You can also declare an array local to a process, which allows only that process to access the array. Example 5-2 shows a multidimensional array declaration local to a process in bold.

Example 5-2 Array Local to a Process

```
void process1() {
   sc_int<8> arr2[64] [32];
   ...
}
```

Reading From and Writing to Variable Arrays

SystemC Compiler creates dedicated decode hardware for each read from or write to an array location. The hardware decodes the index used to reference the array location.

If SystemC Compiler can statically determine that the array access index is a constant, it creates significantly less decode hardware. Example 5-3 shows (in bold) declaring a variable array and reading from it with a constant index and a nonconstant index.

Example 5-3 Reading From a Variable Array

```
sc_int<8> a[16];
sc_int<8> temp;
sc_int<4> i;
...
temp = a[5];  // read constant index 5
...
temp = a[i];  // read nonconstant index i
```

Example 5-4 shows (in bold) declaring a variable array and writing to it with a constant index and a nonconstant index.

Example 5-4 Writing to a Variable Array

```
sc_int<8> a[16];
sc_int<8> temp;
sc_int<4> i;
...
a[5] = temp;  // write constant index 5
...
a[i] = temp;  // write  nonconstant index i
```

Reading From and Writing to Signal Arrays

You can declare an array of signals as sc_signal, sc_out, or sc_in types. Use signal arrays to communicate data between different processes in your design.

Access signal arrays in the same way that you access I/O ports and other signals in your design, and adhere to the coding style rules described in “Behavioral Coding Style Rules” on page 3-10.

As with variable arrays, SystemC Compiler creates significantly less hardware for decoding a constant index. Example 5-5 shows (in bold) declaring a signal array and reading from it with a constant and nonconstant index.

Example 5-5 Reading From a Signal Array

```
sc_signal<sc_int<8> > a[16];
sc_int<8> temp;
sc_int<4> i;
...
temp = a[5].read(); // read constant index 5
...
temp = a[i].read(); // read nonconstant index I
```

Example 5-6 shows (in bold) declaring a signal array and writing to it with a constant index and a nonconstant index.

Example 5-6 Writing to a Signal Array

```
sc_signal<sc_int<8> > a[16];
sc_int<8> temp;
sc_int<4> i;
...
a[5].write(temp);  // write with a constant index
...
a[i].write(temp);  // write with non-constant index i
```

Accessing Slices of an Array Location

SystemC Compiler generates decoding hardware for each read and write access to an array location, even if you are accessing a single bit or a range of bits (called a slice) of the data contained in that array location. If you access multiple slices of the same array location by using separate reads to that location, decode hardware is generated for each read.

Example 5-7 shows accesses to multiple slices within the same array location.

Example 5-7 Multiple Accesses to Slices in the Same Array

```
sc_int<8> a[16];
sc_int<4> temp1, temp2;
sc_int<4> i;
...
temp1 = a[i].range(3,0); // array read of first slice
                           // in location a[i]
temp2 = a[i].range(7,4);  // array read of second slice
                           // in location a[i]
...
a[i].range(3,0) = temp1;  // array write of first slice
                           // in location a[i]
a[i].range(7,4) = temp2;   // array write of second slice
                           // in location a[i]
```

To improve the efficiency of the hardware created, copy the array location into a temporary variable, and access the various slices from the temporary variable. This coding style requires just one array access and creates one instance of decode hardware.

Example 5-8 shows an example of this alternate coding style.

Example 5-8 Multiple Array Accesses Using a Variable

```
sc_int<8> a[16];
sc_int<8> temp;
sc_int<4> temp1, temp2;
sc_int<4> i;
...
temp = a[i];
temp1 = temp.range(3,0);
temp2 = temp.range(7,4);
...
temp.range(3,0) = temp1;
temp.range(7,4) = temp2;
a[i] = temp;
```

Unlike a variable array, you cannot access slices of array locations in a signal array. It is not allowed. Example 5-9 shows the coding style you need to use to access slices of signal array locations.

Example 5-9 Accessing Slices of a Signal Array Location

```
sc_signal<sc_int<8> > a[16];
sc_int<8> temp;
sc_int<4> temp1, temp2;
sc_int<4> i;
...
temp = a[i].read();
temp1 = temp.range(3,0);
temp2 = temp.range(7,4);
...
temp.range(3,0) = temp1;
temp.range(7,4) = temp2;
a[i].write(temp);
```

Array Implementations

By default, SystemC Compiler generates registers and logic for indexing into the arrays (including multidimensional arrays) in the behavioral code. SystemC Compiler generates dedicated logic for each read from or write to an array. This can result in a large amount of logic.

You can improve the synthesis of designs that have large arrays by mapping an array to a register file or memory. If your design includes large arrays (more than 1024 elements) that are not mapped to a register file or memory, SystemC Compiler will issue a warning, because large unmapped arrays can cause long runtimes.

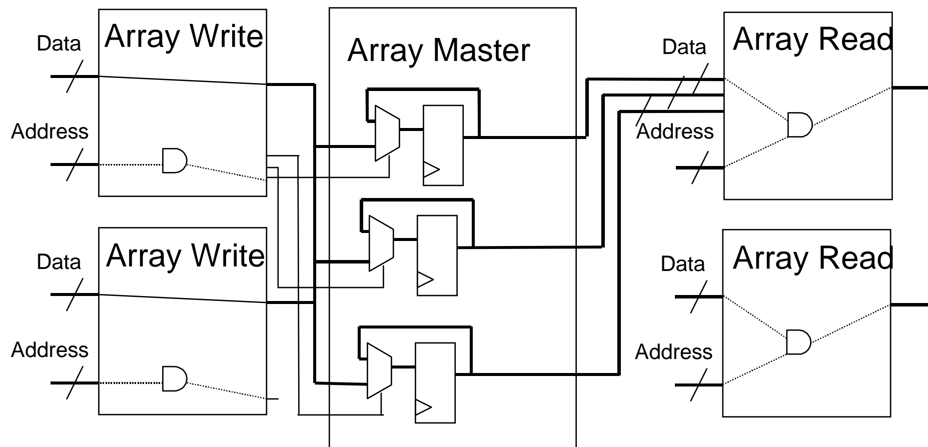
It is generally more efficient to map arrays to memory than to map them to register files. However, unless you have ready access to the appropriately sized memory and all the models you need (a timing model for synthesis and a behavioral model for simulation), it is easier to map arrays to register files. For details about how to use memories, constrain designs with memories, obtain reports about memories, and generate a memory wrapper interface, see the *CoCentric SystemC Compiler User Guide*.

SystemC Compiler can use only synchronous memories. For information about converting an asynchronous memory to a synchronous memory, see the *CoCentric SystemC Compiler User Guide*.

Mapping Arrays to Register Files

Register files are similar to memories, except that SystemC Compiler builds the read and write operators and the register array on the fly. Figure 5-1 shows the architecture of a register file.

Figure 5-1 Register File Architecture



Mapping All Arrays to Register Files

To map all the arrays in your code to register files, set the `bc_use_registerfiles` variable to true.

You can quickly compare the runtime of the `compile_systemc` command with this variable set to true and then to false to see if your design would benefit from mapping arrays to register files or memories. For details about using register files, see the *CoCentric SystemC Compiler User Guide*.

Mapping Specific Arrays to Register Files

To map specific arrays to register files, use the `synopsys resource` compiler directive and the `map_to_registerfiles` attribute in your code to specify the arrays that are to be mapped to register files.

Example 5-10 shows a section of code that uses the `synopsys resource` compiler directive and the `map_to_registerfiles` attribute (shown in bold) to map an array named mem. In this example, R1 is a resource in the synthetic library.

Example 5-10 Mapping Specific Arrays to Register Files

```
sc_int<16> mem[16];
     sc_int<32> mem [16];
     /*synopsys resource  R1:
      variables="mem",
       map_to_registerfiles = "TRUE";*/
   //The following are all mapped to memory.
     //Write to mem
     mem[0] = a;
     mem[1] = b;
     // and so forth

     //Read from mem
     a = mem[0];
     b = mem[1];
     // and so forth
```

Mapping Arrays to Memories

You can map read or write operations of arrays to memory read or write operations. A memory (RAM) contains accessing logic that is transparent to your design.

Map arrays of variables to memories.

You can declare memory locally, which means the memory is accessed only by the process in which it is declared. You can alternatively declare memory as a data member so that the memory is shared by all processes in a design.

To map a specific array to a local or shared memory, use the `synopsys resource` compiler directive and the `map_to_module` attribute in your code to specify the array that is to be mapped to memory.

Local Memory

Example 5-11 shows a section of code that maps the array named amem to a memory local to the process. In a local memory declaration, place the compiler directives immediately after the array declaration, as shown in bold.

Example 5-11 Declaring Local Memory Resources

```
//SystemC code fragment
while (true){

sc_int<32> amem[16];

     /* synopsys
          resource RAM_A:
          variables = "amem",
          map_to_module = "my_mem_model"
     */
     // array amem mapped to a single RAM
          amem[i] = ser_in;
          a = amem[j];
}
```

The statements in Example 5-11 collectively declare a resource named RAM_A. Accesses to array amem map to this memory. The my_mem_model is the memory wrapper interface (described in *CoCentric SystemC Compiler User Guide*).

DesignWare libraries provide some synchronous memory models such as DW_ram_r_w_s_dff that you can use.

The address range declarations must match the actual memory address range. If multiple arrays map to one memory, SystemC Compiler automatically places them in non-overlapping address spaces in the memory.

Multiple Arrays Accessing One Memory

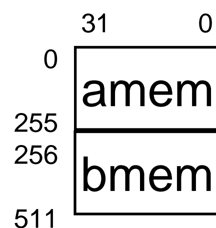
You can use one memory for multiple arrays. Example 5-12 shows two arrays mapped to one memory resource. The memory must be as large as or larger than the combined size of the arrays mapped to it.

Example 5-12 Multiple Arrays Accessing One Memory

```
// Mapping arrays to a RAM
void my_proc() {
  sc_int<8> amem[256];
  sc_int<8> bmem[256];
  /* synopsys resource RAM_A:
     variables="amem bmem",
     map_to_module = "ram1_s_d";
  */
```

When you map multiple arrays to one memory, SystemC Compiler automatically places them in non-overlapping address spaces in the memory. Figure 5-2 shows a representation of the address space mapping of Example 5-12, with two arrays accessing the same memory resource. Address space is allocated in the order the variables are declared.

Figure 5-2 Multiple Array Address Space Mapping



Exploring Alternative Memory Types

You can explore the tradeoffs of using various memory types, such as single port, dual port, or pipelined memories. If the synthetic library descriptions are available for each memory type, you can explore the impact of the different memory types by changing only the `map_to_module` attribute, as shown in bold in Example 5-13.

Example 5-13 Changing Memory Types

```
//Single port memory
while (true){

sc_int<32> amem[16];

     /* synopsys
          resource RAM_A:
               variables = "amem",
               map_to_module = "my_single_port_mem_model"
     */
     // array amem mapped to a single-port RAM
          amem[i] = ser_in;
          a = amem[j];
}

//Change to dual-port memory
while (true){

sc_int<32> amem[16];

     /* synopsys
          resource RAM_A:
               variables = "amem",
               map_to_module = "my_dual_port_mem_model"
     */
     // array amem mapped to a dual-port RAM
          amem[i] = ser_in;
          a = amem[j];
}
```

Accessing Register Files and Memories Efficiently

Minimize the number of array read and write operations accessing a register files or a memory to improve the latency of your design.

By default, SystemC Compiler constrains all accesses to a memory or register file so that they occur one at a time. This prevents multiple accesses from reading or writing the same array location simultaneously. Redundant memory accesses, however, can inflate the latency of your design, so you should avoid them.

You can prevent SystemC Compiler from constraining reads and writes so that they occur one at a time, by using the `ignore_array_precedences` command for register files and the `ignore_memory_precedences` command for memories. See the *CoCentric SystemC Compiler User Guide* for information about using this command.

Accessing Memory

Each memory access requires one or more clock cycles, which has an effect on design latency. For example, if a memory read takes two clock cycles, the circuit needs time to access the memory. In superstate-fixed I/O scheduling mode, clock cycles are automatically inserted. In cycle-fixed scheduling mode, you need to insert wait statements in your code. Example 5-14 shows a memory read that requires a second clock cycle, which is inserted correctly in Example 5-15 in bold.

Example 5-14 Incorrect Memory Read Timing for Cycle-Fixed

```
...
while (true) {
   ...
   wait(); // one cycle
   addr = input_port.read();
   // Need another cycle before write to output
   output_port.write(memory[addr]);
   wait();
```

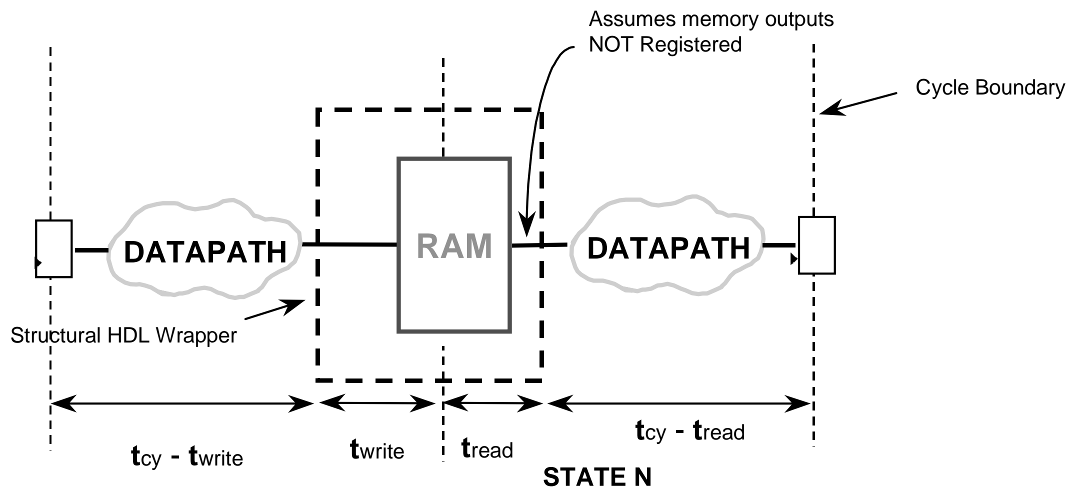
Example 5-15 Correct Memory Read Timing for Cycle-Fixed

```
...
while (true) {
   ...
   wait(); // one cycle
   addr = input_port.read();
   wait(); // second cycle needed
   output_port.write(memory[addr]);
   wait();
```

Allowing for Vendor Memory Timing

Unlike with arithmetic operations, SystemC Compiler does not reserve time in the clock period for vendor timing specifications such as read, write, off-chip, or BIST logic access (see Figure 5-3).

Figure 5-3 Memory Access Time Specification



You need to provide this timing information, using the `set_memory_output_delay` and `set_memory_input_delay` commands. See the *CoCentric SystemC Compiler User Guide* for information about these commands.

Eliminating Redundant Memory Accesses

Every array access infers a memory read or memory write operation. Redundant memory operations result in longer schedules to avoid memory contention.

Example 5-16 creates a redundant memory read.

Example 5-16 Redundant Memory Read

```
x = a[i] + 5;
y = a[i] + 11;
```

A more efficient coding style assigns the array location to a temporary variable, as shown in Example 5-17.

Example 5-17 Array Location Assigned to Temporary Variable

```
temp = a[i];
x = temp + 5;
y = temp + 11;
```

Accessing Bit Slices of Memory Data

Variable and signal accesses (such as assignment or use in an expression) operate on the entire value. When a single bit or a bit slice of a variable or signal is assigned a value, the following steps occur:

1. The original value of the variable or signal is retrieved.
2. The new bit value is patched in.
3. The resulting value is assigned to the variable or signal.

This process is inefficient when you need to access only a bit or slice of memory data. For example, assume that

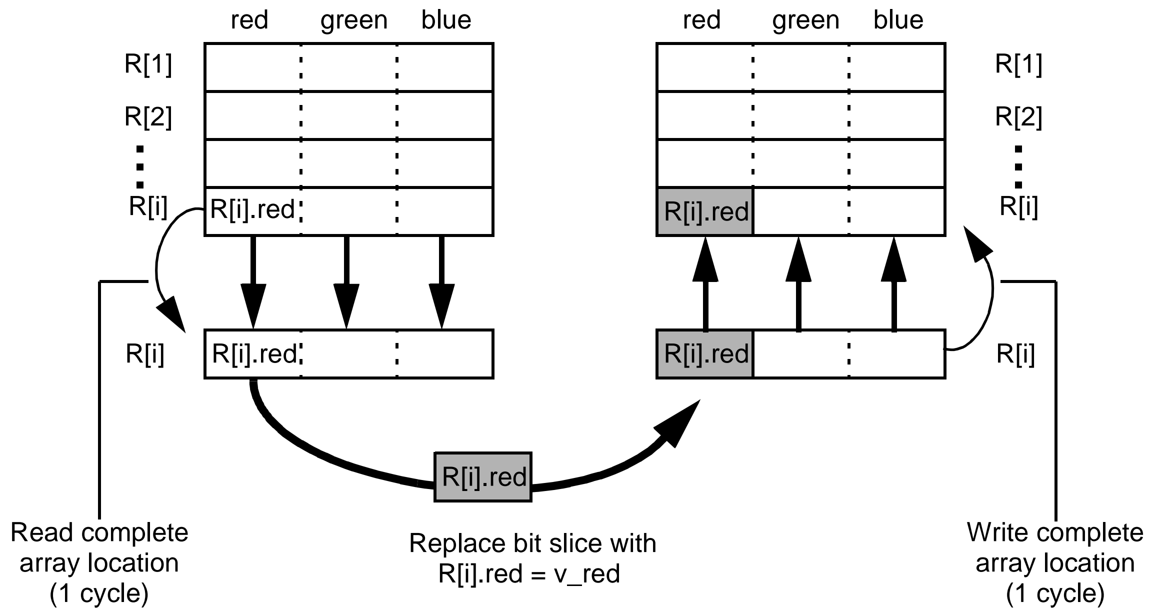
- R is an array of struct types with fields red, green, and blue.
- R maps to a memory with one-cycle read and write operations.

The following assignment requires two cycles—one to read the original value of R[i] and one to write back the new value of R[i]:

```
R[i].red = v_red;
```

SystemC Compiler first reads the array location from memory, because when SystemC Compiler is writing, the full array location is updated (see Figure 5-4).

Figure 5-4   Bit Slice Accesses



The following code takes six cycles to execute—two cycles for each line of code:

```
R[i].red = v_red;
R[i].green = v_green;
R[i].blue = v_blue;
```

You can accomplish this task by using a variable to collect the bit or slice values prior to writing the complete array location.This method requires only one memory write, which executes in one cycle. In the following example, v is a variable of the appropriate struct type:

```
v.red = v_red;
v.green = v_green;
v.blue = v_blue;
R[i] = v;
```

6

Using Handshaking in the Circuit and Testbench

In the superstate-fixed scheduling mode, SystemC Compiler may insert clock cycles in addition to those you specify, in order to properly schedule the design. Therefore, a testbench that you use for verification at the behavioral level may no longer work at the RTL or gate level. To ensure that the same testbench can be used throughout the design process, it is strongly recommended that you use handshaking in both the design and the testbench. Using handshaking also ensures that the block you are designing can communicate with other blocks, regardless of the number of clock cycles introduced during scheduling.

This chapter contains the following sections:

- Using Handshake Protocols
- Using One-Way Handshake Protocols

- Using Two-Way Handshake Protocols
- Fast Handshaking
- Using a Pipeline Handshake Protocol

Using Handshake Protocols

The I/O scheduling mode you use to schedule a design affects your simulation methodology. SystemC Compiler can allocate additional cycles in the superstate-fixed I/O scheduling mode, so you need to use handshake protocols to test and verify the functionality at the register transfer and gate levels.

Handshake protocols allow you to use the same testbench to test the circuit at the behavioral, register transfer, and gate levels of abstraction. You do not have to modify the testbench to compare the behavioral simulation results with the RTL simulation results after scheduling.

Use handshake signals to communicate between the behavioral block and the other blocks in the system. The behavioral block can use handshake signals to notify the other blocks in the system when

- The behavioral block can accept new data
- The outputs of the behavioral block are ready

This ensures that the behavioral block operates the same way before and after scheduling in the context of other blocks in the design.

Using One-Way Handshake Protocols

Use one-way handshake protocols to communicate with other blocks in the system that have a fixed response time. A block with fixed response time is one that can grant a request after a fixed number of cycles from the time the behavioral block issues the request, and the behavioral block does not need an acknowledgement signal from the other block.

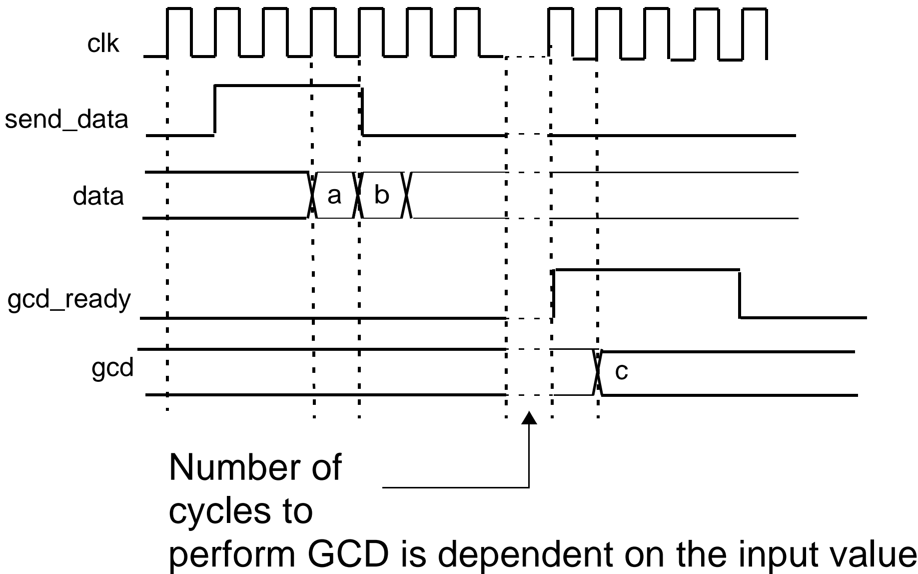
One-Way Handshake Initiated From Behavioral Block

Figure 6-1 shows a timing diagram for a greatest common divisor (GCD) behavioral block that uses one-way handshake protocols to get data and to write data out. The GCD block initiates the handshake with the testbench. Example 6-1 shows the code for the GCD block, the testbench, and the main routine for simulation.

Note:

> The number of cycles needed to compute the GCD is not fixed, because it depends on the value of the two numbers for which the GCD is computed. Therefore, this example requires the use of handshaking.

Figure 6-1 One-Way Handshake Protocol



Example 6-1 One-Way Handshake Protocol Behavioral Block

```
// gcd1.h header file

#define READ_LATENCY 2
#define WRITE_LATENCY 3


SC_MODULE(gcd_mod) {
  // Ports
  sc_in_clk  clk;          // Clock input
  sc_in<bool> reset;       // Reset input
  sc_in<int> data;         // Port to get data
  sc_out<bool> send_data;  // Handshake signal to
                           // request input
  sc_out<bool> gcd_ready;  // Handshake signal to
                           // indicate output is ready

  sc_out<int> gcd;         // Port to send GCD value

  // Process
  void gcd_algo();         // The process that does GCD

  // Internal functions
  int do_gcd(int a, int b);// Function of actual
                            // gcd algorithm

    SC_CTOR(gcd_mod) {
      SC_CTHREAD(gcd_algo, clk.pos());
      watching(reset.delayed() == true);
    }
};


/*********************************/
// gcd1.cc implementation file

#include "systemc.h"
#include "gcd1.h"

void gcd_mod::gcd_algo()
{
    int a, b;  // The two variables to compute gcd
    int c;     // The GCD

    // Reset operations
    gcd.write(0);
    send_data.write(false);
    gcd_ready.write(false);
```

```
    wait();

    while (true) {
         // First get the two inputs
         // using receiver initiated handshake.
         send_data.write(true);
         wait();
         // Wait READ_LATENCY cycles before
         // getting first data.
         wait(READ_LATENCY);
         send_data.write(false);
         a = data.read();
         wait();
         b = data.read();

         // Now do the algorithm
         c = do_gcd(a, b);

         // Now write the output
         // using sender initiated handshake.
         gcd_ready.write(true);
         gcd.write(c);
         wait(WRITE_LATENCY);
         gdc_ready.write(false);
         wait();
    }
}

int gcd_mod::do_gcd(int a, int b)
{
    int temp;

    if (a != 0 && b != 0) {
         while (b != 0) {
              while (a >= b) {
                a = a - b;
                wait();
              }
              temp = a;
              a = b;
              b = temp;
              wait();
         }
    }
    else {
         a = 0;
         wait();
    }
```

```
    return a;
}


/**********************************/
// gcd1_test.h header file.

#ifndef READ_LATENCY
#define READ_LATENCY 2
#endif

#ifndef WRITE_LATENCY
#define WRITE_LATENCY 2
#endif

SC_MODULE(testbench) {
    sc_in_clk clk;
    sc_in<bool> send_data;
    sc_in<bool> gcd_ready;
    sc_in<int> gcd;
    sc_out<bool> reset;
    sc_out<int> data;

    // Process
    void do_run();

    // Internal function
    void do_handshake(int a, int b);

    SC_CTOR(testbench) {
        SC_CTHREAD(do_run, clk.pos());
    }
};


/**********************************/
// gcd1_test.cc implementation file

#include "systemc.h"
#include "gcd1_test.h"

void testbench::do_run()
{
    reset.write(false);
    wait();
    reset.write(true);
    wait();
```

```
    reset.write(false);
    wait();

    cout << "*** Reset Done. Begin Testing. ***\n";

    do_handshake(12, 6);
    do_handshake(172, 36);
    do_handshake(36, 172);
    do_handshake(19, 5);
    do_handshake(2584, 4712);
    do_handshake(15, 0);
    cout << " *** Testing Done ***\n";
    sc_stop();
}

void testbench::do_handshake(int a, int b)
{
    cout << "GCD of " << a << " and " << b
         << " is = ";

    // Receiver initiated handshake.
    // Wait until receiver is ready
    wait_until(send_data.delayed() == true);
    wait(READ_LATENCY - 1);  // Wait for latency
    // Now write data in 2 consecutive cycle.
    data.write(a);
    wait();
    data.write(b);
    wait();

    // Sender initiated handshake.
    // Wait until sender is ready.
    wait_until(gcd_ready.delayed() == true);
    wait(WRITE_LATENCY - 1); // Wait for latency
    // Now read data.
    cout << gcd.read() << endl;
}


/*******************************************/
// gcd1_main.cc top-level simulation model.

#include "systemc.h"
#include "gcd1.h"
#include "gcd1_test.h"

int
main()
```

```
{
    sc_signal<int> data, gcd;
    sc_signal<bool> reset, send_data, gcd_ready;
    sc_clock clock("Clock", 20, 0.5);

    gcd_mod G("GCD");
    G(clock, reset, data, send_data, gcd_ready, gcd);

    testbench TB("TB");
    TB(clock, send_data, gcd_ready, gcd, reset, data);

    sc_trace_file *tf = sc_create_vcd_trace_file("gcd1");
    sc_trace(tf, clock.signal(), "Clock");
    sc_trace(tf, reset, "Reset");
    sc_trace(tf, send_data, "Send Data");
    sc_trace(tf, data, "In");
    sc_trace(tf, gcd_ready, "Out Ready");
    sc_trace(tf, gcd_ready, "Out");

    sc_start(-1);
    return 0;
}
```

The following steps describe how the handshaking protocol works:

1. The behavioral block asserts the handshake signal send_data (high), to indicate that it can process new data. It waits for READ_LATENCY cycles, which gives the testbench time to respond. In this example, READ_LATENCY is 2.
2. The testbench sees the send_data signal and responds with the first piece of data after READ_LATENCY cycles.
3. The behavioral block reads the first piece of data, and in the cycle immediately following, it de-asserts send_data (low) and reads the second piece of data.
4. This process repeats each time the behavioral block can process the next data set.

5. The behavioral block proceeds to compute the GCD of the two numbers it has read. This computation can take an indeterminate, but finite, number of cycles.

The following steps are used to implement the output handshake protocol:

1. The behavioral block asserts the handshake signal gcd_ready (high), to indicate that it can send new output data, and it waits WRITE_LATENCY cycles – where WRITE_LATENCY is 3 in this example. This gives the testbench time to read the output.
2. After the testbench sees the gcd_ready signal, it has WRITE_LATENCY cycles within which to sample the result of GCD.
3. This process repeats each time the behavioral block can send new output data.

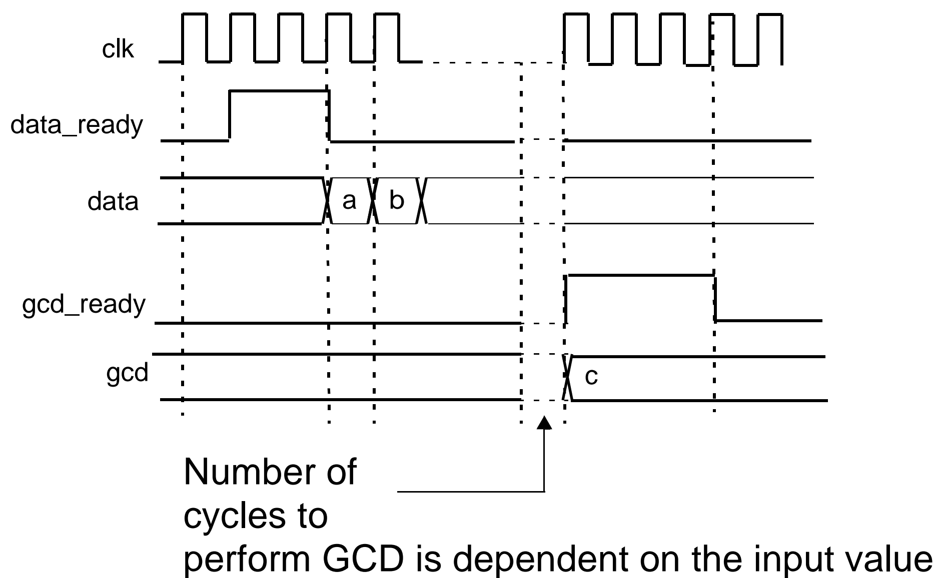
Looking at the testbench code in Example 6-1, you can see that the handshaking for the testbench is done in the do_handshake function. To send data to the behavioral block, the testbench waits until the send_data signal is asserted (high). To model the testbench latency for a read, the code has a wait (READ_LATENCY - 1) statement. At the end of this wait, the testbench writes two consecutive values on the data port, which the behavioral block reads.

To get the output of the behavioral block, the testbench waits until the behavioral block asserts the gcd_ready signal (high). To model the testbench latency for a write, the code contains a wait (WRITE_LATENCY - 1) statement. At the end of this wait, the testbench reads the output of the behavioral block from the gcd port.

One-Way Handshake Initiated From Testbench

Figure 6-2 shows a timing diagram for the GCD behavioral block with a slightly different handshaking mechanism than Figure 6-1. In this example, the testbench initiates the handshake to the GCD block before sending new input. After computing the output, the GCD block initiates sending the output to the testbench. Example 6-2 shows the code for the GCD block, the testbench, and the top-level simulation executable.

Figure 6-2 Testbench-Initiated One-Way Handshake



Example 6-2 Behavioral Block Responding to One-Way Handshake

```
// gcd2.h header file

#define READ_LATENCY 2
#define WRITE_LATENCY 3


SC_MODULE(gcd_mod) {
  // Ports
  sc_in_clk  clk;          // Clock input
  sc_in<bool> reset;       // Reset input
  sc_in<int> data;         // Port to get data
  sc_in<bool> data_ready;  // Handshake signal to
                           // indicate input ready
  sc_out<bool> gcd_ready;  // Handshake signal to
                           // indicate output is ready
  sc_out<int> gcd;         // Port to send GCD value

  // Process
  void gcd_algo();            // Process to do GCD

  // Internal functions
  // Function to compute gcd algorithm
  int do_gcd(int a, int b);

  SC_CTOR(gcd_mod) {
     SC_CTHREAD(gcd_algo, clk.pos());
     watching(reset.delayed() == true);
  }
};


/**********************************/
// gcd2.cc implementation file

#include “systemc.h”
#include “gcd2.h”

void gcd_mod::gcd_algo()
{
    int a, b;  // Two variables to compute GCD
    int c;     // The GCD

    // Reset operations
    gcd.write(0);
    gcd_ready.write(false);
```

```
    wait();

    while (true) {
      // First get the two inputs
      // using sender initiated handshake.
      wait_until(data_ready.delayed() == true);
      // Wait READ_LATENCY cycles for the first data
      wait(READ_LATENCY);
      a = data.read();
      wait();
      b = data.read();

      // Now do the algorithm
      c = do_gcd(a, b);

      // Now write the output
      // using sender initiated handshake.
      gcd_ready.write(true);
      gcd.write(c);
      wait(WRITE_LATENCY);
      gcd_ready.write(false);
      wait();
    }
}

int gcd_mod::do_gcd(int a, int b)
{
    int temp;

    if (a != 0 && b != 0) {
      while (b != 0) {
        while (a >= b) {
          a = a - b;
          wait()
        }
        temp = a;
        a = b;
        b = temp;
        wait();
      }
    }
    else {
      a = 0;
      wait();
    }
    return a;
}
```

```
/*****************************/
// gcd2_test.h header file

#ifndef READ_LATENCY
#define READ_LATENCY 2
#endif

#ifndef WRITE_LATENCY
#define WRITE_LATENCY 3
#endif

SC_MODULE(testbench) {
    sc_in_clk clk;
    sc_out<bool> data_ready;
    sc_in<bool> gcd_ready;
    sc_in<int> gcd;
    sc_out<bool> reset;
    sc_out<int> data;

    // Process
    void do_run();

    // Internal function
    void do_handshake(int a, int b);

    SC_CTOR(testbench) {
        SC_CTHREAD(do_run, clk.pos());
    }
};


/**********************************/
// gcd2_test.cc testbench implementation file

#include “systemc.h”
#include “gcd2_test.h”

void testbench::do_run()
{
    reset.write(false);
    data_ready.write(false);
    wait();
    reset.write(true);
    wait();
    reset.write(false);
    wait();

    cout << “*** Reset Done - Begin Testing ***\n”;
```

```
    do_handshake(12, 6);
    do_handshake(172, 36);
    do_handshake(36, 172);
    do_handshake(19, 5);
    do_handshake(2584, 4712);
    do_handshake(15, 0);
    cout << " *** Testing Done ***\n";
    sc_stop();
}

void testbench::do_handshake(int a, int b)
{
    cout << "GCD of " << a << " and " << b << " is = ";

    // Sender initiated handshake - send data ready signal
    data_ready.write(true);
    wait(READ_LATENCY);  // Wait for latency
    // Now write data in 2 consecutive cycles
    data.write(a);
    data_ready.write(false);
    wait();
    data.write(b);
    wait();

    // Sender initiated handshake - wait until sender is ready
    wait_until(gcd_ready.delayed() == true);
    wait(WRITE_LATENCY - 1); // Wait for latency
    // Now read data
    cout << gcd.read() << endl;
    wait();
}



/******************************************/
// gcd2_main.cc top-level simulation model.

#include "systemc.h"
#include "gcd2.h"
#include "gcd2_test.h"

int
main()
{
    sc_signal<int> data, gcd;
    sc_signal<bool> reset, data_ready, gcd_ready;
    sc_clock clock("Clock", 20, 0.5);
```

```
    gcd_mod G("GCD");
    G(clock, reset, data, data_ready, gcd_ready, gcd);

    testbench TB("TB");
    TB(clock, data_ready, gcd_ready, gcd, reset, data);

    sc_trace_file *tf = sc_create_vcd_trace_file("gcd");
    sc_trace(tf, clock.signal(), "Clock");
    sc_trace(tf, reset, "Reset");
    sc_trace(tf, data_ready, "Data Ready");
    sc_trace(tf, data, "In");
    sc_trace(tf, gcd_ready, "Out Ready");
    sc_trace(tf, gcd_ready, "Out");

    sc_start(-1);
    return 0;
}
```

The following steps describe how the input handshaking protocol works:

1. The testbench asserts the data_ready signal (high), to indicate that it can send new data in READ_LATENCY cycles. In this example, READ_LATENCY is 2.
2. The behavioral block waits until it sees the data_ready signal. Then it waits a further READ_LATENCY cycles before it can read the data.
3. At the end of this wait, the behavioral block reads two consecutive values from the data port.
4. This process repeats each time the testbench can send the next data set.

The steps for implementing the output handshake protocol are identical to the steps in Example 6-1 on page 6-6.

Looking at the testbench code in Example 6-2, you see that the handshaking for the testbench is done in the do_handshake function. To send data to the behavioral block, the testbench asserts data_ready (high) and then waits READ_LATENCY cycles. This models the time it takes the testbench to get the data ready. At the end of this wait, the testbench writes two consecutive values on the data port, which the behavioral block reads.

The handshaking for the output of the behavioral block is identical to that in Example 6-1 on page 6-6.

Constraining the Width of Handshake Strobes

Because SystemC Compiler can insert clock cycles in the superstate-fixed scheduling mode, you need to constrain the number of clock cycles used to raise or lower handshake signals, and you also need to constrain the number of cycles between raising a handshake signal and reading or writing data. For example, in Example 6-1 on page 6-6, you need to constrain the number of cycles between reading *a* and reading *b* to one cycle. And you also need to constrain the number of cycles between assertion of send_data and reading the value of *a*. Figure 6-3 shows the section of code with the input handshaking where you need to constrain the cycles. Line labels are added to make it more convenient to use the `set_cycles` command.

Figure 6-3 Constraining Input Handshake Signals

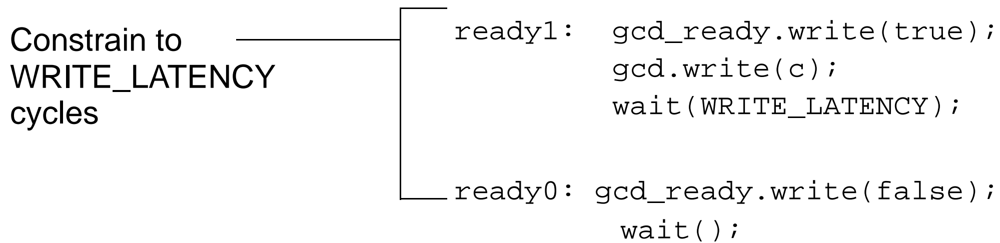

```
send_d:    send_data.write(true);
            wait();
           wait(READ_LATENCY);
           send_data.write(false);
read_d1:   a = data.read();
           wait();
read_d2:   b = data.read();
```

Use the `set_cycles` and `find` commands, described in the *CoCentric SystemC Compiler User Guide*, to set these constraints before scheduling. For example,

```
dc_shell> data = find cell *send_data_send_d* -hier
dc_shell> a = find cell *a_read_d1* -hier
dc_shell> b = find cell *b_read_d2* -hier
dc_shell> set_cycles 2 -from data -to a
dc_shell> set_cycles 1 -from a -to b
```

Similarly, you need to constrain the number of cycles for the output handshake. Figure 6-4 shows the output handshake section of code from Example 6-1 with line labels added.

Figure 6-4 Constraining Output Handshake Signals



You can use the `set_cycles` command to constrain the output handshake cycles. For example,

```
dc_shell> r1 = find cell *gcd_ready_ready1* -hier
dc_shell> r0 = find cell *gcd_ready_ready0* -hier
dc_shell> set_cycles 3 -from r1 -to r0
```

Using Two-Way Handshake Protocols

Use two-way handshake protocols for modules that do not have a fixed response time and are scheduled with the superstate-fixed I/O scheduling mode. A two-way handshake protocol means the block requesting to send or receive data waits for an acknowledgement signal from the other block. Therefore, two-way handshake uses an acknowledgement signal rather than using a fixed latency like a one-way handshake.

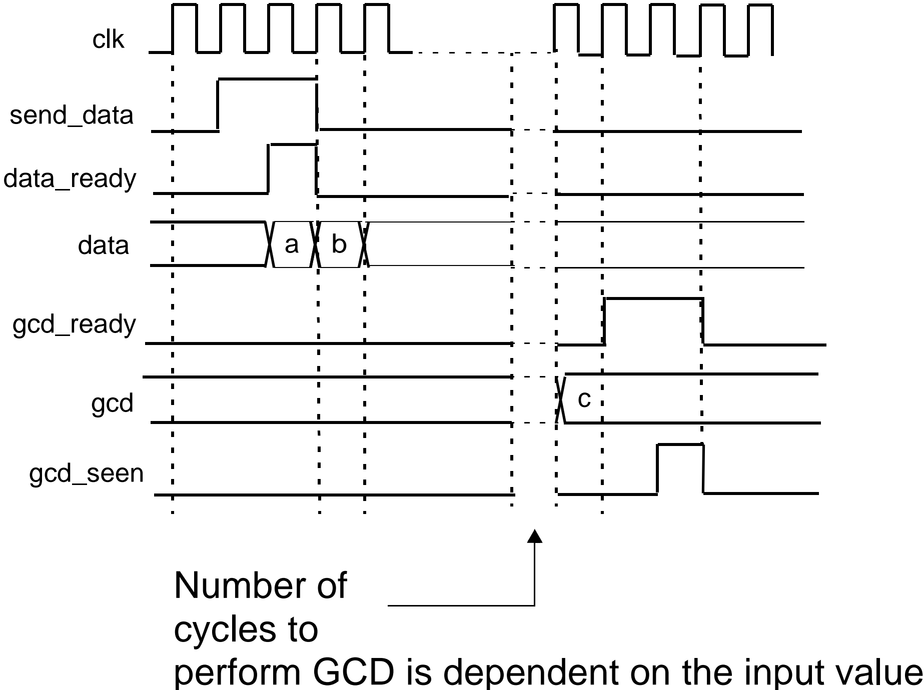
Two-Way Handshake Initiated From Behavioral Block

Figure 6-5 shows a timing diagram for a GCD behavioral block that uses two-way handshake protocols to get data and to write data out. The GCD block initiates the handshake with the testbench. Example 6-3 shows the code for the GCD block, the testbench, and the main routine for simulation.

Note:

The number of cycles needed to compute the GCD is not fixed because it depends on the value of the two numbers for which the GCD is computed. Therefore, this example requires the use of handshaking.

Figure 6-5   Two-Way Handshake Protocol



Example 6-3 Two-Way Handshake Protocol From GCD Block

```
// gcd3.h Two-way handshake header file.

SC_MODULE(gcd_mod) {
  // Ports
  sc_in_clk  clk;            // Clock input
  sc_in<bool> reset;         // Reset input
  sc_in<int> data_in;        // Port for getting data

  // Handshake signals
  sc_out<bool> send_input;   // Request input
  sc_in<bool> data_ready;    // Data is ready
  sc_out<bool> output_ready; // Output is ready
  sc_in<bool> output_seen;   // Output is seen

  sc_out<int> gcd_out;      // Port to send GCD value

  // Process
  void gcd_algo();           // The process that does GCD

  // Internal functions
  int do_gcd(int a, int b);  // Function of gcd algorithm

  SC_CTOR(gcd_mod) {
    SC_CTHREAD(gcd_algo, clk.pos());
    watching(reset.delayed() == true);
  }
};

/**********************************/
// gcd3.cc two-way handshake implementation file.

#include “systemc.h”
#include “gcd3.h”

void gcd_mod::gcd_algo()
{
    int a, b;  // Two variables to compute gcd
    int c;     // The GCD

    // Reset operations
    gcd_out = 0;
    send_input = false;
    output_ready = false;
    wait();

    while (true) {
```

```
    // First get the two inputs
    // using receiver initiated handshake.
    send_input.write(true);
    wait();
    wait_until(data_ready.delayed() == true);
    // Read data and deassert send_input
    send_input.write(false);
    a = data_in.read();
    wait();
    b = data_in.read();

    // Now do the algorithm
    c = do_gcd(a, b);

    // Now write the output
    // using sender initiated handshake.
    output_ready = true;
    gcd_out = c;
    wait();
    wait_until(output_seen.delayed() == true);
    output_ready = false;
    wait();
    }
}

int gcd_mod::do_gcd(int a, int b) {
    int temp;

    if (a != 0 && b != 0) {
      while (b != 0) {
        while (a >= b) {
          a = a - b;
          wait();
        }
      temp = a;
      a = b;
      b = temp;
      wait();
    }
   }
   else {
      a = 0;
      wait();
   }
   return a;
}
```

```
/***********************************/
// gcd3_test.h header file

SC_MODULE(testbench) {
    sc_in_clk clk;
    sc_in<bool> send_data;
    sc_out<bool> data_ready;
    sc_in<bool> gcd_ready;
    sc_out<bool> gcd_seen;
    sc_in<int> gcd;
    sc_out<bool> reset;
    sc_out<int> data;

    // Process
    void do_run();

    // Internal function
    void do_handshake(int a, int b);

    SC_CTOR(testbench) {
         SC_CTHREAD(do_run, clk.pos());
    }
};

/***********************************/
// gcd3_test.cc testbench implementation file.

#include "systemc.h"
#include "gcd3_test.h"

void testbench::do_run()
{
    reset = false;
    gcd_seen = false;
    data_ready = false;
    wait();
    reset = true;
    wait();
    reset = false;
    wait();

    cout << "*** Reset Done - Begin Testing ***\n";

    do_handshake(12, 6);
    do_handshake(172, 36);
    do_handshake(36, 172);
    do_handshake(19, 5);
    do_handshake(2584, 4712);
```

```
    do_handshake(15, 0);
    cout << " *** Testing Done ***\n";
    sc_stop();
}

void testbench::do_handshake(int a, int b)
{
    cout << "GCD of " << a << " and " << b << " is = ";

    // Receiver initiated handshake
    // Wait until receiver is ready
    wait_until(send_data.delayed() == true);

    // Indicate data is ready and
    // write data in 2 consecutive cycles
    data_ready = true;
    data = a;
    wait();
    data_ready = false; // Deassert data_ready
    data = b;
    wait();

    // Sender initiated handshake
    // Wait until sender is ready
    wait_until(gcd_ready.delayed() == true);

    // Now read data
    cout << gcd.read() << endl;
    gcd_seen = true;
    wait();
    gcd_seen = false;
    wait();
}

/******************************************/
// gcd3_main.cc simulation executable.

#include "systemc.h"
#include "gcd3.h"
#include "gcd3_test.h"

int
main()
{
    sc_signal<int> data, gcd;
    sc_signal<bool> reset, send_data, data_ready,
                    gcd_ready, gcd_seen;
    sc_clock clock("Clock", 20, 0.5);
```

```
    gcd_mod G("GCD");
    G(clock, reset, data, send_data, data_ready,
      gcd_ready, gcd_seen, gcd);

    testbench TB("TB");
    TB(clock, send_data, data_ready, gcd_ready,
       gcd_seen, gcd, reset, data);

    sc_trace_file *tf = sc_create_vcd_trace_file("gcd");
    sc_trace(tf, clock.signal(), "Clock");
    sc_trace(tf, reset, "Reset");
    sc_trace(tf, send_data, "Send Data");
    sc_trace(tf, data_ready, "Data Ready");
    sc_trace(tf, data, "In");
    sc_trace(tf, gcd_ready, "Out Ready");
    sc_trace(tf, gcd_seen, "Out Seen");
    sc_trace(tf, gcd_ready, "Out");

    sc_start(-1);
    return 0;
}
```

The following steps describe how the input handshaking protocol works:

1. The behavioral block asserts the handshake signal send_data (high), to indicate that it can process new data. It waits until it sees a data_ready signal from the testbench. When it sees the data_ready signal asserted, it reads the first piece of data.
2. In the next cycle, the module de-asserts (low) the send_data signal and reads the second piece of data.
3. The behavioral block proceeds to compute the GCD of the two numbers it has read. This computation can take an indeterminate, but finite, number of cycles.
4. This process repeats each time the behavioral block is ready to accept new data.

The following steps are used to implement the output handshake protocol:

1. The behavioral block asserts the gcd_ready signal, to indicate that it can send new output data, and writes the output to the gcd port.
2. The testbench waits until it sees the gcd_ready signal, reads the output on the gcd port, and asserts (high) the gcd_seen signal.
3. The behavioral block waits until it sees the gcd_seen signal and de-asserts (low) the gcd_ready signal. The testbench also de-asserts the gcd_seen signal.
4. This process repeats each time the behavioral block can send new output data.

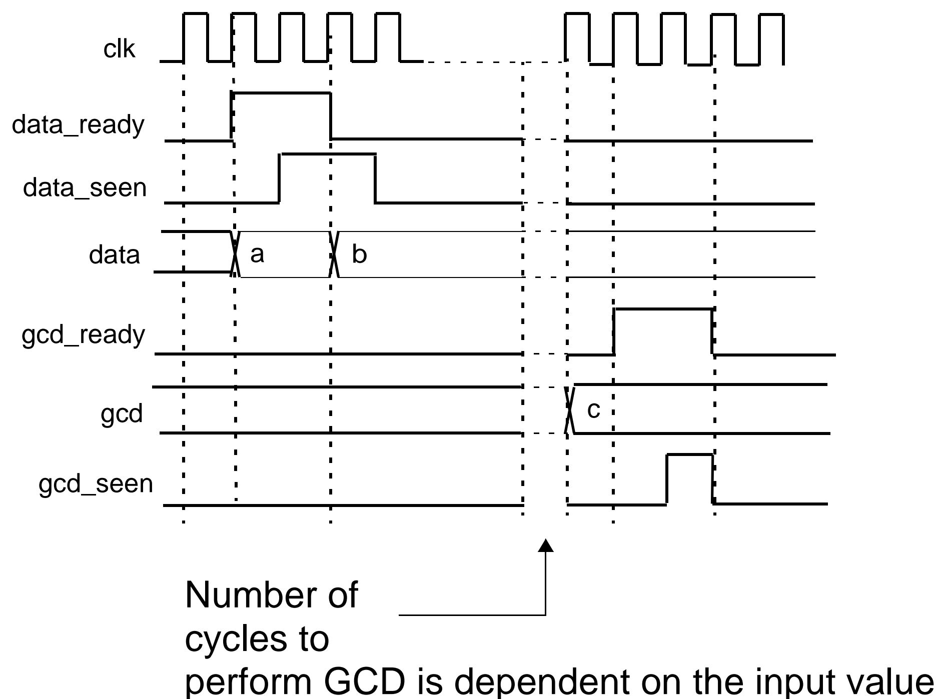
Two-Way Handshake Initiated From Testbench

Figure 6-6 shows a timing diagram for a GCD behavioral block that uses two-way handshake protocols to get data and to write data out. The testbench initiates the input handshake with the GCD block, and the GCD block initiates the output handshake to send the GCD output to the testbench. Example 6-4 shows the code for the GCD block, the testbench, and the main routine for simulation.

Note:

> The number of cycles needed to compute the GCD is not fixed, because it depends on the value of the two numbers for which the GCD is computed. Therefore, this example requires the use of handshaking.

Figure 6-6 Two-Way Handshake Protocol



Example 6-4 Two-Way Handshake Protocol From Testbench

```
// gcd4.h Two-way handshake from testbench header file.


SC_MODULE(gcd_mod) {
  // Ports
  sc_in_clk  clk;            // Clock input
  sc_in<bool> reset;         // Reset input
  sc_in<int> data_in;        // Port to get data

  // Handshake signals
  sc_out<bool> data_seen;    // Data read
  sc_in<bool> data_ready;    // Data ready
  sc_out<bool> output_ready; // Output is ready
  sc_in<bool> output_seen;   // Output is seen

  sc_out<int> gcd_out;       // Port to send GCD value

  // Process
  void gcd_algo();           // The GCD process

  // Internal functions
  int do_gcd(int a, int b);  // GCD algorithm function

  SC_CTOR(gcd_mod) {
     SC_CTHREAD(gcd_algo, clk.pos());
     watching(reset.delayed() == true);
  }
};


/**********************************/
// gcd4.cc behavioral module implementation file.

#include “systemc.h”
#include “gcd4.h”

void gcd_mod::gcd_algo()
{
  int a, b;  // Two variables to compute gcd
  int c;     // The GCD

  // Reset operations
  gcd_out = 0;
  data_seen = false;
  output_ready = false;
  wait();
```

```
  while (true) {
    // First get the two inputs
    // using sender initiated handshake.
    wait_until(data_ready.delayed() == true);
    // Read data and assert data_seen
    data_seen = true;
    a = data_in.read();
    wait(2);  // Two cycles for new data to arrive
    b = data_in.read();
    data_seen = false;
    wait();

    // Now do the algorithm
    c = do_gcd(a, b);

    // Now write the output
    // using sender initiated handshake
    output_ready = true;
    gcd_out = c;
    wait();
    wait_until(output_seen.delayed() == true);
    output_ready = false;
    wait();
  }
}

int gcd_mod::do_gcd(int a, int b)
{
    int temp;

    if (a != 0 && b != 0) {
       while (b != 0) {
         while (a >= b) {
           a = a - b;
           wait();
         }
         temp = a;
         a = b;
         b = temp;
         wait();
       }
    }
    else {
       a = 0;
       wait();
    }
    return a;
}
```

```
/***********************************/
// gcd4_test.h testbench header file.

SC_MODULE(testbench) {
    sc_in_clk clk;
    sc_in<bool> data_seen;
    sc_out<bool> data_ready;
    sc_in<bool> gcd_ready;
    sc_out<bool> gcd_seen;
    sc_in<int> gcd;
    sc_out<bool> reset;
    sc_out<int> data;

    // Process
    void do_run();

    // Internal function
    void do_handshake(int a, int b);

    SC_CTOR(testbench) {
         SC_CTHREAD(do_run, clk.pos());
    }
};

/***********************************/
// gcd4_test.cc testbench implementation file.

#include "systemc.h"
#include "gcd4_test.h"

void testbench::do_run()
{
    reset = false;
    gcd_seen = false;
    data_ready = false;
    wait();
    reset = true;
    wait();
    reset = false;
    wait();

    cout << "*** Reset Done - Begin Testing ***\n";

    do_handshake(12, 6);
    do_handshake(172, 36);
    do_handshake(36, 172);
```

```
    do_handshake(19, 5);
    do_handshake(2584, 4712);
    do_handshake(15, 0);
    cout << " *** Testing Done ***\n";
    sc_stop();
}

void testbench::do_handshake(int a, int b)
{
    cout << "GCD of " << a << " and " << b << " is = ";

    // Sender initiated handshake.
    // Send data ready signal and first data.
    data_ready = true;
    data = a;
    wait();
    wait_until(data_seen.delayed() == true);
    // Now send the second data
    // and deassert data_ready.
    data_ready = false; // Deassert data_ready
    data = b;
    wait();

    // Sender initiated handshake.
    // Wait until sender is ready.
    wait_until(gcd_ready.delayed() == true);
    // Now read data
    cout << gcd.read() << endl;
    gcd_seen = true;
    wait();
    gcd_seen = false;
    wait();
}


/*****************************************/
// gcd4_main simulation executable.

#include "systemc.h"
#include "gcd4.h"
#include "gcd4_test.h"

int
main()
{
    sc_signal<int> data, gcd;
    sc_signal<bool> reset, data_seen, data_ready,
                    gcd_ready, gcd_seen;
```

```
    sc_clock clock("Clock", 20, 0.5);

    gcd_mod G("GCD");
    G(clock, reset, data, data_seen, data_ready,
      gcd_ready, gcd_seen, gcd);

    testbench TB("TB");
    TB(clock, data_seen, data_ready, gcd_ready,
       gcd_seen, gcd, reset, data);

    sc_trace_file *tf = sc_create_vcd_trace_file("gcd");
    sc_trace(tf, clock.signal(), "Clock");
    sc_trace(tf, reset, "Reset");
    sc_trace(tf, data_seen, "Data Seen");
    sc_trace(tf, data_ready, "Data Ready");
    sc_trace(tf, data, "In");
    sc_trace(tf, gcd_ready, "Out Ready");
    sc_trace(tf, gcd_seen, "Out Seen");
    sc_trace(tf, gcd_ready, "Out");

    sc_start(-1);
    return 0;
}
```

The following steps describe how the input handshaking protocol works:

1. The testbench asserts the handshake signal data_ready (high), to indicate that it has new data to process.
2. The behavioral module waits until it sees the data_ready signal. Then it asserts (high) the data_seen signal and reads the first piece of data in the same cycle.
3. The testbench waits until it sees the data_seen signal asserted (high). Then it sends the second piece of data and de-asserts data_ready (low).

4. Because the testbench needs to see the data_seen signal before it sends the next piece of data, which takes two cycles, the behavioral module waits two cycles before reading the second piece of data, and it de-asserts (low) the data_seen signal to indicate that it has finished reading the data set.
5. The behavioral block proceeds to compute the GCD of the two numbers it has read. This computation can take an indeterminate, but finite, number of cycles.
6. This process repeats each time the testbench has the next data to send to the behavioral module.

The output handshake protocol is the same as Example 6-3 on page 6-23, which initiates the output handshake from the behavioral module.

Fast Handshaking

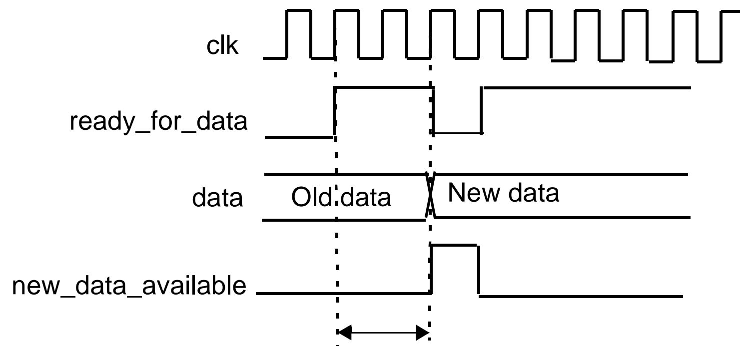
Example 6-5 shows a fragment of code for a behavioral block that needs to wait for a ready_for_data handshake signal before it can assert a new_data_available signal and write the new data in the same clock cycle. General coding rule 5 (“General Coding Rule 5” on page 3-20) requires a wait statement (shown in bold) immediately after a while loop.

Example 6-5 Two-Way Handshake Using a while Loop

```
new_data_available.write(0);
while(ready_for_data.read() == 0) {
     wait();
}
wait(); // Wait required after loop
new_data_available.write(1);
data.write(...);
wait();
...
```

The requirement of general coding rule 5 produces the timing diagram shown in Figure 6-7. Two cycles are required between the assertion of the ready_for_data signal and the new_data_available signal.

Figure 6-7 Timing Diagram of while Loop



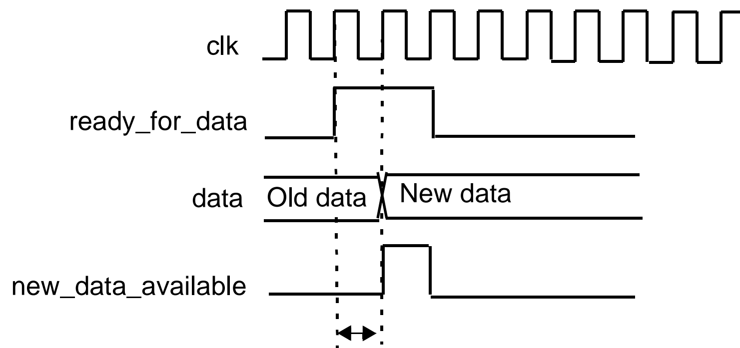
Using if…else

You can use an infinite while loop that contains an if...else conditional branch to accomplish the handshake in one cycle. This method moves the wait statement required for loop exit into the loop. Example 6-6 shows a fragment of code for this alternative code with the wait statement in bold and Figure 6-8 shows the timing diagram.

Example 6-6 Fast Two-Way Handshake Using while Loop

```
new_data_available.write(0);
while (true) {
   if (ready_for_data.read() == 0) {
        wait();
   }
   else {
        new_data_available.write(1);
        data.write(...);
        wait();
        break;
   }
}
wait();
```

Figure 6-8 Timing Diagram Using if…else



SystemC provides an convenient, alternative syntax for fast handshaking, the wait_until statement, described in the next section.

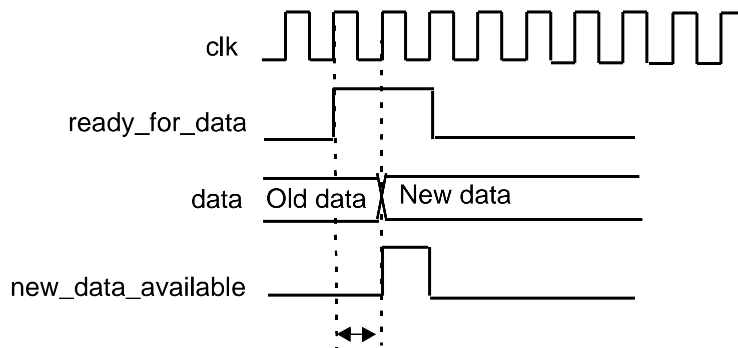
Using wait_until

You can use a wait_until statement rather than a while loop to accomplish a fast handshake, as shown in bold in the code fragment in Example 6-7. This method eliminates the extra wait statement required to exit the while loop hierarchy, and it produces the timing diagram shown in Figure 6-9.

Example 6-7 Fast Two-Way Handshake Using wait_until

```
new_data_available.write(0);
wait_until (rdy_for_data.delayed() == 0);
new_data_available.write(1);
data.write(...);
wait();
```

Figure 6-9 Timing Diagram Using wait_until



Using a Pipeline Handshake Protocol

Loop pipelining has two restrictions that affect how handshake protocols are implemented:

1. The pipelined loop can contain only unrolled loops. Therefore, pipelined loops use only one-way handshake protocols, because two-way handshake requires a rolled loop.
2. During execution, iterations of a pipelined loop overlap. You cannot have a signal write operation in one loop iteration within the same clock cycle as a write operation to the same signal in any overlapping iteration.

Example 6-8 shows, in bold, a pipelined loop with a handshake signal assertion in the first clock cycle of the loop. This is followed by a de-assertion of the same signal in the next clock cycle of the loop.

Example 6-8 Incorrect Loop Pipeline With Handshake

```
...
while (true) {
    // Loop to pipeline with handshake
  for (int i = 0; i < 3; i++){
      send_data.write(true);
      wait();
      send_data.write(false);
      wait();
      ...
   }
}
```

If you pipeline the loop with an initiation interval of one clock cycle, the three loop iterations overlap, as shown in Figure 6-10.

Figure 6-10 Incorrect Loop Pipeline With Handshake

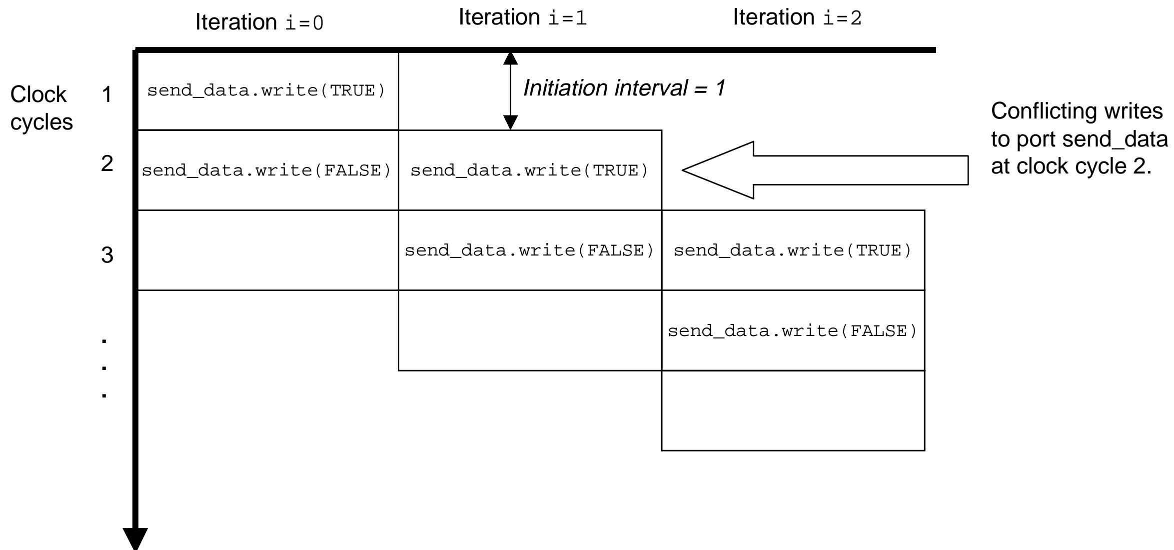
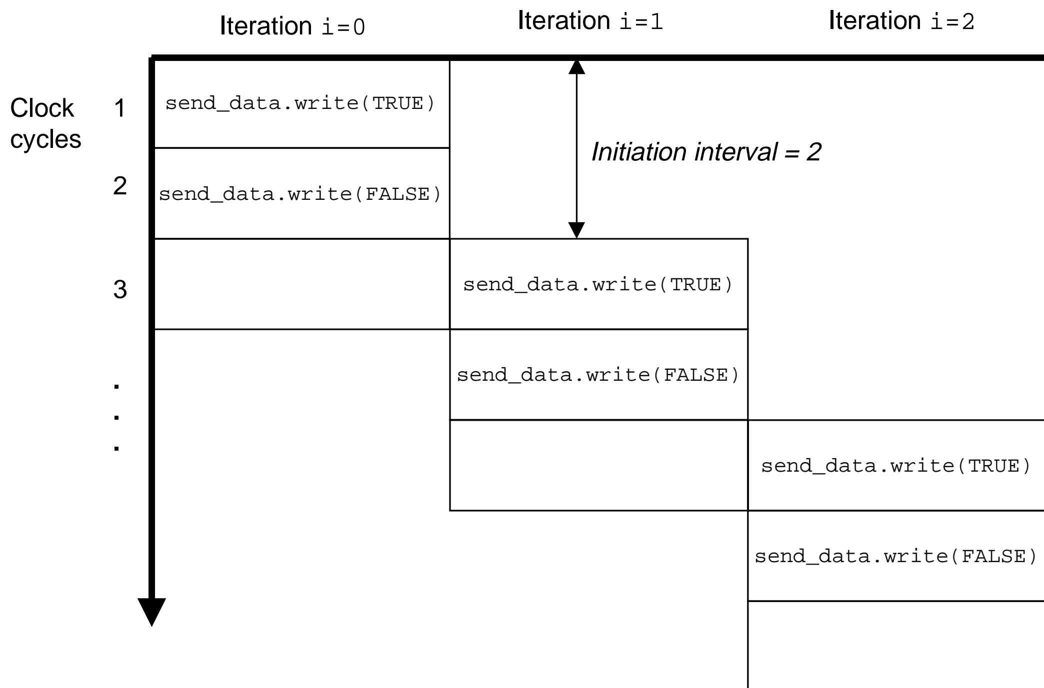


Figure 6-10 shows an incorrect handshake protocol for a pipelined loop that causes resource contention on the send_data port when the initiation interval is one clock cycle. In clock cycle 2, the send_data port writes a FALSE in iteration 1 and a TRUE in iteration 2. In clock cycle 3, the situation is similar. SystemC Compiler reports a resource contention error on the send_data port.

You can resolve this resource contention by extending the initiation interval to two clock cycles, as shown in Figure 6-11.

Figure 6-11   Correct Loop Pipeline With Extended Initiation Interval



Changing the initiation interval to 2, as shown in Figure 6-11, however, is not acceptable if the initiation interval must be one clock cycle.

To resolve a pipelined loop handshake resource contention with an initiation interval of 1, assert the handshake signal in the first iteration of the pipelined loop, and de-assert it after the pipelined loop exits from the outer loop. Example 6-9 shows (in bold) this method of correctly handshaking in a pipelined loop.

Example 6-9 Correct Handshake in a Pipelined Loop

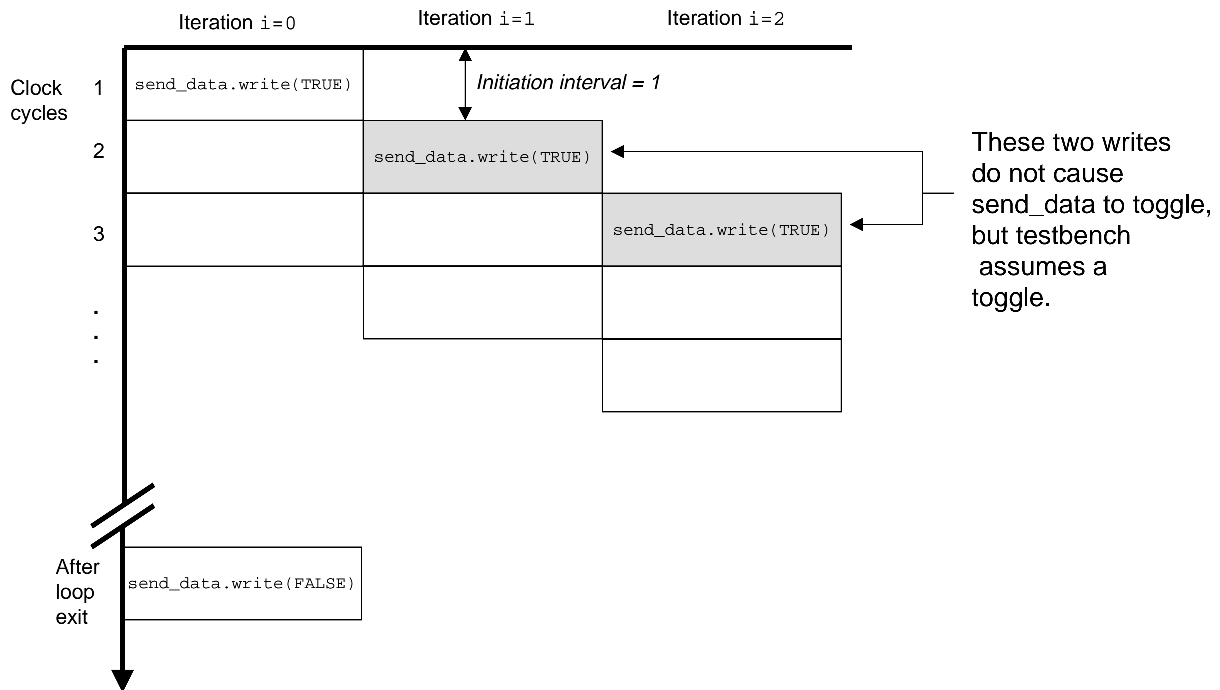
```
...
while (true){
   // Loop to pipeline with handshake
   for (int i = 0; i < 3; i++){
      ...
      send_data.write(true); //Assert handshake
      ...
   }
   wait();
   send_data.write(false); //De-assert
   ...
}
```

For Example 6-9, create a testbench that detects assertion of send_data, and assume that the next assertion of the signal happens after the initiation interval. This assumption is valid because of the periodicity of loop pipelining.

When the testbench detects de-assertion of send_data, it recognizes that the pipelined loop has exited. There are no more loop iterations for the testbench to process, and therefore no more assumed assertions of send_data.

Figure 6-12 illustrates loop pipelining of Example 6-9 with an initiation interval of 1 clock cycle. Resource contention is prevented by de-asserting send_data after the loop, rather than inside the loop.

Figure 6-12 Correct Loop Pipeline Without Handshake Signal De-assertion



For coding rules about pipelining loops, see “Pipelining Loop Rules” on page 3-45.

A

Compiler Directives

This appendix provides a list of the compiler directives you can use with SystemC Compiler, and it tells you where you can find further details about using them.

Synthesis Compiler Directives

To specify a compiler directive in your SystemC code, insert a comment in which the first word is either `synopsys` or `snps`. You can use either a multiple-line comment enclosed in /* and */ characters or a single-line comment beginning with two slash (//) characters.

Table A-1 lists the compiler directives in alphabetical order.

Table A-1 SystemC Compiler Compiler Directives

Compiler Directive	Details
/* snps inout_param */	page A-5
/* snps line_label *string* */	page A-3
/* snps map_to_operator *dw_part* */	page A-3
/* snps preserve_function */	page A-4
/* snps resource *name*: variables = *var*, map_to_module = *memory_module_name*; [memory_address_ports = *port_name*] */	page A-6
/* snps resource *name*: variables = *var*, map_to_registerfiles = “TRUE”; */	page A-6
/* snps return_port_name *port* */	page A-4
/* snps synthesis_off */ and /* snps synthesis_on */	page A-7
/* snps translate_off */ and /* snps translate_on */	Use synthesis_off and synthesis_on instead of translate_off and translate_on.
/* snps unroll */	page A-8

line_label

Use the `line_label` compiler directive to label a loop or a line of code. In SystemC Compiler generated reports, the label is reflected in the report hierarchy.

```
my_module2 :: entry {
         // Synopsys compiler directive
     while (true) { //snps line_label reset_loop2
       ...
       wait();
       ...
       wait();
     }
}
```

See “Labeling a Loop” on page 3-34.

map_to_operator

Use the `map_to_operator` compiler directive to use a standard DesignWare synthetic operator to implement a function. Place the compiler directive in the first line of the function body.

```
sc_int<16> my_mult (const sc_int<8> A,
                    const sc_int<8> B) {

      // snps map_to_operator MULT2_TC_OP
      // snps return_port_name P
      // Function code block
      ...
       return (A*B);
}
```

See “Using DesignWare Components” on page 4-11.

return_port_name

When you use the `map_to_operator` compiler directive, the name of the return port, if any, is Z by default. You can override the name by using the `return_port_name` compiler directive.

```
int xyz(int a, const int& b) {
   /* snps map_to_operator XYZ_OP */
   /* snps return_port_name P */
   ...
}
```

See "Using DesignWare Components" on page 4-11.

preserve_function

Use the `preserve_function` compiler directive to preserve a function as a separate level of hierarchy. Place the compiler directive in the first line of the function body.

```
// Define my_func
int my_func(int y, int& x) {
  /* synopsys preserve_function */
  x = x + y;
  return x;
}

void my_module::entry() {
  int a, b, c;
  ...
  c = my_func(a , b);
  ...
}
```

During synthesis, the level of hierarchy is compiled into a component that is treated exactly the same way as any other combinational component, such as an adder or a multiplier. Only functions that describe purely combinational RTL designs can be preserved. See “Using Preserved Functions” on page 4-4.

inout_param

Use the `inout_param` compiler directive with the `preserve_function` compiler directive.

SystemC Compiler maps nonconstant C++ reference parameters to the output ports of the design corresponding to a preserved function. If the preserved function contains a read from a reference parameter, SystemC Compiler assumes that you are trying to read an output port and issues an error message unless you use the `inout_param` compiler directive. Notice that the `inout_param` is placed immediately after the reference parameter and is inside the parentheses. The `preserve_function` directive is placed in the first line of the function body.

```
void my_func (int y, int& x /*snps inout_param */) {
  /* snps preserve_function */
  x = x + y;
}
```

When you use the `inout_param` compiler directive, SystemC Compiler creates an input port x and an output port x’ for the x reference parameter so it can perform the read and the write.

resource

Use the `resource` compiler directive and the `map_to_module` attribute in your code to specify the array that is to be mapped to a memory. See "Mapping Arrays to Memories" on page 5-11.

```
while (true){

sc_int<32> amem[16];

     /* synopsys
          resource RAM_A:
               variables = "amem",
               map_to_module = "my_mem_model"
     */
     // array amem mapped to a single RAM
          amem[i] = ser_in;
          a = amem[j];
}
```

Use the `synopsys resource` compiler directive and the `map_to_registerfiles` attribute in your code to specify the array that is to be mapped to a register file. See "Mapping Arrays to Register Files" on page 5-8.

```
sc_int<32> mem [16];
/*synopsys resource R1
 variables="mem",
   map_to_registerfiles="TRUE";*/
//The following are all mapped to memory.
//Write to mem
mem[0] = a;
mem[1] = b;
// and so forth

//Read from mem
a = mem[0];
b = mem[1];
// and so forth
```

synthesis_off and synthesis_on

Use the `synthesis_off` and `synthesis_on` compiler directives to isolate simulation-specific code and prevent the code from being interpreted for synthesis.

```
/* synopsys synthesis_off */
... //Simulation-only code
/* snps synthesis_on */
```

translate_off and translate_on

Use `synthesis_off` and `synthesis_on` compiler directives instead.

unroll

Use the synopsys unroll compiler directive to unroll a for loop. Place the synopsys unroll compiler directive as a comment in the first line in the body of the for loop. See “Unrolling for Loops” on page 3-37.

```
...
for (int i=0; i < 8; i++) {
  // synopsys unroll
  .. // loop operations
}
...
```

C/C++ Compiler Directives

You can use C/C++ compiler directives instead of or in addition to the equivalent `synopsys` compiler directives.

C Line Label

Use the C line label instead of the line_label compiler directive. See “Labeling a Loop” on page 3-34.

```
my_module1 :: entry {
     // C style line label
     reset_loop1: while (true) {
       ...
       wait();
       ...
       wait();
     }
}
```

C Conditional Compilation

Use the C/C++ language #if, #ifdef, #ifndef, #elif, #else, and #endif conditional compilation directives to isolate blocks of code and prevent them from being included during synthesis.

```
    //C directive
    #ifdef SIM
    ...//Simulation-only code
    #endif
```

B

First-In-First-Out Example

This appendix provides a simple first-in-first-out (FIFO) circular buffer example that shows you a behavioral model with a testbench and the equivalent RTL model that uses the same testbench.

This chapter contains the following sections:

- FIFO Description
- Architectural Model
- Behavioral Model
- RTL Model

FIFO Description

The FIFO is a circular buffer that accepts a 32-bit integer value from the input and writes an integer to the output. A reset port clears all data in the buffer.

Architectural Model

The architectural model describes the FIFO algorithm. The size of the FIFO is specified with the BUFSIZE macro. The number of bits required to address the FIFO is specified with the LOGBUFSIZE macro. Example B-1 shows the architectural simulation model, which works for a FIFO with a size that is a power of 2.

Example B-1 Architectural Simulation Model

```
/*
 fifo.cc executable specification.

 This model works for a FIFO
 with a size that is a power of 2.
 */


#include "systemc.h"

#define BUFSIZE 4
#define LOGBUFSIZE 2

struct circ_buf {
  int buffer[BUFSIZE];       // The FIFO buffer
  sc_uint<LOGBUFSIZE> headp; // Pointer to head of FIFO
  sc_uint<LOGBUFSIZE> tailp; // Pointer to tail of FIFO
  int num_in_buf;            // Number of buffer elements

  // Routine to initialize the FIFO
  void init() {
     num_in_buf = 0;
     headp = 0;
     tailp = 0;
  }

  // Constructor
  circ_buf() {
    init();
  }

  void status();        // Status of the FIFO
  int read();           // To read from the FIFO
  void write(int data); // To write to the FIFO
  bool is_full();       // To determine if FIFO is full
  bool is_empty();      // To determine if FIFO is empty
};

int
circ_buf::read() {
  if (num_in_buf) {
    num_in_buf--;
    return (buffer[headp++]);
  }
    // Otherwise ignore read request
}
```

```
void
circ_buf::write(int data) {
    if (num_in_buf < BUFSIZE) {
        buffer[tailp++] = data;
         num_in_buf++;
    }
    // Otherwise ignore write request
}

bool
circ_buf::is_full() {
    return (num_in_buf == BUFSIZE);
}

bool
circ_buf::is_empty() {
    return (num_in_buf == 0);
}

void
circ_buf::status() {
    cout << "FIFO is ";
    if(is_empty()) cout << "empty\n" ;
    else if (is_full()) cout << "full\n" ;
    else cout << "neither full nor empty\n";
}

int
main()
{
    circ_buf fifo;  // instantiate buffer

    // This is the testbench for the FIFO

    fifo.status();

    cout << "FIFO write 1\n"; fifo.write(1);
    cout << "FIFO write 2\n"; fifo.write(2);
    cout << "FIFO write 3\n"; fifo.write(3);
    fifo.status();
    cout << "FIFO write 4\n"; fifo.write(4);
    fifo.status();

    cout << "FIFO read " << fifo.read() << endl;
    fifo.status();
    cout << "FIFO read " << fifo.read() << endl;
    cout << "FIFO read " << fifo.read() << endl;
    cout << "FIFO read " << fifo.read() << endl;
```

```
    fifo.status();

    cout << "FIFO write 1\n"; fifo.write(1);
    cout << "FIFO write 2\n"; fifo.write(2);
    cout << "FIFO write 3\n"; fifo.write(3);
    fifo.status();
    cout << "FIFO read " << fifo.read() << endl;
    cout << "FIFO read " << fifo.read() << endl;
    fifo.status();

    cout << "FIFO write 4\n"; fifo.write(4);
    cout << "FIFO write 5\n"; fifo.write(5);
    fifo.status();
    cout << "FIFO write 6\n"; fifo.write(6);
    fifo.status();

    cout << "FIFO read " << fifo.read() << endl;
    fifo.status();
    cout << "FIFO read " << fifo.read() << endl;
    cout << "FIFO read " << fifo.read() << endl;
    fifo.status();
    cout << "FIFO read " << fifo.read() << endl;
    fifo.status();

    return 0;
}
```

Behavioral Model

The behavioral description of the FIFO has one SC_CTHREAD clocked thread process.

Ports and Signals

Several signals are added for the hardware description.

The FIFO has the following ports and signals:

- data_in

  An sc_in port of type int to write 32-bit data into the FIFO.

- data_out

  An sc_out port of type int to read 32-bit data from the FIFO.

- clk

  An sc_in_clk port for the SC_CTHREAD process

- reset

  An sc_in port of type bool to clear the data from the buffer, which is specified as a global reset.

- read_info

  An sc_in port of type bool that indicates a read from the FIFO.

- write_info

  An sc_in port of type bool that indicates a write to the FIFO.

- full

  An sc_out port of type bool that indicates that the FIFO is full.

- empty

  An sc_out port of type bool that indicates that the FIFO is empty.

Behavioral Description

Example B-2 shows the header file, Example B-3 shows the implementation file, and Example B-4 shows a command script to synthesize to gates for the behavioral model of the FIFO.

Example B-2 Behavioral Header File

```
/* fifo_bhv.h header file */

#define BUFSIZE 4
#define LOGBUFSIZE 2
#define LOGBUFSIZEPLUSONE 3

SC_MODULE(circ_buf) {
    sc_in_clk clk;          // The clock
    sc_in<bool> read_fifo;  // Indicate read from FIFO
    sc_in<bool> write_fifo; // Indicate write to FIFO
    sc_in<int> data_in;     // Data written to FIFO
    sc_in<bool> reset;      // Reset the FIFO

    sc_out<int> data_out;   // Data read from the FIFO
    sc_out<bool> full;      // Indicate FIFO is full
    sc_out<bool> empty;     // Indicate FIFO is empty

    int buffer[BUFSIZE];       // FIFO buffer
    sc_uint<LOGBUFSIZE> headp; // Pointer to FIFO head
    sc_uint<LOGBUFSIZE> tailp; // Pointer to FIFO tail
    // Counter for number of elements
    sc_uint<LOGBUFSIZEPLUSONE> num_in_buf;

    void read_write(); // FIFO process

    SC_CTOR(circ_buf) {
            SC_CTHREAD(read_write, clk.pos());
            watching(reset.delayed() == true);
    }
};
```

Example B-3 Behavioral Implementation File

```
  /* fifo_bhv.cc implementation file */

#include "systemc.h"
#include "fifo_bhv.h"

void
circ_buf::read_write() {
    // Reset operations
    headp = 0;
    tailp = 0;
    num_in_buf = 0;
    full = false;
    empty = true;
    data_out = 0;
    wait();

    // Main loop
    while (true) {
        if (read_fifo.read()) {

            // Check if FIFO is not empty
            if (num_in_buf != 0) {
                num_in_buf--;
                data_out = buffer[headp++];
                full = false;
                if (num_in_buf == 0) empty = true;
            }
            // Ignore read request otherwise
            wait();
        }
        else if (write_fifo.read()) {

            // Check if FIFO is not full
            if (num_in_buf != BUFSIZE) {
                buffer[tailp++] = data_in;
                num_in_buf++;
                empty = false;
                if (num_in_buf == BUFSIZE) full = true;
            }
            // Ignore write request otherwise
            wait();
        }
        else {
           wait();
        }
    }
}
```

Example B-4 Behavioral Synthesis to Gates Script

```
search_path       = search_path + "$SYNOPSYS/libraries/syn"
target_library    = {"tc6a_cbacore.db"};
synthetic_library = {"dw01.sldb","dw02.sldb"}
link_library    = {"*"} + target_library + synthetic_library

bc_enable_analysis_info    = "false"
effort_level               = medium
io_mode                    = super
top_unit                   = "fifo_bhv"

sh date
compile_systemc top_unit + ".cc"

create_clock clk -p 10

bc_time_design

schedule -io io_mode -effort effort_level

compile

write -hier -f db -o top_unit + "_gate.db"
```

Behavioral Testbench

Example B-5 shows the header file for the FIFO testbench; Example B-6 shows the implementation file; and Example B-7 shows the top-level simulation file, main.cc.

Example B-5 Behavioral Testbench Header File

```
/* fifo_bhv_test.h header file */

SC_MODULE(testbench) {
    sc_in_clk clk;
    sc_in<int> data_in;
    sc_in<bool> full;
    sc_in<bool> empty;

    sc_out<bool> read_fifo;
    sc_out<bool> write_fifo;
    sc_out<int> data_out;
    sc_out<bool> reset;

    void stim();
    void monitor();

    SC_CTOR(testbench) {
         SC_CTHREAD(stim, clk.pos());
    }
};
```

Example B-6 Behavioral Testbench Implementation File

```
/* fifo_bhv_test.cc testbench implementation file */

#include "systemc.h"
#include "fifo_bhv_test.h"

void
testbench::stim()
{
  reset.write(0);
  write_fifo.write(false);
  read_fifo.write(false);
  wait();
```

```
reset.write(true);
wait(2);
reset.write(false);
cout << " *** Reset Done - starting test ***\n";
wait();

write_fifo.write(true);
cout << "FIFO write 1"; data_out.write(1); monitor();
wait();
cout << "FIFO write 2"; data_out.write(2); monitor();
wait();
cout << "FIFO write 3"; data_out.write(3); monitor();
wait();
cout << "FIFO write 4"; data_out.write(4); monitor();
wait();
write_fifo.write(false); monitor(); wait();

read_fifo.write(true); monitor();
wait(2);
cout << "FIFO read " << data_in.read(); monitor();
wait();
cout << "FIFO read " << data_in.read(); monitor();
wait();
cout << "FIFO read " << data_in.read(); monitor();
read_fifo.write(false); wait();
cout << "FIFO read " << data_in.read(); monitor();
wait();

write_fifo.write(true);
cout << "FIFO write 1"; data_out.write(1); monitor();
wait();
cout << "FIFO write 2"; data_out.write(2); monitor();
wait();
cout << "FIFO write 3"; data_out.write(3); monitor();
wait();
write_fifo.write(false); monitor(); wait();

read_fifo.write(true); monitor(); wait(2);
read_fifo.write(false);
cout << "FIFO read " << data_in.read() ; monitor();
wait();
cout << "FIFO read " << data_in.read() ; monitor();
wait();

write_fifo.write(true);
cout << "FIFO write 4"; data_out.write(4); monitor();
wait();
cout << "FIFO write 5"; data_out.write(5); monitor();
```

```
  wait();
  cout << "FIFO write 6"; data_out.write(6); monitor();
  wait();
  write_fifo.write(false); monitor(); wait();

  read_fifo.write(true); monitor();
  wait(2);
  cout << "FIFO read " << data_in.read() ; monitor();
  wait();
  cout << "FIFO read " << data_in.read() ; monitor();
  wait();
  cout << "FIFO read " << data_in.read() ; monitor();
  read_fifo.write(false);  wait();
  cout << "FIFO read " << data_in.read() ; monitor();
  monitor();
  wait(10);

  sc_stop();
}

void
testbench::monitor()
{
  cout << "     FULL = " << full.read();
  cout << " EMPTY = " << empty.read();
  cout << endl;
}
```

Example B-7 Behavioral Top-Level Simulation File

```
/* main.cc simulation file for behavioral fifo testbench. */

#include "systemc.h"
#include "fifo_bhv.h"
#include "fifo_bhv_test.h"

int
main()
{
    sc_signal<bool> reset, write_fifo,
                    read_fifo, full, empty;
    sc_signal<int> data_in, data_out;

    sc_clock clock("Clock", 20.0, 0.5);

    testbench T("Testbench");
    T(clock, data_out, full, empty, read_fifo,
      write_fifo, data_in, reset);

    circ_buf FIFO("FIFO");
    FIFO(clock, read_fifo, write_fifo, data_in,
         reset, data_out, full, empty);

    sc_trace_file *tf =
             sc_create_vcd_trace_file("bhv");
    sc_trace(tf, reset, "Reset");
    sc_trace(tf, write_fifo, "WRITE");
    sc_trace(tf, read_fifo, "READ");
    sc_trace(tf, full, "FULL");
    sc_trace(tf, empty, "EMPTY");
    sc_trace(tf, data_in, "DATA-IN");
    sc_trace(tf, data_out, "DATA-OUT");
    sc_trace(tf, clock.signal(), "Clock");
    sc_start(-1);

    return 0;
}
```

RTL Model

To create an RTL model instead of a behavioral model, you need to do the following:

- Separate the control logic and data path
- Define an explicit FSM for the control logic
- Refine the module to be cycle accurate internally

Example B-8 shows the header file for the RTL version of the FIFO, and Example B-9 shows the implementation file. This RTL example shows the level of detail you need in order to describe an RTL model, which is automatically created by SystemC Compiler from a behavioral description. The RTL coding style has separate processes for the FSM control and data path. The FIFO RTL model has the following separate processes:

- ns_logic

  The process for describing the next state logic.

- update_regs

  The process for updating all the FIFO registers.

- gen_full

  The process for generating a buffer full signal.

- gen_empty

  The process for generating a buffer empty signal.

RTL Description

The I/O communication for the RTL model is identical to the I/O for the behavioral model. Because the FIFO RTL description has four separate processes, the RTL description has extra internal signals to communicate between the processes.

Example B-8 RTL Header File

```
/* fifo_rtl.h header file */

#define BUFSIZE 4
#define LOGBUFSIZE 2
#define LOGBUFSIZEPLUSONE 3


SC_MODULE(circ_buf) {
    // Same I/O as behavioral
    sc_in<bool> clk;
    sc_in<bool> read_fifo;
    sc_in<bool> write_fifo;
    sc_in<int> data_in;
    sc_in<bool> reset;
    sc_out<int> data_out;
    sc_out<bool> full;
    sc_out<bool> empty;

    // Internal signals
    sc_signal<int> buf0, buf0_next;
    sc_signal<int> buf1, buf1_next;
    sc_signal<int> buf2, buf2_next;
    sc_signal<int> buf3, buf3_next;
    sc_signal<sc_uint<LOGBUFSIZEPLUSONE> >
              num_in_buf, num_in_buf_next;
    sc_signal<bool> full_next, empty_next;
    sc_signal<int> data_out_next;

    // Declare processes
    void ns_logic();   // Next-state logic
    void update_regs();// Update all registers
    void gen_full();   // Generate a full signal
    void gen_empty();  // Generate an empty signal

    // Constructor
    SC_CTOR(circ_buf) {
```

```
        SC_METHOD(ns_logic);
        sensitive << read_fifo << write_fifo
          << data_in << num_in_buf;

        SC_METHOD(update_regs);
        sensitive_pos << clk;

        SC_METHOD(gen_full);
        sensitive << num_in_buf_next;

        SC_METHOD(gen_empty);
        sensitive << num_in_buf_next;
    }
};
```

Example B-9 RTL Implementation File

```
/* fifo_rtl.cc implementation file */

#include "systemc.h"
#include "fifo_rtl.h"

void circ_buf::gen_full(){
    if (num_in_buf_next.read() == BUFSIZE)
       full_next = 1;
    else
       full_next = 0;
}

void circ_buf::gen_empty(){
    if (num_in_buf_next.read() == 0)
       empty_next = 1;
    else
       empty_next = 0;
}

void circ_buf::update_regs(){
    if (reset.read() == 1) {
       full = 0;
       empty = 1;
       num_in_buf = 0;
       buf0 = 0;
       buf1 = 0;
       buf2 = 0;
       buf3 = 0;
       data_out = 0;
```

```
    }
    else {
       full = full_next;
       empty = empty_next;
       num_in_buf = num_in_buf_next;
       buf0 = buf0_next;
       buf1 = buf1_next;
       buf2 = buf2_next;
       buf3 = buf3_next;
       data_out = data_out_next;
    }
}

void circ_buf::ns_logic(){
    // Default assignments
    buf0_next = buf0;
    buf1_next = buf1;
    buf2_next = buf2;
    buf3_next = buf3;
    num_in_buf_next = num_in_buf;
    data_out_next = 0;

    if (read_fifo.read() == 1) {
       if (num_in_buf.read() != 0) {
         data_out_next = buf0;
         buf0_next = buf1;
         buf1_next = buf2;
         buf2_next = buf3;
         num_in_buf_next = num_in_buf.read() - 1;
       }
    }
    else if (write_fifo.read() == 1) {
       switch(int(num_in_buf.read())) {
         case 0:
            buf0_next = data_in.read();
            num_in_buf_next = num_in_buf.read() + 1;
            break;
         case 1:
            buf1_next = data_in.read();
            num_in_buf_next = num_in_buf.read() + 1;
            break;
         case 2:
            buf2_next = data_in.read();
            num_in_buf_next = num_in_buf.read() + 1;
            break;
         case 3:
            buf3_next = data_in.read();
            num_in_buf_next = num_in_buf.read() + 1;
```

```
            default:
               // ignore the write command
               break;
         }
      }
}
```

RTL Testbench

The RTL testbench is identical to the behavioral testbench, Example B-5 on page B-11 and Example B-6 on page B-11. Example B-10 shows the top-level RTL simulation file.

Example B-10 RTL Top-Level Simulation File

```
/* main_rtl.cc simulation run file. */

#include "systemc.h"
#include "fifo_rtl.h"
#include "fifo_rtl_test.h"

int
main()
{
    sc_signal<bool> reset, write_fifo,
                    read_fifo, full, empty;
    sc_signal<int> data_in, data_out;

    sc_clock clock("Clock", 20.0, 0.5);

    testbench T("Testbench");
    T(clock, data_out, full, empty, read_fifo,
      write_fifo, data_in, reset);

    circ_buf FIFO("FIFO");
    FIFO(clock, read_fifo, write_fifo, data_in,
         reset, data_out, full, empty);

    sc_trace_file *tf =
            sc_create_vcd_trace_file("bhv");
    sc_trace(tf, reset, "Reset");
    sc_trace(tf, write_fifo, "WRITE");
    sc_trace(tf, read_fifo, "READ");
    sc_trace(tf, full, "FULL");
    sc_trace(tf, empty, "EMPTY");
    sc_trace(tf, data_in, "DATA-IN");
    sc_trace(tf, data_out, "DATA-OUT");
    sc_trace(tf, clock.signal(), "Clock");
    sc_start(-1);

    return 0;
}
```

C

Memory Controller Example

This appendix provides a simple memory controller example.

It contains the following sections:

- Memory Controller Description
- Functional Simulation Model
- Refined Behavioral Model

Memory Controller Description

The memory controller handles all internal memory accesses in a system and provides a simple, command-based interface that lets the testbench or other modules read to and write from memory.

Commands

The memory controller responds to the following four commands (other commands are illegal):

- WTBYT

  The WTBYT command writes a byte of memory. It is a 3-byte sequence of the WTBYT command, address, and data.

- WTBLK

  The WTBLK command writes a block of memory. It is a 6-byte sequence of the WTBLK command, address, and 4 bytes of data.

- RDBYT

  The RDBYT command reads a byte of memory. It is a 3-byte sequence of the RDBYT command, address, and getting the new data.

- NOP

  The NOP command is an idle or no operation state.

Ports

The memory controller has the following ports:

- into

  An sc_in port for reading the command, address, and data

- outof

  An sc_out port for returning the data read from memory

- clk

  An sc_in_clk port for the process

- reset

  An sc_in port for global reset, which de-asserts the com_complete signal

- new_command

  An sc_in port handshake signal that asserts high when a new command is available for processing

- com_command

  An sc_out port handshake signal that asserts high when a command is complete and the memory controller can accept a new command

Communication Protocol

The communication protocol between the memory controller and a testbench or another module executes in the following sequence:

1. A testbench or module communicating with the memory controller writes a token (defined in Example C-4) on the into port, and then it asserts the new_command signal.
2. The memory controller reacts to the assertion of the new_command in the following sequence:
   a. Reads the token from the into port
   b. Executes the WTBYT, WRBLK, or RDBYT command. In the case of a NOP command, it does nothing and skips step 3.
   c. Asserts the com_complete signal
3. The testbench or communicating module reacts to the com_complete assertion and de-asserts the new_command signal
4. The memory controller reacts to the de-assertion of the new_command signal by de-asserting the com_complete signal.

Functional Simulation Model

Example C-1 shows the header file, and Example C-2 shows the implementation file for the functional simulation model of the memory controller. At the functional level of abstraction, the input and output data is a user-defined token, defined in Example C-3. A token is 8 bits wide and implemented with a struct. The token consists of a command_t (defined in Example C-4) with an address of type unsigned char and a four-element array of type unsigned char (defined in Example C-3).

In Example C-1, notice that the process is an SC_METHOD sensitive to the new_command and reset signals.

Example C-1   Memory Controller Header File

```
//mem_controller.h header file
#ifndef _MEM_CONTROLLER_H_
#define _MEM_CONTROLLER_H_

SC_MODULE(mem_controller) {

  sc_in<token>  into;
  sc_in<bool>   reset;
  sc_in<bool>   new_command;
  sc_out<token> outof;
  sc_out<bool>  com_complete;

  // Internal variables
  unsigned char memory[256];

  void entry();
  SC_CTOR(mem_controller) {
      SC_METHOD(entry);
      sensitive << new_command << reset;
  }

};
 #endif
```

Example C-2 Memory Controller Implementation File

```
//mem_controller.cpp implementation file
#include "systemc.h"
#include "memc_types.h"
#include "token.h"
#include "mem_controller.h"

void mem_controller::entry() {
  token com_pkt;

  if (reset == true) {
    com_complete.write(false);
  }
  else if (new_command.posedge()) {

    com_pkt = into.read();
    switch (com_pkt.command) {// opcode
     case NOP:
      break;
     case RDBYT:
      // get data out of memory
      com_pkt.data[0] =
         (memory[com_pkt.address]);
      outof.write(com_pkt);
      break;
     case WTBYT:
      memory[com_pkt.address] = com_pkt.data[0];
      break;
     case WTBLK:
      for(short i=0;i<4;i++) {
        memory[com_pkt.address+i] =
          com_pkt.data[i];
      }
      break;
     default:
      cout << "Illegal opcode : "
           << com_pkt.command << endl;
      break;
    } // end switch
    com_complete.write(true);  // handshake
  } // end else if
```

```
  else if (new_command.negedge()) {
    com_complete.write(false);  //handshake
  }   // end else if else
} // end entry
```

Example C-3   Token Header File

```
//token.h header file
struct token {
    command_t command;
    unsigned char address;
    unsigned char data[4];

    // Define the == operator
    inline bool operator ==
         (const token& rhs) const{
      return
         (command == rhs.command && address == rhs.address &&
          data[0] == rhs.data[0] && data[1] == rhs.data[1] &&
          data[2] == rhs.data[2] && data[3] == rhs.data[3] );
    }
 };
```

Example C-4   Memory Controller Command Types

```
//memc_types.h
#ifndef _MEMC_TYPES_
#define _MEMC_TYPES_

 enum command_t {
  NOP,
  RDBYT,
  WTBYT,
  WTBLK
};

#endif
```

Refined Behavioral Model

Refining the functional simulation model to a behavioral synthesizable model means clarifying the data types and the communication protocol.

The process type is changed to a SC_CTHREAD clocked thread process instead of an SC_METHOD process.

Data Types

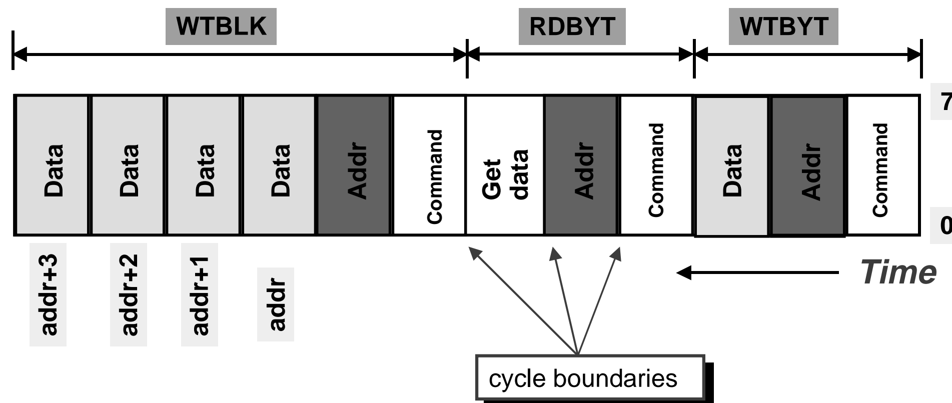
To refine the abstract data types in Example C-3 to specified bit-width data types, Example C-5 shows,

- The into and outof data ports are declared as sc_in<sc_uint<8> > and sc_out<sc_uint<8> > types instead of the abstract token type.
- The memory is declared as an array of type sc_int<8> instead of an array of type unsigned char.

Communication Protocol

Figure C-1 illustrates the data flow into and out of the memory controller.

Figure C-1   Behavioral Input Data Flow



Clock Placement

Notice the placement of wait statements in the implementation file (Example C-6) that implicitly define the control.

Behavioral Model

Example C-5 shows the header file and Example C-6 shows the implementation file for the refined behavioral model of the memory controller. Example C-7 shows a command script for behavioral synthesis to gates.

The functional and behavioral abstraction levels use the same memc_types.h file to define the memory controller commands, shown in Example C-4

Example C-5 Behavioral Header File

```
//mem_controller.h header file

#ifndef _MEM_CONTROLLER_H_
#define _MEM_CONTROLLER_H_

SC_MODULE(mem_controller) {

  sc_in<sc_uint<8> >    into;
  sc_in<bool>           reset;
  sc_out<sc_uint<8> >   outof;
  sc_in_clk             clk;

  // Internal variables
  sc_int<8> memory[32];
 /* snps resource RAM_A: variables = "memory",
  map_to_registerfiles = "TRUE"; */

  void entry();

  SC_CTOR(mem_controller) {
      SC_CTHREAD(entry, clk.pos());
      watching(reset.delayed() == true);
  }
};
#endif
```

Example C-6 Behavioral Implementation File

```
//mem_controller.cpp implementation file
#include <math.h>
#include "systemc.h"
#include "memc_types.h"
#include "mem_controller.h"

void mem_controller::entry() {
  sc_uint<8> data_tmp;
  sc_uint<8> address;
  data_tmp = 0;
  address = 0;
  wait();
  while (true) {
    data_tmp = into.read();
    switch (data_tmp) {// determine opcode
    case NOP:
      wait(); // do nothing
      break;
    case RDBYT:
      wait();
      address = into.read(); // get address
      wait(); //wait one to mimic latency
      data_tmp = memory[address]; // get data out of memory
      outof.write(data_tmp);
      wait();
      break;
    case WTBYT:
      wait();
      address = into.read(); // get address
      wait();
      data_tmp = into.read(); // get data
      memory[address] = data_tmp; // write data
      wait();
      break;
    case WTBLK:
      wait();
      address = into.read(); // get address
      wait();
      for (short i=0; i<4; i++) {
         data_tmp = into.read(); // get data
```

```
            memory[address+i] = data_tmp;
            // write data
            wait();
          }
          break;
        default:
          wait();
          break;
      } // end switch
    } // end while
  } // end entry
```

Example C-7 Behavioral Synthesis to Gates Script

```
top_unit          = "mem_controller"
search_path       = search_path + {"../ram"}
synthetic_library = {"dw01.sldb" "ram.sldb"}
bc_enable_analysis_info  = "true"
target_library    = "tc6a_cbacore.db";
link_library      = target_library + synthetic_library ;

compile_systemc mem_controller.cpp
write -f db -hier -o top_unit + "_elab.db"

create_clock -p 20.0 clk
bc_time_design
write -f db -hier -o top_unit + "_time.db"

schedule -io super
write -f db -hier -o top_unit + "_rtl.db"

compile
write -f db -hier -o top_unit + "_gate.db"
```

D

Fast Fourier Transform Example

This appendix provides a 16-point fast Fourier transform (FFT) example that shows you a functional floating-point model and a behavioral fixed-point model that uses numerous arrays and bit manipulations.

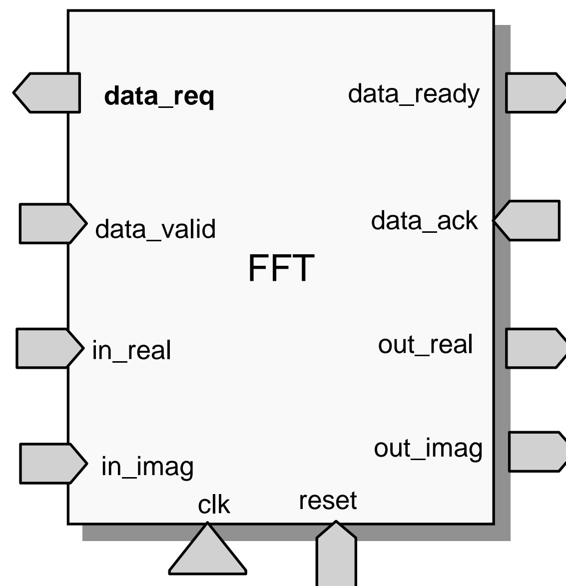
This chapter contains the following sections:

- FFT Description
- FFT Functional Model
- FFT Behavioral Model
- FFT Testbench

FFT Description

Figure D-1 shows the input and output ports and data types for the 16-point FFT.

Figure D-1 FFT Ports and Data Types



FFT Computation

The FFT block computes a 16-point FFT on a sequence of complex inputs by using a radix-2 decimation in frequency algorithm. The input data is read as a signed 16-bit fixed-point number with 10 fractional bits. Twiddle factors and output values are in the same representation.

Internally in the block, computation is performed with fixed-point arithmetic. The input samples and output transformations are externally inferred as 16-bit integers.

Refining From Functional to Behavioral

The floating-point functional version of the FFT was developed to prove the algorithm and verify the results, working at the highest level of abstraction. To use behavioral synthesis, the floating-point version was refined into the fixed-point behavioral version. The data ports were refined from an infinite precision representation to a finite bit-width representation. The computations were refined to fixed-point arithmetic.

Data Read Two-Way Handshake

The FFT block initiates reading of a data sample by assertion of the data_req signal. Next it waits for the data_valid signal to assert. Then it de-asserts the data_req signal and reads from the in_real and in_imag ports. The FFT block reads 16 samples of data.

Data Write Two-Way Handshake

After the FFT calculation is performed, the block writes the transformed values to a sink block in the testbench. It writes the real and imaginary components of the transformed value on the out_real and out_imag ports. Next it asserts the data_ready signal, indicating that the FFT is ready to read data from its ports. It waits for the data_ack signal to assert, then it sends the next set of 16 values.

FFT Functional Model

Example D-1 shows the header file, and Example D-2 shows the implementation file of the 16-point FFT functional model. This model uses floating-point data types, which are refined to fixed-point data types for the behavioral model.

Example D-1 FFT Functional Header File

```
struct fft: sc_module {

  sc_in<sc_int<16> >  in_real;
  sc_in<sc_int<16> >  in_imag;
  sc_in<bool> data_valid;
  sc_in<bool> data_ack;
  sc_out<sc_int<16> > out_real;
  sc_out<sc_int<16> > out_imag;
  sc_out<bool> data_req;
  sc_out<bool> data_ready;
  sc_in<bool> reset;
  sc_in_clk CLK;

  SC_CTOR(fft)
    {
      SC_CTHREAD(entry, CLK.pos());
      watching(reset.delayed()==true);
    }

 void entry();
};
```

Example D-2   FFT Functional Description File

```
// fft.cpp floating-point functional model.

#include "systemc.h"
#include "fft.h"
#include "math.h"
void fft::entry()
{ float sample[16][2];
  unsigned int index;

  while(true) {
    data_req.write(false);
    data_ready.write(false);
    index = 0;
    //Reading in the Samples
      cout << endl << "Reading in the samples..." << endl;
      while( index < 16 ) {
       data_req.write(true);
       wait_until(data_valid.delayed() == true);
       sample[index][0] = in_real.read();
       sample[index][1] = in_imag.read();
       index++;
       data_req.write(false);
       wait();
      }
      index = 0;


      /*  Computation:
          1D Complex DFT In-Place DIF
          Computation Algorithm
      */

      //Size of FFT, N = 2**M
       unsigned int N, M, len ;
       float theta;
       float W[7][2], w_real, w_imag, w_rec_real;
       float w_rec_imag, w_temp;

       //Initialize
       M = 4; N = 16;
       len = N/2;
       theta = 8.0*atan(1.0)/N;

       cout << "Computing..." << endl;

       //Calculate the W-values recursively
        w_real =  cos(theta);
```

```
    w_imag =  -sin(theta);

    w_rec_real = 1;
    w_rec_imag = 0;

    index = 0;
    while(index < len-1)
    {
       w_temp = w_rec_real*w_real - w_rec_imag*w_imag;
       w_rec_imag =  w_rec_real*w_imag + w_rec_imag*w_real;
       w_rec_real = w_temp;
       W[index][0] = w_rec_real;
       W[index][1] = w_rec_imag;
       index++;
    }


   float tmp_real, tmp_imag, tmp_real2, tmp_imag2;
   unsigned int stage, i, j,index2, windex, incr;

  //Begin Computation
   stage = 0;

   len = N;
   incr = 1;

   while (stage < M)
   {
    len = len/2;

    //First Iteration :  With No Multiplies
      i = 0;


      while(i < N)
      {
         index =  i; index2 = index + len;

         tmp_real = sample[index][0] + sample[index2][0];
         tmp_imag = sample[index][1] + sample[index2][1];

         sample[index2][0] = sample[index][0] -
                             sample[index2][0];
         sample[index2][1] = sample[index][1] -
                             sample[index2][1];

         sample[index][0] = tmp_real;
         sample[index][1] = tmp_imag;
```

```
        i = i + 2*len;

     }


   //Remaining Iterations: Use Stored W
    j = 1; windex = incr - 1;
    while (j < len) // This loop executes N/2 times
                    // at first stage, and once at last stage.
    {
       i = j;
       while (i < N)
       {
         index = i;
         index2 = index + len;

         tmp_real = sample[index][0] + sample[index2][0];
         tmp_imag = sample[index][1] + sample[index2][1];
         tmp_real2 = sample[index][0] - sample[index2][0];
         tmp_imag2 = sample[index][1] - sample[index2][1];

         sample[index2][0] = tmp_real2*W[windex][0] -
                             tmp_imag2*W[windex][1];
         sample[index2][1] = tmp_real2*W[windex][1] +
                             tmp_imag2*W[windex][0];

         sample[index][0] = tmp_real;
         sample[index][1] = tmp_imag;

         i = i + 2*len;

       }
       windex = windex + incr;
       j++;
    }
     stage++;
     incr = 2*incr;
  }

/////////////////////////////////////////////////////

// Writing out the normalized transform values
// in bit reversed order
 sc_uint<4> bits_i;
 sc_uint<4> bits_index;
 bits_i = 0;
```

```
      i = 0;

      cout << "Writing the transform values..." << endl;
      while( i < 16)
      {
       bits_i = i;
       bits_index[3]= bits_i[0];
       bits_index[2]= bits_i[1];
       bits_index[1]= bits_i[2];
       bits_index[0]= bits_i[3];
       index = bits_index;
       out_real.write(sample[index][0]);
       out_imag.write(sample[index][1]);
       data_ready.write(true);
       wait_until(data_ack.delayed() == true);
       data_ready.write(false);
       i++;
       wait();
      }
      index = 0;
      cout << "Done..." << endl;
  }
 }
```

FFT Behavioral Model

Example D-3 shows the header file, and Example D-4 shows the implementation file of the 16-point FFT model. Example D-5 shows a command script for behavioral synthesis to gates.

Example D-3 FFT Header File

```
// fft.h fft_module header file
SC_MODULE(fft_module) {

  // Input ports Declaration
  sc_in<sc_int<16> >  in_real;
  sc_in<sc_int<16> >  in_imag;
  sc_in<bool> data_valid;
  sc_in<bool> data_ack;
  sc_in<bool> reset;

  // Output ports Declaration
  sc_out<sc_int<16> > out_real;
  sc_out<sc_int<16> > out_imag;
  sc_out<bool> data_req;
  sc_out<bool> data_ready;


  // Clock Declaration
  sc_in_clk clk;

  // Declare implementation functions
  void fft_process();

  // Constructor
  SC_CTOR(fft_module)
    {
      SC_CTHREAD(fft_process, clk.pos());
      watching(reset.delayed()==true);
    }

};
```

Example D-4 FFT Implementation File

```
// fft.cc FFT implementation file
#include "systemc.h"
#include "fft.h"

/*********************************************
         Function Definition for butterfly computation
*********************************************/
 void func_butterfly
    ( const sc_int<16>& w_real,
      const sc_int<16>& w_imag,
      const sc_int<16>& real1_in,
      const sc_int<16>& imag1_in,
      const sc_int<16>& real2_in,
      const sc_int<16>& imag2_in,
      sc_int<16>& real1_out,
      sc_int<16>& imag1_out,
      sc_int<16>& real2_out,
      sc_int<16>& imag2_out
    ) {

   // Variable declarations
     sc_int<17> tmp_real1;
     sc_int<17> tmp_imag1;
     sc_int<17> tmp_real2;
     sc_int<17> tmp_imag2;
     sc_int<34> tmp_real3;
     sc_int<34> tmp_imag3;

    // Begin Computation
    tmp_real1 = real1_in + real2_in;

    // <s,6,10> = <s,5,10> + <s,5,10>
    tmp_imag1 = imag1_in + imag2_in;

    // <s,6,10> = <s,5,10> - <s,5,10>
    tmp_real2 = real1_in - real2_in;

    // <s,6,10> = <s,5,10> - <s,5,10>
    tmp_imag2 = imag1_in - imag2_in;

    // <s,13,20> = <s,6,10>*<s,5,10> -
    // <s,6,10>*<s,5,10>
    tmp_real3 = tmp_real2*w_real - tmp_imag2*w_imag;

    //  <s,13,20> = <s,6,10>*<s,5,10> +
    //  <s,6,10>*<s,5,10>
    tmp_imag3 = tmp_real2*w_imag + tmp_imag2*w_real;
```

```
    // assign the sign-bit(MSB)
    real1_out[15] = tmp_real1[16];
    imag1_out[15] = tmp_imag1[16];

    // assign the rest of the bits
    real1_out.range(14,0) = tmp_real1.range(14,0);
    imag1_out.range(14,0) = tmp_imag1.range(14,0);

   // assign the sign-bit(MSB)
    real2_out[15] = tmp_real3[33];
    imag2_out[15] = tmp_imag3[33];

   // assign the rest of the bits
    real2_out.range(14,0) = tmp_real3.range(24,10);
    imag2_out.range(14,0) = tmp_imag3.range(24,10);

 }; // end func_butterfly

/*************************************
             Process Definition Begin
**************************************/

void fft_module::fft_process() {

/*************************************
             Variable Declarations
**************************************/
   sc_int<16> real[16];
   /* snps resource reg_a: variables = "real",
      map_to_registerfiles = "TRUE"; */
   sc_int<16> imag[16];
   /* snps resource reg_b: variables = "imag",
      map_to_registerfiles = "TRUE"; */
   sc_int<16> W_real[7];
   /* snps resource reg_c: variables = "W_real",
      map_to_registerfiles = "TRUE"; */
   sc_int<16> W_imag[7];
   /* snps resource reg_d: variables = "W_imag",
      map_to_registerfiles = "TRUE"; */
   sc_int<16> w_real;
   sc_int<16> w_imag;
   sc_int<16> real1_in;
   sc_int<16> imag1_in;
   sc_int<16> real2_in;
   sc_int<16> imag2_in;
   sc_int<16> real1_out;
   sc_int<16> imag1_out;
```

```
   sc_int<16> real2_out;
   sc_int<16> imag2_out;
   sc_int<4>  stage;
   sc_int<6> N;
   sc_int<4> M;
   sc_int<6> len;
   sc_uint<4> bits_i;
   sc_uint<4> bits_index;
   short i;
   short j;
   short index;
   short index2;
   short windex;
   short incr;

/********************************************
              Reset/Initializion of signals and variables
********************************************/
   data_req.write(0);
   data_ready.write(0);
   index = 0;
   W_real[0] = 942;// Precomputed twiddle factors for 16 point FFT
   W_imag[0] = -389 ;
   W_real[1] = 718;
   W_imag[1] = -716;
   W_real[2] = 388;
   W_imag[2] = -932;
   W_real[3] = 2;
   W_imag[3] = -1005;
   W_real[4] = -380;
   W_imag[4] = -926;
   W_real[5] = -702;
   W_imag[5] = -708;
   W_real[6] = -915;
   W_imag[6] = -385;

   wait();

/*********************************************
              Overall Functionality Loop
*********************************************/

 while(true) {
     wait();
    /*****************************************
     Read Input Samples Look
     *****************************************/
      cout << endl << "Reading in the samples..."
```

```
      << endl;

while( index < 16 ) {
  data_req.write(1);
  wait_until(data_valid.delayed() == 1);
  real[index] = in_real.read();
  imag[index] = in_imag.read();
  index++;
  data_req.write(0);
  wait();
 }
  // Initialize
  index = 0;
  M = 4; N = 16;
  len = N >> 1;
  stage = 0;
  len = N;
  incr = 1;

  cout << "Computing..." << endl;

/****************************************
  Stages Loop
  ***************************************/
  // Loop iterates over the number of stages. There are M stages
  // where M = log2(N), already defined above. Loop control variable
  // is "stages". For every iteration stage it is incremented by 2
// and incr is multiplied by 2.

  while (stage < M)
  {
    len = len >> 1;
    i = 0;

 /***************************************
  First Pass Loop
  ***************************************/
  // Loop does the following:
     // a. loop execute condition (checked before executing anything)
  //  is i < N
     // b. "i" is updated for every next iteration as i = i + len*2

    while(i < N) {
      index =  i; index2 = i + len;

      real1_out = real[index] + real[index2];
      imag1_out = imag[index] + imag[index2];
```

```
        real[index2] = (real[index] - real[index2]);
        imag[index2] = (imag[index] - imag[index2]);

       real[index] = real1_out;
       imag[index] = imag1_out;

       i = i + (len << 1);
       wait();
     }
      j = 1; windex = incr - 1;

 /***************************************
  Remaining Passes Loop
  ***************************************/
    // This loop executes N/2 times at the first stage,
  // N/2 times at the second, and once at last stage.

    while (j < len)
    {
       i = j;
       while (i < N)
       {
         index = i;
         index2 = i + len;

// Read in the data and twiddle factors

w_real  = W_real[windex];
w_imag  = W_imag[windex];
         real1_in = real[index];
imag1_in = imag[index];
         real2_in = real[index2];
imag2_in = imag[index2];

// Call butterfly computation function
func_butterfly(w_real, w_imag, real1_in,
     imag1_in, real2_in, imag2_in, real1_out, imag1_out,
  real2_out, imag2_out);

// Store back the results
         real[index]  = real1_out;
         imag[index]  = imag1_out;
         real[index2] = real2_out;
         imag[index2] = imag2_out;

         i = i + (len << 1);
       }
       windex = windex + incr;
```

```
            j++;
         }

          stage++;
          incr = incr << 1;
       }

      bits_i = 0;
      bits_index = 0;
      i = 0;
      cout << "Writing the transform values..." << endl;

 /**************************************
             Write Transform Values Loop
  **************************************/
    // Write loop that writes the transform values to output
   // ports out_real and out_imag.

     while( i < 16)
      {
       bits_i = i;
       bits_index[3]= bits_i[0];
       bits_index[2]= bits_i[1];
       bits_index[1]= bits_i[2];
       bits_index[0]= bits_i[3];
       index = bits_index;
       out_real.write(real[index]);
       out_imag.write(imag[index]);
       data_ready.write(1);
       wait_until(data_ack.delayed() == true);
       data_ready.write(0);
       i++;
       wait();
      }

      index = 0;
      cout << "Done..." << endl;
 }
}
/****************************************
              End Process Definition
****************************************/
```

Example D-5 Behavioral Synthesis to Gates Script

```
search_path       = search_path + "$SYNOPSYS/libraries/syn"
target_library    = {"tc6a_cbacore.db"};
synthetic_library = {"dw01.sldb","dw02.sldb"}
link_library    = {"*"} + target_library + synthetic_library

bc_enable_analysis_info    = "false"
effort_level               = medium
io_mode                    = super
top_unit                   = "fft"

sh date
compile_systemc top_unit + ".cc"

create_clock clk -p 25

bc_time_design

schedule -io io_mode -effort effort_level

compile

write -hier -f db -o top_unit + "_gate.db"
```

FFT Testbench

The FFT testbench consists of three files: source.cc, sink.cc, and main_fft.cc.

- The source.cc file reads in real and imaginary samples from files named in_real and in_imag, which are ASCII files containing values. The source block interacts with the FFT behavioral block using two-way handshake.
- The sink.cc file reads the real and imaginary components of the output transform values from the FFT block. It writes the values to output files named out_real and out_imag, which are ASCII format files of the output values. The sink block also interacts with the FFT block by using two-way handshake.

Example D-6 shows the source block, Example D-7 shows the sink block, and Example D-8 shows the top-level simulation executable main_fft.cc.

Example D-6 FFT Testbench Source

```
// source.h header file

SC_MODULE(source_module) {
  sc_in<bool> data_req;
  sc_out<sc_int<16> >  out_real;
  sc_out<sc_int<16> >  out_imag;
  sc_out<bool> data_valid;
  sc_out<bool> reset;
  sc_in_clk CLK;

  void source_process();
  SC_CTOR(source_module)
    {
      SC_CTHREAD(source_process, CLK.pos());
    }

};


/****************************/
// source.cc implementation file

#include "systemc.h"
#include "source.h"
void source_module::source_process()
{ FILE *fp_real, *fp_imag;

    int tmp_val;

    fp_real = fopen("in_real", "r");
    fp_imag = fopen("in_imag", "r");

  reset.write(true);
  wait(5);
  reset.write(false);
  data_valid.write(false);

  while(true)
  {
    wait_until(data_req.delayed() == true);
    if (fscanf(fp_real,"%d", &tmp_val) == EOF)
    { cout << "End of Input Stream: Simulation Stops" << endl;
        sc_stop();
        break;
        };
    out_real.write(tmp_val);
    if (fscanf(fp_imag,"%d", &tmp_val) == EOF)
```

```
    { cout << "End of Input Stream: Simulation Stops" << endl;
         sc_stop();
         break;
         };
     out_imag.write(tmp_val);
     data_valid.write(true);
     wait_until(data_req.delayed() == false);
     data_valid.write(false);
     wait();
   }
 }
```

Example D-7 FFT Testbench Sink

```
// sink.h header file

SC_MODULE(sink_module) {
 sc_in<bool>  data_ready;
 sc_out<bool> data_ack;
 sc_in< sc_int<16> > in_real;
 sc_in< sc_int<16> > in_imag;
 sc_in<bool> reset;
 sc_in_clk CLK;

 void sink_process();

 SC_CTOR(sink_module) {
     SC_CTHREAD(sink_process, CLK.pos());
     watching(reset.delayed() == 1);
  }

};
/*****************************/
// sink.cc implementation file

#include "systemc.h"
#include "sink.h"

void sink_module::sink_process(){
 FILE *fp_real, *fp_imag;
 sc_int<16> tmp;
 int tmp_out;
 fp_real = fopen("out_real","w");
 fp_imag = fopen("out_imag","w");

 data_ack.write(false);

 while(true){
   wait_until(data_ready.delayed() == true);
   tmp = in_real.read();
   tmp_out = tmp;
   fprintf(fp_real,"%d  \n",tmp_out);
   tmp = in_imag.read();
   tmp_out = tmp;
   fprintf(fp_imag,"%d  \n",tmp_out);
   data_ack.write(true);
   wait_until(data_ready.delayed() == false);
   data_ack.write(false);
 }
}
```

Example D-8 FFT Testbench Top-Level Model

```
// Filename: main_fft.cc
// This file instantiates all modules and ties them together with signals

#include "systemc.h"
#include "fft.h"
#include "source.h"
#include "sink.h"

int sc_main(int ac, char* av[])
{
 sc_signal<sc_int<16> > in_real;
 sc_signal<sc_int<16> > in_imag;
 sc_signal<bool> data_valid;
 sc_signal<bool> data_ack;
 sc_signal<sc_int<16> > out_real;
 sc_signal<sc_int<16> >  out_imag;
 sc_signal<bool> data_req;
 sc_signal<bool> data_ready;
 sc_signal<bool> reset;
 sc_clock clock("CLOCK", 10, 0.5, 0.0);

 fft_module FFT1("FFTPROCESS");
 FFT1.in_real(in_real);
 FFT1.in_imag(in_imag);
 FFT1.data_valid(data_valid);
 FFT1.data_ack(data_ack);
 FFT1.out_real(out_real);
 FFT1.out_imag(out_imag);
 FFT1.data_req(data_req);
 FFT1.data_ready(data_ready);
 FFT1.reset(reset);
 FFT1.clk(clock);

 source_module SOURCE1("SOURCEPROCESS");
 SOURCE1.data_req(data_req);
 SOURCE1.out_real(in_real);
 SOURCE1.out_imag(in_imag);
 SOURCE1.data_valid(data_valid);
 SOURCE1.reset(reset);
 SOURCE1.CLK(clock);

 sink_module SINK1("SINKPROCESS");
 SINK1.data_ready(data_ready);
 SINK1.data_ack(data_ack);
 SINK1.in_real(out_real);
 SINK1.in_imag(out_imag);
 SINK1.reset(reset);
```

```
 SINK1.CLK(clock);

 sc_start(clock, -1);

 return 0;
}
```

E

Inverse Quantization Example

This appendix provides an inverse quantization (IQ) example that shows you a complex behavioral model. This model uses many member functions to describe the functionality, which makes it easier to understand the functional complexity.

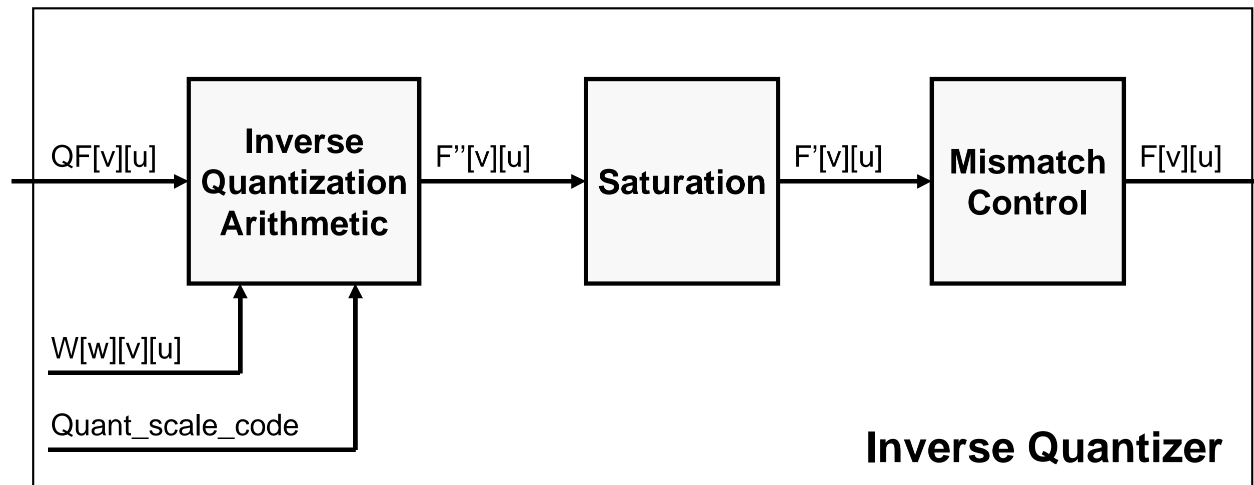
This chapter contains the following sections:

- IQ Description
- IQ Behavioral Model

IQ Description

The IQ is a block in an MPEG-2 that contains subblocks for inverse quantization arithmetic, saturation, and mismatch control, as shown in Figure E-1.

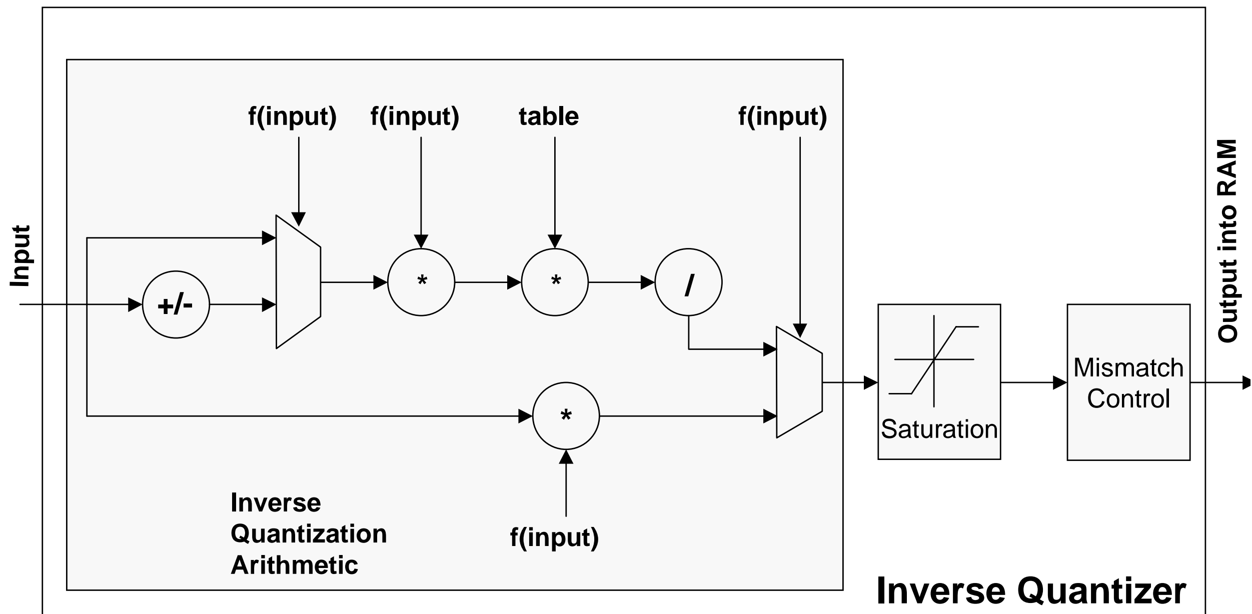
Figure E-1 IQ Blocks



IQ Data Flow

Figure E-2 shows the IQ arithmetic block and the data flow into the saturation and mismatch control blocks.

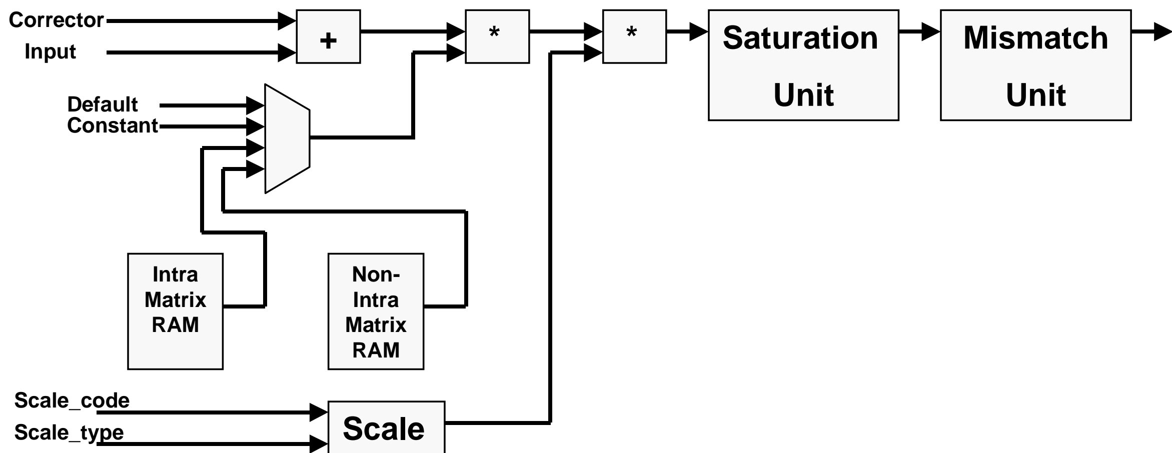
Figure E-2 IQ Data Flow



IQ Block Diagram

Figure E-3 shows the IQ block diagram.

Figure E-3 IQ Block Diagram



IQ Behavioral Model

Example E-1 shows the header file, and Example E-2 shows the implementation file for the IQ behavioral model. Example E-3 shows a command script to synthesize the model to gates.

Example E-1 IQ Header File

```
SC_MODULE (VD_iq) {
  // Declare ports
  sc_in_clk           CLK;
  sc_in<bool>         reset;
  sc_in<bool>         iq_start;
  sc_in<bool>         slice;
  sc_in<bool>         load_intra_quantizer_matrix;
  sc_in<bool>         load_non_intra_quantizer_matrix;
  sc_in<sc_uint<8> >  W;
  sc_in<bool>         run_level_valid;
  sc_in<sc_uint<5> >  quantizer_scale_code;
  sc_in<sc_uint<2> >  intra_dc_precision ;
  sc_in<sc_uint<5> >  dct_dc_size;
  sc_in<sc_uint<11> > dct_dc_differential;
  sc_in<bool>         q_scale_type;
  sc_in<bool>         alternate_scan;
  sc_in<bool>         end_of_block;
  sc_in<bool>         mblock_intra;
  sc_in<sc_uint<4> >  block_count;
  sc_in<sc_uint<6> >  run;
  sc_in<sc_uint<12> > level;
  sc_out<sc_int<12> > f;
  sc_out<bool>        f_valid;
  sc_out<sc_uint<6> > f_addr;
  sc_out<bool>        iq_block_ready;
  sc_out<bool>        iq_calc_ready;
  sc_in<bool>         iq_write_block;
  sc_out<bool>        iq_error;
  sc_in<bool>         iq_skip;
  sc_in<bool>         iq_clear;

  // Internal signals
  sc_signal<bool>        acknowledge_data;
  sc_signal<bool>        valid_data;
  sc_signal<sc_uint<8> > error;
  sc_signal<bool>        skip_block_out;

  // Data members
  sc_uint<7> q_scale[32];
  sc_uint<6> scan_zigzag[64];
  sc_uint<6> scan_alternate[64];
  sc_uint<7> default_intra_quant[64];

  bool       iq_sleep;
  bool       eob_tmp;
  sc_uint<4> wait_for_ack;
  bool       previous_tmp;
```

```
  sc_uint<5> quantizer_scale_code_inp;
  sc_uint<7> current_address;
  bool       macroblock_intra;
  bool       q_scale_type_inp;
  bool       alternate_scan_inp;
  bool       reset_dct_pred_inp;
  sc_int<12> level_inp;
  sc_uint<6>  run_inp;
  sc_uint<4>  dct_size_inp;
  sc_uint<11> dct_diff_inp;
  sc_int<12> dc_dct_pred[3];
  sc_int<26> f_t_t;
  sc_int<12> f_t;
  sc_uint<2> corrector;
  sc_uint<3> block_count_tmp;
  sc_uint<7> quantizer_scale;
  sc_uint<7> matrix_value;
  sc_uint<2> intra_dc_precision_inp;
  sc_uint<4> block_count_inp;
  bool       mismatch_control;
  sc_uint<7> intra_matrix_ram[64];
  sc_uint<7> non_intra_matrix_ram[64];
  bool       next_value[64];
  sc_int<12> f_out;
  sc_uint<6> f_addr_out;
  bool       use_load_intra_matrix_inp;
  bool       use_load_non_intra_matrix_inp;
  sc_uint<8> offset;

  sc_int<12> f_mat[64];

  sc_signal<bool> CLK_iqgate;

  // Declare implementation functions
  void entry();
  void generate_valid();

  SC_CTOR (VD_iq) {
        SC_CTHREAD (entry, CLK.pos());
        SC_CTHREAD (generate_valid, CLK.pos());
         watching(reset.delayed() == true);
      }

  // Declare member functions
  sc_uint<2> corrector_calc();
  sc_uint<7> quantizer ();
  sc_int<12> saturation ();
  sc_int<26> quantization ();
```

```
  sc_int<12> mismatch ();
  sc_int<26> intra_mult ();
  sc_uint<6> inverse_scan ();
  void       reset_prediction();
  void       reset_action();
  sc_uint<7> W_lookup ();

}; //End of SC_MODULE
```

Example E-2 IQ Implementation File

```
// VD_iq.cc implementation file

#define MAX_WAIT_FOR_ACK 10
#define MAX_ADDRESS 64
#include <systemc.h>
#include "VD_iq.h"

void VD_iq::entry() {

  // Define local variables
  sc_uint<7> next_hit;
  bool       fill;
  bool       skip_inp;
  bool       clear_inp;
  sc_uint<6> mblock_intra_inp;

  // Synthesis attributes
  /* synopsys resource RAM_A:
     variables = "intra_matrix_ram",
     map_to_module = "lsi_6_7"; */
  /* synopsys resource RAM_B:
     variables = "non_intra_matrix_ram",
     map_to_module = "lsi_6_7"; */

  // Reset behavior in member function
  reset_action();
  wait();

  // Main functionality
  while (true) {

    // This start signal comes from the stream parser
    iq_block_ready.write(false);
    acknowledge_data.write(false);
    wait_until(iq_start.delayed()==true);
    wait();
```

```
    skip_inp  = iq_skip.read();
    clear_inp = iq_clear.read();

    wait();
    if(iq_write_block.read()==true)
      offset = MAX_ADDRESS;
    else
      offset = 0;

    if (skip_inp==false) {
      // Sample data that are block constant
      mblock_intra_inp         = mblock_intra.read();
      macroblock_intra         = mblock_intra_inp;
      alternate_scan_inp       = alternate_scan.read();
      quantizer_scale_code_inp =
          quantizer_scale_code.read();
      q_scale_type_inp         = q_scale_type.read();
      intra_dc_precision_inp   = intra_dc_precision.read();
      block_count_inp          = block_count.read();

      // Initial values for a block
      mismatch_control   = 0;

      // Special treatment for dc values in intra pictures,
      // see 7.4.1 in MPEG spec
      if (macroblock_intra==true) {
        dct_diff_inp   = 0;
        dct_diff_inp   = dct_dc_differential.read();
        dct_size_inp   = dct_dc_size.read();
        f_t_t = intra_mult();
        // Saturation
        f_t = saturation();

        // Mismatch control
        if (f_t[0] == 1)
          mismatch_control = !mismatch_control;
        f_out =  f_t;

        // Output assignment
        f_addr_out = 0;
        f_addr.write(f_addr_out);
        f.write(f_out);
        f_valid.write(true);
        f_mat[f_addr_out+offset] = f_out;
        current_address    = 1;
      } else {
        // If we don't have a intra block
        current_address    = 0;
```

```
}
wait();

// Pipeline that does the main computation
main_loop:while(1) {

  f_valid.write(false);
  acknowledge_data.write(false);
  // Wait for a valid signal for end of the block
  wait_until(valid_data.delayed()==true);
  wait();
  // Checking if block ends
  if (end_of_block.read()==true)
    break;
  acknowledge_data.write(true);

  // Sample inputs of normal block behavior and compensate for run
  level_inp   = level.read();
  run_inp     = run.read();
  if(run_inp!=0)
    current_address   = current_address+run_inp;

  next_value[current_address] = true;

  // Correct the values for dequantization
  corrector       = corrector_calc();
  quantizer_scale = quantizer();

  // Memory access
  matrix_value = W_lookup();

  // Multiplications and correction
  f_t_t = quantization();

  // Saturation
  f_t = saturation();

  // Mismatch control
  if (f_t[0] == 1)
    mismatch_control = !mismatch_control;

  if(current_address==(MAX_ADDRESS - 1))
    f_out =  mismatch();
  else
    f_out =  f_t;

  // Address assignment
  f_addr_out = inverse_scan();
```

```
      // Output assignment
      f_addr.write(f_addr_out);
      f.write(f_out);
      f_valid.write(true);
      f_mat[f_addr_out+offset] = f_out;
      wait();
      // Prepare for next iteration
      current_address++;
    }
  } else
    wait();

  iq_calc_ready.write(true);
  wait();

  // This part isn't necessary when skipping a
  // block and the output is already cleared.
  if ((skip_inp==false) || (clear_inp==true)) {

    acknowledge_data.write(false);
    // Fill the empty spots, otherwise
    // the positions of the second block are not
    // correct as written in the previous cycle.
    // This could be done more efficiently
    // if the RAM has a reset.
    current_address = 0;
    iq_calc_ready.write(false);
    f_out           = 0;
    wait();
    do {

      current_address++;
      if((next_value[current_address]==true) && (clear_inp==false))
        fill     = false;
      else
        fill     = true;
      next_value[current_address]=false;

      if(fill==true) {
        f_addr_out = inverse_scan();
        f_addr.write(f_addr_out);

        f_t = 0;
        if(current_address==(MAX_ADDRESS - 1))
          f_out = mismatch();
        else
          f_out = f_t;
```

```
          f.write(f_out);
          f_valid.write(true);
          f_mat[f_addr_out+offset] = f_out;
        }
        wait();
      } while(current_address< (MAX_ADDRESS - 1));
    } else
      wait();

    // One block is done.
    iq_block_ready.write(true);
    f_valid.write(false);
    wait();

  }
}

// Define member functions

sc_uint<2> VD_iq::corrector_calc () {
   //  synopsys preserve_function
  unsigned tmp;

  if ((macroblock_intra==true) || (level_inp==0))
    tmp = 0;
  else {
    if (level_inp>0)
      tmp = 1;
    else
      tmp = 2;
    }
  return tmp;
}

sc_uint<7> VD_iq::quantizer () {
  //  synopsys preserve_function
  unsigned tmp;

  if (q_scale_type_inp==true)
    tmp = q_scale[quantizer_scale_code_inp];
  else
    tmp = quantizer_scale_code_inp<<1;

  return tmp;
}

sc_int<12> VD_iq::saturation () {
```

```
  //  synopsys preserve_function
  int tmp;
  int divide;

  divide = f_t_t;
  if (divide<-2048)
    tmp = -2048;
  else {
    if (divide>2047)
      tmp = 2047;
    else
      tmp = divide;
    }
  return tmp;

}

sc_int<26> VD_iq::quantization () {
  //  synopsys preserve_function
  int tmp1;
  sc_int<26> tmp2;
  tmp1 = (level_inp<<1) + corrector;
  tmp2 = tmp1 * matrix_value * quantizer_scale;
  // proper rounding
  if (((tmp2&0x1f) != 0) && (tmp2<0)) {
    tmp2 = (tmp2>>5) + 1;
  } else {
    tmp2 = tmp2>>5;
  }
  // cout << " test " << tmp1 << " "
          << tmp2 << endl;
  return (tmp2);
}

sc_int<12> VD_iq::mismatch () {
  //  synopsys preserve_function
  int   tmp;
  bool  f_t_bit;

  f_t_bit = f_t[0];
  tmp = f_t;
  if (mismatch_control==false) {
    if (f_t_bit==false)
      tmp = f_t+1;
    else
      tmp = f_t-1;
  }
```

```
  return tmp;
}

void VD_iq::reset_prediction() {
  dc_dct_pred[0] = 0;
  dc_dct_pred[1] = 0;
  dc_dct_pred[2] = 0;
}

void VD_iq::generate_valid() {
  bool tmp = false;

  wait();
  while(1) {
    tmp     = (run_level_valid.read()) ||
              (end_of_block.read());
    eob_tmp = end_of_block.read();
    previous_tmp = tmp;
    valid_data.write(tmp);
    if ((tmp==true) && (previous_tmp!=true)) {
      wait_for_ack=0;
    }

    if (wait_for_ack<MAX_WAIT_FOR_ACK)
      wait_for_ack++;

    if (wait_for_ack==(MAX_WAIT_FOR_ACK-1)) {
      cout << "Error : VD_iq(generate_valid)"
           << " Valid without acknowledge at time "
           << sc_time_stamp() << endl;
      wait_for_ack = MAX_WAIT_FOR_ACK;
    }

    if ((acknowledge_data.read()==true) &&
        (wait_for_ack!=MAX_WAIT_FOR_ACK)) {
      wait_for_ack = MAX_WAIT_FOR_ACK;
    }
    wait();
  }
}

sc_int<26> VD_iq::intra_mult () {
  //  synopsys preserve_function
  sc_int<16>  tmp ;
  sc_uint<14> half_range ;
  sc_int<14>  dct_diff ;

  dct_diff = 0;
```

```
  tmp      = 0;
  if (dct_size_inp!=0) {
    half_range = 1 << (dct_size_inp-1);

    if (dct_diff_inp>=half_range)
      dct_diff.range(10,0) = dct_diff_inp;
    else
      dct_diff = dct_diff_inp+1-(half_range<<1);
  }

  if (block_count_inp<4) {
    dc_dct_pred[0] = dc_dct_pred[0]+dct_diff;
    tmp = (int) dc_dct_pred[0];
  } else {
    if (block_count_inp==4) {
      dc_dct_pred[1] = dc_dct_pred[1]+dct_diff;
      tmp  = (int) dc_dct_pred[1];
    } else {
      dc_dct_pred[2] = dc_dct_pred[2]+dct_diff;
      tmp  = (int) dc_dct_pred[2];
    }
  }

  // intra multiplication
  tmp = tmp << (3-intra_dc_precision_inp);
  return(tmp);
}

sc_uint<6> VD_iq::inverse_scan() {
  unsigned tmp;

  if (alternate_scan_inp==false)
    tmp = scan_alternate[current_address];
  else
    tmp = scan_zigzag[current_address];

  return tmp;
}

void VD_iq::reset_action() {
  f_t       = 0;
  f_t_t     = 0;
  f_out     = 0;
  level_inp = 0;
  run_inp   = 0;
  corrector = 0;
  quantizer_scale = 0;
  matrix_value    = 0;
```

```
  iq_sleep        = false;
  f_addr.write(0);
  f.write(0);
  f_valid.write(false);
  iq_calc_ready.write(false);
  acknowledge_data.write(false);
}

sc_uint<7> VD_iq::W_lookup() {
  unsigned matrix_value_tmp;

  if (macroblock_intra==false && use_load_non_intra_matrix_inp==true)
      matrix_value_tmp =
      non_intra_matrix_ram[inverse_scan()];
  else {
     if (macroblock_intra==false && use_load_non_intra_matrix_inp==false)
      matrix_value_tmp = 16;
    else {
      if (macroblock_intra==true && use_load_non_intra_matrix_inp==true)
          matrix_value_tmp =
          intra_matrix_ram[inverse_scan()];
      else
          matrix_value_tmp =
          default_intra_quant[inverse_scan()];
    }
  }
  return matrix_value_tmp;
}
```

Example E-3 Behavioral Synthesis to Gates Script

```
search_path      = search_path + "$SYNOPSYS/libraries/syn"
+ "./ram"
target_library    = {"tc6a_cbacore.db"};
synthetic_library = {"dw01.sldb","ram.sldb"}
link_library    = {"*"} + target_library + synthetic_library

bc_enable_analysis_info    = "false"
effort_level               = medium
io_mode                    = super
top_unit                   = "VD_iq"

sh date
define_design_lib RAMS -path ./ram
define_design_lib DBS -path ./db

compile_systemc top_unit + ".cc"

compile_preserved_functions

create_clock CLK -p 20

bc_time_design

schedule -io io_mode -effort effort_level

compile -map low

write -hier -f db -o top_unit + "_gate.db"
```

F

Expressions and Operations

This appendix provides basic information about using expressions and operators in a SystemC behavioral description.

Using Expressions

In C++, an expression is a combination of operators and operands that can be evaluated according to the semantic rules of the language. Operators specify the computation to perform. In the following code fragment, A and B are operands, + is an operator, and A + B is an expression. Expressions are often enclosed within parentheses, but they do not have to be.

```
C = (A + B);
```

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

- Assign them to variables or signals or use them as initial values of constants
- Use them as operands to other operators
- Use them for the return value of functions
- Use them as input parameters in a function call
- Use them to control the actions of statements such as if, loop, and case

For complex expressions, enclose the expression in parentheses and use nested parentheses to specify the order of evaluation.

Operator Precedence

Typical operations in an expression are

- Arithmetic operations such as +, -, *, /, and %
- Equality, relational, and logic operations !, <, <=, >, >=, ==, !=, &&, and || where the result is either a 1 (true) or a 0 (false)
- User-defined operations such as functions

SystemC Compiler evaluates expressions in the same precedence and order of evaluation as C++. Table F-1 shows the C++ operator precedence from highest to lowest; the nonsynthesizable operators are excluded from this list.

Table F-1 Operator Precedence

Operator	Function	Use
::	Class scope	class::name
.	Member selectors	object.member
[ ]	Subscript	variable[expr]
( )	Function all	name(expr_list)
++	Postfix increment	lvalue++
--	Postfix decrement	lvalue--
typeid	Type identification	typeid(type)
const_cast	Type conversion	const_cast<type>(expr)
static_cast	Type conversion	static_cast<type>(expr)
++	Prefix increment	++lvalue
--	Prefix decrement	--lvalue
~	Bitwise NOT	~expr
!	Logical NOT	!expr
-	Unary minus	-expr
+	Unary plus	+expr
&	Address of	&expr (parameter passing only)
( )	Type conversion	(type) expr
*	Multiply	expr * expr
/	Divide	expr / expr
%	Modulo (remainder)	expr % expr

Table F-1 Operator Precedence (continued)

Operator	Function	Use
+	Add	expr + expr
-	Subtract	expr - expr
<<	Bitwise shift left	expr << expr
>>	Bitwise shift right	expr >> expr
<	Less than	expr < expr
<=	Less than or equal	expr <= expr
>	Greater than	expr > expr
>=	Greater than or equal	expr >= expr
==	Equality	expr == expr
!=	Inequality	expr != expr
&	Bitwise AND	expr & expr
^	Bitwise XOR	expr ^ expr
|	Bitwise OR	expr | expr
&&	Logical AND	expr && expr
||	Logical OR	expr || expr
=	Assignment	lvalue = expr
=, *=, /=, %=, +=, -=, <<=, >>=, &=, |=, ^=	Compound assignment	lvalue += expr, and similar for each operator
?:	Conditional expression	expr ? expr : expr
,	Comma	expr, expr

Glossary

abstract data type
An abstract data type is a data type, such as a floating-point number, that does not readily translate to hardware.

aggregate data type
An aggregate data type contains multiple data types that are grouped together in a C/C++ structure (struct).

allocation
Allocation means assignment of hardware resources such as components, memory, and registers to scheduled operations and variables.

behavioral synthesis
Behavioral synthesis is the process of transforming a behavioral description at the unclocked algorithmic level with few or no implementation details into a clocked netlist of components. Behavioral synthesis automatically schedules the operations in the behavioral description into clock cycles, allocates hardware to execute them, and generates a state machine representing the control logic.

chained operation

A chained operation is two or more data-dependent operations scheduled in the same clock period without the need to register the intermediate results.

clock cycle

A clock cycle represents one clock period.

compiler directive

A compiler directive is a user-specified directive to SystemC Compiler that is placed in the source code as a comment.

constraint

Constraint are user-specified parameters such as the clock period, I/O timing, number and type of data path elements, and desired number of clock cycles.

control

The control portion of the design represents the FSM or control structure, which is implied from the conditional constructs and loops.

cycle-accurate model

A cycle-accurate model of a design is an abstract model that represents the cycle-to-cycle behavior of a design. It is not necessarily the exact structure of the hardware that implements the design.

data flow graph (DFG)

A DFG depicts the data dependencies, the inputs and outputs of a design, the operations used in the design, and the flow of data from the inputs to the outputs.

data path

A data path is the portion of the design that operates on data that is flowing into the design. Typically, the data path is controlled by the control portion of the design or FSM.

enumerated data type
An enumerated data type is an abstract type with a discrete set of values. When an enumerated type is synthesized, a unique bit pattern is assigned to each possible value of the enumerated type.

high-level synthesis
High-level synthesis (HLS) is synthesis from a behavioral description of the design into a clocked netlist of components. See behavioral synthesis.

latency (delay)
Latency means the number of clock cycles for performing a calculation.

lifetime analysis
Lifetime analysis is the process for determining how many clock cycles need to be reserved for resources or registers to execute a particular operation, or how many clock cycles to hold the value of a particular variable.

memory inferencing
Memory inferencing is a synthesis technique implementing an array in the behavioral description to a memory component, keeping the memory technology independent from design development.

multicycle operation
Multicycle operation is a combinational operation that requires more than one clock cycle to execute.

nonabstract data type
A nonabstract data type is a data type that can be easily translated into hardware.

operation
An operation is an instance of an operator in a design.

operator
An operator is an abstract representation of a design function, such as + for addition.

pipelined loop
Loop pipelining is a synthesis technique for partially overlapping loop iterations at circuit runtime to improve design performance.

pipelined component
A pipelined component is a component that executes an operation over several clock cycles. It differs from a multicycle component in that it is sequential. The internal registers break the logic that implements the operation into multiple stages of combinational logic. Each stage executes within a clock cycle. The output of a stage is stored in a register and passed onto the next stage at the next clock cycle.

preserved function
A preserved function is a function that is preserved as a level of hierarchy in synthesis. A preserved function is pre-compiled into a logic netlist prior to behavioral synthesis. Each call to the function is treated as a single operation by behavioral synthesis. It is scheduled and has hardware allocated for it.

register sharing
Register sharing means variables with sharing non-overlapping lifetimes can share the same register.

resource allocation
Resource allocation is the process for deciding how many and what kind of resources are used or needed for a given design.

resource sharing
Resource sharing is a synthesis optimization technique that allows multiple operations to be executed on the same resource.

RTL
RTL is an acronym for *register-transfer level*.

RTL synthesis
RTL synthesis, also known as logic synthesis, is the process of transforming an RTL description into a gate-level, technology-specific netlist.

scheduling
Scheduling is the synthesis process of assigning each operation to a control step.

superstate
A superstate represents one or more clock cycles for the `schedule` command in the superstate_fixed scheduling mode.

unrolled loop
Loop unrolling means the code body for each loop iteration is replicated as many times as there are iterations.

wait statement
A wait statement causes a wait for the next active clock edge, which defines a clock cycle in cycle-fixed scheduling mode or the boundary of a superstate in the superstate-fixed scheduling mode.

Index

Symbols

#elif compiler directive A-9
#else compiler directive A-9
#endif compiler directive A-9
#if compiler directive A-9
#ifdef compiler directive 2-31, A-9
#ifndef compiler directive A-9
? operator 3-32

A

abstraction level
  architectural 1-3, 1-4
  behavioral 1-3, 1-9
  choosing for synthesis 1-19
  RTL 1-3, 1-14
access
  memory 5-15
  memory bit slice 5-19
  multiple arrays 5-13
  register file 5-15
aggregate data type 2-43
architectural
  FIFO example B-2
  model 1-4
architecture refinement 2-8
arithmetic operation F-3
array
  assigning 5-5
  declaring 5-2
  implementation 5-7
  large 5-7
  mapping 5-1
    memory 5-11
    register file 5-9, A-6
  reading 5-3
  writing 5-3
atomic block 2-9
attribute
  map_to_module A-6
  map_to_registerfiles 5-9, A-6

B

bc_check_design command 3-31
bc_use_registerfiles variable 5-9
behavioral
  coding style 1-10, 3-1
  design attributes 1-19
  FFT example D-9
  FIFO example B-6
  IQ example E-4
  memory controller
    example C-9
  model 1-9

refine
architectural model 1-10
for synthesis 2-1
block
atomic 2-9
hierarchical 2-9

C

C line label 3-34, A-9
C/C++
compiler directives A-9
data types 2-43
language elements 2-1
nonsynthesizable constructs 2-34
refine model 2-6
synthesizable subset 2-32
case 3-32
clock 3-2, 3-7
clocked thread
example 3-4
process 2-26
code
simulation-specific 2-30
synthesis-specific 2-30
coding rules 3-10
cycle-fixed 3-12
cycle-fixed examples 3-23
finding timing errors 3-31
general examples 3-13
pipelined loop 3-45
superstate-fixed 3-12
superstate-fixed examples 3-29
terms 3-10
coding rules, general 3-11
coding style
behavioral 1-10, 3-1
RTL 1-15
command
bc_check_design 3-31
compile_preserve_functions 4-8
compile_systemc 3-14, 5-9
ignore_array_precedences 5-15
read_preserve_function_netlist 4-8
schedule 3-8, 3-26, 3-31
set_behavioral_reset 3-48
set_cycles 6-19
set_memory_input_delay 5-17
set_memory_output_delay 5-17
compare
design attributes 1-22
I/O schedule modes 3-9
compile_preserve_functions command 4-8
compile_systemc command 3-14, 5-9
compiler directive 2-31, A-2
#elif, #else, #endif A-9
#if, #ifdef, #ifndef A-9
#ifdef C language 2-31
C/C++ A-9
inout_param 4-10, A-5
line_label 3-34, A-3
map_to_operator 4-11, A-3
preserve_function 4-6, A-4
resource 5-9, A-6
return_port_name 4-11, A-4
synthesis_off 2-31, A-7
synthesis_on 2-31, A-7
translate_off A-7
translate_on A-7
unroll 3-37, A-8
components, DesignWare 4-11
conditional statements 3-32
constrain cycles
handshake 6-19
constructor 2-24
control refinement 2-3, 2-47
cycle-fixed
coding rules 3-12
schedule 3-8

D

data
  aggregate type 2-43
  C/C++ types 2-43
  enumerated type 2-43
  lifetime of value 3-54
  nonsynthesizable types 2-38
  recommended types 2-46
  resource sharing 3-53
  sc_bigint 2-42
  sc_biguint 2-42
  sc_bit 2-41
  sc_bv 2-41
  sc_int 2-42
  sc_uint 2-42
  synthesizable types 2-37, 2-39
  SystemC
    bit types 2-41
    integer types 2-42
  variable 2-18
data refinement 2-3, 2-30, 2-37
  C/C++ 2-32
  SystemC 2-32
define process 2-20
design
  behavioral attributes 1-19
  compare attributes 1-22
  RTL attributes 1-19, 1-21
DesignWare components 4-11
do-while loop 3-33, 3-36

E

else 3-32
enumerated data type 2-43
equality operation F-3
example
  architectural
    FIFO 1-5, B-2
  behavioral
    FFT D-9
    FIFO 1-11, B-6
    IQ E-4
    memory controller C-9
  clocked thread 3-4
  cycle-fixed coding rules 3-23
  fast handshake 6-36
  FFT D-1
  FIFO B-1
  functional
    FFT D-4
    memory controller C-5
  general coding rules 3-13
  IQ E-1
  local memory 5-11
  memory controller C-1
  one-way handshake 6-4, 6-12
  RTL
    FIFO 1-15, B-15
  superstate-fixed coding rules 3-29
  testbench
    FIFO B-11, B-20
  two-way handshake 6-21, 6-29
expression F-2

F

fast handshake 6-36
FFT
  behavioral example D-9
  example D-1
  functional example D-4
  testbench example D-17
FIFO
  architectural example 1-5, B-2
  behavioral example 1-11, B-6
  example B-1
  RTL example 1-15, B-15
  testbench example B-11
for loop 3-33, 3-36
  unrolled 3-37
function
  member 2-23, 4-2

nonmember 4-4
preserve 4-4
process 2-21
using 4-1
functional model 1-4
timed 1-5
untimed 1-5

G

global reset 3-46

H

handshake
constrain cycles 6-19
fast 6-36
loop pipelining 6-40
one-way protocol 6-4
protocols 6-1, 6-3
signals 6-3
two-way protocol 6-21
hardware
allocation F-3
behavioral synthesis process 2-21
header file, module 2-13
hierarchical block 2-9

I

I/O
read and write 3-7
schedule mode 3-8
specify 3-7
if 3-32
ignore_array_precedences command 5-15
implementation file
module 2-26
infinite loop 2-26
do-while 3-4
for 3-4
SC_CTHREAD 2-26
types 3-4
while 3-4, 3-35
initialize variable 3-49
inout_param compiler directive 4-10, A-5
input 3-6
nonregistered 3-6
read 3-7
interrupt behavior 3-46
introduction 1-1
introduction to refinement 2-3
IQ
behavioral example E-4
example E-1

L

label
C line label 3-34, A-9
source code 3-34
large array 5-7
lifetime
of data value 3-54
line_label compiler directive 3-34, A-3
local memory
declaring 5-11
example 5-11
logic operation F-3
loop
conditional 3-10
continue 3-10
do-while 3-33, 3-36
for 3-33, 3-36
for unrolled 3-37
infinite while 2-26, 3-33, 3-35
iteration 3-10
label 3-34
pipelining 3-45
pipelining handshake protocol 6-40
unroll 3-37, A-8
while 2-26, 3-33, 3-35

C/C++ constructs 2-34
data types 2-38
subset 2-30
SystemC constructs 2-33

M

map_to_module attribute A-6
map_to_operator compiler directive 4-11, A-3
map_to_registerfiles attribute 5-9, A-6
mapping arrays 5-1
member
  function 2-23, 4-2
  variables 2-18
memory 5-1
  access 5-15
  access bit slice 5-19
  access redundancy 5-18
  array mapping 5-11
  contention 5-18
  explore types 5-14
  local 5-11
  multiple array access 5-13
  resources 5-11
  timing 5-17
memory controller
  example C-1
    behavioral C-9
    functional C-5
method process 2-21
module 2-12
  constructor 2-24
  header file 2-13
  implementation file 2-26
  port 2-14
  signal 2-16
  syntax 2-13
  variable 2-18

N

nonmember
  function 4-4
  preserve_function 4-9
nonregistered inputs 3-6
nonsynthesizable

O

one-way handshake 6-4
  example 6-4, 6-12
operand F-2
operation F-3
  arithmetic F-3
  equality F-3
  logic F-3
  relational F-3
  user-defined F-3
operator F-2
  precedence F-3
  SystemC
    bit types 2-41
    integer types 2-42
output 3-6
  registered 3-6
  write 3-7

P

pipelining
  handshake restrictions 6-40
  loops 3-45
port 2-14, 3-6
  data types 2-15
  read and write 2-17
  sc_in 2-14
  sc_in_clk 2-14
  sc_inout 2-14
  sc_out 2-14
  syntax 2-15
pragma
  See compiler directive
precedence of operators F-3

preserve_function
  compiler directive 4-4, 4-6, A-4
  nonmember 4-9
  restrictions 4-5
  using 4-5
process 2-12, 2-20
  creating 2-22
  method 2-21
  refine 2-28
  SC_CTHREAD 2-21, 3-2
  SC_THREAD 2-21
  synchronizing with clock 3-2

R

read
  array 5-3
  input 3-7
  port 2-17
  signal 2-17
read_preserve_function_netlist command 4-8
reducing runtime 5-7
refine
  advanced techniques 2-5, 2-48
  behavioral from architectural 1-10
  C/C++ model 2-6
  control 2-3, 2-47
  data 2-3, 2-30, 2-37
  data type recommendation 2-46
  detailed architecture 2-8
  for behavioral synthesis 2-1
  internal communication 2-7
  internal structure 2-6
  overview 2-3
  process 2-28
  recommended practices 2-49
  RTL from behavioral 1-15, B-15
  structure 2-3, 2-6
  SystemC model 2-28
register file 5-1
  access 5-15
  array mapping 5-9, A-6
registered output 3-6
relational operation F-3
reset behavior 3-46
resource compiler directive 5-9, A-6
resource sharing 3-53
restrictions
  pipeline handshake 6-40
return_port_name compiler directive 4-11, A-4
rolled and unrolled loops 3-37, A-8
RTL
  coding style 1-15
  design attributes 1-19, 1-21
  model 1-14
  synthesis process 2-21

S

sc_bigint 2-42
sc_biguint 2-42
sc_bit 2-41
sc_bv 2-41
SC_CTHREAD 2-21, 3-2
  infinite loop 2-26
SC_CTHREAD process 2-21
SC_CTOR 2-24
sc_in 2-14
sc_in_clk 2-14
sc_inout port 2-14
sc_int 2-42
SC_METHOD 2-21
SC_MODULE 2-12
SC_MODULE syntax 2-13
sc_out 2-14
SC_THREAD process 2-21
sc_uint 2-42
schedule
  command 3-8
  compare I/O modes 3-9

cycle-fixed 3-8
I/O mode 3-8
superstate-fixed 3-8
schedule command 3-26, 3-31
set_behavioral_reset command 3-48
set_cycles command 6-19
set_memory_input_delay command 5-17
set_memory_output_delay command 5-17
signal 3-6, 3-49
data types 2-17
internal 2-16
read and write 2-17
syntax 2-16
wait statement 3-50
writing to 5-5
simulation-specific code 2-30
snps compiler directive 2-31, A-2
struct GL-1
structure GL-1
structure refinement 2-3, 2-6
superstate-fixed
coding rules 3-12
schedule 3-8
switch 3-32
synopsys compiler directive 2-31, A-2
syntax
module 2-13
port 2-15
signal 2-16
synthesis
choosing abstraction level 1-19
synthesis_off compiler directive 2-31, A-7
synthesis_on compiler directive 2-31, A-7
synthesis-specific code 2-30
synthesizable
data types 2-37, 2-39
subset 2-30
SystemC
bit data types 2-41
bit operator types 2-41
class library 1-1
integer data types 2-42
integer operator types 2-42
language elements 2-1
nonsynthesizable constructs 2-33
refine model 2-28
synthesizable subset 2-32

T

testbench
FFT D-17
FIFO example B-20
handshake 6-1
process types 2-21
timed model 1-5
timing errors 3-31
timing of memories 5-17
translate_off compiler directive A-7
translate_on compiler directive A-7
two-way handshake
example 6-21, 6-29

U

unroll compiler directive 3-37, A-8
untimed model 1-5
user-defined operation F-3

V

variable 3-49
bc_use_registerfiles 5-9
guidelines for using 3-49
initializing 3-49
mapping to register 3-53
member 2-18
module 2-18

W

wait statement 3-2, 3-3, 3-7
wait_until statement 3-2, 3-3
watching 3-46
while loop 2-26, 3-33, 3-35
write
  array 5-3
  output 3-7
  port 2-17
  signal 2-17