

CoCentric™

SystemC Compiler

Behavioral User 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, CBAll, 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: 37444-000 JB
CoCentric™ SystemC Compiler Behavioral User Guide, v2000.11-SCC1

Contents

What's New in This Release	xxviii
About This Guide	xxx
Customer Support	xxxiii
1. Introduction to SystemC Compiler Behavioral Synthesis	
Understanding What SystemC Compiler Does	1-3
Synthesis With SystemC Compiler	1-4
Timing	1-6
Scheduling	1-7
Allocating Hardware	1-9
Creating an FSM and Data Path	1-10
Pipelining Loops	1-11
Inferring Memories	1-11
Libraries and Other Inputs	1-12
Behavioral Description	1-13
Technology Library	1-13
Synthetic Library	1-14

Outputs From SystemC Compiler	1-14
2. Using SystemC Compiler	
Usage and Commands	2-3
Defining Libraries	2-5
Compiling and Elaborating the Source Code	2-5
Preparing to Use BCView	2-5
Using the compile_systemc Command	2-6
Elaborating a Design With a Single Behavioral Module.	2-7
Elaborating a Hierarchical Design With Multiple Behavioral Modules	2-7
Elaborating a Design With Multiple Files	2-8
Assigning Timing and Area Design Constraints	2-8
Setting the Clock Period	2-8
Setting Other Initial Constraints	2-9
Checking the Design	2-10
Running Check Design	2-10
Changing the Code	2-10
Estimating Time and Area	2-11
Reporting Timing and Area Estimates	2-12
Saving the Timed Design.	2-12
Scheduling the Design and Allocating Resources.	2-13
Scheduling for Smallest Area.	2-13
Changing the Effort Level	2-14
Setting Schedule Constraints	2-14

Using BCView to Analyze Scheduling Errors.	2-15
Analyzing Scheduling Results	2-15
Generating Summary Reports.	2-16
Removing Designs from SystemC Compiler Memory	2-16
Resuming Synthesis From a Saved .db File	2-17
Writing the RTL Files	2-18
Writing the RTL .db File	2-19
Writing a Synthesizable RTL HDL File.	2-20
Writing an RTL Simulation File	2-21
Specifying VHDL Packages.	2-23
Specifying Verilog Include Files	2-23
Compiling and Writing a Gate-Level Netlist.	2-24
Preparing for Place and Route	2-24
Preparing for Physical Compiler	2-25
Preparing RTL for Physical Synthesis	2-25
Preparing Gate-Level for Physical Synthesis.	2-26
3. Timing and Area Estimation	
Understanding Clock Cycle, I/O, and Operation Relationships.	3-2
Operation Delay and Clock Cycle	3-2
I/O Protocol	3-3
Operations and Clock Cycles.	3-5
Setting Your Timing Environment	3-7
Setting Clocks	3-7
Setting Input Delays.	3-8

Setting Operating Conditions	3-9
Listing Libraries	3-9
Listing Operating Conditions	3-10
Setting Wire Loads	3-12
Timing the Design	3-13
Timing Through the Components	3-13
Computing the Clock Cycle Margin	3-14
Interpreting the Timing and Area Resource Report.	3-17
Evaluating the Resource Estimate Report.	3-17
Looking at Parallel Paths	3-19
Area Estimates	3-21
4. Scheduling and Scheduling Constraints	
Scheduling for Synthesis.	4-2
Operation Scheduling	4-3
Resource Sharing	4-4
Inferred Registers	4-5
Register Sharing	4-6
Controller (FSM) Generation	4-7
Controlling Synthesis	4-10
Selecting an I/O Scheduling Mode	4-10
Cycle-Fixed I/O Scheduling Mode	4-12
Using Cycle-Fixed I/O Scheduling Mode	4-12
Superstate-Fixed I/O Scheduling Mode	4-14
Using Superstate-Fixed I/O Scheduling Mode.	4-15
Comparing the I/O Scheduling Modes.	4-16

Performing Scheduling	4-18
Scheduling Objectives	4-18
Using Timing-Constrained Scheduling.	4-18
Using Resource-Driven Scheduling.	4-19
Analyzing the Scheduling Report	4-20
Schedule Summary Report	4-20
Schedule Report of Operations	4-22
Schedule Report of Variables	4-26
Schedule Report of the FSM	4-28
Adding Scheduling Constraints	4-31
Matching Cells to Operations and Loops.	4-31
Naming Conventions	4-31
Using Line Labels	4-32
Using Find	4-33
Reporting Hierarchy.	4-34
Constraining Loops and Operations	4-37
Constraining Between Two Operations	4-37
Constraining a Loop	4-38
Constraining Nested Loops	4-39
Placing Constraints Across Loop Boundaries	4-41
Using the Set Cycles Commands and Options	4-42
Pipelining a Loop	4-44
Restrictions and Limitations For Pipelining Loops	4-47
Determining the Initiation Interval.	4-47
Pipelining a Loop With Handshake Signals	4-50
Determining Current Scheduling Constraints.	4-53

Removing Scheduling Constraints	4-54
Constraining Resource Allocations	4-55
Setting Common Resources	4-55
Setting Exclusive Registers	4-57
5. Optimizing Latency and Area	
Exploring Architectures and Improving the Quality of Results	5-2
Looking at Architectural Tradeoffs	5-2
Architectural Exploration Guidelines	5-6
Controlling Operation and Implementation Selection	5-7
Operation Chaining	5-8
Operation Chaining With Bitwise Timing	5-8
Determining Operation Chaining	5-10
Controlling Operation Chaining	5-11
Controlling Margin Calculation	5-12
Removing Unnecessary Registers	5-18
Using Multicycle Operations	5-19
Reporting Multicycle Operations	5-20
Increased Latency of Multicycle Operations	5-21
Replacing Multicycle Components	5-23
Using Preserved Functions	5-23
When to Preserve Functions	5-24
Determining Which Functions to Preserve	5-24
Creating Preserved Functions	5-25
Preserving a Function	5-27

Using a Precompiled Netlist for a Preserved Function	5-28
Compiling Preserved Functions	5-29
Using Preserved Functions for Behavioral Synthesis	5-31
Limitations of Preserved Functions	5-33
Bit-Width Restrictions	5-33
Hierarchy	5-33
Sequential Logic	5-34
Using DesignWare Components	5-34
Listing DesignWare Components	5-35
Finding and Implementing Pipelined Components	5-37
6. Analyzing Designs With BCView	
Using BCView	6-2
Preparing Designs for BCView	6-2
Starting BCView	6-3
Removing BCView Analysis Information	6-3
Using BCView Windows	6-3
Recommended Usage for BCView	6-8
Examining Scheduling Errors	6-9
Identifying Errors to Analyze	6-9
Using the Scheduling Error Analyzer	6-10
Viewing the Selection Inspector Window	6-11
Determining the Operations That Bound the Error	6-12
Examining the Graphic Information	6-13
Fixing the Code and Rescheduling	6-21
Evaluating the Architecture Generated by SystemC Compiler	6-21

Reviewing FSM Operation	6-22
Stepping Through the FSM	6-23
Reviewing State Transitions and Actions	6-24
Evaluating the Scheduled Design	6-26
Understanding the Reservation Table Window	6-26
Viewing Resources, Latencies, and Operation Sharing	6-31
Viewing Clocks, Chaining, and Combinational Delay	6-36
Examining Paths	6-38
Reviewing Register Use	6-42
Viewing Loops	6-44
Identifying Constraints and Data Dependencies	6-49
Exploring Architectural Improvements	6-51
Reducing Latency	6-51
Identifying Multicycle Operations	6-51
Identifying Chaining Opportunities	6-53
Viewing Clock-Cycle Utilization	6-54
Reducing Area	6-55
Reviewing Critical Paths	6-61
Viewing the Design Summary	6-61
7. Using Register Files and Memories for Arrays	
Comparing Array Implementations	7-2
Comparing Arrays, Register Files, and Memories	7-3
Array Implementation Recommendations	7-6
Mapping Arrays to Register Files	7-6
Mapping All Arrays to Register Files	7-6
Mapping Specific Arrays to Register Files	7-7

Understanding the Effects of Mapping to Register Files	7-8
Reporting Array Access Conflicts	7-8
Allowing Multiple Accesses in the Same Cycle	7-10
Identifying Register File Operations.	7-13
Finding Array Operation Cells	7-14
Mapping Arrays to Memory	7-15
Preparing to Use Memories	7-15
Using Memory in Your Design	7-17
Getting Memory and Library Information	7-18
Using Asynchronous Memories	7-24
Allowing for Vendor Memory Timing	7-25
Setting Memory Input Delay for Vendor Memory Timing . . .	7-26
Setting Memory Output Delay for the Vendor Timing Specifications	7-27
Constraining Read and Write Operations on Memory	7-28
Reporting Conflicting Memory Accesses	7-29
Using the ignore_memory_precedences Command	7-31
Using the ignore_memory_loop_precedences Command . .	7-32
Generating Memory Wrappers	7-34
Understanding the Memory Wrapper Generator Tool	7-34
Using the Memory Wrapper Generator Tool	7-35
Creating a Memory Wrapper for a Vendor Memory	7-39
Defining the Memory Type and Properties.	7-40
Assigning Memory Pins to the Wrapper Logical Ports	7-46
Defining the Memory Wrapper Properties	7-52
Reviewing the Memory Wrapper	7-56
Editing the Waveform Values	7-56

Adding Registers to the Memory Wrapper	7-58
Adding Custom Logic to the Memory Wrapper	7-59
Viewing and Editing the Wrapper Properties	7-61
Saving the Memory Wrapper Files	7-64
Using Generated Vendor Memory Wrappers	
With SystemC Compiler	7-66
Creating a Memory Wrapper for an Exploratory Memory	7-67
Defining the Memory Type and Properties	7-68
Assigning Pins to the Memory Logical Ports	7-73
Defining the Exploratory Memory Wrapper Properties	7-75
Reviewing and Editing the Exploratory Memory Wrapper	7-78
Saving the Exploratory Memory Wrapper Files	7-78
Generating a Memory Wrapper Testbench	7-79
8. Advanced Techniques	
Using Multiple Files to Describe a Design	8-2
Using #include	8-2
Using Precompiled Netlists	8-2
Speculative Execution	8-4
Setting a Specific Implementation for Components	8-6
Externalize a Cell	8-8
Appendix A. Setting Up SystemC Compiler	
Defining Environment Variables and Paths	A-2
Defining Libraries and Other Variables	A-3
Starting the SystemC Compiler Command Interface	A-4

Creating a command.log File	A-4
Recording Your Command Session	A-5
Issuing SystemC Compiler Commands	A-5
Listing SystemC Compiler Variables	A-6
Using Scripts	A-6
Creating Scripts	A-6
Script Example	A-7
Using the Script	A-8
Using UNIX Shell Commands	A-9
Using compile_systemc Command Preprocessor Options	A-9
Starting BCView	A-11
Starting BCView From dc_shell	A-11
Starting BCView From a UNIX Shell	A-12
Using BCView in Your Script	A-12
Opening BCView Windows	A-13
Starting the Memory Wrapper Tool	A-14
Getting Command, Variable, and Error Help	A-15
System Prompt	A-15
SystemC Compiler Command Prompt	A-15
 Appendix B. Complex Number Multiplier Example Files	
Complex Number Multiplier Source Code	B-2
Command Script	B-4
Reports Created During Synthesis	B-6

Estimated Resources	B-6
Schedule Report	B-9
Area Report	B-11
Timing Report	B-12
Report Resource	B-14

Figures

Figure 1-1	Behavioral Synthesis Compared to RTL Synthesis	1-3
Figure 1-2	Structure of the Circuit Generated by SystemC Compiler During Behavioral Synthesis.	1-5
Figure 1-3	Scheduling Into Specific Clock Cycles	1-7
Figure 1-4	Allocation of Resources	1-9
Figure 1-5	An Algorithm and the Created Data Path and FSM	1-10
Figure 1-6	SystemC Compiler Input and Output Flow	1-12
Figure 2-1	SystemC Compiler Commands Use in the Flow.	2-4
Figure 3-1	Timing Diagram of the Complex Multiplier I/O Protocol.	3-4
Figure 3-2	Operations of the Complex Multiplier	3-6
Figure 3-3	Typical Timing Path.	3-15
Figure 3-4	Estimated Resources Report (Partial)	3-18
Figure 3-5	Parallel Paths in the Estimated Resources Report (Partial)	3-20
Figure 4-1	Scheduling Into Specific Clock Cycles	4-2

Figure 4-2	Operation Scheduling	4-3
Figure 4-3	Resource Allocation Reservation Table	4-4
Figure 4-4	Register Allocation Reservation Table	4-7
Figure 4-5	Shared Component.	4-8
Figure 4-6	Shared Register	4-8
Figure 4-7	FSM Control Signals.	4-9
Figure 4-8	Synthesized Design Representation.	4-9
Figure 4-9	Behavioral Code and I/O Operation	4-11
Figure 4-10	Cycle-Fixed I/O Mode.	4-12
Figure 4-11	Superstate-Fixed I/O Mode.	4-15
Figure 4-12	Source Code and I/O Scheduling Mode Simulation	4-17
Figure 4-13	Resources With Loops	4-40
Figure 4-14	Nonpipelined Loop	4-44
Figure 4-15	Pipelined Loop	4-45
Figure 4-16	Invalid Loop Initiation Value	4-48
Figure 4-17	Valid Loop Initiation Value.	4-48
Figure 4-18	Invalid Memory and I/O Access	4-49
Figure 4-19	Valid Memory and I/O Access.	4-50
Figure 4-20	Handshake Signal Preventing Loop Pipelining.	4-50
Figure 4-21	Pipelined Loop With Handshake Signal	4-51
Figure 4-22	Exit From a Pipelined Loop.	4-52
Figure 5-1	Architectural Exploration.	5-3
Figure 5-2	Bitwise Timing for Operation Chaining	5-9
Figure 5-3	Chained Operations in the Estimated Resources	

	Report (Partial)	5-10
Figure 5-4	Typical Timing Path.	5-13
Figure 5-5	Chaining Operation Timing	5-15
Figure 5-6	Multicycle Operation	5-19
Figure 5-7	Multicycle Operations in Conditional Statements	5-22
Figure 5-8	Flow for Preserving Functions	5-27
Figure 5-9	Command Flow With Preserved Functions	5-32
Figure 6-1	BCView Windows	6-5
Figure 6-2	BCView Recommended Usage	6-8
Figure 6-3	Selection Inspector With Error Information	6-11
Figure 6-4	Scheduling Error Analyzer With Bounding Operations	6-12
Figure 6-5	Scheduling Error Analyzer Paths and Clock Cycles	6-13
Figure 6-6	Expanded Derived Edge.	6-17
Figure 6-7	Selection Inspector Window With Edge Information.	6-18
Figure 6-8	Code Browser With Behavioral Code	6-20
Figure 6-9	FSM Viewer With States and Transitions	6-22
Figure 6-10	Selected Transition With Conditions and Actions	6-25
Figure 6-11	Reservation Table Window	6-27
Figure 6-12	Reservation Table Toolbar Buttons	6-30
Figure 6-13	Resource Utilization in Reservation Table	6-32
Figure 6-14	Resource Delay in Reservation Table.	6-33
Figure 6-15	Operation Delay in Reservation Table	6-34
Figure 6-16	Operation Delay Detail in Selection Inspector	6-34
Figure 6-17	Shared Resources in Reservation Table	6-35

Figure 6-18	Operation Delays in Clock Cycles	6-37
Figure 6-19	Derived Edge Example	6-39
Figure 6-20	Registers in the Reservation Table	6-42
Figure 6-21	Loops in the Reservation Table	6-44
Figure 6-22	Loop Information Tips	6-46
Figure 6-23	Loop Details in Selection Inspector	6-47
Figure 6-24	Loop Operations Zoomed View	6-48
Figure 6-25	Clock Cycle Utilization	6-54
Figure 6-26	Little Resource Sharing	6-56
Figure 6-27	Shared Resources	6-57
Figure 6-28	Shareable Resources That Are Not Shared	6-58
Figure 6-29	Forced Resource Sharing.	6-60
Figure 6-30	Design Summary in Selection Inspector Window	6-62
Figure 7-1	Array Generation	7-2
Figure 7-2	Register File Architecture	7-3
Figure 7-3	Dual-Port Memory Operations	7-4
Figure 7-4	Multiple Accesses in the Same Cycle That May Conflict	7-12
Figure 7-5	Asynchronous Memory With Registered Input	7-24
Figure 7-6	Manually Adding Registers to an Asynchronous Memory.	7-25
Figure 7-7	Memory Access Time Specification	7-25
Figure 7-8	Pipelined Memory Accesses.	7-29
Figure 7-9	Invalid Schedule With Loop Carry Dependency	7-32

Figure 7-10	Empty Memory Wrapper Window	7-36
Figure 7-11	Completed Memory Wrapper	7-37
Figure 7-12	Memory Selection Dialog Box.	7-40
Figure 7-13	Memory Selection from a DB File Dialog Box.	7-41
Figure 7-14	Memory Definition Dialog Box	7-42
Figure 7-15	Completed Memory Definition	7-45
Figure 7-16	Memory Pin Definition Dialog Box	7-47
Figure 7-17	Completed Memory Pin Definition	7-49
Figure 7-18	Completed Wrapper Properties Dialog Box	7-51
Figure 7-19	Wrapper Summary	7-54
Figure 7-20	Memory Wrapper Displayed in Main Window.	7-55
Figure 7-21	Read Port Protocol Waveforms	7-57
Figure 7-22	Manually Adding Registers to an Asynchronous Memory.	7-59
Figure 7-23	Code Editor Dialog Box With Default Code	7-60
Figure 7-24	Properties Dialog Boxes	7-63
Figure 7-25	Export Wrapper Dialog Box	7-65
Figure 7-26	Exploratory Memory Selection Dialog Box	7-68
Figure 7-27	Exploratory Memory Definition Dialog Box	7-69
Figure 7-28	Completed Exploratory Memory Definition	7-71
Figure 7-29	Exploratory Memory Pin Definition Dialog Box.	7-72
Figure 7-30	Exploratory Wrapper Properties Dialog Box.	7-74
Figure 7-31	Exploratory Memory Wrapper Summary	7-76
Figure 7-32	Exploratory Memory Wrapper in Main Window	7-77

Figure 8-1 Externalize a Cell 8-8

Tables

Table 6-1	Edges Representing Constraints	6-14
Table 6-2	Reservation Table Symbols	6-28
Table 7-1	Comparing Arrays, Register Files, and Memories	7-5

Examples

- Example 3-1 Complex Multiplier I/O Protocol. 3-3
- Example 3-2 Complex Multiplier Arithmetic Operations. 3-5
- Example 3-3 Report Clock 3-7
- Example 3-4 Listing Libraries. 3-10
- Example 3-5 Library Report (Partial) 3-11
- Example 3-6 Wire Load Model (Partial) 3-12
- Example 3-7 Timing Report (Partial) 3-14
- Example 3-8 Clock Margin in the Resource Estimate Report 3-16
- Example 3-9 Estimated Resource Report 3-21
- Example 4-1 Schedule Report Summary. 4-21
- Example 4-2 Report Schedule Operations. 4-24
- Example 4-3 Report Schedule Variables 4-27
- Example 4-4 Report Schedule Abstract FSM 4-29
- Example 4-5 Report Hierarchy Before Scheduling. 4-35
- Example 4-6 Report Hierarchy After Scheduling 4-36
- Example 4-7 Constraining Between Two Operations. 4-38

Example 4-8	Constraining a Loop	4-38
Example 4-9	Nested Loops With Operations	4-39
Example 4-10	Passing a Constraint Between Loops	4-42
Example 4-11	Pipelined Loop Timing Summary (Partial).	4-46
Example 4-12	Commands for Minimum Resource-Driven Scheduling	4-56
Example 4-13	Commands for Maximum Resource-Driven Scheduling	4-56
Example 4-14	Commands for Forced Maximum Resource-Driven Scheduling	4-57
Example 5-1	Clock Margin in the Resource Estimate Report	5-14
Example 5-2	Multicycle Report (Partial).	5-21
Example 5-3	Creating a Preserved Function	5-26
Example 5-4	Defining a Preserved Function in a Separate File.	5-26
Example 5-5	Using the read_preserved_function_netlist Command	5-28
Example 5-6	Using the compile_preserved_functions Command	5-30
Example 5-7	Using DesignWare Components.	5-35
Example 5-8	Reporting DesignWare Components	5-36
Example 5-9	Listing Pipelined Components	5-38
Example 6-1	HLS-52 Error Message	6-10
Example 7-1	Defining a Register File for a Specific Array	7-7
Example 7-2	Report of Array Conflicts.	7-9
Example 7-3	Accesses That May or May Not Conflict.	7-11

Example 7-4	Declaring a Local Memory Resource	7-18
Example 7-5	Report of Synthetic Memory Wrapper.	7-19
Example 7-6	Report of a Synthetic Library	7-22
Example 7-7	Report of Memories Used in a Design	7-23
Example 7-8	Set Memory Input Delay	7-27
Example 7-9	Report Nonconflicting Memory Accesses	7-30
Example 7-10	Memory Array Definition	7-66
Example 8-1	Executing Without Speculative Execution.	8-5
Example 8-2	Executing With Speculative Execution	8-6
Example A-1	SystemC Compiler Command Script	A-7
Example A-2	Using BCView in a Script	A-12
Example B-1	Complex Multiplier Source Code.	B-2
Example B-2	Command Script for Complex Number Multiplier	B-4
Example B-3	Report Resource Estimates	B-6
Example B-4	Schedule Report	B-9
Example B-5	Report Area.	B-11
Example B-6	Report Timing	B-12
Example B-7	Report Resources.	B-14

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 “Writing the RTL Files” on page 2-18.

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 “Editing the Waveform Values” on page 7-56.

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 User Guide* explains how to synthesize a SystemC behavioral description of a hardware module into an RTL description or gate-level netlist using the CoCentric SystemC Compiler.

Audience

The *CoCentric™ SystemC Compiler Behavioral User 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 User Guide*, see the following manuals:

- The *CoCentric™ SystemC Compiler Behavioral Modeling Guide*, which provides information about how to develop or refine a SystemC behavioral model for synthesis with SystemC Compiler
- 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.

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.

Convention	Description
<code>Courier</code>	Indicates command syntax.
<i>Courier italic</i>	Indicates a user-defined value in Synopsys syntax, such as <i>object_name</i> . (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.)
Courier bold	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.)
[]	Denotes optional parameters, such as <i>pin1 [pin2 ... pinN]</i>
	Indicates a choice among alternatives, such as <i>low medium high</i> (This example indicates that you can enter one of three possible values for an option: low, medium, or high.)
—	Connects terms that are read as a single term by the system, such as <i>set_annotated_delay</i>
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

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 to SystemC Compiler Behavioral Synthesis

CoCentric™ SystemC Compiler synthesizes a SystemC behavioral hardware module into RTL or a gate-level netlist, and it can synthesize a SystemC RTL module into a gate-level netlist. After synthesis, you can use other Synopsys tools for test insertion, power optimization, and other tasks to complete the physical design.

For information about setting up your environment to use SystemC Compiler, see Appendix A, “Setting Up SystemC Compiler.”

This chapter describes the synthesis process, the inputs required by SystemC Compiler, and the outputs it produces in the following sections:

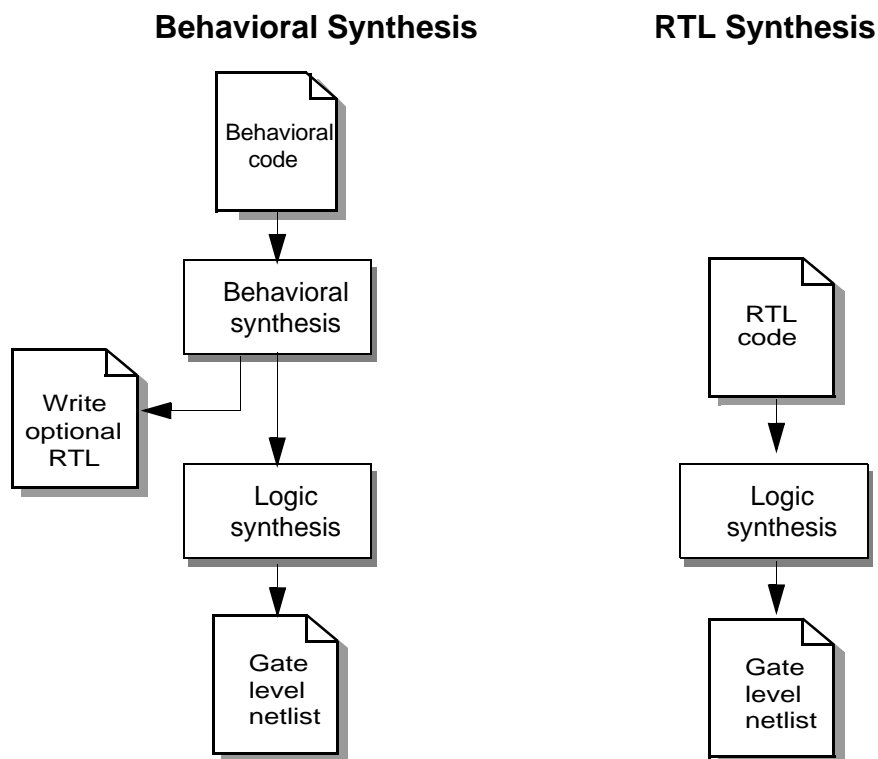
- Understanding What SystemC Compiler Does
- Libraries and Other Inputs

- Outputs From SystemC Compiler

Understanding What SystemC Compiler Does

SystemC Compiler is a tool that can accept both behavioral and RTL SystemC descriptions and performs behavioral or RTL synthesis, as required, to create a gate-level netlist. You can also use SystemC Compiler to create an HDL description for simulation or to use with other HDL tools in your flow. Figure 1-1 shows behavioral and RTL synthesis paths to gate-level netlists.

Figure 1-1 Behavioral Synthesis Compared to RTL Synthesis



Synthesis With SystemC Compiler

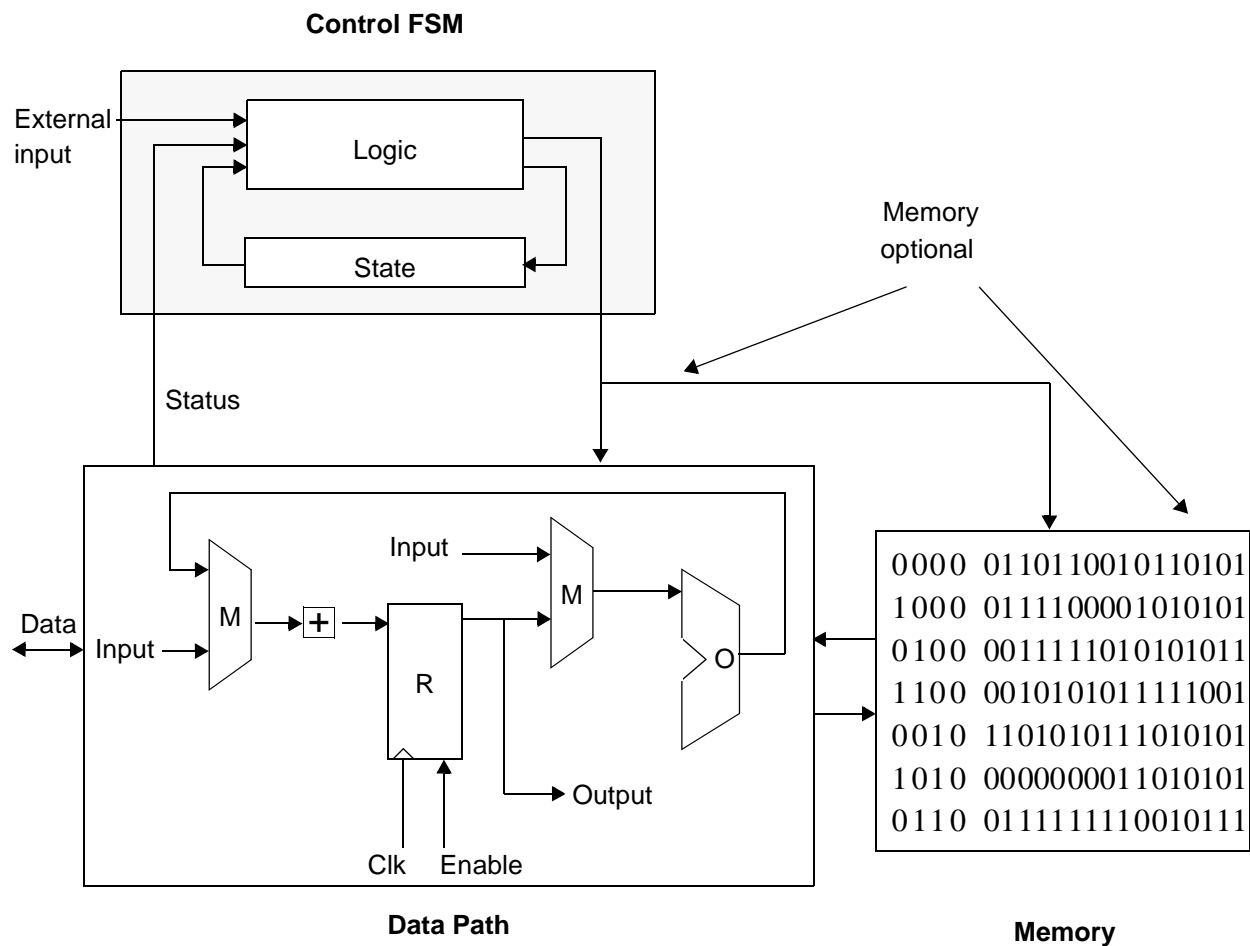
This guide explains how to do behavioral synthesis with SystemC Compiler. For information about doing RTL synthesis with SystemC Compiler, see the *CoCentric™ SystemC Compiler RTL User and Modeling Guide*.

SystemC Compiler synthesizes hardware from a behavioral description by

- Timing all operations, based on a technology library
- Scheduling operations, I/O, and memory accesses into clock cycles
- Allocating hardware by assigning variables and signals to registers and assigning operations to synthetic components
- Creating a finite state machine (FSM) and memory interface control logic
- Pipelining loops for higher throughput, which typically increases the size of the ASIC
- Inferring memory for arrays
- Chaining and multicycling operations

SystemC Compiler generates a design that consists of an FSM, a data path, and memory, as shown in Figure 1-2.

Figure 1-2 Structure of the Circuit Generated by SystemC Compiler During Behavioral Synthesis



Timing

During timing, SystemC Compiler determines the delay through each component that is used in the design shows the critical paths in the design. To accurately determine timing, it uses

- ASIC vendor libraries
- Wire load models
- Operating conditions
- DesignWare component libraries

The timing estimates that are created are used during scheduling and allocation to determine an appropriate architecture. Timing is described in more detail in “Timing and Area Estimation” in Chapter 3.

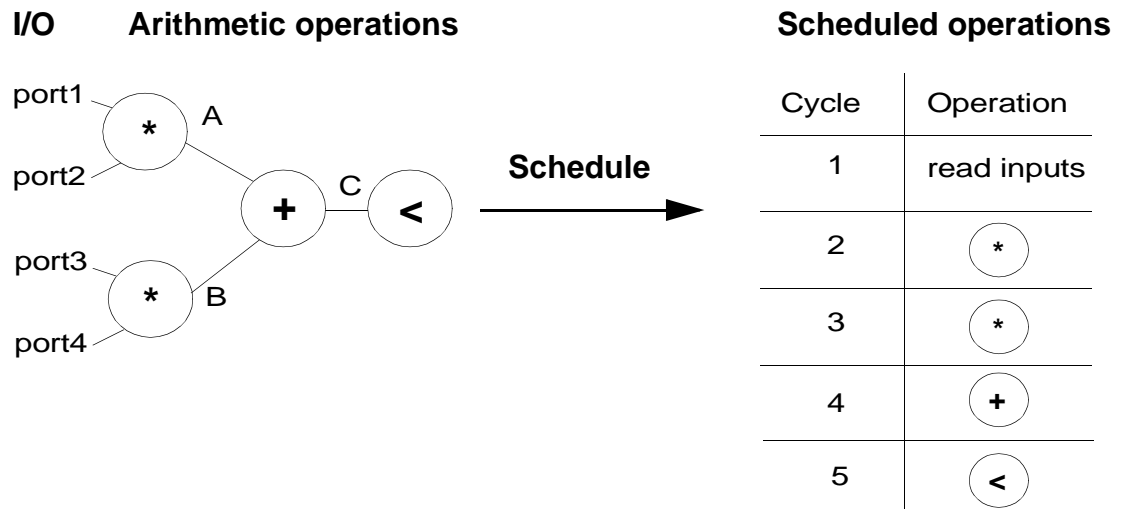
Scheduling

During scheduling, SystemC Compiler schedules I/O operations, arithmetic operations, and memory accesses into specific clock cycles, as shown in Figure 1-3.

Figure 1-3 Scheduling Into Specific Clock Cycles

Behavioral code

```
wait_until(start.delayed() == true);  
A = port1.read() * port2.read();  
B = port3.read() * port4.read();  
C = A + B;  
if (C < 0) {...}
```



SystemC Compiler objectives during scheduling are to

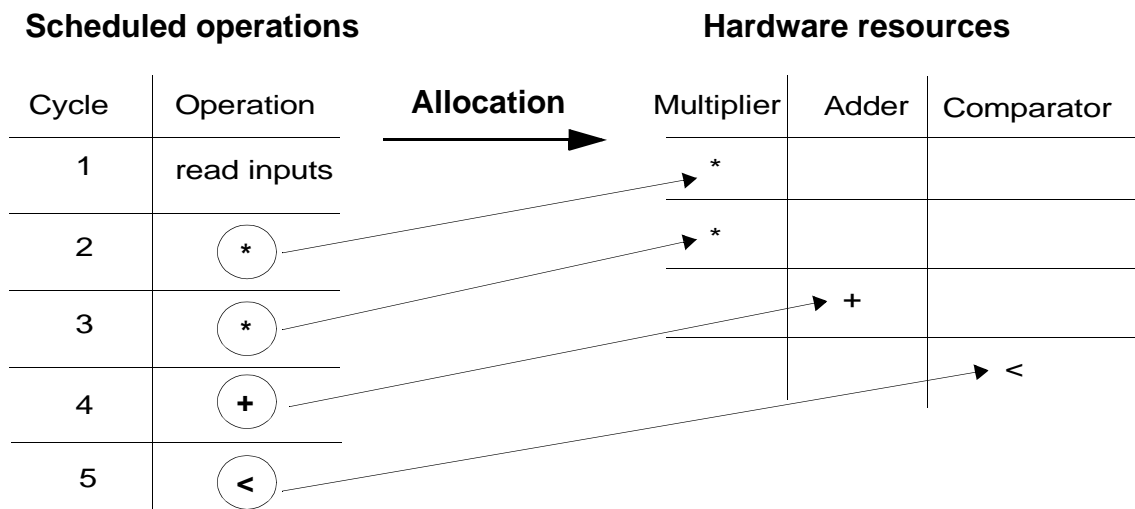
- Satisfy the data and control dependencies between operations
- Ensure that the scheduling constraints of latency, throughput, and clock period are met
- Facilitate maximum resource sharing by distributing operations over the allowed number of cycles
- Allow for maximum register sharing by producing and consuming variables intelligently

The scheduling mode defines how SystemC Compiler handles I/O operations. You control the scheduling mode to be either cycle-fixed or superstate-fixed. In cycle-fixed scheduling mode, the I/O operations are left in the exact clock cycle specified in the behavioral description. In superstate-fixed scheduling mode, SystemC Compiler preserves the relative order of I/O operations defined in the behavioral description, but it can insert clock cycles between I/O operations. Scheduling modes are described in Chapter 4, “Scheduling and Scheduling Constraints.”

Allocating Hardware

After the design is scheduled into clock cycles, data values are assigned to specific registers, and operations are allocated to specific hardware resources. To achieve the best overall hardware cost, SystemC Compiler calculates whether sharing a resource and adding multiplexers is more expensive than duplicating the resources during allocation. Figure 1-4 shows allocation of hardware for a set of constraints. The allocation uses one multiplier, one adder, and one comparator to execute the operations

Figure 1-4 Allocation of Resources



Creating an FSM and Data Path

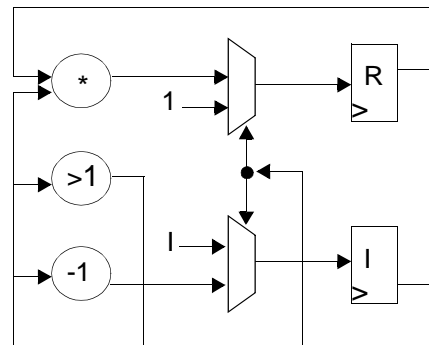
Data path operations are specified explicitly in the behavioral description, and FSM actions are implied from the control statements such as if, while, and loop statements. Figure 1-5 shows a simple algorithm and the data path and FSM that SystemC Compiler creates.

Figure 1-5 An Algorithm and the Created Data Path and FSM

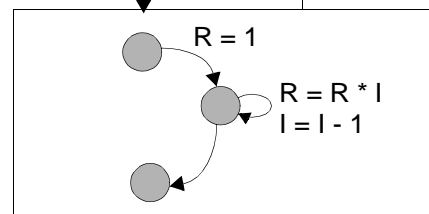
Algorithm

```
R = 1;
while (I > 1){
    R = R * I;
    I = I - 1;
}
```

Data Path



FSM



Pipelining Loops

You can increase the throughput of your design by pipelining loops. During scheduling, SystemC Compiler generates the required loop pipelining controls in the FSM. By pipelining loops, your design can execute more operation per time unit, however the resulting ASIC implementation is usually larger. Pipelining loops is described in “Pipelining a Loop” on page 4-44.

Inferring Memories

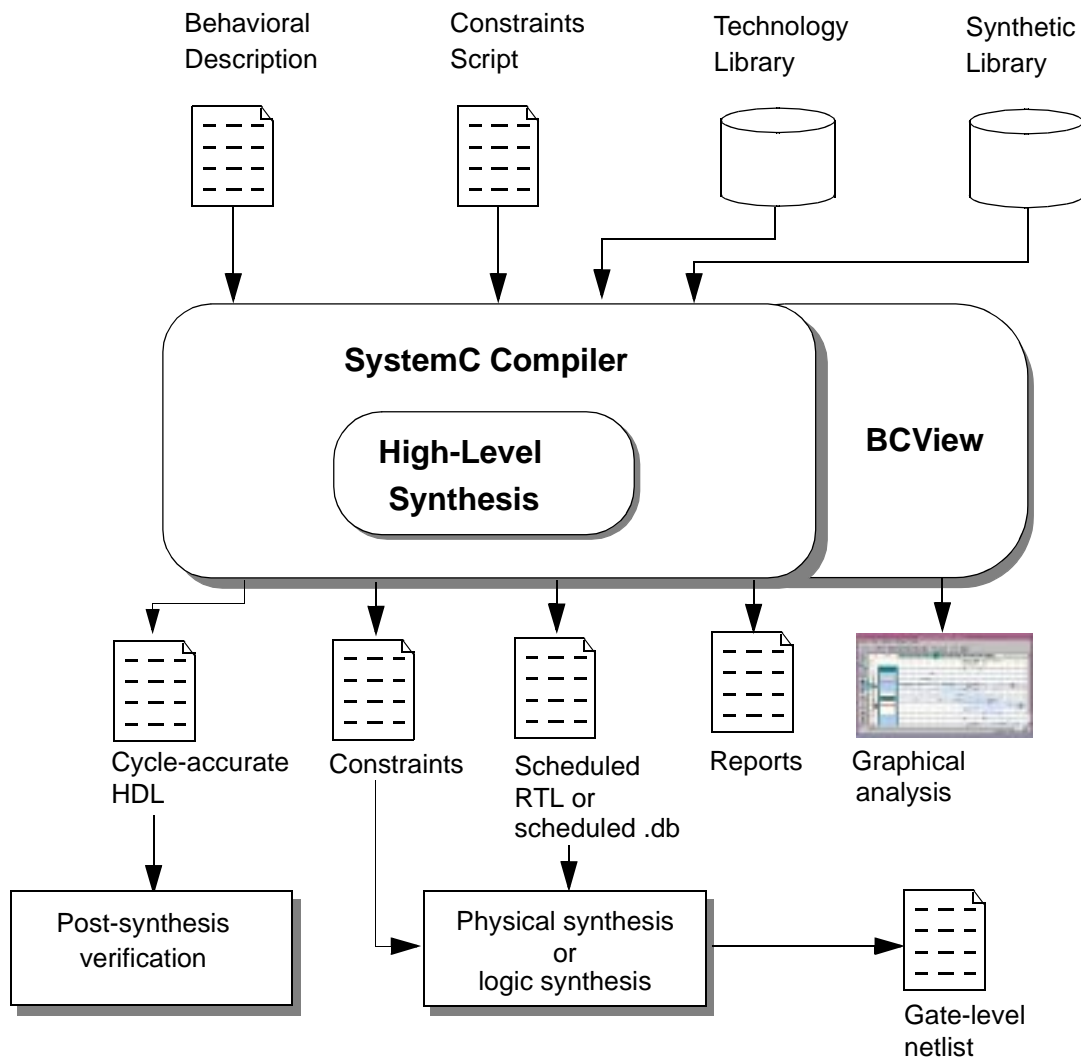
You can map an array to a memory and use a simple array access statement in the behavioral description to access the memory. SystemC Compiler automatically schedules memory accesses and generates the control for memory access. Data dependencies between memory read and write and other operations in the data flow are respected. Use memory inferencing to explore the tradeoffs of different memory architectures. Inferring memories and register files is described in Chapter 7, “Using Register Files and Memories for Arrays.”

Libraries and Other Inputs

SystemC Compiler requires a SystemC behavioral description following the coding guidelines described in the *CoCentric™ SystemC Compiler Behavioral Modeling Guide*, a technology library, and a synthetic library.

Figure 1-6 shows the flow into and out of SystemC Compiler.

Figure 1-6 SystemC Compiler Input and Output Flow



Behavioral Description

Write and refine the behavioral hardware description in SystemC using the SystemC Class Library according to the guidelines in the *CoCentric™ SystemC Compiler Behavioral Modeling Guide*.

The behavioral description is independent from the technology and implementation architecture. Using SystemC Compiler, you can change the target technology library or constrain the implementation architecture without modifying the behavioral description. This allows you to explore various implementation architectures and target technologies, which is particularly useful for FPGAs.

This manual uses the example designs that are available in the *CoCentric™ SystemC Compiler Behavioral Modeling Guide*. The files for these examples are available in the SystemC Compiler installation in the `SYNOPSIS/doc/syn/scc` directory.

Technology Library

A technology library is provided by an ASIC vendor in Synopsys .db database format. It provides the area, timing, wire load models, and operating conditions. You provide the path to your chosen technology library for your design by defining the `target_library` variable in `dc_shell`.

Sample technology libraries are provided in the SystemC Compiler installation at `SYNOPSIS/libraries/syn`. For the examples in this manual, the `tc6a_cbacore.db` sample technology library is defined as the target library.

Synthetic Library

The DesignWare synthetic library is a technology-independent library of logic components. SystemC Compiler maps your design operations to the synthetic library components. You provide the path to your chosen synthetic libraries for your design by defining the `synthetic_library` variable in `dc_shell`.

The DesignWare libraries are provided in the SystemC Compiler installation at `$SYNOPSIS/libraries/syn`. The synthetic libraries have names such as `standard.sldb`, `dw01.sldb`, `dw02.sldb`, and so forth. For information about the DesignWare libraries, see the DesignWare online documentation.

Outputs From SystemC Compiler

The output from SystemC Compiler is a cycle-true, fully constrained RTL architecture that includes the FSM control logic and constraints (such as multicycle constraints and resource sharing constraints) needed for logic synthesis, as shown in Figure 1-6 on page 1-12.

You can write out the RTL in three styles

- An RTL `.db` file, which is recommended for compilation to a gate-level netlist.
- A synthesizable RTL HDL file in Verilog or VHDL, which you can use for compilation to gates, for verification, or for any other aspect of the design flow that requires an HDL input.
- An RTL HDL or SystemC file optimized for simulation, which is recommended for verification.

SystemC Compiler has a graphical analysis environment called BCView that you can use to quickly and effectively analyze the architecture generated by SystemC Compiler and to identify the causes of common scheduling errors, if they should occur. BCView is described in Chapter 6, “Analyzing Designs With BCView.”

2

Using SystemC Compiler

This chapter describes the SystemC Compiler commands required to synthesize a SystemC behavioral description into a gate-level netlist or an RTL description.

In this chapter, a complex number multiplier design is used to show the typical command usage. The source code, command script, and reports generated are available in Appendix B, “Complex Number Multiplier Example Files.” Other example designs are available in the *CoCentric™ SystemC Compiler Behavioral Modeling Guide*, and you can access the design files in the SystemC Compiler installation at `$SYNOPTSYS/doc/syn/scc`.

This chapter contains the following sections:

- Usage and Commands
- Defining Libraries

- Compiling and Elaborating the Source Code
- Assigning Timing and Area Design Constraints
- Checking the Design
- Estimating Time and Area
- Scheduling the Design and Allocating Resources
- Generating Summary Reports
- Removing Designs from SystemC Compiler Memory
- Resuming Synthesis From a Saved .db File
- Writing the RTL Files
- Compiling and Writing a Gate-Level Netlist

Usage and Commands

This chapter uses the complex number multiplier example (“Complex Number Multiplier Source Code” on page B-2) to show how to use the commands. Figure 2-1 illustrates the primary commands that you use to perform behavioral synthesis with SystemC Compiler and compile the design into gates. The diagram also shows the inputs you provide and the outputs SystemC Compiler can provide.

The commands used in this chapter show the typical options you use. For a full description of the command and all its options, see the Synopsys online man pages. Accessing and using man pages is described in “Getting Command, Variable, and Error Help” on page A-15.

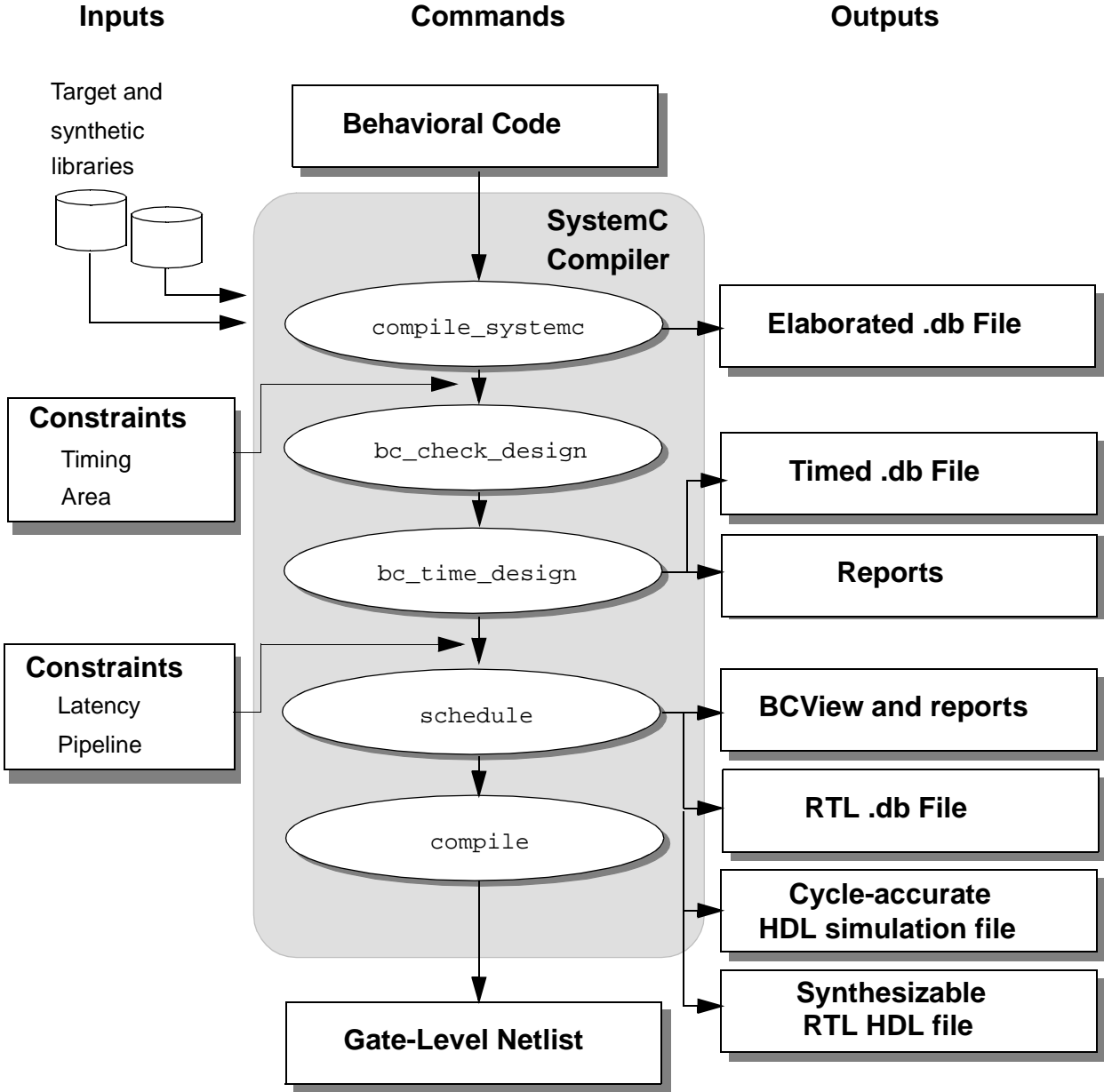
You do not need to complete this design flow in a single session. Start the session at the top of the flow. If you stop, reenter the flow at a later time either at the `compile_systemc` command, at the `bc_time_design` command, or at the `schedule` command.

Enter the SystemC Compiler commands at the `dc_shell` prompt or use the `include` command to run a script that contains the commands. To start `dc_shell`, enter the following at a UNIX prompt:

```
unix% dc_shell
```

If this is the first time you are using SystemC Compiler, see Appendix A, “Setting Up SystemC Compiler” for information about setting up your environment, entering commands, and using scripts.

Figure 2-1 SystemC Compiler Commands Use in the Flow



Defining Libraries

Before you use SystemC Compiler, you need to define the target library, synthetic library, link library, and search path that are appropriate for your design by setting the `target_library`, `synthetic_library`, `link_library`, and `search_path` variables. For example, to use the `tc6a_cbacore` library and the DesignWare libraries, enter

```
dc_shell> target_library = {"tc6a_cbacore.db"}
dc_shell> synthetic_library = {"dw01.sldb" "dw02.sldb"}
dc_shell> link_library = target_library + synthetic_library
dc_shell> search_path = search_path + "your_library.db"
```

Other variables you can set are described in “Defining Libraries and Other Variables” on page A-3.

Compiling and Elaborating the Source Code

Before you use SystemC Compiler, simulate your design with a standard C++ compiler. This ensures that your design is functionally correct and meets the functional specification. This is also valuable to detect and correct any C++ syntax and semantic errors.

Preparing to Use BCView

To use BCView for evaluating your design, set the `bc_enable_analysis_info` variable to true before you use the `compile_systemc` command so SystemC Compiler creates the additional analysis data. Enter

```
dc_shell> bc_enable_analysis_info = true
```

After executing the `schedule` command, use BCView to determine scheduling errors or to evaluate your scheduled design. You can later remove the additional analysis data with the `remove_analysis_info` command.

Using the `compile_systemc` Command

Use the `compile_systemc` command to read your SystemC source code and check it for compliance with synthesis policy, C++ syntax, and C++ semantics. If there are no errors, it produces an internal database (.db) ready for timing analysis. This process is called elaboration.

The `compile_systemc` command, and the other SystemC Compiler commands, respond with 1 if no errors were encountered or a 0 if an error was encountered. It also displays explanatory messages for errors and warnings.

The `compile_systemc` command performs the following:

- Checks C++ syntax and semantics
- Replaces source code arithmetic operations with DesignWare components
- Performs optimizations such as constant propagating, constant folding, dead code elimination, and algebraic simplification
- Performs the necessary elaboration steps to prepare the SystemC description for timing analysis and scheduling

For information about issuing C++ compiler preprocessor options with the `compile_systemc` command, see “Using `compile_systemc` Command Preprocessor Options” on page A-9.

Elaborating a Design With a Single Behavioral Module

If your design has a single behavioral module with one or more behavioral processes, use the `compile_systemc` command to elaborate the design. For example, to elaborate the `cmult` design, enter

```
dc_shell> compile_systemc cmult.cc
```

Elaborating a Hierarchical Design With Multiple Behavioral Modules

If your design is hierarchical and contains multiple behavioral modules, you need to use the `compile_systemc` command to elaborate each module separately. Then use the `link` command to link the internal databases. The top-level module must be an RTL module that instantiates the behavioral modules. Each behavioral module can contain one or more processes. (For details about creating a hierarchical module, see the *CoCentric™ SystemC Compiler Behavioral Modeling Guide* or the *CoCentric™ SystemC Compiler RTL User and Modeling Guide*.) Enter

```
dc_shell> compile_systemc beh_module1.cc
dc_shell> compile_systemc beh_module2.cc
dc_shell> compile_systemc -rtl -rtl_format db top_rtl.cc
dc_shell> link
```

The current design name is taken from the most recently executed `compile_systemc` command. In this example, the current design name is `top_rtl`. You can elaborate the files in any order. To change the current design name after you link the elaborated files, enter

```
dc_shell> current_design new_design_name
```

The `compile_systemc` command provides several other options related to RTL synthesis. For information about these options, see the *CoCentric™ SystemC Compiler RTL User and Modeling Guide*.

Elaborating a Design With Multiple Files

If your design has multiple modules that are defined in separate files, you can use either the `#include` directive or preserved functions to bring the external files into the primary design. The commands to use either method are described in “Using Multiple Files to Describe a Design” on page 8-2

Assigning Timing and Area Design Constraints

Before timing the design, you can enter constraints that affect timing. The `create_clock` command is the only constraint that is required by SystemC Compiler at this stage.

Setting the Clock Period

Use the `create_clock` command to mark an existing design port as the clock and set the clock period, which is specified in the same unit defined in the target technology library. For example, to mark a port named `clk` as a clock port and set the clock period to 20 units, enter

```
dc_shell> create_clock clk -period 20
```

Setting Other Initial Constraints

You do not need to set other design constraints at this stage. You can, however, set constraints including environmental conditions that affect delays (for example, the operating conditions and wire load model). If you do not specify operating conditions and a wire load model, the target library default values are used. The constraints you can set are described in “Setting Your Timing Environment” on page 3-7.

Checking the Design

Use the `bc_check_design` command to check for errors that will prevent your design from being synthesized with SystemC Compiler.

Before using the `bc_check_design` command, you need to specify the clock period for the design so the analysis is accurate (see “Setting the Clock Period” on page 2-8).

Running Check Design

Run the `bc_check_design` command to quickly check for SystemC Compiler scheduling errors. This check determines whether your module can be scheduled using the selected I/O scheduling mode.

```
dc_shell> bc_check_design -io_mode mode
```

The `io_mode` can be either `cycle_fixed` or `superstate_fixed`. The default `io_mode` is `cycle_fixed`. For the complex number multiplier example, the I/O mode is chosen to be `superstate_fixed` mode.

Selecting a scheduling I/O mode is described in “Selecting an I/O Scheduling Mode” on page 4-10. Finding and fixing scheduling errors is described in “Using BCView to Analyze Scheduling Errors” on page 2-15.

Changing the Code

If the result of `bc_check_design` indicates a need to change the source code, make the necessary changes and repeat the steps from “Compiling and Elaborating the Source Code” on page 2-5.

Estimating Time and Area

The `bc_time_design` command estimates the timing and area used by the design based on the initial design constraints. The Design Compiler timing engine and the target library default settings are used for accurate estimation. By default, the calculation is based on the implementation with the smallest area. Enter

```
dc_shell> bc_time_design
```

This command annotates the current design with the timing and area data for later use by the `schedule` command.

To change the default behavior of the `bc_time_design` command, use the `-fastest` option. Enter

```
dc_shell> bc_time_design -fastest
```

The `-fastest` option uses the fastest available implementation for each synthetic operation (+, *, and so forth) instead of the default, which is the implementation with the smallest area.

To force the `bc_time_design` command to recompute and overwrite the existing timing and area estimates, use the `-force` option. Use it, for example, to recalculate timing and area estimates when you change the target library. Enter

```
dc_shell> bc_time_design -force
```

Commands, variable settings, and other techniques you can use to improve latency and area are described in Chapter 3, “Timing and Area Estimation,” and Chapter 5, “Optimizing Latency and Area.”

Reporting Timing and Area Estimates

The `report_resource_estimates` command displays the timing and area estimates generated by the `bc_time_design` command. Enter

```
dc_shell> report_resource_estimates
```

The report shows the delays through the synthetic components required by the current design. These delays are used for scheduling and allocation.

An example of the report and an explanation are provided in “Interpreting the Timing and Area Resource Report” on page 3-17.

Saving the Timed Design

Save the timed `.db` file so you can explore different architectures without running `bc_time_design` each time. Use the `write` command to write out the timed `.db` file. Enter

```
dc_shell> write -hierarchy  
             -output cmult_timed.db
```

Resuming synthesis from this saved `.db` file is described in “Resuming Synthesis From a Saved `.db` File” on page 2-17.

Scheduling the Design and Allocating Resources

The `schedule` command invokes the scheduling and allocation functions of SystemC Compiler. If you have not already invoked the `bc_time_design` command, the `schedule` command executes it. Enter

```
dc_shell> schedule -io_mode mode
```

By default, the `schedule` command

- Makes tradeoffs to achieve a design with the fastest latency as the top priority
- Creates a design with the smallest area as a secondary priority
- Performs scheduling and allocation with low effort
- Performs cycle-fixed I/O scheduling

The `-io_mode` can be an I/O scheduling mode of either `cycle_fixed` or `superstate_fixed`. Selecting a scheduling I/O mode is described in “Selecting an I/O Scheduling Mode” on page 4-10. For the complex number multiplier example, the `superstate_fixed` mode is chosen.

Scheduling for Smallest Area

To change the scheduling priority to smallest area scheduling as the top priority and fastest latency as a secondary priority, use the `-extend_latency` option. Enter

```
dc_shell> schedule -io_mode superstate_fixed
            -extend_latency
```

When you use the `-extend_latency` option with the superstate-fixed scheduling mode, the `schedule` command adds clock cycles (latency) to the design whenever possible to minimize resources needed by the design and produce the smallest area. The `-extend_latency` option is not relevant for the cycle-fixed scheduling mode, because placement of clock cycles are controlled by the source code.

Changing the Effort Level

To control the CPU effort level for scheduling, use the `-effort` option. Define the effort as `quick`, `low`, `medium`, or `high`. For example,

```
dc_shell> schedule -io_mode superstate_fixed -effort medium
```

To control the CPU effort level for allocation, use the `-allocation_effort` option. Define the effort as `quick`, `low`, `medium`, or `high`. For example,

```
dc_shell> schedule -io_mode superstate_fixed
           -effort high
           -allocation_effort medium
```

Setting Schedule Constraints

You can apply other scheduling constraints before using the `schedule` command, as described in Chapter 4, "Scheduling and Scheduling Constraints."

Using BCView to Analyze Scheduling Errors

If you get scheduling errors when running the `bc_check_design` or `schedule` commands, use the BCView Scheduling Error Analyzer to obtain graphic information that can help you determine where and why the scheduling errors occur. The Scheduling Error Analyzer shows where design specification requirements conflict. Such conflicts arise when user constraints and the inherent requirements of the design are incompatible.

For details about using BCView, see “Examining Scheduling Errors” on page 6-9.

When BCView can analyze scheduling errors, SystemC Compiler prints a message directing you to launch BCView. Otherwise, SystemC Compiler provides informative messages.

Analyzing Scheduling Results

After scheduling the design, analyze the results using one or both of the following methods:

- Use BCView to perform analysis after scheduling, which is described in “Evaluating the Architecture Generated by SystemC Compiler” on page 6-21.
- Use the `report_schedule` command to display the results of scheduling.

The `report_schedule` command displays the results of scheduling and allocation. Examine the scheduling reports to determine whether the synthesized design is satisfactory. Enter

```
dc_shell> report_schedule
```

An example scheduling report and an explanation are provided in “Analyzing the Scheduling Report” on page 4-20.

Generating Summary Reports

To generate summary reports of the design after it is compiled to gates, use one or more of the following commands:

```
dc_shell> report_area
dc_shell> report_resources
dc_shell> report_timing
```

Examples of these reports are shown in “Report Area” on page B-11, “Report Resources” on page B-14, and “Report Timing” on page B-12 show.

Removing Designs from SystemC Compiler Memory

You might want to remove all the current designs from SystemC Compiler memory to synthesize a different design or resume synthesis from a .db file you saved at some point in the flow. To remove designs from SystemC Compiler memory, use either the `remove_design` or `free` commands. Enter,

```
dc_shell> remove_design -designs
```

Or enter,

```
dc_shell> free -designs
```

Resuming Synthesis From a Saved .db File

To resume synthesis of a design from a .db file you saved after elaboration, timing, or scheduling, use the `read` command to bring the .db file into SystemC Compiler. Before reading the .db file, you can optionally remove all designs from SystemC Compiler. For example, if you want to resume synthesis of the complex number multiplier from the .db file saved after timing, enter

```
dc_shell> free -designs
dc_shell> read cmult_timed.db
```

Then resume synthesis starting at the next step in the synthesis flow. For this example, the next step is described in “Scheduling the Design and Allocating Resources” on page 2-13.

Writing the RTL Files

When you are satisfied with the results of scheduling, write out an RTL .db file and HDL format files for

- Verification of behavioral synthesis results
- A future logic synthesis session
- Formal verification
- RTL sign-off
- Use with other Synopsys tools

You can write out the RTL in three styles

1. Write an RTL .db file, which is recommended for compilation to a gate-level netlist.
2. Write a synthesizable RTL file in Verilog or VHDL, which you can use for compilation to gates, for verification, or for any other aspect of the design flow that requires an HDL input.
3. Write an RTL HDL or SystemC file optimized for simulation, which is recommended for verification. This file is not appropriate for synthesis.

Writing the RTL .db File

The RTL .db file is a scheduled and constrained database file you can use for logic synthesis with Synopsys tools, such as Design Compiler and Physical Compiler, that accept a .db file. To write out this file, use the following command:

```
dc_shell> write
          -hierarchy
          -output design_sch_rtl.db
```

where

- The `-hierarchy` option specifies to write all designs in the hierarchy. It is recommended that you always use the `-hierarchy` option for writing out the RTL .db file of a design synthesized with SystemC Compiler.
- The `-output` option specifies the output file name. It is recommended that you create a file name with `_sch` to indicate the design is scheduled, `_rtl` to indicate RTL, and the `.db` extension to indicate it is a database file.

Resuming synthesis from this saved .db file is described in “Resuming Synthesis From a Saved .db File” on page 2-17.

Writing a Synthesizable RTL HDL File

Write the RTL design in an HDL format file for a future logic synthesis session with other Synopsys synthesis tools such as Design Compiler and Physical Compiler. Synthesizable RTL is a register transfer level description of a design generated by SystemC Compiler. To write out this file, use the following command:

```
dc_shell> write_rtl
    [-format [ verilog | vhdl ]
    [-output [design_sch_rtl.vhd | design_sch_rtl.v ]]
    [-rtl_script design_sch_rtl.scr]
```

where

- The `-format` option specifies the output format as Verilog or VHDL.
- The `-output` option specifies the output file name. It is recommended that you create the file name with the typical extensions of `.v` for Verilog or `.vhd` for VHDL.
- The `-rtl_script` option specifies the file name for the automatically generated `dc_shell` script, which contains RTL synthesis constraints. (If you are running `dc_shell` in the `dctcl` mode, the script is generated with the appropriate Tcl syntax.)

Prior to performing logic synthesis, read in the RTL design and the automatically generated script using the following commands:

```
dc_shell> read -f [vhdl | verilog] design_sch_rtl.v[hd]
dc_shell> include design_sch_rtl.scr
dc_shell> compile
```

Writing an RTL Simulation File

You can write the scheduled design to an HDL or SystemC format file optimized for simulation speed using the `-simulation` option with the `write_rtl` command. Use this file as input to a simulator such as Verilog Compiled Simulator (VCS) or Scirocco VHDL Simulator. If you write a SystemC file, use it with a C++ language compiler.

You can also use the synthesizable RTL file, described in “Writing a Synthesizable RTL HDL File” on page 2-20, for simulation. It is not, however, optimized for simulation speed.

When you use the `-simulation` option with the `write_rtl` command, SystemC Compiler generates a cycle-accurate, leveled RTL netlist for simulation purposes. The design hierarchy is flattened and the RTL netlist is written out to contain the least possible number of processes. Each process is sensitive to a clock, which means that only recognized simulation events are clock edges. Because there are only a few processes, the total number of simulation events is significantly reduced, and simulation executes much faster. With the clock edges limited to only simulation events, the simulation is cycle accurate.

Note:

This style of RTL is not suitable for synthesis with logic synthesis tools.

To write out an RTL HDL or SystemC file optimized for simulation, use the following command:

```
dc_shell> write_rtl
    [-format [verilog | vhdl | systemc]
    [-simulation]
    [-debug_mode]
    [-output [design_sch_rtl.vhd | design_sch_rtl.v
    design_sch_rtl.cc]]
```

where

- The `-output` option specifies the output file name.
- The `-format` option specifies the output format as Verilog, VHDL, or SystemC.
- The `-simulation` option specifies the output format as optimized for simulation speed.
- The `-debug_mode` option specifies that the simulation RTL output contains additional code to print diagnostics and enhance debugging of the RTL simulation model. The `-debug_mode` option can only be used with the `-simulation` option. The debug enhancements include
 - A process that traces the execution of the FSM generated by SystemC CompilerThe process contains variables that you can monitor during simulation for the current state of the FSM, the behavioral design loop currently being executed, and the number of clock cycles spent in the loop.
 - Warnings about registers set to unknown values and multiplexers with invalid values on their control lines

- Warnings about input ports with values that are assumed to be constant because the `bc_dont_register_input_port` command is applied to the input port, but the values are changing during simulation
- A trace of all reads and writes to memories
- A trace of all I/O operations that the design executes (To generate I/O traces, set the `bc_add_io_trace` variable to true before executing the `schedule` command.)

Specifying VHDL Packages

If you are writing a VHDL RTL file, you can use the `-use_packages` option of the `write_rtl` command to specify a list of VHDL packages to use in the RTL output. To specify VHDL packages, enter

```
dc_shell> write_rtl
          -format vhd1
          -output design_sch_rtl.vhd
          -rtl_script design_sch_rtl.scr
          -use_packages {dw02.dw02_components,
                        synopsys.attributes}
```

Specifying Verilog Include Files

If you are writing a Verilog RTL file, you can use the `-include_files` option to specify a list of Verilog include files to use in the RTL output. To specify include files, enter

```
dc_shell> write_rtl
          -format verilog
          -output design_sch_rtl.v
          -rtl_script design_sch_rtl.scr
          -include_files {my_mult.v, test_decl.v}
```

Compiling and Writing a Gate-Level Netlist

At this point in the flow, the behavioral description has been synthesized into RTL. You can prepare the design for either place and route or physical synthesis.

Preparing for Place and Route

Use the `compile` command to create a gate-level netlist for place and route. The `compile` command performs logic synthesis and optimization on the current design.

```
dc_shell> compile
        -map_effort [low | medium | high]
```

Use the following command to write the gate-level netlist:

```
dc_shell> write
        -hierarchy
        -output cmult_netlist.db
```

For verification at the gate level, write a Verilog (or VHDL) simulation file using the following command:

```
dc_shell> write
        -format verilog
        -hierarchy
        -output cmult_netlist.v
```

Preparing for Physical Compiler

Physical Compiler accepts an RTL or gate-level input and performs logical and physical synthesis. This process results in placed gates.

To use Physical Compiler for physical synthesis, you need to perform behavioral synthesis of the design using SystemC Compiler with a target library that contains physical information in .lef or .pdb format. For information about using Physical Compiler, see the Physical Compiler documentation.

Preparing RTL for Physical Synthesis

To perform physical synthesis from an RTL netlist with Physical Compiler, you need to provide a synthesizable RTL netlist and a constraints file in Tcl format.

After you execute the SystemC Compiler `compile_systemc`, `bc_time_design`, and `schedule` commands, use the following command to write the synthesizable RTL database file:

```
dc_shell> write -format db
           -hierarchy
           -output cmult_rtl.db
```

Use the following command to write the synthesizable Verilog file and a Tcl constraints script for an HDL-based flow:

```
dc_shell> write_rtl -format verilog
           -output cmult_rtl.v
           -rtl_script synrtl.tcl
```

For verification of the RTL, write a Verilog (or VHDL) simulation file using the following command:

```
dc_shell> write
        -format verilog
        -hierarchy
        -output cmult_rtl.v
```

Preparing Gate-Level for Physical Synthesis

To perform physical synthesis from a mapped, gate-level database with Physical Compiler, you need to provide a .db file.

After you execute the SystemC Compiler `compile_systemc`, `bc_time_design`, and `schedule` commands, use the following `compile` command to create a mapped, gate-level netlist of the design:

```
dc_shell> compile
        -map_effort [low | medium | high]
```

Use the following command to write the gate-level netlist:

```
dc_shell> write -format db
        -hierarchy
        -output cmult_gate.db
```

For verification at the gate level, write a Verilog (or VHDL) simulation file using the following command:

```
dc_shell> write
        -format verilog
        -hierarchy
        -output cmult_gate_netlist.v
```

3

Timing and Area Estimation

This chapter describes how SystemC Compiler calculates timing and area estimates before scheduling the design. It also explains how to influence the estimation of timing and area.

This chapter contains the following sections:

- Understanding Clock Cycle, I/O, and Operation Relationships
- Setting Your Timing Environment
- Timing the Design
- Interpreting the Timing and Area Resource Report

Understanding Clock Cycle, I/O, and Operation Relationships

Your behavioral description defines the I/O protocol of your design and the operations required to execute the required functionality.

Operation Delay and Clock Cycle

To understand how SystemC Compiler uses timing and area estimates, you need to understand the relationships between operation delays and the clock period.

SystemC Compiler produces a circuit that is synchronous. The synchronous design uses edge-triggered flip-flops and a single-phase, single-clock clocking scheme.

Each process in the behavioral description is sensitive to the positive or negative edge of a single clock, which is defined with the `create_clock` command. The relevant clock edge is called the active clock edge. At the active clock edge,

- Inputs to registers are sampled
- Outputs change

In addition to the edge-triggered flip-flops, the design contains combinational logic. The combinational logic is used to implement components, multiplexers, and finite-state machine (FSM) logic.

The combinational logic resides between registers. The combinational logic must compute its output data before the data is sampled by the registers on the next active clock edge. Therefore,

your choice of a clock period determines how much combinational logic SystemC Compiler can place in each clock cycle. This in turn affects the architecture produced by SystemC Compiler.

I/O Protocol

You define the I/O protocol in your behavioral description by specifying when data is read from the input ports and when data is written to the output ports. Inputs are read and outputs are written at the active clock edge.

Active edges are represented in the behavioral description by `wait()` statements. The number of `wait()` statements between I/O reads and writes determines the number of clock cycles between them.

Example 3-1 shows the I/O protocol of the complex number multiplier highlighted in bold. Figure 3-1 shows a timing diagram of the I/O protocol implied by this SystemC description.

Example 3-1 Complex Multiplier I/O Protocol

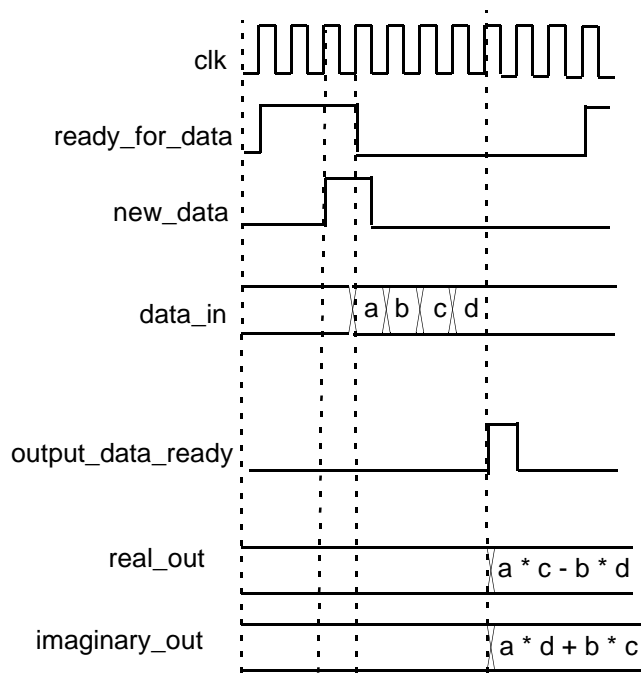
```
// cmult.cc implementation file
#include "systemc.h"
#include "cmult.h"
void cmult_hs :: entry()
{
    sc_int<8> a, b, c, d;
    //Initialize and reset if reset asserts
    ready_for_data.write(false);
    output_data_ready.write(false);
    real_out.write(0);
    imaginary_out.write(0);
    wait(); //required clock before while loop
    while (true)
    {
        ready_for_data.write(true);
        output_data_ready.write(false);
    }
}
```

```

wait_until(new_data.delayed() == true);
ready_for_data.write(false);
// 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 * c);
output_data_ready.write(true);
wait();
}
}

```

Figure 3-1 Timing Diagram of the Complex Multiplier I/O Protocol



Operations and Clock Cycles

Operations in a behavioral description manipulate data received from the input ports to produce the output data, as required by the design functionality. SystemC Compiler determines in which clock cycle it is possible to execute each operation, and then it executes the operation in the most beneficial clock cycle. This is called operation scheduling.

Example 3-2 shows the arithmetic expressions of the complex number multiplier highlighted in bold. Figure 3-2 shows the individual operations that compose the expressions and also shows one possible schedule that maps the operations to clock cycles.

Example 3-2 Complex Multiplier Arithmetic Operations

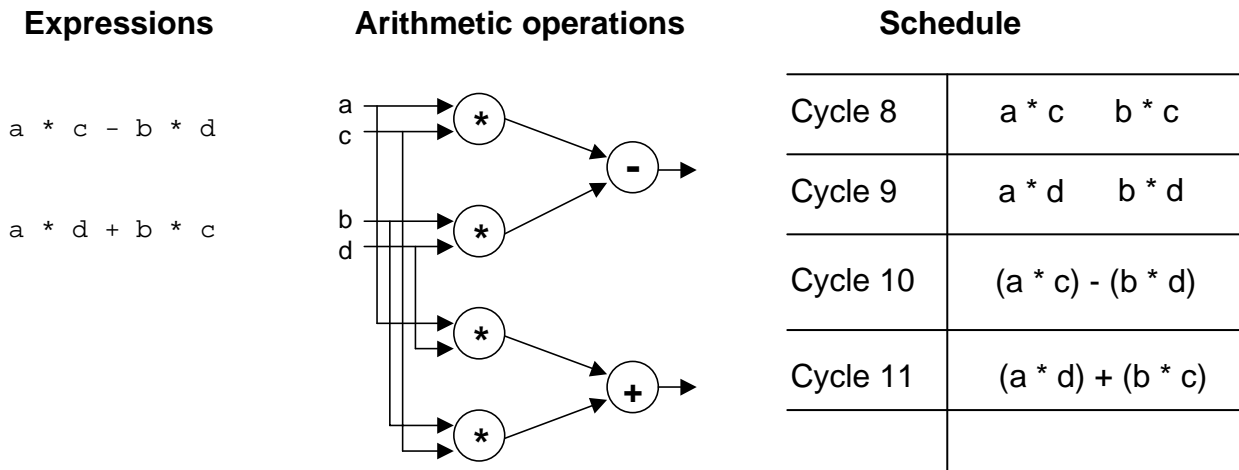
```
// cmult.cc implementation file
#include "systemc.h"
#include "cmult.h"
void cmult_hs :: entry()
{
    sc_int<8> a, b, c, d;
    //Initialize and reset if reset asserts
    ready_for_data.write(false);
    output_data_ready.write(false);
    real_out.write(0);
    imaginary_out.write(0);
    wait(); //required clock before while loop
    while (true)
    {
        ready_for_data.write(true);
        output_data_ready.write(false);
        wait_until(new_data.delayed() == true);
        ready_for_data.write(false);
        // 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 * c);
    output_data_ready.write(true);
    wait();
  }
}

```

Figure 3-2 Operations of the Complex Multiplier



Setting Your Timing Environment

Set your timing environment before you use either the `bc_time_design` or `bc_margin` commands. You need to define the clock period, and you can optionally set the input delay, wire load, and operating conditions.

Setting Clocks

Define the clock period in the units of the technology library using the `create_clock` command. To create a 20 time unit clock, enter

```
dc_shell> create_clock clk -period 20
```

To display the current clock setting, use the `report_clock` command to generate a clock report similar to Example 3-3. Enter

```
dc_shell> report_clock
```

Example 3-3 Report Clock

```
dc_shell> report_clock
*****
Report : clocks
Design : cmult_hs
Version: 2000.11-PROD
Date   : Fri Dec 15 11:37:14 2000
*****
```

```
Attributes:
  d - dont_touch_network
  f - fix_hold
  p - propagated_clock
  G - generated_clock
```

Clock	Period	Waveform	Attrs	Sources
clk	20.00	{0 10}		{clk}

To remove a clock, use the `remove_clock` command. Enter

```
dc_shell> remove_clock clk
```

You can also remove all clocks by using the `-all` argument. Enter

```
dc_shell> remove_clock -all
```

Setting Input Delays

Inputs to the design may be coming from other circuits on or off the chip. These inputs will not arrive exactly at the active clock edge, rather they will arrive some time after the active edge. You can use the `set_input_delay` command to specify the exact time after the active clock edge when the inputs will arrive. This constraint is optional, but it is highly recommended that you set it to enable you to reach timing closure later in the flow.

For example, to set an input delay of 1.0 for the `data_in` port with respect to the `clk` clock, enter

```
dc_shell> set_input_delay 1.0 -clock clk data_in
```

The delay value specifies the input delay, which is in the units of the technology library. SystemC Compiler assumes that the specified inputs, in this case `data_in`, are available at the specified input delay after the active clock edge. SystemC Compiler uses this information to compute the combinational path delays.

The `-clock` option specifies the name of the clock; the specified input delay is added to its active edge. If the design has only one clock, it is not necessary to use the `-clock` option.

The port or list of ports option defines the input ports in the current design to which the input delay is assigned.

You can use the `all_inputs()` function in place of a port list to automatically extract the port names. For example,

```
dc_shell> set_input_delay 1.0 -clock clk
          all_inputs() - clk
```

The `all_inputs()` function returns all input port names, and minus `clk` removes the clock name from the list of all inputs.

Setting Operating Conditions

Use the `set_operating_conditions` command to set the interconnect model as part of the operating conditions. The operating conditions are specified as best case, worst case, and typical case. If you do not specify an operating condition, the technology library default typical case is used. For example, to set the operating conditions to worst case, enter

```
dc_shell> set_operating_conditions WORST
```

Although the `set_operating_conditions` command is optional, it is highly recommended that you select the same operating conditions that you will supply to other tools in the backend flow.

Listing Libraries

To list the names of libraries you have in memory, their file names, and path, use the `list` command with the `-lib` option. Enter

```
dc_shell> list -lib
```

Example 3-4 shows a typical list of libraries.

Example 3-4 Listing Libraries

```
dc_shell> list -lib
```

Library	File	Path
-----	----	----
cba_core	tc6a_cbacore.db	/remote/dtg332/scp/src/ 2000.11-SCC2/2000.11-SCC2/ libraries/syn
dw01.sldb	dw01.sldb	/remote/dtg332/scp/src/ 2000.11-SCC2/2000.11-SCC2/ libraries/syn
dw02.sldb	dw02.sldb	/remote/dtg332/scp/src/ 2000.11-SCC2/2000.11-SCC2/ libraries/syn
gtech	gtech.db	/remote/dtg332/scp/src/ 2000.11-SCC2/2000.11-SCC2/ libraries/syn
standard.sldb	standard.sldb	/remote/dtg332/scp/src/ 2000.11-SCC2/2000.11-SCC2/ libraries/syn

Listing Operating Conditions

To list the operating conditions defined in a technology library, use the `report_lib` command. Enter

```
dc_shell> report_lib cba_core
```

Example 3-5 shows a partial library report.

Example 3-5 Library Report (Partial)

```
dc_shell> report_lib cba_core
*****
Report : library
Library: cba_core
Version: 2000.11-PROD
Date   : Wed Nov 22 14:18:34 2000
*****

Library Type       : Technology
Tool Created      : v3.3b
Date Created      : Fri Aug  9 17:02:36 1996
Library Version   : tc6a_r06
Comments         : Operating condition (25.00 C, 5.00 V, typical)
Time Unit        : lns

Capacitive Load Unit : 1.000000pf
Pulling Resistance Unit : 1kilo-ohm
Voltage Unit       : 1V
Current Unit       : 1mA
Leakage Power Unit : Not specified.
Bus Naming Style   : %s[%d] (default)
```

Operating Conditions:

Name	Library	Process	Temp	Volt	Interconnect Model
typ_25_5.00	cba_core	1.00	25.00	5.00	
typ_-40_4.50	cba_core	1.00	-40.00	4.50	
...					

Setting Wire Loads

SystemC Compiler uses statistically generated wire load models to estimate the wire lengths of nets, their capacitance, resistance, and area. The wire load models, provided in the technology library, define a fanout-to-length relationship. If you do not specify a wire load, the technology library default is used.

Use the `set_wire_load` command to specify the wire load model. Although this constraint is optional, it is recommended that you use the appropriate wire load model for the size of the design you are going to synthesize.

```
dc_shell> set_wire_load 90x90 -lib cba_core
```

Example 3-6 shows a partial report of a wire load model in the `cba_core` technology library. The report was generated by the `report_lib` command. For example,

```
dc_shell> report_lib cba_core
```

Example 3-6 Wire Load Model (Partial)

Wire Loading Model:

```
Name           : tc6a120m2
Location        : cba_core
Resistance      : 0
Capacitance    : 0.02
Area           : 1.4375
Slope          : 2.5
Fanout   Length  Points Average Cap Std Deviation
-----
          1      2.50
          2      5.00
          ...
          10     25.00
```

Timing the Design

SystemC Compiler performs timing to obtain the bit-level timing through the components that are necessary to implement the operations in the behavioral description. It also reserves time from the clock period for hardware that is placed on every timing path during synthesis. The reserved time is called the timing margin or cycle margin.

Timing Through the Components

To perform timing of the design, use the `bc_time_design` command. The `bc_time_design` command computes the timing delays through all chains of operations in the behavioral description. Operations are chained when the output of one operation is used by another operation. Enter,

```
dc_shell> bc_time_design
```

You need to run the `bc_time_design` command only once. You may want to force it to recompute the timing delays, for example after you change the timing environment. Enter

```
dc_shell> bc_time_design -force
```

While the `bc_time_design` command is executing, it displays messages that show which component it is currently building. When SystemC Compiler finishes executing the command, it generates a timing report showing the computed delays through all chains of operations. Example 3-7 shows a partial timing report for the complex number multiplier.

Example 3-7 Timing Report (Partial)

```
...
Cumulative delay starting at mul_36:
  mul_36 = 6.357016
  add_36 = 10.150984
  imaginary_out_36 = 10.150984

Cumulative delay starting at add_36:
  add_36 = 8.784022
  imaginary_out_36 = 8.784022

Cumulative delay starting at data_in_26:
  data_in_26 = 0.000000
  mul_36 = 6.357016
  mul_35 = 6.357016
  add_36 = 10.150984
  sub_35 = 10.150984
  imaginary_out_36 = 10.150984
  real_out_35 = 10.150984
...
```

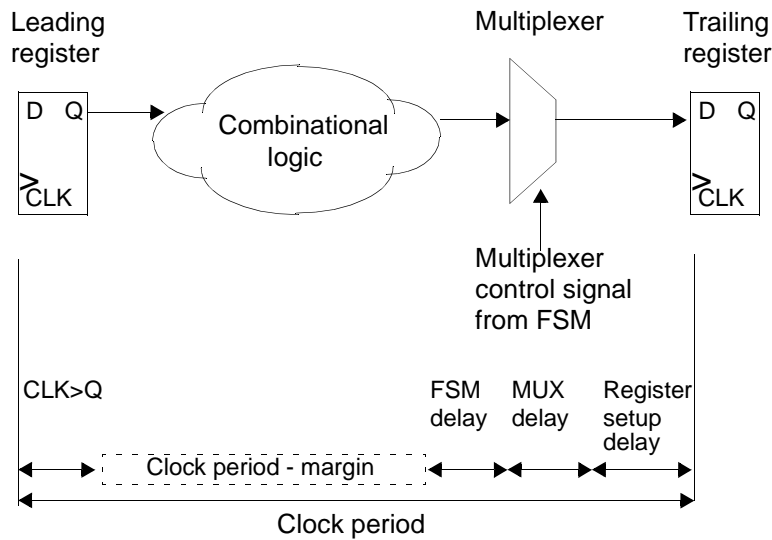
A detailed description of the report is provided in “Interpreting the Timing and Area Resource Report” on page 3-17.

Computing the Clock Cycle Margin

The `bc_time_design` command reserves time in the clock period as a clock cycle margin for the hardware that SystemC Compiler adds to every timing path in the design during synthesis. SystemC Compiler extracts the required time to be reserved from the target technology library. The clock period less the reserved clock cycle margin is available for combinational logic.

The timing path starts at the clock pin of a register, passes through the combinational logic, and terminates at the data input pin of a register. Figure 3-3 shows a typical timing path.

Figure 3-3 Typical Timing Path



Each timing path, as illustrated in Figure 3-3, contains common hardware components. SystemC Compiler reserves a clock cycle margin in the clock period for the following components:

- Register margin

The leading register requires time at the beginning of the clock period to respond to the clock edge and make the data available on its Q output pin. This is called clock-to-Q delay.

Data must arrive at the D input pin to the trailing register a certain time before the end of the clock cycle. This is called setup time.

The register margin is also referred to as the flip-flop (FF) margin, because registers are implemented as FFs from the target library.

- Multiplexer margin

The trailing register can get its input from several different sources. A multiplexer controls which of the different sources provides input to the register. The reserved timing margin includes time for the multiplexer.

- FSM margin

At each clock cycle, the FSM generated by SystemC Compiler moves into a new state. The reserved timing margin includes time for the FSM to decode its state and generate the control signals to control the data path portion of the synthesized design.

The `bc_time_design` command reports the clock cycle margin value based on the current target library. SystemC Compiler looks for all available flip-flops in the target library and uses the average clock-to-Q delay and setup delay. Example 3-8 shows the relevant data in the report.

Example 3-8 Clock Margin in the Resource Estimate Report

```
Clock Cycle Margin      : 2.86 (Default)
      FSM                : 0.55
      MUX                : 1.21
      FF                 : 1.11
Clock Uncertainty      : 0.00
```

To control clock cycle margin calculation, see “Controlling Margin Calculation” on page 5-12.

Interpreting the Timing and Area Resource Report

SystemC Compiler uses operation delays during scheduling to estimate timing. It uses area estimates of components that implement the operations, multiplexers, and registers to calculate the total area of the synthesized design.

SystemC Compiler reports timing and area resource estimates in two ways:

- The report is automatically displayed when you run the `bc_time_design` command.
- The resource report is displayed when you run the `report_resource_estimates` command after SystemC Compiler has calculated timing and area estimates.

A complete example of a resource estimate report is provided in “Estimated Resources” on page B-6.

Evaluating the Resource Estimate Report

The resource estimate report shows paths through chains of operations and the delays at all points in the path. The report is divided into sections, where each section reports on the paths starting at a specific operation.

Figure 3-4 shows a partial report, the related behavioral description fragment, and the related data flow diagram.

Figure 3-4 Estimated Resources Report (Partial)

```

Cumulative delay starting at data_in_32: ← Starting point
data_in_32 = 0.000000
mul_36 = 6.340029
mul_36_2 = 6.340029
mul_35_2 = 6.340029
add_36 = 10.138293
imaginary_out_36 = 10.138293
sub_35 = 10.417433
real_out_35 = 10.417433 ← Ending points

```

Intermediate points in the path

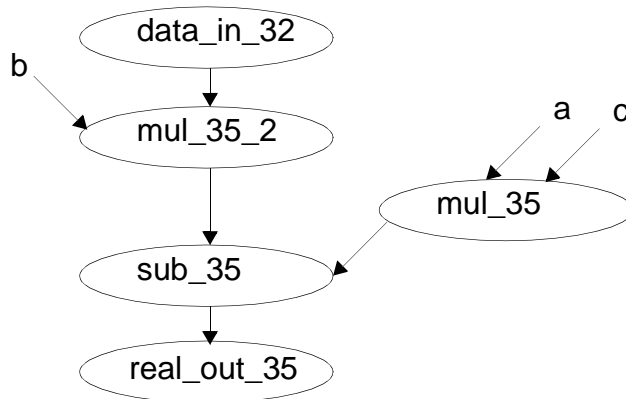
Related code fragment

```

32     d = data_in.read();
33     wait();
34     //Calculate and write output ports
35     real_out.write(a * c - b * d);
36     imaginary_out.write(a * d + b * c);

```

Related data flow diagram



This report shows paths in the design starting at the input read of the `data_in` port on line 32 of the behavioral description. Starting from this input read, the maximum bit-level delay to any output of the

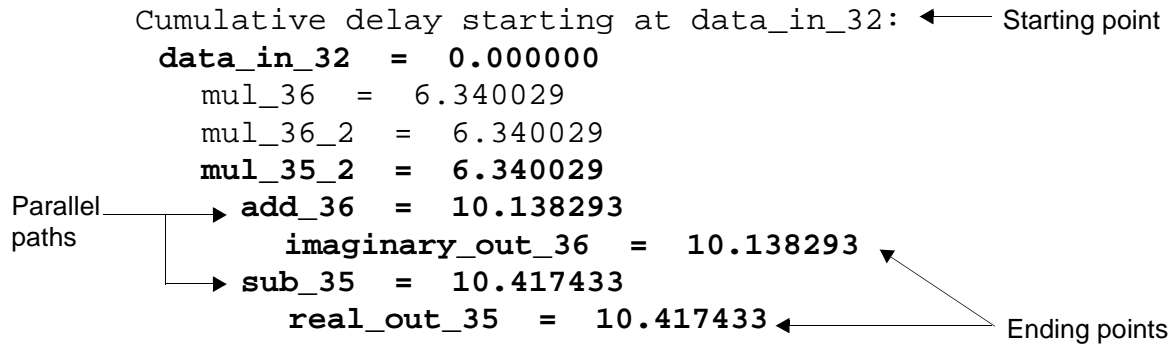
operation `add_36` is 10.1 time units. Operation `add_36` is the addition on line 36. Indentation in the report indicates intermediate points in the same timing path.

Looking at Parallel Paths

The resource estimates report indicates parallel paths by using the same level of indentation.

Figure 3-5 shows a data flow graph of the two parallel paths in the related code fragment. The relevant code and lines of the report are highlighted in bold. The `add_36` and `sub_35` operations have the same level of indentation, indicating they are parallel paths starting at the output of the `mul_35_2` operation.

Figure 3-5 Parallel Paths in the Estimated Resources Report (Partial)

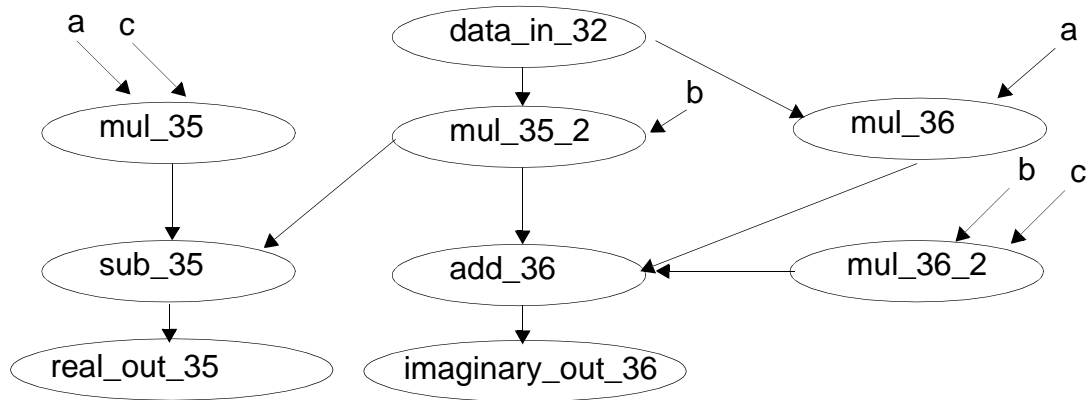


Related code fragment

```

32     d = data_in.read();
33     wait();
34     //Calculate and write output ports
35     real_out.write(a * c - b * d);
36     imaginary_out.write(a * d + b * c);
  
```

Related data flow diagram



Area Estimates

The area section of the report displays the area estimates for all the components (processors) that can implement that operation. Example 3-9 shows the timing and area resource report with an addition operation that has two possible components; an asterisk indicates the component used to calculate the timing. By default, the smallest component is used for the estimate.

Example 3-9 Estimated Resource Report

```
Area for processors that can implement mul_36
(* = used for timing):
```

```
  *DW02_mult(nbw) = 2750.742432
```

```
Area for processors that can implement add_36
(* = used for timing):
```

```
  *DW01_add(rpl) = 94.239998
```

```
  DW01_addsub(rpl) = 503.678986
```

Note:

The target technology library specifies the units of time and area.

4

Scheduling and Scheduling Constraints

This chapter describes how to use the SystemC Compiler I/O scheduling modes and other methods to improve scheduling. During the scheduling step in synthesis, SystemC Compiler determines the specific clock cycle in which to execute the I/O operations, arithmetic operations, and memory accesses. This chapter contains the following sections:

- Scheduling for Synthesis
- Selecting an I/O Scheduling Mode
- Performing Scheduling
- Analyzing the Scheduling Report
- Adding Scheduling Constraints
- Constraining Resource Allocations

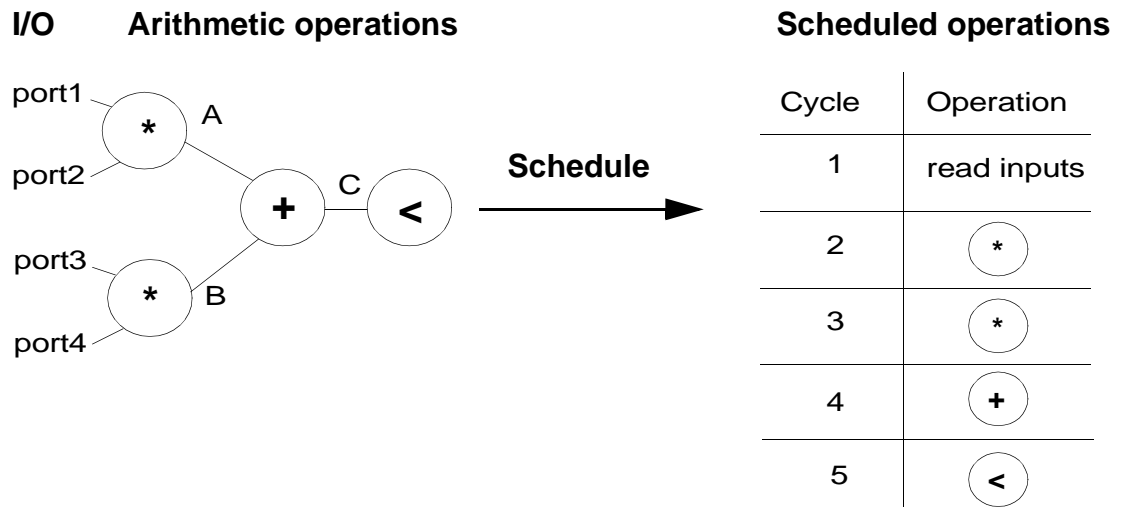
Scheduling for Synthesis

The main synthesis step is scheduling the design. During scheduling, SystemC Compiler schedules I/O operations, arithmetic operations, and memory accesses into specific clock cycles, as shown in Figure 4-1.

Figure 4-1 Scheduling Into Specific Clock Cycles

Behavioral code

```
wait_until(start.delayed() == true);  
A = port1.read() * port2.read();  
B = port3.read() * port4.read();  
C = A + B;  
if (C < 0) {...}
```



Before executing the `schedule` command, the design must be timed. If you have not executed the `bc_time_design` command, SystemC Compiler executes it before it starts the `schedule` command.

Operation Scheduling

During scheduling, SystemC Compiler selects the most beneficial clock cycle in which to execute each operation in the behavioral description. Figure 4-2 shows a sample schedule for the operations in the complex number multiplier example.

Figure 4-2 Operation Scheduling

Cycle	Operation
1	<code>a = data_in.read()</code>
2	<code>b = data_in.read()</code>
3	<code>c = data_in.read()</code>
4	<code>d = data_in.read()</code> <code>(a x c)</code> <code>(b x c)</code>
5	<code>(a x d)</code> <code>(b x d)</code>
6	<code>(a x c) - (b x d)</code>
7	<code>(a x d) + (b x c)</code>
8	<code>real_out.write()</code> <code>imaginary_out.write()</code>

SystemC Compiler ensures that the set of operations that are placed into the same clock cycle can be executed within the clock period that you specify. SystemC Compiler uses the timing information from the `bc_time_design` command to determine the timing.

SystemC Compiler schedules operations to minimize the latency (the number of clock cycles) to execute the synthesized design.

Resource Sharing

SystemC Compiler shares resources whenever possible. This means if two operations can execute on the same component, SystemC Compiler uses one component to execute both operations. This part of the schedule command is called resource allocation. Figure 4-3 shows a possible allocation for the complex number multiplier, based on the schedule shown in Figure 4-2.

Figure 4-3 Resource Allocation Reservation Table

Cycle	Resource					
	Port data_in	Multiplier 1	Multiplier 2	Adder/Subtractor	Port real_out	Port imaginary_out
1	a = data_in.read()					
2	b = data_in.read()					
3	c = data_in.read()					
4	d = data_in.read()	a x c	b x c			
5		a x d	b x d			
6				(a x c) - (b x d)		
7				(a x d) + (b x c)		
8					real_out.write()	imaginary_out.write()

Resource allocations are typically expressed as a reservation table. The columns represent individual components and the rows represent clock cycles. The location of an operation in the table indicates the component that performs the operation and the clock cycle when the operation is executed.

SystemC Compiler shares resources whenever it is beneficial to do so. Sharing resources means allowing the resource to accept inputs from multiple sources, and sharing may require additional multiplexers that can increase the overall area of the synthesized design. SystemC Compiler shares resources if it results in a reduction of the overall area of the synthesized design; otherwise, it does not share resources.

Inferred Registers

SystemC Compiler infers registers for the following behavioral constructs:

- Output ports

SystemC Compiler places a register immediately before each output port of the synthesized design. This ensures that the output data is held stable over the clock cycle when the outputs are asserted.

- Signals

Signals are used in a behavioral description to communicate between processes in the same design. Registers are used to implement signals.

- Variables

Variables that have data created in one clock cycle and used in a later clock cycle are assigned a register to hold the data. The duration when the data must be held is called the lifetime of the variable.

Variables can be defined in the behavioral description. In addition, SystemC Compiler automatically infers variables for the intermediate results of complex, single-line expressions. For example,

```
x = a + b + c
```

This expression has one variable x that is defined in the behavioral description. SystemC Compiler infers an additional variable for the result of $a + b$. If the intermediate result has a lifetime beyond a clock cycle, SystemC Compiler also assigns a register to store the result.

Register Sharing

Registers that are inferred for output ports and signals are dedicated registers. Registers that are inferred to hold variables can be shared between variables, similar to components that are shared by operations.

If two variables have lifetimes that do not overlap, a single register can be used to hold both variables. When the data of one variable becomes irrelevant before the data of a second variable becomes relevant, their lifetimes do not overlap. The irrelevant data no longer needs to be stored in the design, and the data can be overwritten with the relevant data when it becomes available.

SystemC Compiler performs lifetime analysis on each variable to determine if registers can be shared. Variable lifetime is measured from the first clock cycle when it is produced to the last clock cycle in which it is used.

Figure 4-4 shows a reservation table representation of register sharing for the complex number multiplier. Notice how the variable lifetimes are represented and how the variables with non-overlapping lifetimes share the same register.

Figure 4-4 Register Allocation Reservation Table

Cycle	Registers				
	R1	R2	R3	R4	R5
1					
2	a				
3		b			
4			c		
5			d	v1	v2
6		v3	v4		
7	v5				
8		v6			

Automatically generated variables for intermediate results:

$$v1 = (a \times c)$$

$$v2 = (b \times c)$$

$$v3 = (a \times d)$$

$$v4 = (b \times d)$$

$$v5 = v1 - v4 = (a \times c) - (b \times d)$$

$$v6 = v2 + v3 = (a \times d) + (b \times c)$$

SystemC Compiler shares registers when sharing reduces the overall area of the synthesized design.

Controller (FSM) Generation

When SystemC Compiler shares a component or register, it automatically inserts multiplexers at their inputs, if needed. A multiplexer is inserted if the component or register needs to accept its inputs from multiple sources. Figure 4-5 shows a shared

component, and Figure 4-6 shows a shared register. In both cases, each port gets its input from one of two possible sources. A multiplex is inserted on each port to enable this switching.

Figure 4-5 Shared Component

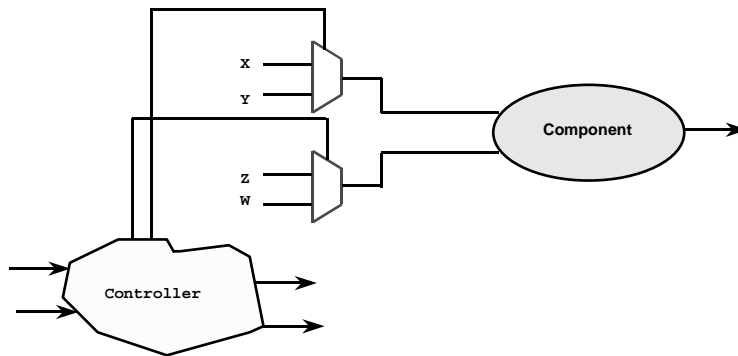
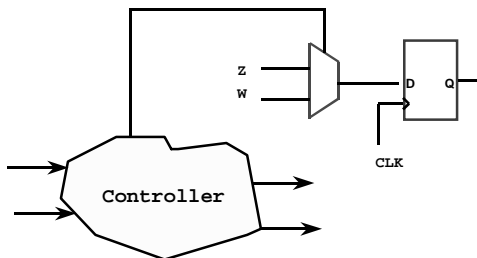
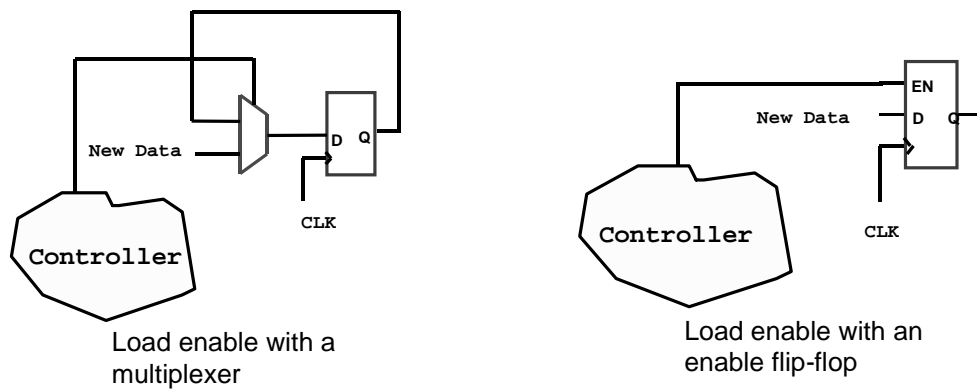


Figure 4-6 Shared Register



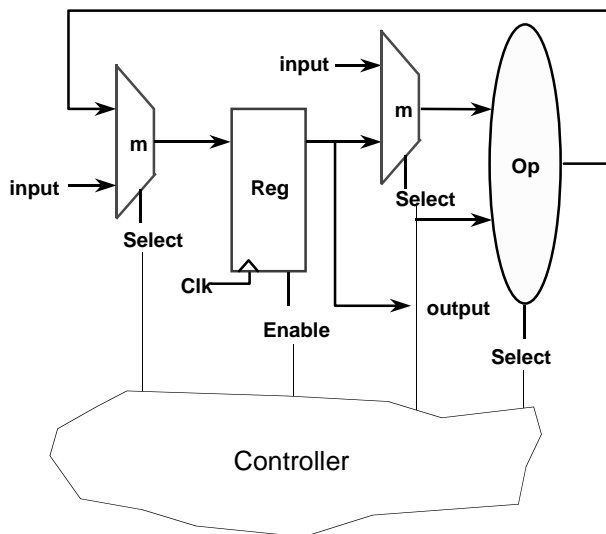
SystemC Compiler synthesizes a controller, in the form of a finite-state machine (FSM), that supplies the multiplexer signals to correctly switch multiplexers at the appropriate clock cycles. The controller is also used to generate the control signals for the components and the registers. Figure 4-7 shows how the controller might supply the control signal for a multiplexer and the load enable signal for a register.

Figure 4-7 FSM Control Signals



The final synthesized design has a data path that contains a netlist of components, multiplexers, registers, and an FSM to control the data path. Figure 4-8 shows a representative fragment of the synthesized design.

Figure 4-8 Synthesized Design Representation



Controlling Synthesis

By default, SystemC Compiler performs scheduling and allocation to minimize design latency and area. You can control the synthesis to better achieve your design objectives. Constraining scheduling and resource allocation is described in more detail in “Adding Scheduling Constraints” on page 4-31 and “Constraining Resource Allocations” on page 4-55.

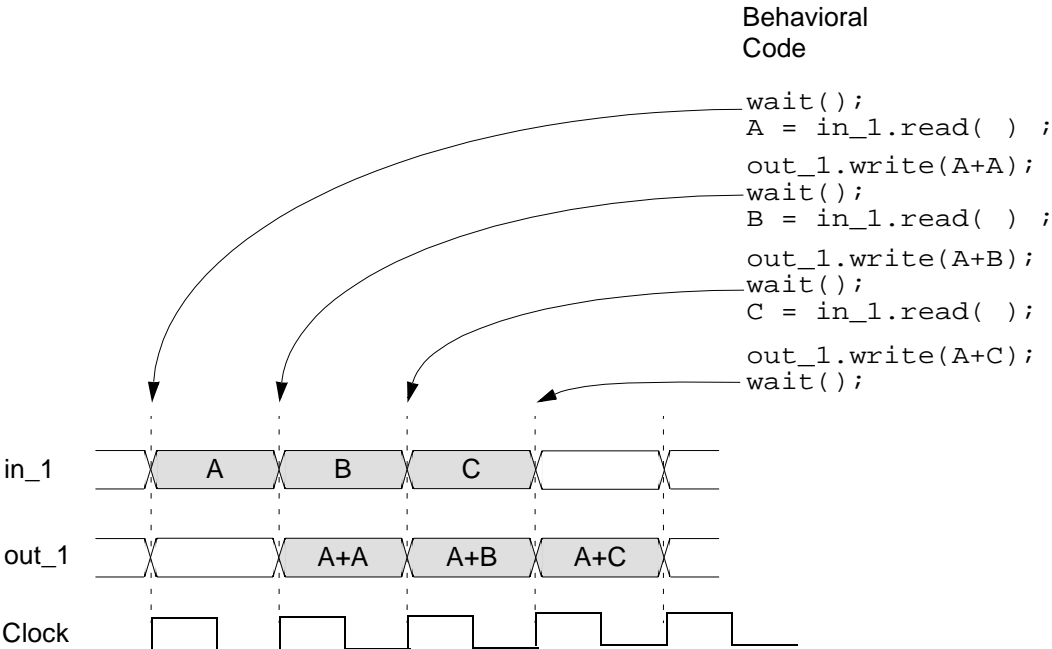
Selecting an I/O Scheduling Mode

For scheduling, SystemC Compiler considers every port or signal read or write statement in the behavioral code to be an I/O operation. Variables are considered to be local to a process. Variable read and write statements are not considered as I/O.

I/O operations are special in that they are the design’s interface to external design modules and testbenches.

In the behavioral code, the `wait()` and `wait_until()` statements delineate clock cycles. All input ports are read during a clock cycle, while the data from the output ports appear at the active edge of the next clock cycle, as illustrated in Figure 4-9. This figure shows simple behavioral code statements, the clock (in this case, a positive-edge sensitive design is assumed), and the resulting I/O operations.

Figure 4-9 Behavioral Code and I/O Operation



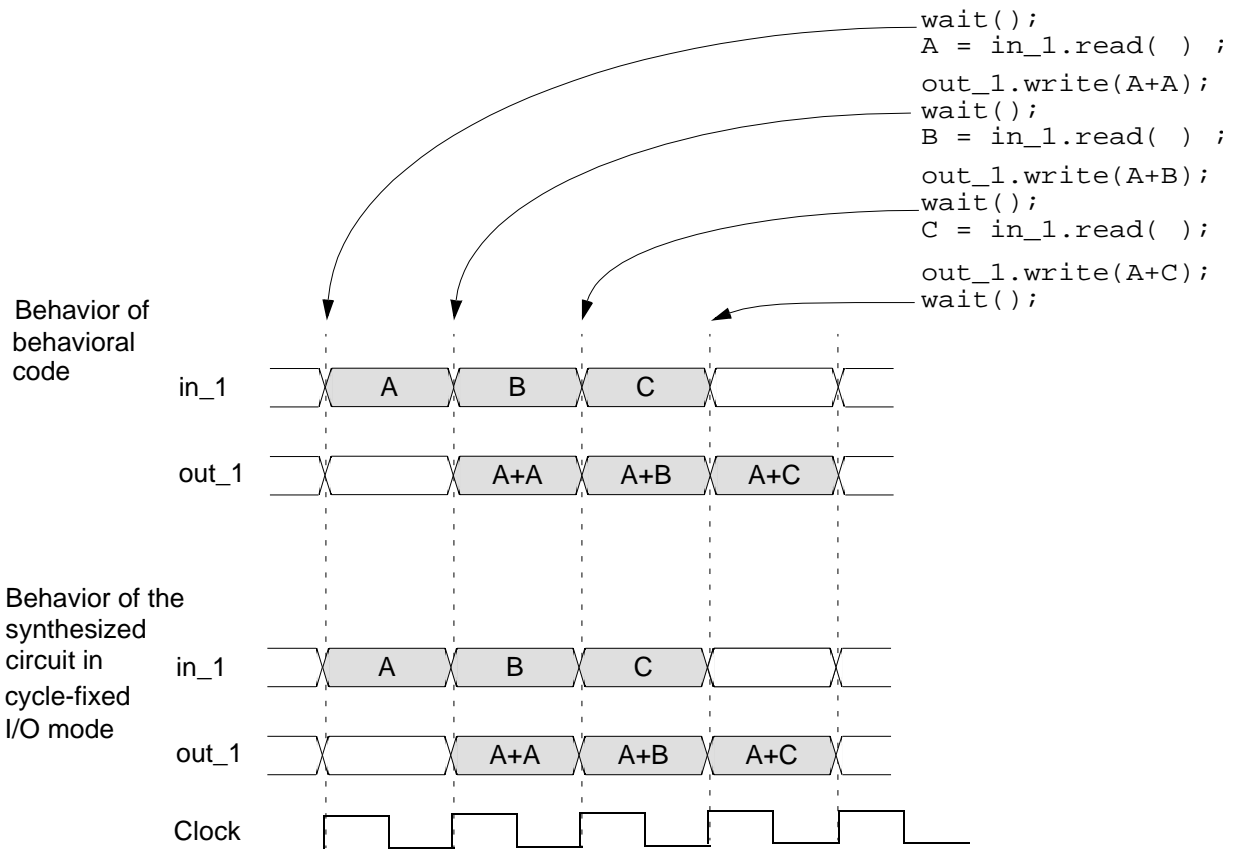
Inputs must be stable during the entire clock cycle so the synthesized circuit behavior matches the original behavioral description.

SystemC Compiler provides cycle-fixed and superstate-fixed modes for scheduling I/O operations. You need to set an I/O scheduling mode for the `bc_check_design` and `schedule` commands.

Cycle-Fixed I/O Scheduling Mode

In cycle-fixed I/O scheduling mode, SystemC Compiler preserves the cycle-to-cycle behavior of I/O as defined in the behavioral description, as shown in Figure 4-10.

Figure 4-10 Cycle-Fixed I/O Mode



Using Cycle-Fixed I/O Scheduling Mode

The goal of cycle-fixed I/O scheduling is to allow simulation of the SystemC behavioral description and the synthesized design side by side with no differences in observed I/O behavior. Cycle-fixed mode is a good choice when you want to specify cycle-accurate behavior

and you are confident that the operations between I/O can be completed in the number of cycles that you specify. The cycle-fixed I/O mode gives you complete control over the I/O schedule so you can use the same testbench for behavioral simulation and for verifying the results of synthesis.

In cycle-fixed I/O scheduling mode, I/O operations are constrained to occur in the same cycle in the synthesized design as in the original behavioral description. The operations required to compute the outputs from the inputs must be completed in the number of cycles between the inputs and outputs that are specified in the source code, but SystemC Compiler determines the exact clock cycle in which each operation is performed.

Cycle-fixed I/O scheduling is appropriate, for example, if input data always arrives at a fixed frequency and the output is required in a fixed number of cycles from the input arrival. Specify the I/O schedule in the behavioral description, and direct SystemC Compiler to use cycle-fixed mode to schedule the design.

In cycle-fixed I/O scheduling mode,

- Each `wait()` statement generates a clock cycle.
- Signal read and write operations remain in the cycle where the source code defines them.
- Operations that are not I/O can float from cycle to cycle as allowed by data and control dependencies, and by constraints.
- If the operations that compute an output from the input data cannot be accommodated within the number of clock cycles between the input and output, SystemC Compiler issues an error message.

For coding rules and recommendations about coding style using this mode, see Chapter 3, “Behavioral Coding Guidelines” in the *CoCentric™ SystemC Compiler Behavioral Modeling Guide*.

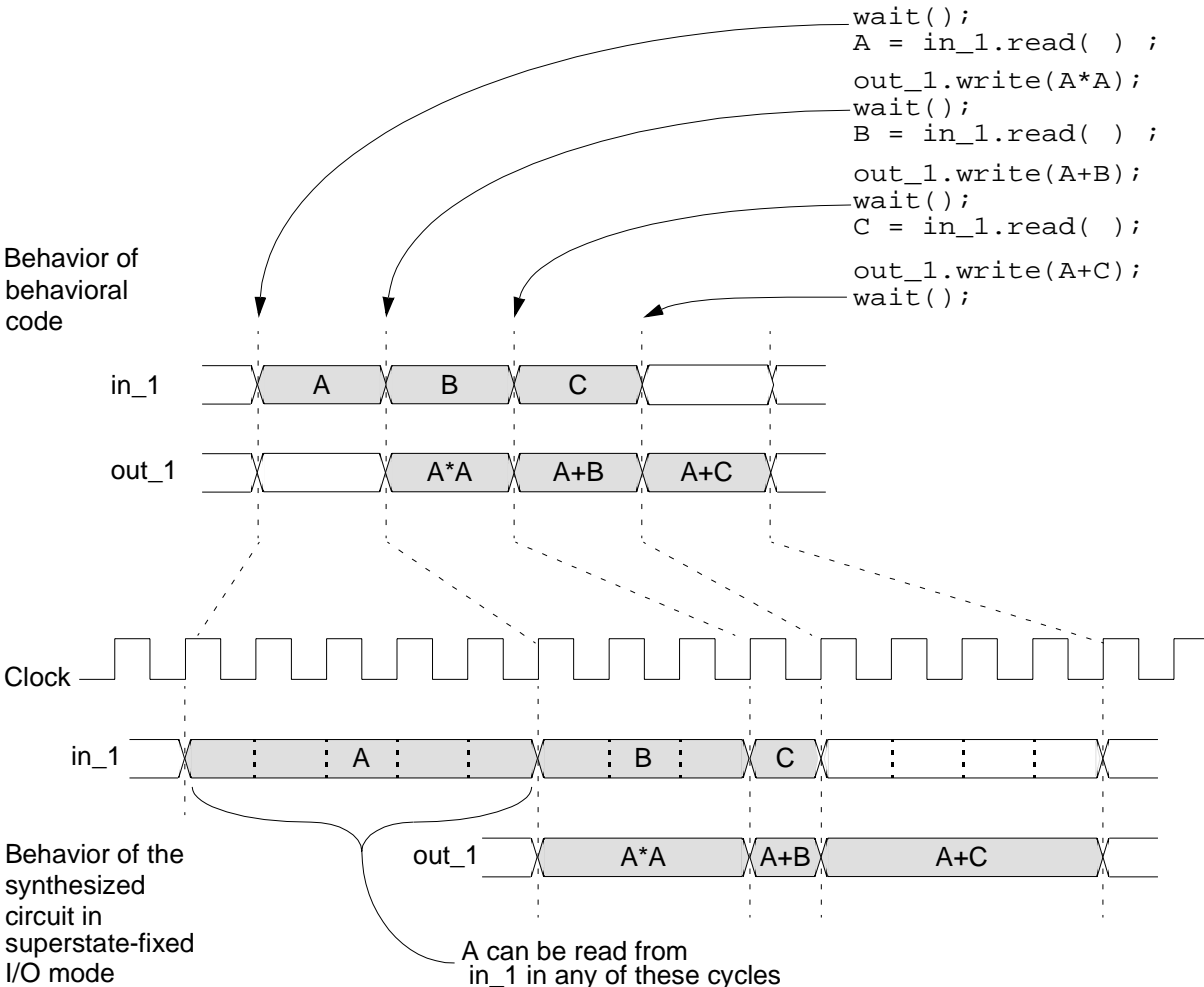
Superstate-Fixed I/O Scheduling Mode

In superstate-fixed mode, SystemC Compiler preserves the logical relationship of read and write operations, but it can add clock cycles to lengthen the time between I/O operations. Figure 4-11 shows an example where SystemC Compiler takes five clock cycles to read A and perform the multiplication, three to read B and perform the addition, and so forth. Therefore, the I/O schedule differs from the original behavioral code because several clock cycles are added.

The segment between two wait() statements in the behavioral description is called a superstate. This segment executes in one clock cycle in the behavioral simulation, but may execute in several clock cycles in the synthesized design.

I/O operations that occur between a pair of consecutive wait() statements in the behavioral description belong to the same superstate. Input reads can be scheduled at any clock cycle in the superstate, but outputs appear after the last clock cycle of the superstate. Notice that A*A in Figure 4-11 appears at the output at the end of the superstate.

Figure 4-11 Superstate-Fixed I/O Mode



Using Superstate-Fixed I/O Scheduling Mode

The superstate-fixed I/O scheduling mode is useful for specifying the sequence of I/O operations while retaining some flexibility in the length of the schedule. In some cases this allows you to change the length of the schedule to minimize the hardware required for implementing the design.

Use superstate-fixed mode when latency-based design exploration is important for your design. It allows you to quickly perform clock period, latency, and resource tradeoffs without modifying the behavioral description.

Use `wait()` statements in superstate-fixed I/O scheduling mode to segment the process into superstates, as illustrated in Figure 4-11 on page 4-15.

Because a superstate corresponds to multiple clock cycles, you cannot determine the exact cycle when the inputs are read; the exact cycle can only be determined after SystemC Compiler schedules the design. When you use superstate-fixed scheduling mode, use handshaking for all data transfers to and from the circuit.

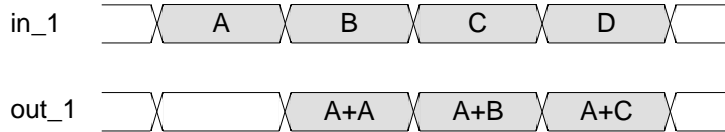
For coding rules about this mode, recommendations about placing clocks in your SystemC source code, and details about handshaking, see Chapter 3, “Behavioral Coding Guidelines” and Chapter 6, “Using Handshaking in the Circuit and Testbench” in the *CoCentric™ SystemC Compiler Behavioral Modeling Guide*.

Comparing the I/O Scheduling Modes

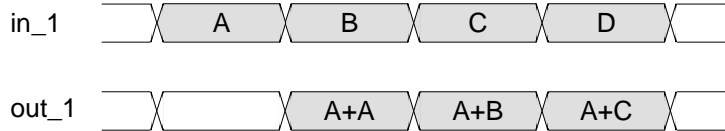
Figure 4-12 shows a comparison of the simulation of the original SystemC code with simulations of the possible results of SystemC Compiler when using different I/O scheduling modes.

Figure 4-12 Source Code and I/O Scheduling Mode Simulation

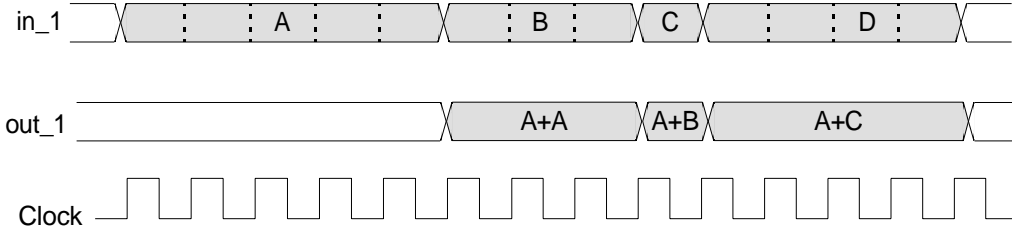
Original Source Code



Cycle-Fixed Mode



Superstate-Fixed Mode



Performing Scheduling

After you select an appropriate scheduling I/O mode, perform scheduling by executing the `schedule` command. Enter

```
dc_shell> schedule
```

By default, SystemC Compiler schedules with the cycle-fixed I/O mode. If you want to schedule with the superstate-fixed I/O mode, enter

```
dc_shell> schedule -io_mode superstate_fixed -effort medium
```

Scheduling Objectives

By default, the `schedule` command makes tradeoffs to achieve a design schedule with the fastest latency as the top priority, and it tries to create the smallest area as a secondary priority. To change the default scheduling priorities, apply scheduling constraints before using the `schedule` command.

Using Timing-Constrained Scheduling

With timing-constrained scheduling, SystemC Compiler minimizes the latency of the design in the superstate-fixed I/O mode. In the cycle-fixed I/O mode, the latency is determined by the number of `wait()` statements in the SystemC description.

Latency is defined as the number of clock cycles required to execute one iteration of a loop or to execute the set of operations.

In timing-constrained scheduling, SystemC Compiler does the following:

- Calculates the minimum number of cycles required to execute the loop or set of operations. The number of cycles is the minimum latency.
- Minimizes the hardware required to achieve the minimum latency

This scheduling technique is most beneficial for designs that have short latency requirements. Short latency is achieved by creating a parallel design implementation. This may require more area than a minimum area design.

Using Resource-Driven Scheduling

You can direct SystemC Compiler to perform resource-driven scheduling instead of timing-constrained scheduling. When performing resource-driven scheduling, SystemC Compiler minimizes hardware resources such as adders, multipliers, multiplexers, and registers while trading off latency. Because this changes the latency that is specified in the behavioral description, this method can be used only with the superstate-fixed I/O mode.

For this type of scheduling, SystemC Compiler increases loop latency. Increasing the latency of a loop distributes operations across more cycles, allowing greater resource sharing. Because fewer resources are allocated, this type of scheduling reduces area.

To implement resource-driven scheduling, use the `-extend_latency` option with the `schedule` command.

Analyzing the Scheduling Report

After you run the `schedule` command, use the `report_schedule` command to generate a scheduling report. For example,

```
dc_shell> schedule -io_mode superstate_fixed
              -effort medium
dc_shell> report_schedule
```

Schedule Summary Report

Example 4-1 shows a report generated by the `report_schedule` command for the complex number multiplier.

In the schedule report in Example 4-1,

- The timing summary indicates the latency in clock cycles for one loop iteration. In this example, the entry loop takes 8 cycles, the loop beginning on line 17 takes 7 cycles, and the loop beginning on line 22 takes 1 cycle.
- The report also indicates that the loop beginning on line 22 has a `continue` statement that occurs in cycle 3 and a loop exit that may occur in cycle 2.
- The area summary shows the total estimated design area and the FSM summary (number of control state, basic transitions, control inputs, and control outputs).
- The resource types show the total number of registers in the design, the number of operations, the associated synthetic library cell, and the I/O ports. The bit-width of each resource type is included in the report.

Example 4-1 Schedule Report Summary

```
/******report_schedule*****/

*****
Date       : Wed Nov  8 13:18:51 2000
Version    : 2000.11-PROD
Design     : cmult_hs
*****

*****
* Summary report for process entry: *
*****

-----
Timing Summary
-----

Clock period 20.00
Loop timing information:
  entry.....8 cycles (cycles 0 - 8)
  loop_17.....7 cycles (cycles 1 - 8)
  loop_22.....1 cycle (cycles 2 - 3)
    (continue) skip_short_branch_1..... (cycle 3)
    (exit) EXIT_L22_1..... (cycle 2)

-----
Area Summary
-----

Estimated combinational area    6127
Estimated sequential area      1734
TOTAL                          7861

9 control states
11 basic transitions
2 control inputs
7 control outputs

-----
Resource types
-----

Register Types
=====
8-bit register.....3
16-bit register.....1

Operator Types
=====
(8_8->16)-bit DW02_mult.....2
(16_16->16)-bit DW01_add.....1
(16_16->16)-bit DW01_sub.....1
```

I/O Ports

```
=====
1-bit input port.....1
1-bit registered output port.....2
8-bit input port.....1
16-bit registered output port.....2
-----
```

Schedule Report of Operations

The `-operations` option reports the scheduling and allocations of operations in a reservation table format. Enter

```
dc_shell> report_schedule -operations
```

Example 4-2 on page 4-24 shows a report of the scheduled operations for the complex number multiplier. It reports all I/O operations, arithmetic operations, and loops that are scheduled in the design. SystemC Compiler reports loops in the design as resources to show the clock cycle in which they are scheduled.

In the reservation table, resources are listed in the horizontal axis, and the clock cycles in which the resources are used are listed in the vertical axis. Operations are placed within the reservation table in the row that corresponds to the clock cycle in which they are executed, and the column that corresponds to the resource that executes it.

In the reservation table, resource type and operation names are abbreviated. The abbreviations are expanded in the report as follows

- Rn means to read from an I/O resource at line number n in the description. For example, the report in Example 4-2 shows in cycle 2 that the `new_data` port is read according to line 22 in the description. This is indicated by `R22` in the row labeled 2 and the column labeled `p5`.
- Wn means to write to an I/O resource at line number n in the description. In cycle 3 of Example 4-2, the output is written to the `ready_for_data` port in line 23 in the description. This is indicated by `W23` in the row labeled 3 and the column labeled `p2`.
- on means an arithmetic operation is performed at line number n in the description. In cycle 5 of Example 4-2, the multiply operation in line 35 is performed with `DW02_mult`. This is indicated by `o35` in the row labeled 5 and the column labeled `r407`.
- Ln means a loop begin, loop end, loop continue, or loop exit boundary. A loop begin means the cycle in which the loop starts, and a loop end means the cycle in which the loop ends. A loop continue means the cycle in which the decision is made to branch back for the next iteration of the loop. A loop exit means the cycle in which the decision is made to exit the loop and jump to the loop end cycle. The number after the loop is a sequential number assigned by SystemC Compiler, and it indicates a loop boundary abbreviation rather than a line number in the description. In Example 4-2, `L6` is the beginning of the loop specified on line 17 in the description, `L7` is the end of this loop, and `L8` is the loop continue boundary for this loop, and `L9` is the loop exit decision. Notice that `L6` loop begin and `L9` exit decision are in cycle 2. The `L8` loop continue decision and the `L7` loop end decision are in cycle 3.

Example 4-2 Report Schedule Operations

```
*****
* Operation schedule of process entry: *
*****
```

```
Resource types
=====
loop.....loop boundaries
p0.....16-bit registered output port imaginary_out
p1.....1-bit registered output port output_data_ready
p2.....1-bit registered output port ready_for_data
p3.....16-bit registered output port real_out
p4.....8-bit input port data_in
p5.....1-bit input port new_data
r35.....(8_8->16)-bit DW02_mult
r120.....(16_16->16)-bit DW01_add
r350.....(16_16->16)-bit DW01_sub
r407.....(8_8->16)-bit DW02_mult
```

```

                                     D      D
                                     W      W
                                     0      0
                                     2      2
                                     -      -
           p      p      -      -      m      m      p      p      p      p
           o      o      a      s      u      u      o      o      o      o
           r      r      d      u      l      l      r      r      r      r
           t      t      d      b      t      t      t      t      t      t

```

cycle	loop	p4	p5	r120	r350	r407	r35	p0	p1	p2	p3
0	..L0..W14.	.W12.	.W11.	.W13.
1	..L3..W20.	.W19.
2	..L9..R22.
	..L6..
3	..L8..	.R26.W23.
	..L7..
4R28.
5R30.o35..
6R32.o35b.	.o35a.
7o36..o36b.	.o36a.	.W36.	.W37.W35.
8	..L5..
	..L4..
	..L2..
	..L1..

Operation name abbreviations

=====

L0.....loop boundaries entry_design_loop_begin
L1.....loop boundaries entry_design_loop_end
L2.....loop boundaries entry_design_loop_cont
L3.....loop boundaries loop_17/loop_17_design_loop_begin
L4.....loop boundaries loop_17/loop_17_design_loop_end
L5.....loop boundaries loop_17/loop_17_design_loop_cont
L6.....loop boundaries loop_17/loop_22/loop_22_design_loop_begin
L7.....loop boundaries loop_17/loop_22/loop_22_design_loop_end
L8.....loop boundaries loop_17/loop_22/loop_22_design_loop_cont
L9.....loop boundaries loop_17/loop_22/EXIT_L22_1
R22.....1-bit read loop_17/loop_22/new_data_22
R26.....8-bit read loop_17/data_in_26
R28.....8-bit read loop_17/data_in_28
R30.....8-bit read loop_17/data_in_30
R32.....8-bit read loop_17/data_in_32
W11.....1-bit write ready_for_data_11
W12.....1-bit write output_data_ready_12
W13.....16-bit write real_out_13
W14.....16-bit write imaginary_out_14
W19.....1-bit write loop_17/ready_for_data_19
W20.....1-bit write loop_17/output_data_ready_20
W23.....1-bit write loop_17/ready_for_data_23
W35.....16-bit write loop_17/real_out_35
W36.....16-bit write loop_17/imaginary_out_36
W37.....1-bit write loop_17/output_data_ready_37
o35.....(8_8->16)-bit MULT_TC_OP loop_17/mul_35
o36.....(16_16->16)-bit ADD_TC_OP loop_17/add_36
o35a.....(8_8->16)-bit MULT_TC_OP loop_17/mul_35_2
o35b.....(16_16->16)-bit SUB_TC_OP loop_17/sub_35
o36a.....(8_8->16)-bit MULT_TC_OP loop_17/mul_36
o36b.....(8_8->16)-bit MULT_TC_OP loop_17/mul_36_2

Schedule Report of Variables

To generate useful reports of variables, operations, and abstract FSM, use `report_schedule` command options.

The `-variables` option reports the scheduled lifetimes of variables and register allocations in a reservation table format. Enter

```
dc_shell> report_schedule -variables
```

Example 4-3 shows a report of the variables from the complex number multiplier. It lists the storage resources. These are automatically generated registers. It also lists the variables that are stored in the registers and the cycles in which a variable occupies a register. Variable names are abbreviated in the reservation table, for example `v0`, and are expanded in the “Data value name abbreviations” section of the report where `v0` means the output of the multiplication at line 35 (`mul_35/Z`).

Variables in the scheduling report do not necessarily match the variables in the original behavioral description. SystemC Compiler introduces variables for data that needs to be stored across several clock cycles, and it removes variables specified in the behavioral description if its lifetime does not span clock edges.

Example 4-3 Report Schedule Variables

```
*****
* Register usage of process entry: *
*****
```

```
Storage resource types
=====
r357.....8-bit register
r428.....16-bit register
r1271.....8-bit register
r1272.....8-bit register
```

cycle	r428	r1271	r357	r1272
	(16)	(8)	(8)	(8)
0
1
2v5...
3v2...
4v2...	..v3...
5	..v0...	..v2...	..v3...
6	..v1...	..v2...	..v3...	..v4...
7
8

Data value name abbreviations

```
=====
v0.....16-bit data value loop_17/mul_35/Z
v1.....16-bit data value loop_17/sub_35/Z
v2.....8-bit data value loop_17/data_in_26/net
v3.....8-bit data value loop_17/data_in_28/net
v4.....8-bit data value loop_17/data_in_32/net
v5.....1-bit data value loop_17/loop_22/U2/Z
```

```
*****
```

Schedule Report of the FSM

The `-abstract_fsm` option reports the FSM generated by SystemC Compiler in a state table format. Enter

```
dc_shell> report_schedule -abstract_fsm
```

Example 4-4 on page 4-29 shows the scheduled abstract FSM report for the complex number multiplier.

The state table is a textual representation of the FSM's state diagram. Each row corresponds to a state transition and includes

- The name of the present state is represented as `s_n_n`. The state name is automatically created by SystemC Compiler.
- The name of the branch condition that transitions out of the present state. The branch conditions are described after the state table.
- The next state that the FSM reaches when it executes the transition.
- The actions column lists the actions that are executed by the synthesized design if this transition is performed. The action name `a_n` is automatically created by SystemC Compiler. The description of the action provides the operation being executed in this transition and the associated line of code. For example, Example 4-4 shows the `s_0_0` present state and lists the following four actions that are executed in this state:
 - `a_6 imaginary_out_14 (write)`, I/O write to port `imaginary_out` on line 14 of the behavioral description
 - `a_10 output_data_ready_12 (write)`, I/O write to port `output_data_ready` on line 12 of the behavioral description

- a_15 ready_for_data_11 (write), I/O write to port ready_for_data on line 11 of the behavioral description
- a_19 real_out_13 (write), I/O write to port real_out on line 13 of the behavioral description

Example 4-4 Report Schedule Abstract FSM

```

*****
* State table style report for process entry: *
*****
=====
present      next
state input  state      actions
-----
s_0_0   c1      s_0_1
          a_6  imaginary_out_14 (write)
          a_10 output_data_ready_12 (write)
          a_15 ready_for_data_11 (write)
          a_19 real_out_13 (write)
s_0_1   c2      s_1_2
          a_9  loop_17/output_data_ready_20
              (write)
          a_14 loop_17/ready_for_data_19
              (write)
s_1_2   c3      s_1_4
          a_0  loop_17/data_in_26 (read)
          a_13 loop_17/ready_for_data_23
              (write)
s_1_2   c5      s_1_4
          a_4  loop_17/loop_22/new_data_22
              (read)
s_1_2   c6      s_2_3
          a_4  loop_17/loop_22/new_data_22
              (read)
s_1_4   c7      s_1_5 a_1  loop_17/data_in_28 (read)
s_1_5   c8      s_1_6 a_2  loop_17/data_in_30 (read)
          a_24 loop_17/mul_35
              (operation)
s_1_6   c9      s_1_7 a_3  loop_17/data_in_32 (read)
          a_27 loop_17/mul_35_2
              (operation)
          a_35 loop_17/sub_35 (operation)
s_1_7   c10     s_1_8
          a_5  loop_17/imaginary_out_36
              (write)
          a_8  loop_17/output_data_ready_37

```

```

        (write)
        a_18 loop_17/real_out_35 (write)
        a_21 loop_17/add_36 (operation)
        a_30 loop_17/mul_36 (operation)
        a_33 loop_17/mul_36_2
            (operation)
s_1_8  c11      s_1_2  a_9  loop_17/output_data_ready_20
            (write)
        a_14 loop_17/ready_for_data_19
            (write)
s_2_3  c12      s_1_4  a_0  loop_17/data_in_26 (read)
        a_13 loop_17/ready_for_data_23
            (write)
s_2_3  c13      s_1_4  a_4  loop_17/loop_22/new_data_22
            (read)
s_2_3  c14      s_2_3  a_4  loop_17/loop_22/new_data_22
            (read)
+++++  c15      s_0_0  a_6  imaginary_out_14 (write)
        a_10 output_data_ready_12
            (write)
        a_15 ready_for_data_11 (write)
        a_19 real_out_13 (write)

```

```

-----
*****          Branch Conditions          *****
-----
state  condition          source
-----
c1      true
c2      true
c3      (and true
        (branch 1 of conditional loop_17/loop_22/SPLIT_L22_1))
c5      (and true
        (branch 1 of conditional loop_17/loop_22/SPLIT_L22_1))
c6      (and true
        (not (branch 1 of conditional loop_17/loop_22/SPLIT_L22_1)))
c7      true
c8      true
c9      true
c10     true
c11     true
c12     (branch 1 of conditional loop_17/loop_22/SPLIT_L22_1)
c13     (branch 1 of conditional loop_17/loop_22/SPLIT_L22_1)
c14     (not (branch 1 of conditional loop_17/loop_22/SPLIT_L22_1))
c15     true
-----
=====

```


The branch conditions section of the report describes the branch conditions under which state transitions are made. Each branch condition is described with the combination of logical events that trigger it. For example, condition c6 happens when branch 1 of the conditional on line 22 of the behavioral description is not taken.

Adding Scheduling Constraints

You can constrain the clock cycles in which operations and loops are scheduled with the set cycles commands (`set_cycles`, `set_min_cycles`, and `set_max_cycles`). These commands allow you to control the number of clock cycles between two operations or loop boundaries.

Matching Cells to Operations and Loops

To constrain the number of cycles between a pair of operations or loops during synthesis, you need to specify the cells that indicate the operation or loop. This section describes how to determine the cell that corresponds to an operation or loop in your behavioral description.

Naming Conventions

SystemC Compiler uses a hierarchical naming convention when creating cells. Levels of hierarchy are loops and preserved functions. To identify an operation further, the line number in the source code is used. For example,

```
cmult_entry/loop_73/sub_88
```

where `cmult_entry` is the name of the process, within the process `loop_73` is the beginning of a loop at line 73 in the source code, and `sub_88` is the subtract operation that implements the subtract operation on line 88 in the source code.

To constrain two operations to be two cycles apart, use the `set_cycles` command, for example:

```
dc_shell> set_cycles 2 -from cmult_entry/loop_73/sub_88
           -to_end cmult_entry/loop_73/sub_88
```

Using Line Labels

If you use the default names of cells generated by SystemC Compiler, the `set_cycles` command is sensitive to the line numbers in the behavioral description. When you add or delete lines from the behavioral description, you will need to update the `set_cycles` command definitions.

Alternatively, you can add labels to lines that contain operations or loops that you want to constrain. SystemC Compiler then replaces the line number with your label in the cell name. This makes the cell names independent from their source code line numbers.

Use a C language label or SystemC Compiler `line_label` compiler directive to label lines in the behavioral description, and refer to these labels in constraints.

For example,

```
//C language label
C_label: y.write(a + b + c + d);

//SystemC Compiler line_label directive
y.write(a + b + c +d); // synopsys line_label my_label
```

Use either line label syntax so the cell name representing the output write operation to port y is named `y_my_label` instead of line number names such as `y_23`.

If more than one operation is on the same line that is labeled as `my_label`, a suffix such as `_2`, `_3`, and so forth are added to the cell names. In the example, the addition operations are `add_my_label`, `add_my_label_2`, `add_my_label_3`, and `add_my_label_4`.

To constrain two of the addition operations to be 5 clock cycles apart, use the following `set_cycles` command:

```
dc_shell> set_cycles 5 -from add_my_label -to add_my_label_3
```

Using Find

You can use the `find` command to locate cells in a design. The `find` command returns all the design or library objects that match the specified names.

```
dc_shell> find type {name_list}
           [-hierarchy]
           [-flat]
```

The *type* specifies the object to be found. The value of *type* can be `design`, `clock`, `port`, `reference`, `cell`, `net`, `pin`, `cluster`, `library`, `lib_cell`, `lib_pin`, `multibit`, `operator`, `module`, `implementation`, or `file`.

By default, the `find` command returns all objects that match the *type* and *name_list* in the current level of design hierarchy. Use the `-hierarchy` option to return all objects matching the *type* and *name_list* within all levels of hierarchy of the current design. If you use the `-hierarchy` option, the *type* must be `design`, `lib_cell`, `net`, `cell`, or `pin`.

```
dc_shell> find cell sub_35 -hier
```

You can limit the search to a particular operation by using a wildcard specification, for example:

```
dc_shell> find cell *mul* -hier
```

The `-flat` option specifies that the command finds only objects in the leaf cells. You need to also use the `-hierarchy` option when using the `-flat` option, for example

```
dc_shell> find cell *sub* -hier -flat
```

Reporting Hierarchy

You can also use the `report_hierarchy` command to show the design hierarchy. Enter

```
dc_shell> report_hierarchy
```

Example 4-5 shows a hierarchy report before scheduling. Each level of hierarchy is indented. In this example, the top-level hierarchy is `cmult_hs`, the second level is `entry_design`, and so forth.

Example 4-5 Report Hierarchy Before Scheduling

```
*****  
Report : hierarchy  
Design : cmult_hs  
Version: 2000.11-PROD  
Date   : Fri Dec 15 11:09:01 2000  
*****
```

Information: This design contains unmapped logic. (RPT-7)
Warning: 10 unresolved references are not included in this report. (RPT-2)

```
cmult_hs  
  entry_design  
    loop_17_design  
      loop_22_design  
        GTECH_BUF          gtech  
        GTECH_NOT         gtech  
        group1
```

Example 4-6 shows a hierarchy report after scheduling the same design.

Example 4-6 Report Hierarchy After Scheduling

```
*****  
Report : hierarchy  
Design : cmult_hs  
Version: 2000.11-PROD  
Date   : Fri Dec 15 11:10:49 2000  
*****
```

Information: This design contains unmapped logic. (RPT-7)

```
cmult_hs  
  GTECH_BUF                gtech  
  GTECH_NOT                gtech  
  GTECH_OR2                gtech  
  cmult_hs_fsm_block_dsg_0  
    GTECH_AND2            gtech  
    GTECH_NOT             gtech  
    GTECH_OR2             gtech  
    GTECH_OR3             gtech  
    GTECH_OR4             gtech  
    GTECH_OR5             gtech  
  cmult_hs_rd_9_0  
  cmult_hs_rd_1_0_0  
  cmult_hs_rd_1_1  
  cmult_hs_rd_8_0_0  
  cmult_hs_rd_8_1_0  
  cmult_hs_rd_8_2  
  cmult_hs_rd_16_0_0  
  cmult_hs_rd_16_1_0  
  cmult_hs_rd_16_2  
  group1_0
```

Constraining Loops and Operations

To constrain the latency through the body of a loop or to specify the latency between two operations in a loop, use one or more of the `set_cycles`, `set_min_cycles`, and `set_max_cycles` commands. These three commands use the same options to specify constraints, which are described in “Using the Set Cycles Commands and Options” on page 4-42

Place constraints on cells after running the `compile_systemc` command and before running `bc_check_design` or `schedule` commands.

Constraining Between Two Operations

To set a constraint of a fixed number of cycles between two operations, use the `set_cycles` command. To select the operations you want to constrain, place line labels on the lines of behavioral description that contain the operations, and use the `find` command before you use one of the set cycle commands.

For example, to set a constraint of 4 clock cycles between the two addition operations on lines line labels M and N in Example 4-7, use the following commands:

```
dc_shell> op1 = find -hier cell *top/add_M*
dc_shell> op2 = find -hier cell *top/add_N*
dc_shell> set_cycles 4 -from op1 -to op2
```

Example 4-7 Constraining Between Two Operations

```
void top(){
    ...
    wait();
    x = in_1.read();
    out1.write(calc_M(x)); //synopsys label M

    wait();
    x = in_1.read();
    out_2.write(calc_N(x)); //synopsys label N
    wait();
    ...
}
```

Constraining a Loop

The `schedule` command sets the minimum latency for each loop by default, which usually results in the largest area. You can define a larger cycle budget for a loop to maximize resource sharing. Example 4-8 shows a loop that SystemC Compiler can schedule in two cycles by using two adders.

Example 4-8 Constraining a Loop

```
void top(){
    ...
    wait();
    loop1: for (int i = 0; i < 4; i++){
    x = in_1.read() + in_2.read();
    wait();
    x = x + in_3.read();
    wait();
    out1.write(calc_M(x));}
    ...
}
```


You can force sharing of a single adder resource by setting a constraint of 4 cycles on the loop. Enter

```
dc_shell> loop1 = find -hier cell *top/loop1
dc_shell> set_cycles 4 -from_beginning loop1 -to_end loop1
```

Constraining Nested Loops

SystemC Compiler schedules designs from the innermost loop to the outermost loop. Constraints placed on the outermost loop in a loop hierarchy do not propagate to the inner loops. It is recommended that you constrain the inner loop first, then constrain the next outer loop, and so forth. Constrain the outermost loop last.

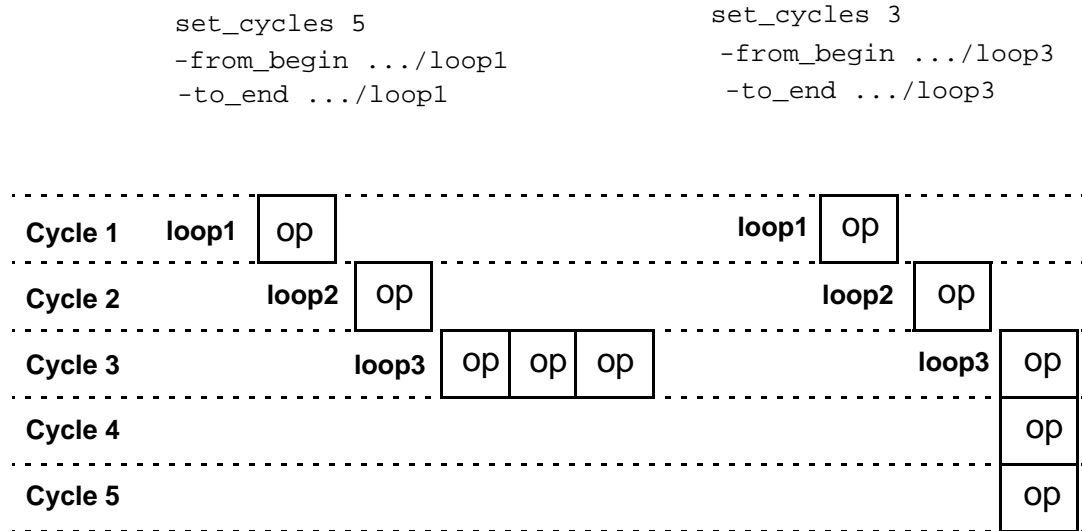
Example 4-9 shows a code fragment with three nested loops.

Example 4-9 Nested Loops With Operations

```
...
loop1: while (true){
    ...
    // 1 operation
    loop2: for (int i = 0; i < 4; i++){
        ...
        // 1 operation
        ...
        loop3: while (out_ready == 1){
            ...
            // 3 operations
        }
    }
}
```

Figure 4-13 illustrates the effects of setting different cycle constraints on the nested loops in Example 4-9.

Figure 4-13 Resources With Loops



On the left side of Figure 4-13, placing a constraint of 5 on the outermost loop (loop1) creates a design latency of five cycles, but it requires three resources for the design. The unconstrained innermost loop (loop3) dictates area, which is implemented in one cycle using three resources. The constraint on the outer loop does not propagate to the innermost loop. Therefore, this loop implements in the shortest possible latency, wasting two unused cycles in loop1.

On the right side of Figure 4-13, the innermost loop (loop3) has a loop constraint of three cycles. SystemC Compiler uses one resource and distributes the three operations over three cycles. This constraint on the innermost loop fully utilizes the five cycles intended for the design.

Placing Constraints Across Loop Boundaries

Infinite loops, while loops, and for loops form a level of hierarchy for SystemC Compiler, and it schedules each hierarchical unit separately. Therefore, operations in one loop cannot reference operations in other loops.

SystemC Compiler does not budget cycles across loop boundaries. Do not set constraints between an operation in one loop and an operation in another loop. Instead, do the following steps:

1. Place a constraint between the operation in a loop and the loop end.
2. Place another constraint between the end of the first loop and the beginning of the second loop.
3. Place a third constraint between the beginning of the second loop and the operation in the second loop.

The following commands show how to pass a constraint between loops for the code segments in Example 4-10.

```
dc_shell> set_cycles 2 -from cmult_entry/label1  
           -to_end cmult_entry/lp1  
dc_shell> set_cycles 1 -from_end cmult_entry/lp1  
           -to_beginning cmult_entry/lp2  
dc_shell> set_cycles 2 -from_beginning cmult_entry/lp2  
           -to cmult_entry/lp2/label2
```

Example 4-10 *Passing a Constraint Between Loops*

```
//SystemC

lp1: for (cond1) {
    .....
    label1: y.write(a + b);
    .....
}

wait( );

lp2: for (cond2) {
    .....
    label2: q.write(result);
    .....
}
```

Using the Set Cycles Commands and Options

The `set_cycles`, `set_min_cycles`, and `set_max_cycles` commands have the same options, described in the following sections. Use the `set_cycles` command to define a fixed number of cycles between two operations. Use the `set_min_cycles` to define a minimum number of cycles between two operations. Use the `set_max_cycles` to define a maximum number of cycles between two operations.

It is easy to overconstrain the schedule by improperly using these commands. For example, if the latency specified by `set_max_cycles` is less than that specified by `set_min_cycles`, scheduling fails. If this occurs when executing the `schedule` command, review the error messages and the constraints, and remove the constraints that make scheduling impossible.

You can set the following options for the set cycles commands:

```
[-process process_name]  
[cycle_offset]  
[[-from | -from_beginning | -from_end] from_operation]  
[[-to | -to_beginning | -to_end] to_operation]
```

The `-process` option specifies the process to which this command applies. Use this option if your behavioral description has multiple processes.

The `cycle_offset` option specifies the number of cycles by which you are constraining the two operations. The number must be a positive integer, and a negative integer is invalid. If you set the `cycle_offset` to zero, it implies that the two operation can happen in the same cycle.

The `-from` and `-from_beginning` options (functionally equivalent options) specify that the beginning of an operation is selected. In the case of a loop, select the first cycle of the loop. In the case of a multicycle operation, select the first cycle of the operation. In the case of a single-cycle operation, select the cycle of the operation.

The `-from_end` option specifies that the ending of an operation is selected. In the case of a loop, select the last cycle of the loop. In the case of a multicycle operation, select the last cycle of the operation.

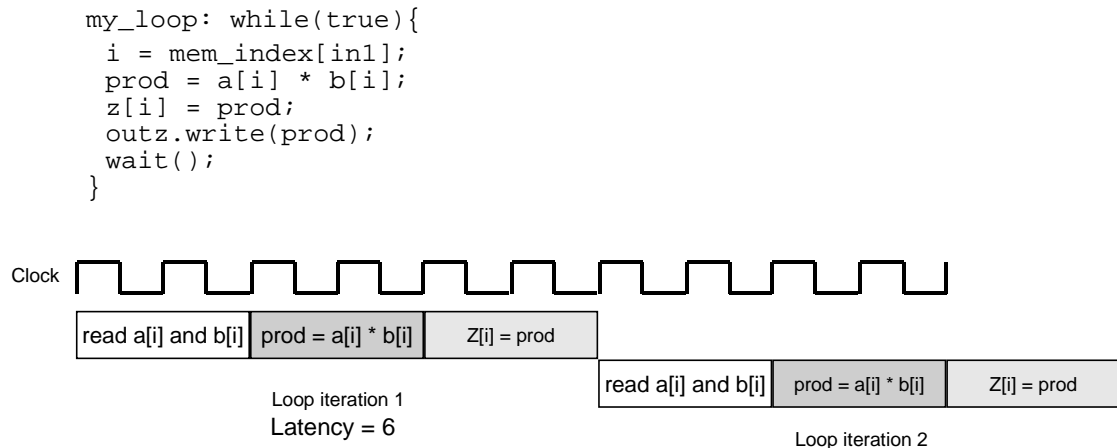
The `-to` and `-to_beginning` options (functionally equivalent options) specify the last cycle of an operation or loop boundary to constrain. In the case of a loop, select the first cycle of the loop. In the case of a multicycle operation, select the first cycle of the operation. In the case of a single-cycle operation, select the cycle of the operation.

The `-to_end` option specifies that the end of an operation is selected. In the case of a loop, select the last cycle of the loop. In the case of a multicycle operation, select the last cycle of the operation.

Pipelining a Loop

You can increase the throughput of your design by pipelining loops. When a loop is pipelined, SystemC Compiler synthesizes the design so that the loop iterations overlap when the synthesized design is operating. Pipelining loops reduces the overall runtime latency of the synthesized design. By default, loops are not pipelined. Figure 4-14 shows code for a loop with a latency of 6 clock cycles and its nonpipelined latency.

Figure 4-14 Nonpipelined Loop

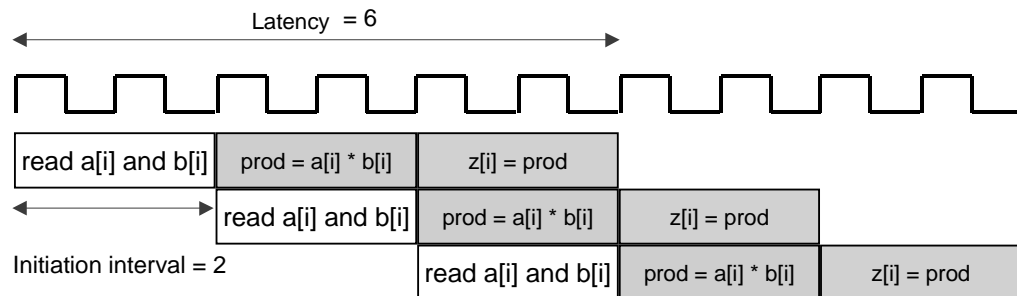


To pipeline a loop, use the `pipeline_loop` command so SystemC Compiler generates the required loop pipelining controls in the FSM during scheduling. The pipeline controls tasks such as filling and flushing the pipeline and overlapping loop iterations.

For the `pipeline_loop` command, you specify the loop name, the initiation interval, and the latency of the loop you want SystemC Compiler to pipeline. The initiation interval is the number of clock cycles until the start of the next loop iteration, and loop latency is the number of clock cycles required to complete one loop iteration.

Figure 4-15 shows the effect of pipelining of the same loop code from Figure 4-14 with an initial interval of 2 and a latency of 6.

Figure 4-15 Pipelined Loop



To pipeline a loop,

1. Schedule the design without pipelining to determine the loop latency, which is provided in the scheduling report timing summary.
2. Determine the initiation interval based on your behavioral description, as described in “Determining the Initiation Interval” on page 4-47.
3. Use the `pipeline_loop` command to specify the pipeline values.
4. Run the `schedule` command again and compare the results, or use BCView and look at the generated FSM.

Use the `find` command to extract the full loop path name. Enter the following commands to pipeline the loop and report the schedule:

```
dc_shell> loop_label = find -hier cell *calc_loop*
dc_shell> pipeline_loop loop_label
           -init_interval 2
           -latency 6
dc_shell> report_schedule -summary
```

Note:

In superstate-fixed scheduling mode, you can set the latency with either the `pipeline_loop` or the `set_cycle` command.

Example 4-11 shows the schedule summary report with pipelined loop information of 2 cycles for the initiation interval and 6 cycles for the pipeline latency for the `calc_loop`.

Example 4-11 Pipelined Loop Timing Summary (Partial)

```
-----
Timing Summary
-----
Clock period 20.00
Loop timing information:
entry.....8 cycles (cycles 0 - 8)
loop_17.....7 cycles (cycles 1 - 8)
  calc_loop.(initiation interval)...2 cycles
    (pipeline latency).....6 cycles
                                (cycles 1-7)
...

```

You can change the initiation interval and latency of pipelined loops to explore tradeoffs such as throughput and area. A smaller loop iteration means a higher loop throughput.

Restrictions and Limitations For Pipelining Loops

Pipelining a loop has the following restrictions:

1. The loop latency must be an integer multiple of the initiation interval. For example, a loop latency of 6 can have an initiation interval of 1, 2, or 3, and a loop latency of 10 can have an initiation interval of 1, 2, or 5.
2. A pipelined loop cannot contain other loops unless the nested loop is unrolled.
3. A loop exit is implicitly constrained to occur only within the initiation interval.

For further information, see Chapter 3, “Behavioral Coding Guidelines” in the *CoCentric™ SystemC Compiler Behavioral Modeling Guide*.

Determining the Initiation Interval

To determine the initiation interval, consider the design throughput requirements and the

- Loop carry dependencies
- Memory and I/O accesses
- Loops with handshake signals
- Exit from a pipelined loop

Loop Carry Dependencies

Loop carry dependencies are data values produced in one iteration of a loop that are consumed by a subsequent iteration. A loop carry dependency can restrict the initiation interval. Figure 4-16 shows a loop with a latency of 5 that produced a result in loop iteration 1, which is needed in iteration 2. The result from iteration 1 is not available until the end of cycle two. An initiation value of 1 is, therefore, not valid.

Figure 4-16 Invalid Loop Initiation Value

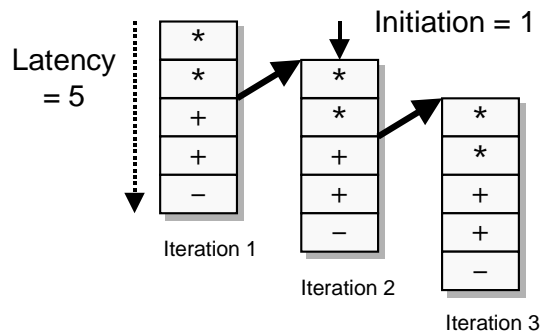
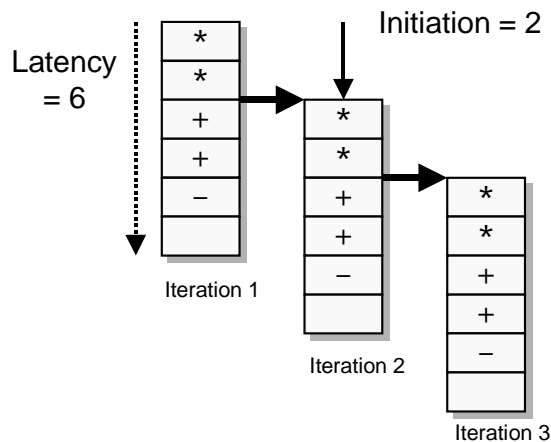


Figure 4-17 shows a corrected version of the loop with an initiation interval of 2. The loop latency is extended to six to satisfy the requirement that the loop initiation must be an integer multiple of the latency.

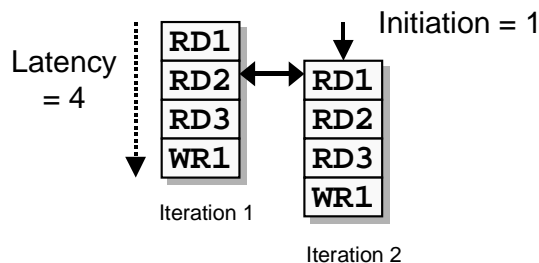
Figure 4-17 Valid Loop Initiation Value



Memory and I/O Accesses

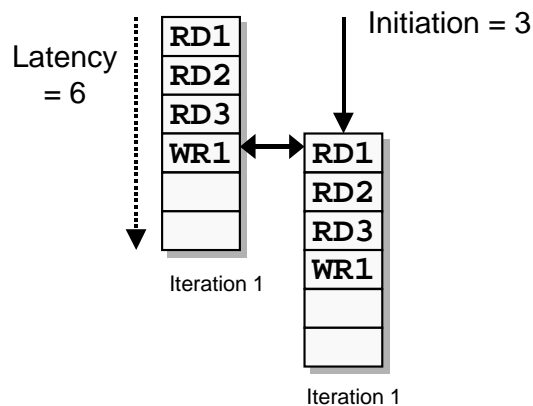
When you pipeline a loop, the original order specified in the behavioral description for reading and writing to the same memory, signal, or port is preserved. Simultaneous reading and writing to the same memory, signal, or port are not possible from different loop iterations. Figure 4-18 illustrates several reads (RD1, RD2, and RD3) from and a write (WR1) to the same memory. This loop cannot be pipelined, because it is attempting to simultaneously read from the same memory in different loop iterations.

Figure 4-18 Invalid Memory and I/O Access



If the memory has two ports (for example, a dual-port RAM), the pipelining shown in Figure 4-18 is valid. If WR1 and RD1 are accessing the same memory location, however, there is a loop-carry dependency from WR1 to RD1. In that case, the initiation interval must be changed to 3, as shown in Figure 4-19, to resolve the loop-carry dependency.

Figure 4-19 Valid Memory and I/O Access



Pipelining a Loop With Handshake Signals

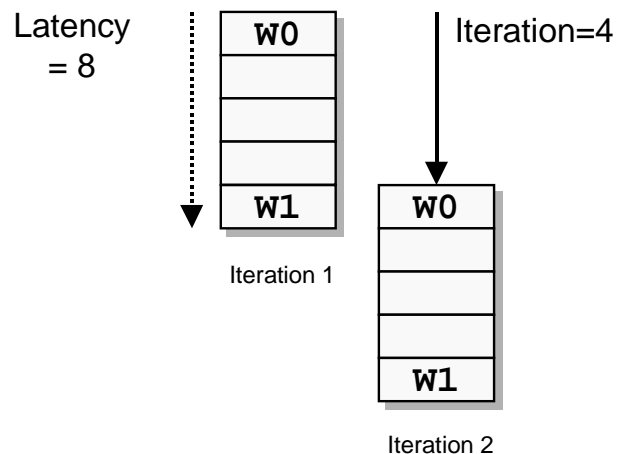
A loop with handshake signals can require modification of the loop code before you can pipeline the loop. Figure 4-20 shows a code example that contains a handshake signal `output_rdy`. A function call, which requires 4 cycles to execute, is between the two writes to the handshake signal. Before iteration 2, the `output_rdy` signal is already a 1. In iteration 2, however, the `output_rdy.write(0)` cannot occur until the iteration 1 `output_rdy.write(1)` has occurred. This loop, therefore, cannot be pipelined.

Figure 4-20 Handshake Signal Preventing Loop Pipelining

```

calc_loop : while (true){
  output_rdy.write(0);
  wait();
  result.write(func(in1));
  output_rdy.write(1);
  wait();
}

```



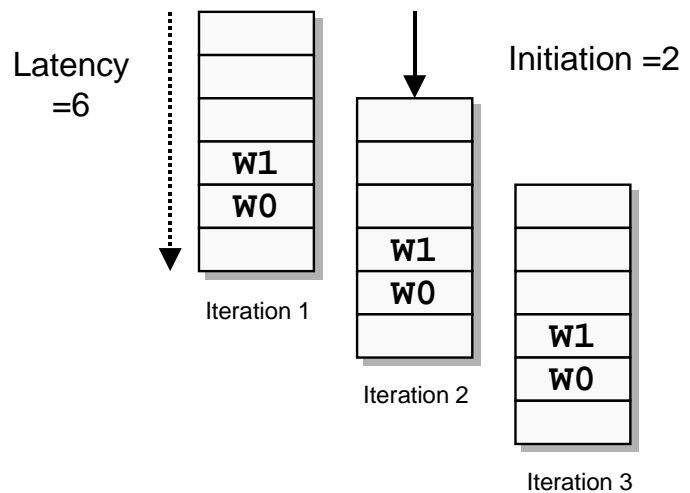
You can change the order of the code in a loop with handshake signals to change the latency and initiation interval. This allows the loop to be pipelined, and it improves the throughput. Figure 4-21 shows the reordered loop code from Figure 4-20. It moves the raising and lowering of the output_rdy signal closer together. The loop latency is extended to 6, allowing an initiation interval of 2.

Note:

In this situation, you might need to rewrite the code to enable pipelining. Changing just the initiation interval and latency will not enable pipelining.

Figure 4-21 Pipelined Loop With Handshake Signal

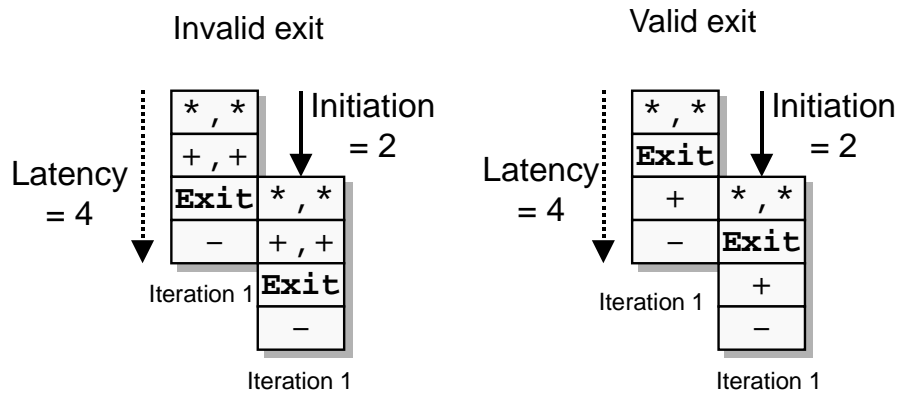
```
output_rdy.write(0);
wait();
calc_loop : while (true){
    result.write(func(in1));
    output_rdy.write(1);
    wait();
    output_rdy.write(0);
    wait();
}
```



Exit From a Pipelined Loop

A loop exit can occur only within the initiation interval of a pipelined loop, because the semantics of the behavioral description forbid the next iteration from being launched. To preserve the semantics, a check is made to determine if the current iteration is the last iteration. If it is the last iteration, the loop is exited before the next iteration begins. Figure 4-21 illustrates an invalid exit and a valid exit from a pipelined loop.

Figure 4-22 Exit From a Pipelined Loop



To exit a pipelined loop within the initiation interval, use a while loop with an implicit conditional exit or a for loop with an implicit conditional exit, because the exit condition of these loops are evaluated in the first cycle of the loop.

Determining Current Scheduling Constraints

To determine all scheduling constraints, use the `report_scheduling_constraints` command to display all SystemC Compiler scheduling constraints on the current design.

```
dc_shell> report_scheduling_constraints
```

This command reports constraints you set with the following commands:

- `set_cycles`
- `set_max_cycles`
- `set_min_cycles`
- `pipeline_loop`
- `preschedule`
- `chain_operations`
- `dont_chain_operations`

Removing Scheduling Constraints

If you want to remove scheduling constraints, use the `remove_scheduling_constraints` command to remove the explicit constraints.

```
dc_shell> remove_scheduling_constraints  
          [-process process_name]
```

If you do not specify a process, the default is all processes.

The `remove_scheduling_constraints` command removes constraints set with the following commands:

- `set_cycles`
- `set_max_cycles`
- `set_min_cycles`
- `set_min_cycles`
- `preschedule`
- `dont_chain_operations`

The `remove_scheduling_constraints` command does not affect timing constraints or constraints inferred from the source description.

Constraining Resource Allocations

SystemC Compiler shares resources whenever possible. You can constraint the amount of resource sharing.

Setting Common Resources

For your design, you might want to instruct SystemC Compiler to share resources. You can do this with the `set_common_resource` command. It provides control to reduce area by allowing you to specify the implementation of a set of operations on a given number of resources.

```
dc_shell> set_common_resource
    [-process process_name] {operation_names}
    [-min_count min_resources]
    [-max_count max_resources]
    [-force_sharing]
    [-exclusive]
```

The `-process` option specifies the process to which this command applies. The default is to apply the command to all behavioral processes in the current design.

The `-min_count` option specifies the minimum number of available resources for operations in *operation_names*. You can specify this option only in combination with the `schedule` command using its `-extend_latency` option. During resource driven scheduling, this prevents SystemC Compiler from increasing the latency of a design to the point where the design can be scheduled with only one resource of each type.

For example, the commands in Example 4-12 schedule the design so that cycles are added to allow the operations `add_2`, `add_27`, `add_33`, `sub_5`, and `sub_21` to be implemented on two resources.

Example 4-12 Commands for Minimum Resource-Driven Scheduling

```
dc_shell> ops = {"add_2" "add_27" "add_33" "sub5" "sub_21"}
dc_shell> set_common_resource ops -min_count 2
dc_shell> schedule -extend_latency
```

The `-max_count` option specifies the maximum number of available resources for operation in *operation_names*. SystemC Compiler terminates scheduling and issues an error message if it cannot find a schedule that uses resources fewer than or equal to the specified `max_count` value.

Example 4-13 shows the commands to schedule the design so that cycles are added to allow the operations `add_2`, `add_27`, `add_33`, `sub_5`, and `sub_21` to be implemented on three or fewer resources.

Example 4-13 Commands for Maximum Resource-Driven Scheduling

```
dc_shell> ops = {"add_2" "add_27" "add_33" "sub5" "sub_21"}
dc_shell> set_common_resource ops -max_count 3
dc_shell> schedule -extend_latency
```

The `-force_sharing` option is used in combination with the `-max_count` option. It forces SystemC Compiler to share the operations even if the cost functions indicate that this would increase the area of the design. Without this option, SystemC Compiler could disregard the `-max_count` option if the sharing results in a design with a larger area, for example if resource sharing introduces large multiplexers.

Example 4-14 shows the commands to schedule the design so that cycles are added to allow the operations `add_2`, `add_27`, `add_33`, `sub_5`, and `sub_21` to be implemented on three or fewer resources with forced sharing of resources.

Example 4-14 Commands for Forced Maximum Resource-Driven Scheduling

```
dc_shell> ops = {"add_2" "add_27" "add_33" "sub5" "sub_21"}
dc_shell> set_common_resource ops -max_count 3
           -force_sharing
dc_shell> schedule -extend_latency
```

The `set_common_resource` command affects the scheduling of operations. Operations grouped into common resources are scheduled in non-overlapping cycles to allow them to be shared on the same resource.

SystemC Compiler attempts to increase the number of cycles (latency) to meet specified resource goals, but these increases in cycles must not violate timing constraints. Timing constraints take priority over resource goals. If the resource goals are not met within the defined timing constraints, SystemC Compiler issues an error.

Setting Exclusive Registers

During your design, you might want a variable to remain in a single register at all times; for example, a variable that you later scan out. You can accomplish this by using a signal instead of a variable. In that case, SystemC Compiler creates a dedicated register to hold the signal value.

To control register sharing, use the `set_exclusive_use` command. When you apply this command to a variable, it directs SystemC Compiler to dedicate a single register to hold the variable.

After executing the `compile_systemc` command and before executing the `schedule` command, enter

```
dc_shell> set_exclusive_use variable_name
```

You can use the `-shared` option with the `set_exclusive_use` command to direct SystemC Compiler to force register sharing by assigning the variable to an existing register. Enter

```
dc_shell> set_exclusive_use variable_name -shared
```

If it is not possible to find a register with variables that do not overlap lifetimes with the specified variable, SystemC Compiler will not force sharing.

5

Optimizing Latency and Area

This chapter describes how to influence the optimization of latency and area to improve the quality of results produced by SystemC Compiler.

This chapter contains the following sections:

- Exploring Architectures and Improving the Quality of Results
- Controlling Operation and Implementation Selection
- Operation Chaining
- Removing Unnecessary Registers
- Using Multicycle Operations
- Using Preserved Functions
- Using DesignWare Components

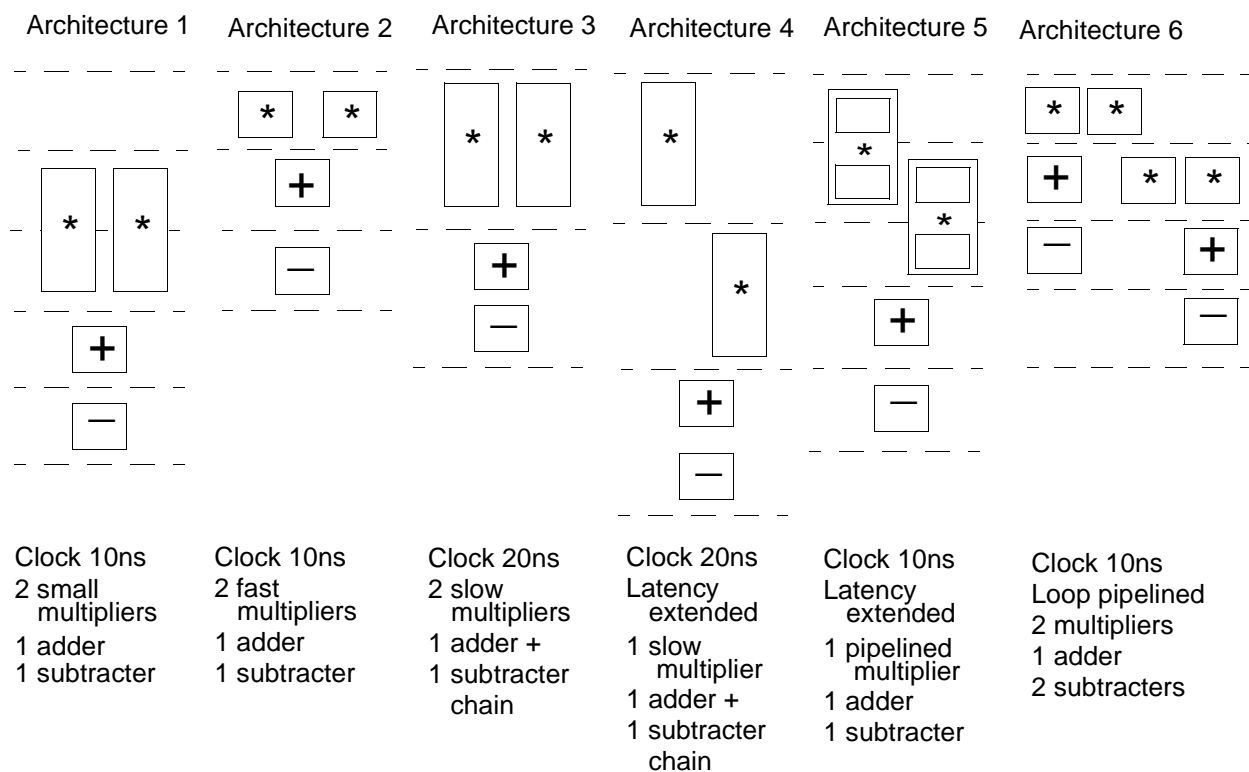
Exploring Architectures and Improving the Quality of Results

Exploring architectures means trading off the clock speed, latency, and resources for a design. Use the visualization capabilities of BCView and the various timing and scheduling reports to find opportunities to improve the quality of results of your design.

Looking at Architectural Tradeoffs

Figure 5-1 shows different implementations of a design that performs two multiplications, one addition, and one subtraction. The figure shows you how to explore different architectural possibilities by setting constraints, using command options, and selecting different components from the synthetic library.

Figure 5-1 Architectural Exploration



In Figure 5-1,

- Architecture 1

Architecture 1 is implemented with the `bc_time_design` command default of smallest area and the `schedule` command default of fastest latency. The clock period is set to 10ns. This architecture requires 5 clock cycles or 50ns.

The multiplication operations require more than one clock cycle to execute, which makes them multicycle operations. Multicycle operations increase latency, because their inputs must be registered prior to the start of the multicycle operation.

Architecture 1 shows an empty first clock cycle when the inputs to the multipliers must arrive. Finding multicycle operations is described in “Identifying Multicycle Operations” on page 6-51

- Architecture 2

Eliminating multicycle operations can reduce the latency of your architecture. In Architecture 2, the `bc_time_design` command is used with the `-fastest` option to select faster components. The multiply operations can now be executed in a single clock cycle, reducing the latency of the architecture to 3 cycles or 30ns. However, faster components are usually larger, so you need to tradeoff a larger area for the faster latency. These commands are described in “Controlling Operation and Implementation Selection” on page 5-7.

- Architecture 3

If your design allows a slower clock period, you can eliminate multicycle operations by increasing the clock period. In Architecture 3, the clock period is set to 20ns. The slower multiply operations are not multicycle and their inputs do not need to be registered. The addition and the subtraction operations are quick enough to be executed in the same cycle, even though they are data-dependent. This reduces the overall latency of the architecture to 2 cycles or 40ns. The scheduling optimization that places several data-dependent operations in one cycle is called operation chaining, and it is described in “Operation Chaining” on page 5-8.

- Architecture 4

If you are concerned about reducing the area rather the improving the latency, use the `schedule` command in the superstate-fixed I/O scheduling mode with the `-extend_latency` option to achieve Architecture 4. SystemC Compiler stretches the latency

of the loop, so that the multiply operations can share the same multiplier. This architecture has one less multiplier and a latency of 3 cycles or 60ns. Reducing area is described in “Using Resource-Driven Scheduling” on page 4-19.

- Architecture 5

You can eliminate multicycle operations without increasing the clock period by using pipelined components to implement the operation. Architecture 5 illustrates the use of a 2-stage pipelined multiplier. For this alternative, use the `schedule` command with the `-extend_latency` option so the two multiply operations can share the same pipelined multiplier. This architecture can be implemented with a 10ns clock in 5 cycles, which is a total of 50ns. In your behavioral description, use the `map_to_operator` compiler directive to instruct the `bc_time_design` command to map a function to a specific component. You can also use the `set_dont_use` command to prevent the `bc_time_design` command from using certain undesirable components. These commands are described in “Controlling Operation and Implementation Selection” on page 5-7.

- Architecture 6

If you are concerned about throughput, loop pipelining may be appropriate for your design. Architecture 6 shows two overlapping iterations of a loop that are pipelined with an initiation interval of 1 clock cycle and a latency of 3 clock cycles. The fast multipliers are used to achieve the highest throughput and fastest latency possible. Loop pipelining usually results in larger area designs, because components and registers need to be duplicated to provide the resources to execute the overlapping iterations of the loop. How to pipeline a loop is described in “Pipelining a Loop” on page 4-44.

Architectural Exploration Guidelines

Explore the architecture of your design looking for opportunities to improve the quality of results by using these general guidelines.

- Use the BCView Selection Inspector detailed area breakdown to find opportunities to reduce area by selecting components and implementations, described in “Controlling Operation and Implementation Selection” on page 5-7.
- Check for under utilization of resources, described in “Resource Utilization” on page 6-32. To improve resource utilization, use component selection and operation chaining.
- Look for operation chaining opportunities, described in “Identifying Chaining Opportunities” on page 6-53.
- Look for multicycle operations, described in “Identifying Multicycle Operations” on page 6-51. In place of multicycle components, use preserved functions, described in “Creating Preserved Functions” on page 5-25. Or use pipelined components, described in “Finding and Implementing Pipelined Components” on page 5-37.
- Analyze critical paths and try to reduce the delay by using preserved functions, described in “Using Preserved Functions” on page 5-23.
- Look for loop pipelining opportunities, described in “Pipelining a Loop” on page 4-44.
- Map arrays to memory, described in Chapter 7, “Using Register Files and Memories for Arrays.”

Controlling Operation and Implementation Selection

For timing estimation, SystemC Compiler uses the components and implementations with the minimum area by default. The smallest components are typically the slowest components. For example, if the design description contains an addition operation (+) and the default synthetic libraries are in use, SystemC Compiler implements the design with the DW01_add operation with the ripple carry (rpl) implementation because it has the smallest area.

To control implementation selection, use one of the following methods:

- Restrict the choice of component by using the `set_dont_use` command on components or implementations that you do not want used for estimation.

```
dc_shell> set_dont_use {object_list}
```

The *object_list* specifies a list of objects (library cells, modules, or implementations) that are excluded in the design. The object names must contain a library prefix. For example, if you do not want SystemC Compiler to consider the bk or csm implementations in the dw01.sldb synthetic library to reduce the execution time of the `bc_time_design` command, you can exclude them with the following commands:

```
dc_shell> set_dont_use dw01.sldb/*/bk  
dc_shell> set_dont_use dw01.sldb/*/csm
```

To decide which synthetic library architectures you might want to exclude, see the descriptions in the DesignWare documentation.

- Use the `bc_time_design` command with the `-fastest` option to instruct SystemC Compiler to use the fastest component available in its timing estimates.

If you want to remove the `set_dont_use` command after defining it, use the `remove_attribute` command. For example,

```
dc_shell> remove_attribute dw01.sldb/*/csm set_dont_use
```

Operation Chaining

Operation chaining is the process of scheduling multiple, data-dependent operations in the same clock cycle if the total delay is less than the clock period. SystemC Compiler automatically looks for opportunities to chain operations to deliver higher performance designs.

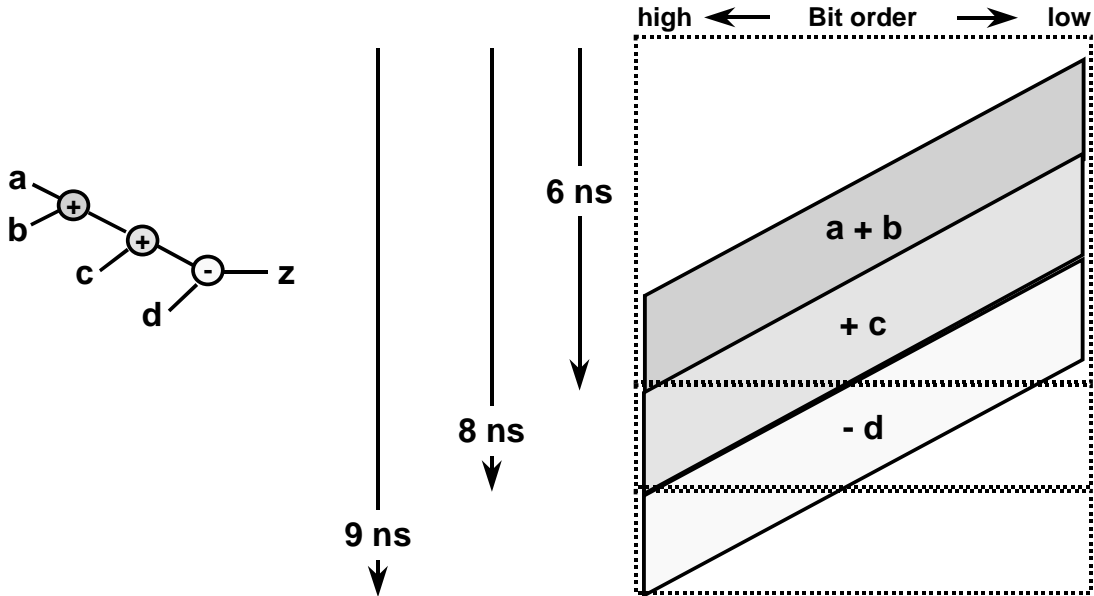
Operation Chaining With Bitwise Timing

SystemC Compiler schedules multiple data dependent operations into the same clock cycle if the total bitwise delay is less than the clock period.

Figure 5-2 shows the bitwise operation chaining possibility for $z = a + b + c - d$ using a clock period of 10 ns. The low order bits of $a + b$ can be used to compute the low order bits of $a + b + c$, before the entire computation of $a + b$ is finished. Therefore, the two addition and subtraction operations can be combined into a chain. Their total delay can be less than 10ns, even if the individual operations have a 6ns delay.

Figure 5-2 Bitwise Timing for Operation Chaining

$$z = a + b + c - d$$



You cannot chain into a multicycle operation because its inputs must be registered, but you can chain out of a multicycle operation.

Determining Operation Chaining

You can determine chained operations by looking at the resource estimates report (described in “Interpreting the Timing and Area Resource Report” on page 3-17). Figure 5-3 shows a partial resource estimates report of the complex number multiplier. The cumulative delay starting at data_in_32 shows the timing path to the sub_35 subtraction operation is 10.4, and the mul_35_2 operation is 6.3. It appears that the subtraction operation delay is 3.8. However, looking further in the report for the individual delay of the sub_35 operation, the operation delay is actually 9.1. The subtraction operation delay of 3.8 is the incremental delay contributed by sub_35 to the chain of data-dependent operations starting at data_in_32.

Figure 5-3 Chained Operations in the Estimated Resources Report (Partial)

```
Cumulative delay starting at data_in_32:
data_in_32 = 0.000000
mul_36    = 6.340029
mul_36_2  = 6.340029
mul_35_2  = 6.340029
add_36    = 10.138293
  imaginary_out_36 = 10.138293
  sub_35   = 10.417433
    real_out_35   = 10.417433
  ...

Cumulative delay starting at sub_35:
sub_35   = 9.158618
  real_out_35   = 9.158618
```

Controlling Operation Chaining

You control operation chaining with the `bc_enable_chaining` variable, which is set to true by default. You can set this variable to false to prevent operation chaining. For example,

```
dc_shell> bc_enable_chaining = "false"
```

SystemC Compiler automatically implements operation chaining when the `schedule` command runs. To force SystemC Compiler to chain certain operations, use the `chain_operations` command before scheduling. However, the `chain_operations` command is ignored if the delay through the operations exceeds the cycle time. For example, to chain the operations labeled `add_1`, `add_2`, and `sub_1`, enter

```
dc_shell> add_1 = find -hier cell *add_1*
dc_shell> add_2 = find -hier cell *add_2*
dc_shell> sub_1 = find -hier cell *sub_1*
dc_shell> chain_operations {add_1 sub_1 add_2 }
```

Similarly, to prevent SystemC Compiler from chaining a particular set of operations with each other, use the `dont_chain_operations` command. For example, to force the operations labeled `op1`, `op2`, and `op3` to be scheduled in different clock cycles, enter

```
dc_shell> op_1 = find -hier cell *op_1*
dc_shell> op_2 = find -hier cell *op_2*
dc_shell> op_3 = find -hier cell *op_3*
dc_shell> dont_chain_operations {op_1 op_2 op_3}
```

SystemC Compiler chains input port reads into an operation by default, even if you specify the operation inputs are not to be chained with the `dont_chain_operations` command. To disable chaining of an input port read into an operation, set the `bc_chain_read_into_oper` variable to false. Enter

```
dc_shell> bc_chain_read_into_oper = "false"
```

SystemC Compiler provides a similar variable for memories, `bc_chain_read_into_mem`. Input port reads are chained into a memory by default, even if the memory `.sldb` file declares they are not chainable. To disable chaining of an input port read into memory, set the `bc_chain_read_into_mem` variable to false. Enter

```
dc_shell> bc_chain_read_into_mem = "false"
```

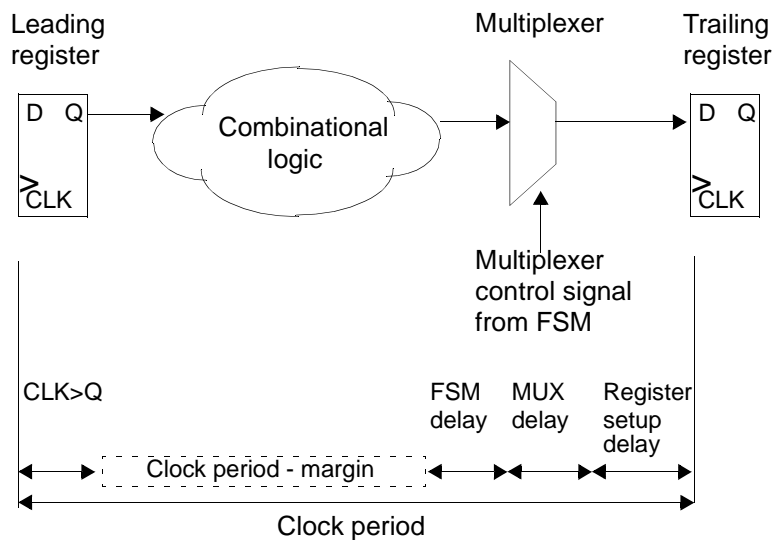
In some cases it may appear as though two operations can be chained, but SystemC Compiler is failing to do so. The most common reason for this is cycle margin, as described in the next section.

Controlling Margin Calculation

The `bc_time_design` command reserves time in the clock period as a clock cycle margin for the hardware that SystemC Compiler adds to every timing path in the design during synthesis. SystemC Compiler extracts the required time to be reserved from the target technology library. The clock period less the reserved clock cycle margin is available for combinational logic.

The timing path starts at the clock pin of a register, passes through the combinational logic, and terminates at the data input pin of a register. Figure 5-4 shows a typical timing path.

Figure 5-4 Typical Timing Path



Each timing path, as illustrated in Figure 5-4, contains common hardware components. SystemC Compiler reserves a clock cycle margin in the clock period for the following components:

- Register margin

The leading register requires time at the beginning of the clock period to respond to the clock edge and make the data available on its Q output pin. This is called clock-to-Q delay.

The trailing register requires data to arrive at its D input pin a certain time before the end of the clock cycle. This is called setup time.

The register margin is also referred to as the flip-flop (FF) margin, because registers are implemented as FFs from the target library.

- Multiplexer margin

The trailing register can get its input from several different sources. A multiplexer controls which of the different sources provides input to the register. The reserved timing margin includes time for the multiplexer.

- FSM margin

At each clock cycle, the FSM generated by SystemC Compiler moves into a new state. The reserved timing margin includes time for the FSM to decode its state and generate the control signals to control the data path portion of the synthesized design.

The `bc_time_design` command reports the clock cycle margin value based on the current target library. SystemC Compiler looks for all available flip-flops in the target library and uses the average clock-to-Q delay and setup delay. Example 5-1 shows the relevant data in the resource estimate report.

Example 5-1 Clock Margin in the Resource Estimate Report

```
Clock Cycle Margin      : 2.86 (Default)
      FSM                : 0.55
      MUX                : 1.21
      FF                 : 1.11
Clock Uncertainty      : 0.00
```

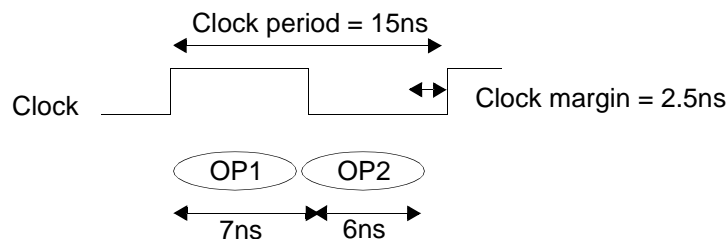
The FSM margin is computed based on the FSM coding style that is specified with the `bc_fsm_coding_style` variable. The default for this variable is the `one_hot` coding style. To compute the register margin, SystemC Compiler looks for all available flip-flops in the target library and uses the average clock-to-Q delay and setup delay unless you specify the `-preferred_FF` option of the `bc_margin` command. To specify a particular flip-flop to use for the margin calculation, enter

```
dc_shell> bc_margin -preferred_FF FF_name
```

You can see the value for the clock cycle margin in the report generated by the `bc_time_design`, `bc_margin`, or `report_resource_estimates` commands.

Figure 5-5 shows a situation where OP1 and OP2 cannot chain because the total delay, including margin, exceeds the clock period.

Figure 5-5 Chaining Operation Timing



Cycle time = Clock period - clock margin = 15 - 2.5 = 12.5ns

Logic delay = OP1 delay + OP2 delay = 7 + 6 = 13ns

Logic delay > cycle time, therefore OP1 cannot chain into OP2

If the clock margin is too conservative, you can make improvements in either of the following ways:

- Remove slower flip-flops from consideration by applying a `set_dont_use` command for them before executing the `bc_time_design` command. For example,

```
dc_shell> set_dont_use my_lib/dff_slow
dc_shell> bc_time_design
```

- Manually override the default value by applying the `bc_margin` command after the `bc_time_design` command completes. Confirm the new value by examining the end of the report produced by the `report_resource_estimates` command (Example 5-2 on page 5-21).

For example,

```
dc_shell> bc_time_design
dc_shell> bc_margin -global 1.5
dc_shell> report_resource_estimates
```

If you provide the global margin value, FSM, multiplex, and register delay is set to 1/3 of the global margin. For example, if you set the cycle margin to 6:

```
bc_margin -global 6

Cycle margin : 6.00
    FSM      : 2.00
    MUX      : 2.00
    FF       : 2.00
```

You can use the following options with the `bc_margin` command:

```
dc_shell> bc_margin [-process process_name] [-global margin]
[-reg margin] [-fsm margin] [-mux margin] [-preferred_FF
cell_name] [-report]
```

The `-process process_name` option specifies the process to which it applies. The default is to apply the command to all behavioral processes in the current design.

The `-global margin` option specifies the total clock cycle margin to be used by SystemC Compiler for the current design.

The `-fsm margin` option specifies the amount of timing margin to be used for FSM decoding logic.

The `-mux margin` option specifies the amount of timing margin to be used for MUX delay.

The `-reg margin` option specifies the amount of timing margin to be used for setup and clock-to-Q pin delay inside flip-flop.

The `-preferred_FF cell_name` option specifies the flip-flop name from the current target library which is used to determine setup and clock-to-Q delay.

The `-report` option generates the detailed information of the flip-flops in the current target library that you can set with the `-preferred_FF` option.

Removing Unnecessary Registers

By default, SystemC Compiler saves the value of input ports in registers after they are read if the value is used in a later clock cycle. When an input port is a static value (for example, a mode-select signal or coefficient value) that never changes during normal operation of your design, the registers are unnecessary.

You can remove these redundant registers from your design and save area by instructing SystemC Compiler that these input values are static and they do not need registers.

Use the `bc_dont_register_input_port` command to specify which ports are static. For example, to prevent registers on ports `name1`, `name2`, and `name3`, enter

```
dc_shell> bc_dont_register_input_port
           {name1 name2 name3}
```

The ports you list with this command will not have their input data registered, and they chain directly into operations in the design.

Note:

If a port is specified as static, its value should change only during the reset state. If the value changes at any other time, the circuit might operate incorrectly because SystemC Compiler constructs an architecture that treats the signal as static.

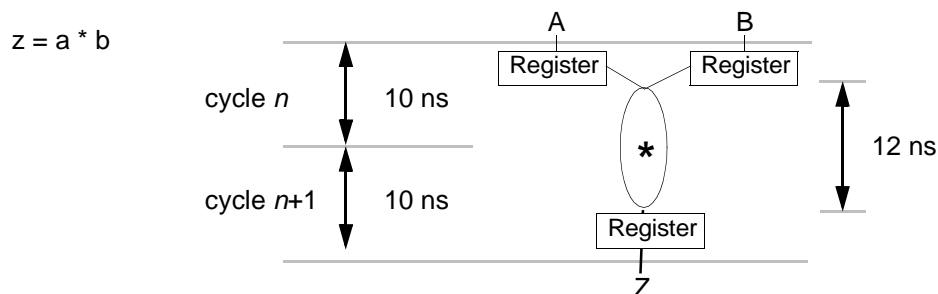
Using Multicycle Operations

SystemC Compiler uses the timing estimates to determine whether to implement single cycle or multicycle operations. If the hardware delay is less than the clock period, SystemC Compiler implements a single cycle operation and may be able to chain single cycle operations together in the same clock cycle; otherwise, SystemC Compiler implements a multicycle operation.

You do not have to indicate whether to schedule operations as multicycle or single cycle. SystemC Compiler schedules these operations automatically if the clock cycle, scheduling constraints, and target technology indicate that multicycling is necessary.

If an operation has a delay greater than the clock period and you are allowing multicycle, SystemC Compiler automatically schedules it as a multicycle operation, as shown in Figure 5-6.

Figure 5-6 Multicycle Operation



In Figure 5-6, the multiplication hardware has a maximum delay of 12 ns, but the clock is only 10 ns. If you use this multiplier in a system that has a clock of 10 ns, the multiplier becomes a multicycle operation.

A multicycle operation affects implementation in these ways:

- The inputs to the multicycle operation must be held stable for as many clock cycles as necessary. For example, in Figure 5-6 the inputs must be held stable for two clock cycles, so they are registered.
- The results are valid in the register corresponding to variable Z two cycles after the input data is valid.
- You need to pass the correct multicycle constraints into logic optimization.
- Optimizing multicycle paths can impact Design Compiler runtime.

SystemC Compiler automatically handles all of these requirements based on the clock period and timing estimates it calculates for the hardware operations.

Reporting Multicycle Operations

You can use the `report_multicycles` command to report multicycle operations for your scheduled design. Example 5-2 shows a partial report, where

- The `cstep` (clock step) indicates the first clock cycle in which the multicycle operation is used.
- The cycle latency is the number of cycles for the multicycle operation.
- The cluster is the automatically created resource name for the component executing the multicycle operation, which is expanded at the end of the report to show the multicycle resource.

- The design name is the name of the synthetic operation that is multicycled.
- The self delay is the operation delay.

Example 5-2 Multicycle Report (Partial)

```

Clock period: 10.00 & margin: 0.00

loop_17/add_36
  cstep: 8
  cycle latency: 2
  cluster: r46
  design name: DW01_add
  self delay: 13.62
...
*****
* Multicycle operators for process cmult_entry: *
*****
r46: DW01_add
  loop_17/add_36
...

```

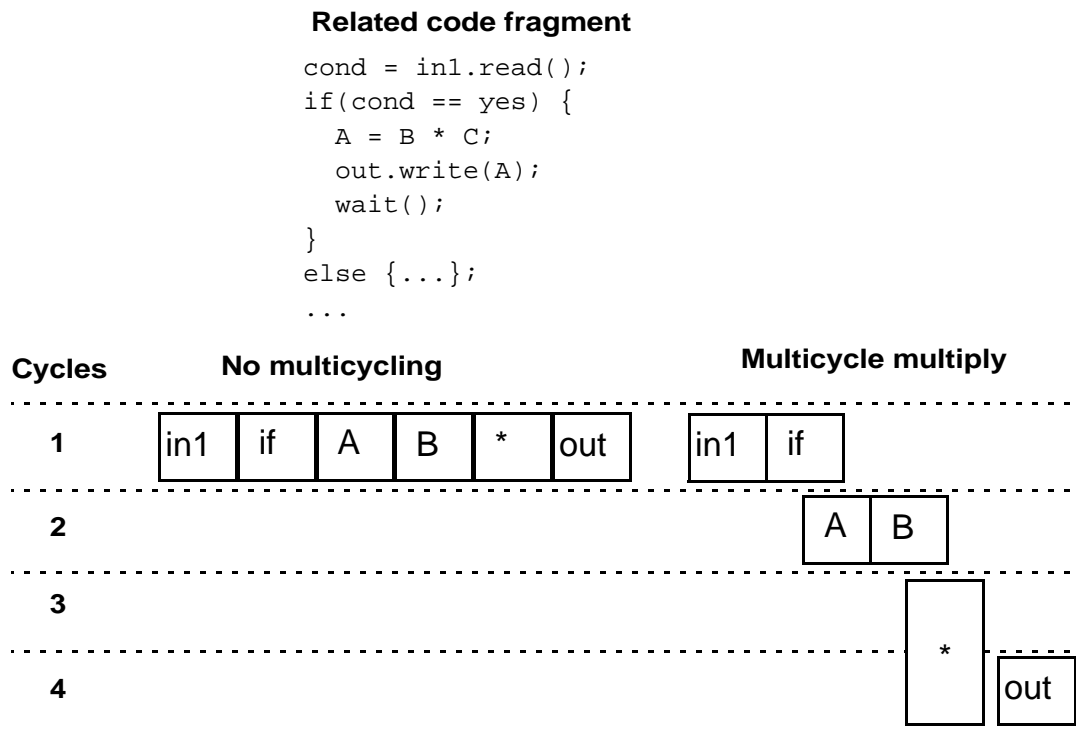
Increased Latency of Multicycle Operations

In many situations, multicycle operations cause an undesirable increase in area or latency. A multicycle operation can increase latency, because

- An extra clock is required before the operation to register the input data and keep the inputs stable for the duration of the multicycle operation.
- If a multicycle operation is in the first clock cycle of a conditional statement, the conditional evaluation needs to be evaluated at least 2 clock cycles earlier.

Figure 5-7 shows a code fragment containing a conditional if statement. When the multiply operation is not multicycle, the entire code fragment can be implemented in 1 clock cycle. When the multiply operation is multicycle, the latency increases to 4 clock cycles, because the if condition must be evaluated in the first cycle before the input data is read, the input data is read and registered in the second cycle, and the multiply operation takes 2 cycles.

Figure 5-7 Multicycle Operations in Conditional Statements



Replacing Multicycle Components

It is highly recommended that you replace multicycle components whenever possible to improve latency.

To replace the multicycle components with a faster combinational component, use the `-fastest` option with the `bc_time_design` command.

If the combinational version is too slow, replace the multicycle component with either a preserved function or a pipelined component, as described in the next sections.

Using Preserved Functions

You can reduce the design complexity and runtime and improve the quality of the resulting hardware by using preserved functions, importing netlists from other tools such as Module Compiler, and using custom DesignWare parts.

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 operation
- Group components that belong in the same cycle into one operation so SystemC Compiler treats the encapsulated function as a single operation
- Incorporate custom netlists into your design (for example, preexisting combinational and pipelined parts)
- Precompile parts and enable more accurate timing estimation
- Precompile groups of operations that would otherwise take more than one cycle with aggressive compile strategies
- Use the preserved function as a resource that can be shared
- Create a pipelined component for an operation

Determining Which Functions to Preserve

During your early attempts to synthesize a design, it might not be obvious where preserved functions are needed. Run an initial timing estimation or schedule the design to help you identify functions for which you might want to preserve hierarchy. After identifying these functions, you can preserve them.

To determine which functions to preserve and to compile them,

1. Apply the `preserve_function` compiler directive to functions that you already know you want to maintain as separate hierarchies.

2. Compile and elaborate the entire design using the `compile_systemc` command.
3. Check the timing estimations or run an initial schedule to find out which other operations might be best kept in their own hierarchies.

For example, look in the resource estimate report for operations that should always be chained into one clock cycle or for groups of logic that have an unrealistic delay estimation. (See “Evaluating the Resource Estimate Report” on page 3-17.) You can usually improve the delay estimation later by precompiling groups of logic with user-defined constraints.

4. Group the logic and operations you want as a preserved function into a function, and insert the `preserve_function` compiler directive into your code, as described in the *CoCentric™ SystemC Compiler Behavioral Modeling Guide*.
5. After elaboration, set the clock period, run the `compile_preserved_functions` command, and time the design.

Creating Preserved Functions

For each preserved function, SystemC Compiler creates a level of hierarchy during elaboration. As a result, right after elaboration you can see the hierarchy, write out elaborated preserved functions, and compile them. Treat a preserved function as you would any other combinational RTL design.

To preserve a function where the function is defined in the design description, annotate it with the `preserve_function` compiler directive, as shown in Example 5-3. Note that the `preserve_function` directive must be the first line in the function body.

Example 5-3 Creating a Preserved Function

```
//SystemC code fragment

my_add ( const sc_int<8> a,
         const sc_int<8> b) {
    // synopsis preserve_function
    //Function code block
    return (a+b);
}
```

If the preserved function is defined in a separate file, declare the preserved function in the header file, as shown in Example 5-4. Place the `preserve_function` compile directive on the function declaration. This compiler directive alerts SystemC Compiler that the preserved function is defined in another file.

Example 5-4 Defining a Preserved Function in a Separate File

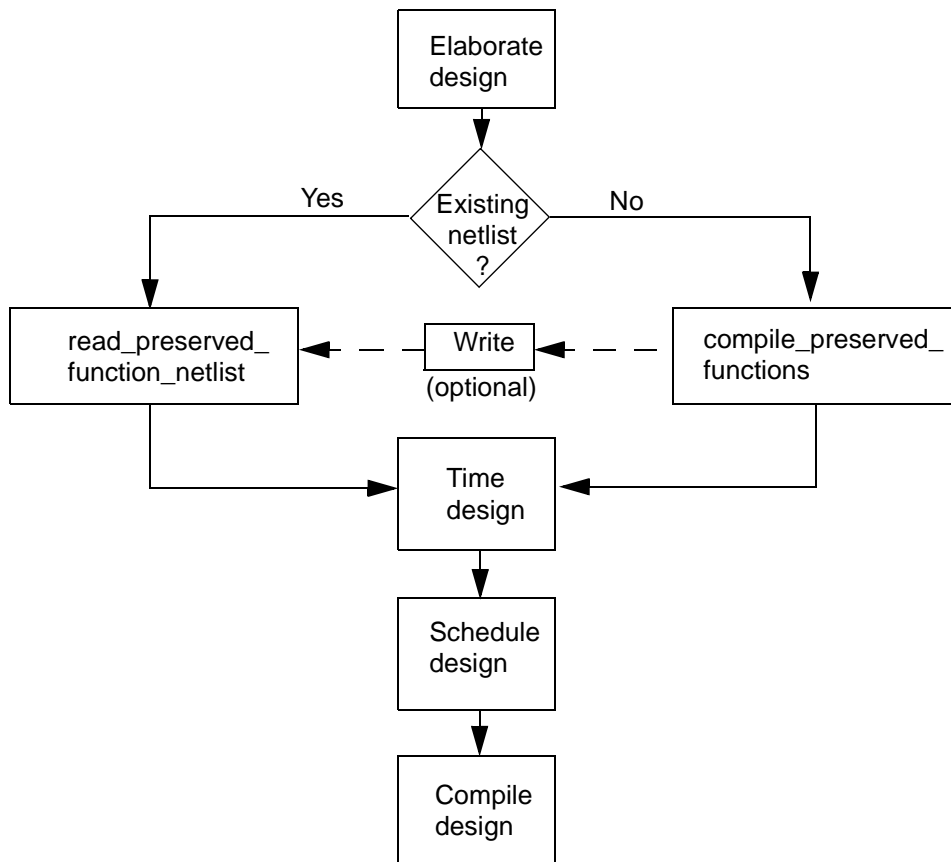
```
//SystemC code fragment
// preserve.h header file
SC_MODULE(pre_example) {
    // Declare ports
    ...
    // Declare member functions
    bool func1(); /* synopsis preserve_function */
    // Declare processes in the module
    void entry();
    // Constructor
    SC_CTOR (pre_example) {
        ...
    }
};
```

For details about creating preserved functions, see the *CoCentric™ SystemC Compiler Behavioral Modeling Guide*.

Preserving a Function

Figure 5-8 shows the standard flow for preserving a function.

Figure 5-8 Flow for Preserving Functions



The typical flow is to elaborate the top-level design, compile the preserved functions, read in any precompiled netlists that are used to implement the preserved functions, then time and schedule the design. If you do not have a precompiled netlist, compile all the preserved functions as described in “Compiling Preserved Functions” on page 5-29.

Using a Precompiled Netlist for a Preserved Function

You can read in an existing precompiled, mapped netlist for a preserved function. The mapped netlist must be in .db format and generated with the `compile_preserved_functions` command or by another Synopsys tool.

To use precompiled netlists as preserved functions,

1. Elaborate the top-level design with the `compile_systemc` command.
2. Execute the `read_preserved_function_netlist` command. This command reads in netlists of all preserved functions in the designated design library. Or you can read in each preserved function netlist separately.
3. Time and schedule the design.

Example 5-5 shows the most commonly used options for the `read_preserved_function_netlist` command.

Example 5-5 Using the read_preserved_function_netlist Command

```
dc_shell> compile_systemc my_design
dc_shell> read_preserved_function_netlist func1
           -design_library my_lib
dc_shell> read_preserved_function_netlist func2
           -design_library my_other_lib
           -return_port new_return_port_name
```

You can provide one or more preserved function names to read. If you do not specify a preserved function name, this command reads all preserved functions in the designated design library. There must be an existing .db file named `func_name.db` file in the designated design library. To provide more than one function name, enclose the names in braces (`{ }`).

The `-design_library` option specifies a design library where the preserved functions are stored. If you do not specify a design library, it reads from the default design library typically named `WORK`. When you designate a design library name, use the `define_design_lib` command to map the logical library name to a physical UNIX path before executing the `read_preserved_function_netlist` command.

For example,

```
dc_shell> define_design_lib library_name1
           -path /remote/design_libraries/library1
dc_shell> read_preserved_function_netlist func1
           -design_library my_lib
```

The `-return_port` option specifies the name of the port to use for the return value of the preserved function. The default return port name of `func_name-return` is used when you do not specify a return port name.

If the preserved function is a pipelined netlist that was created with either the `compile_preserved_function_netlist` or `pipeline_design` command, the `read_preserved_function_netlist` command automatically determines that the netlist is pipelined and not combinational.

Compiling Preserved Functions

When you do not precompile preserved functions, SystemC Compiler automatically compiles them during timing of the design with default compilation strategy and constraints.

You can compile the preserved functions before timing a design to check the results of compilation for preserved functions and make adjustments to get the timing and area you want, if necessary, before performing timing estimation.

Example 5-6 shows the most commonly used options for the `compile_preserved_functions` command. Define the clock period with the `create_clock` command before executing the `compile_preserved_functions` command.

Example 5-6 Using the `compile_preserved_functions` Command

```
dc_shell> create_clock -name clk -period 20
dc_shell> compile_preserved_functions {func1 func2}
    [-write]
    [-design_library mylib]
    [-compile_effort high]
    [-stages number_of_stages]
    [-include_script constraints.scr]
```

The list of function names specifies preserved function names that are to be compiled. The default is to compile all preserved functions in the current top-level design.

The `-write` option specifies to write out each elaborated and compiled design to a file as *function_name.db* for reuse. The `.db` files are written in the default design library, or use the `-design_library` option to specify the design library in which the designs are to be written.

Use the `-compile_effort` option to specify a compilation effort of low, medium, or high. The default is medium.

When the `-stages` option is used, SystemC Compiler automatically runs retiming on the preserved function to generate pipelined preserved functions. Without this option, only combinational

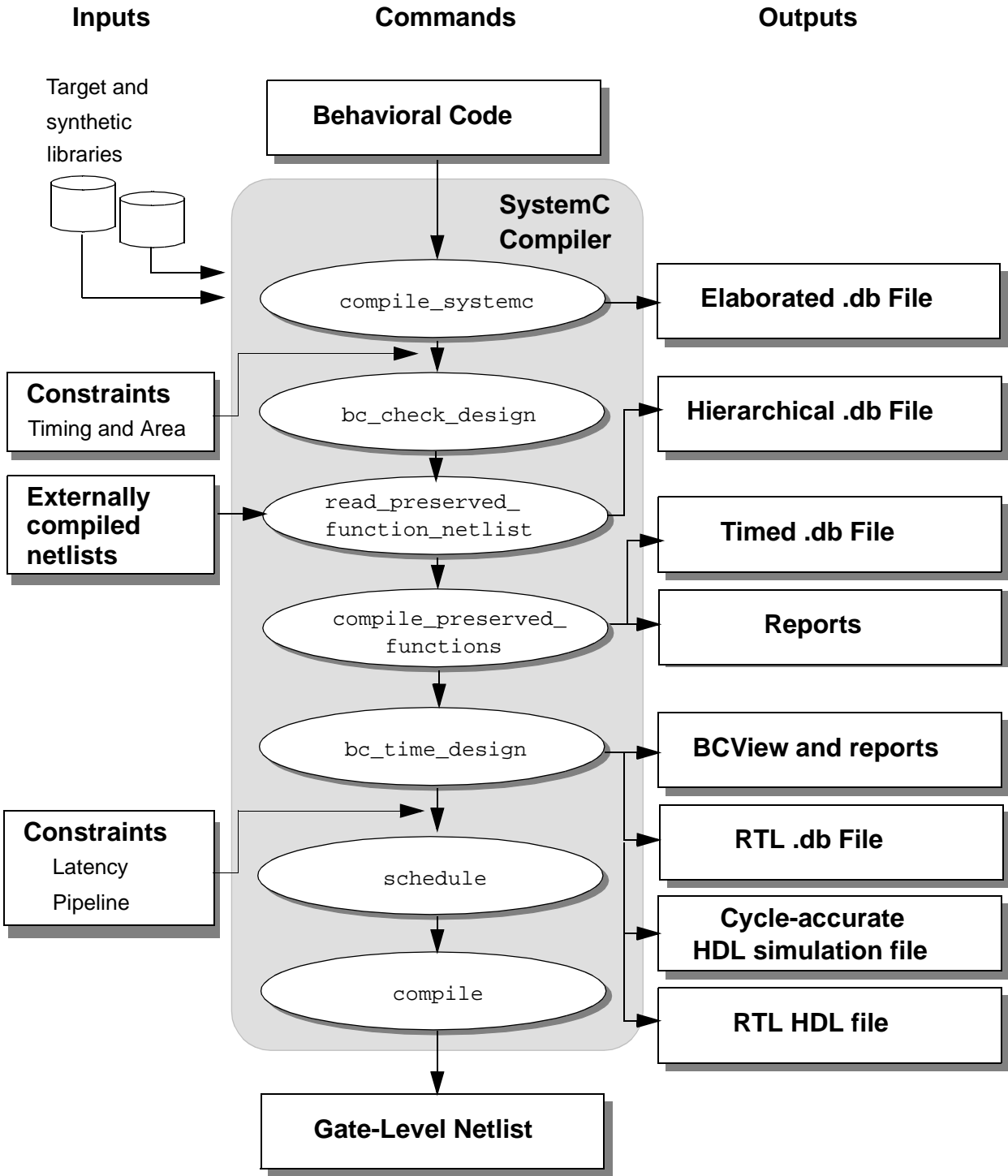
preserved function are created. This option specifies the number of pipeline stages for the preserved functions. The number of stages is one more than the number of registers encountered on any path from any data input port to any data output of the preserved function. The minimum possible value is 2.

The `-include_script` option includes a user-defined `dc_shell` synthesis script file that contains constraints and compilation commands to enable customized compilation of preserved function into components. (For information about using these scripts, see “Using Scripts” in Appendix A.)

Using Preserved Functions for Behavioral Synthesis

Figure 5-9 shows where preserved function fit in the SystemC Compiler command flow.

Figure 5-9 Command Flow With Preserved Functions



Limitations of Preserved Functions

This section describes the restrictions placed on functions that can be preserved.

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
- A `wait()` statement
- Signal reads and writes
- Rolled loops
- Preserved functions (no nesting of preserved functions)

Bit-Width Restrictions

You can describe inlined functions without restricting the bit-width. For preserved functions, however, you need to define the bit-width of every formal parameter and variable used in the functions as a SystemC data types such as `sc_int<n>`, `sc_uint<n>`, and `sc_bv<n>`.

Hierarchy

Preserved functions cannot contain lower levels of hierarchy (such as other preserved functions or rolled loops).

The `compile_preserved_functions` command automatically flattens the design by default.

You cannot call other preserved functions from a preserved function; however, you can have a function call to nonpreserved functions inside a preserved function. The nonpreserved function call will be inlined.

During elaboration, SystemC Compiler checks for hierarchical elements in a preserved function. It issues an error message if you try to call another preserved function or a warning (unresolved reference) if you use a netlist that contains hierarchy.

Sequential Logic

Sequential logic such as rolled loops and `wait()` statements are not allowed in preserved functions. SystemC Compiler issues an error message when structures of these types are encountered.

Using DesignWare Components

In addition to preserved functions, you can also use DesignWare components to create a level of hierarchy in your design. The `map_to_operator` compiler directive performs an action similar to the `preserve_function` compiler directive with the following additional benefits:

- Enables use of memories
- Enables use of standard DesignWare components

DesignWare components cannot contain hierarchy; however, they can contain other DesignWare components that have been compiled down to gates.

Example 5-7 shows code that uses a DesignWare component. For more information about creating a SystemC description that uses DesignWare components, see the *CoCentric™ SystemC Compiler Behavioral Modeling Guide*.

Example 5-7 Using DesignWare Components

```
//SystemC code fragment

sc_int<8> my_add (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);
}
```

Listing DesignWare Components

To list the available DesignWare components,

1. Execute the `list -libs` command to list the available libraries.
2. Execute the `report_synlib` command to list the available components in a DesignWare library.

For example,

```
dc_shell> list -libs
dc_shell> report_synlib standard.sldb
```

Example 5-8 shows fragments of the synthetic library report. For details about this report, see the DesignWare documentation.

Example 5-8 Reporting DesignWare Components

Library Type : Synthetic
Tool Created : 2000.11-PROD
Date Created : Fri Oct 27 20:38:23 PDT 2000
Library Version : 1998.08

Operator Types:

Operator Name	Type
---------------	------

ABS_OP	abs
--------	-----

ADD_TC_CI_OP	add
--------------	-----

...

Operators:

Operator	Ports	Dir
----------	-------	-----

ABS_OP	A	in
	Z	out

ADD_TC_CI_OP	A	in
	B	in
	CI	in
	Z	out

...

Synthetic Modules:

Module

DW01_ADD_AB	design_library: DW01
-------------	----------------------

DW01_ADD_AB1	design_library: DW01
--------------	----------------------

...

Module Pins:

Attributes:

c - clock_pin

Module	Pins	Dir	Width	Default Value	Stall Pin	Pin Attributes
--------	------	-----	-------	---------------	-----------	----------------

DW01_ADD_AB	A	in	1			
	B	in	1			
	S	out	1			
	COUT	out	1			


```
DW01_ADD_AB1    A      in   1
                B      in   1
                S      out  1
                COUT   out  1
```

...

Module Bindings:

Module	Binding
DW01_add	b1 bound_operator: ADD_UNNS_OP Pin Associations (module, oper): A, A B, B CI, "0" SUM, Z b2 bound_operator: ADD_TC_OP Pin Associations (module, oper): A, A B, B CI, "0" SUM, Z

...

Finding and Implementing Pipelined Components

Multicycle operations increase latency, because they require an extra clock cycle to register the input data to keep it stable. To improve latency, you can replace a multicycle component with a pipelined component. Changing to a pipelined component instead of a multicycle component also provides the opportunity to pipeline the loop, as described in “Pipelining a Loop” on page 4-44.

To find and implement a pipelined component,

1. Execute the `report_synlib` command to list the components available in a DesignWare library.
2. Choose a pipelined component.

3. Modify your behavioral description to use the pipelined component with the `map_to_operator` compiler directive. Example 5-7 on page 5-35 shows an example.
4. Start again at the beginning of the SystemC Compiler command flow and execute the `compile_systemc` command (“Compiling and Elaborating the Source Code” on page 2-5).

For example, if you want to replace a multiplier with a pipelined multiplier, use the `report_synlib` command to find a pipelined multiplier component. Example 5-9 shows a fragment of the report for the DW02 synthetic library that lists some of the multiplier components where

- DW02_mult2 is a nonpipelined multiplier
- DW02_mult_s_stage is a 2-stage pipelined multiplier
- DW02_mult_3_stage is a 3-stage pipelined multiplier

Example 5-9 Listing Pipelined Components

...

```
DW02_mult2      design_library: DW02
                HDL parameter: A_width = width('A')
                HDL parameter: B_width = width('B')
                Parameter: PRODUCT_width = B_width + A_width
```

```
DW02_mult_2_stage design_library: DW02
                clocking_scheme: positive_edge
                resource: P1
                resource: P2
                HDL parameter: A_width = width('A')
                HDL parameter: B_width = width('B')
                Parameter: PRODUCT_width = B_width + A_width
```

```
DW02_mult_3_stage design_library: DW02
                clocking_scheme: positive_edge
                resource: P1
                resource: P2
                resource: P3
                HDL parameter: A_width = width('A')
```

```
HDL parameter: B_width = width('B')
Parameter: PRODUCT_width = B_width + A_width
```

...

You might choose, in this case, to replace a DW02_mult2 component with a 2-stage pipelined component, DW02_mult_2_stage.

6

Analyzing Designs With BCView

After you time and schedule your design, use BCView to review common scheduling errors, evaluate the results of scheduling, and evaluate ways to improve the latency and area of your design.

This chapter includes the following sections:

- Using BCView
- Using BCView Windows
- Recommended Usage for BCView
- Examining Scheduling Errors
- Evaluating the Architecture Generated by SystemC Compiler
- Exploring Architectural Improvements

Using BCView

SystemC Compiler has a graphical analysis environment called BCView, which you can use to

- Evaluate your synthesized architecture
- Understand how it corresponds to the original behavioral SystemC description
- Zoom in on specific features of the architecture, such as operations, components, and dataflow paths
- Tune your synthesis constraints to improve the synthesized architecture
- Analyze scheduling errors

Preparing Designs for BCView

To use BCView, you must set a variable that causes SystemC Compiler to generate the analysis information used by BCView. Set the `bc_enable_analysis_info` variable to true before using the `compile_systemc` command. Enter

```
dc_shell> bc_enable_analysis_info = true  
dc_shell> compile_systemc design.cc
```

The default value of the `bc_enable_analysis_info` variable is false.

Starting BCView

To start BCView from the `dc_shell`, enter

```
dc_shell> bc_view
```

For additional information and other ways to start BCView, see “Starting BCView” in Appendix A.

Removing BCView Analysis Information

After your analysis is complete and if you want to reduce the size of the `.db` file, use the `remove_analysis_info` command to delete the analysis information. Deleting this information means you cannot use BCView on the design unless you execute the preparation steps again.

Using BCView Windows

BCView uses cross-linked windows to graphically show information about

- Source code
- Resource allocation and operation scheduling
- The FSM generated by SystemC Compiler
- Clock-cycle (also called a control step) and resource utilization
- Scheduling errors

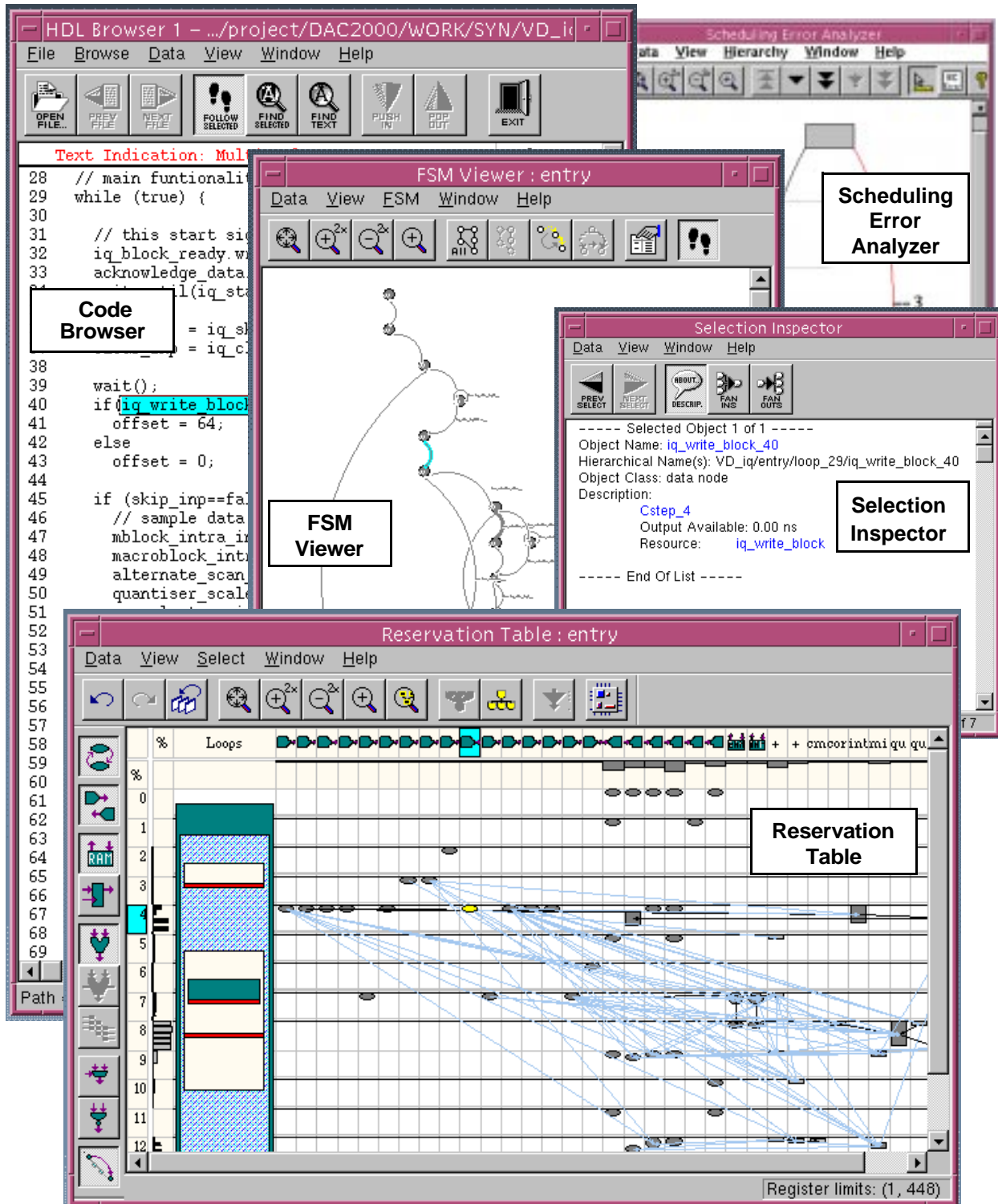
The cross-linking allows you to select an object in one window and view specific information about it in the other windows.

Note:

BCView is not a graphical front end to SystemC Compiler. Also, it is not a design-entry tool, and it does not give explicit information about your SystemC coding style.

Figure 6-1 shows an example of the five BCView windows.

Figure 6-1 BCView Windows



The BCView windows are

Reservation Table

Allows you to view resource allocation, scheduling information, and data dependencies between operations and registers. The table displays allocated resources on the horizontal axis and the clock cycles on the vertical axis. Operations are placed in the reservation table in the column corresponding to the resource on which it is executed and the row corresponding to the clock cycle in which it is executed. The row and column with a percentage (%) heading show the percentage of the clock cycle or resource that is used by the operation.

Code Browser

Displays the behavioral SystemC source file. Whenever you select an object in another window, the line of code corresponding to that object is highlighted in the Code Browser window.

FSM Viewer

Illustrates the FSM generated by SystemC Compiler in a traditional bubble diagram format. You can step through the state transitions, view the actions that the synthesized architecture executes on each transition, and see the corresponding lines of the source code and allocated resources highlighted in the Code Browser and the Reservation Table windows. You can also view the conditions and actions related to a selected transition in the FSM Viewer window.

Selection Inspector

Shows detailed information about an object selected in any of the other windows, including the object name, hierarchy, class, description, fanins, and fanouts.

Scheduling Error Analyzer

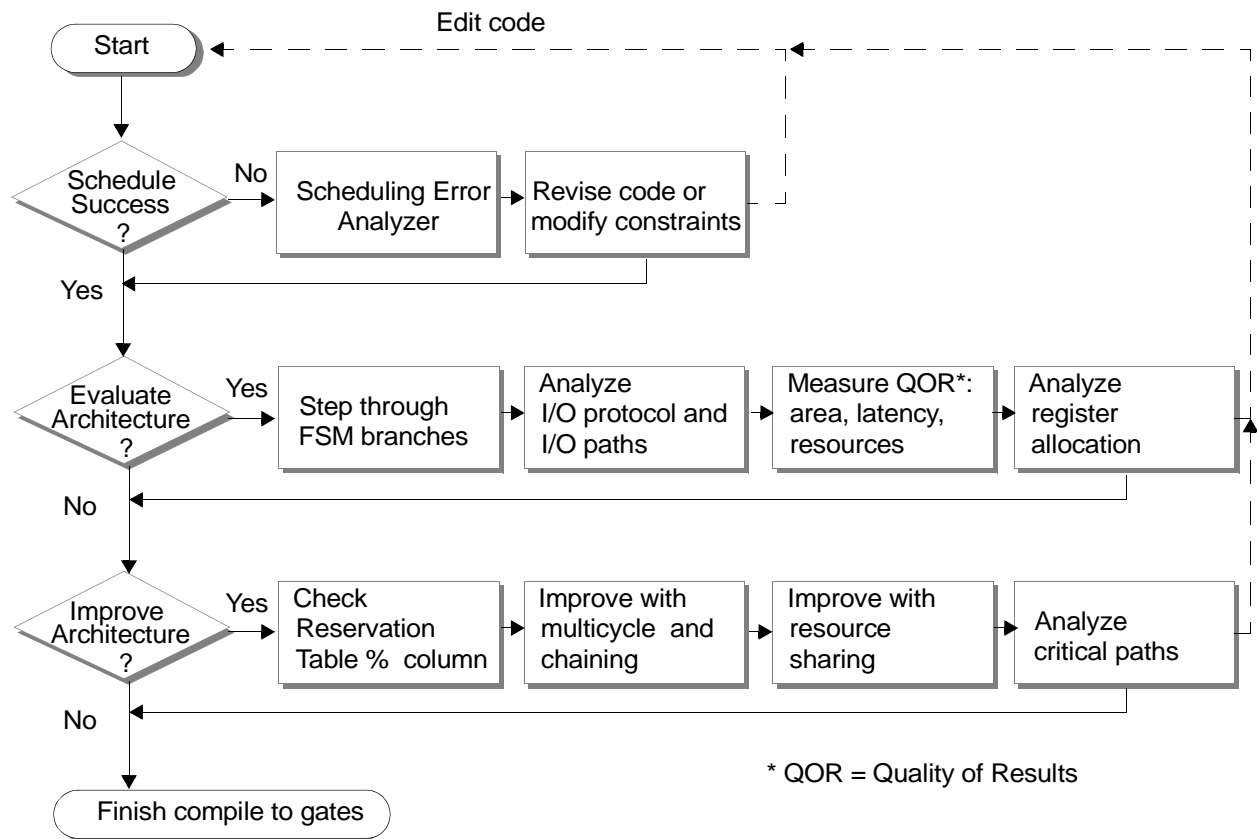
Shows conflicts between user constraints and the inherent constraints in the behavioral description that result in scheduling errors during synthesis, so you can graphically determine the cause of some of these errors. If there are no scheduling errors, this window is not displayed.

You can also find information about BCView in the Synopsys man pages.

Recommended Usage for BCView

Use BCView to quickly find common scheduling errors and to evaluate the results of synthesis. Figure 6-2 shows a recommended usage for BCView.

Figure 6-2 BCView Recommended Usage



Examining Scheduling Errors

The Scheduling Error Analyzer window in BCView presents a graph of the operations involved in a scheduling failure, providing you with a graphical description of what is wrong. This window displays a graph that shows only the operations, data dependencies, and constraints (both user-defined and inherent) involved in the scheduling failure.

The Scheduling Error Analyzer window shows a visual representation of the conflicts that resulted in the scheduling error. Using the Scheduling Error Analyzer window, you can quickly analyze constraints and your code to determine where and why a scheduling failure occurred.

Identifying Errors to Analyze

You can use the Scheduling Error Analyzer window to examine the following scheduling errors:

- Unsatisfiable timing constraints (HLS-51)

The design fails to schedule because conflicting timing constraints cannot be met. For example, a design using the superstate-fixed I/O mode might have a `set_cycles` command (see “Constraining Loops and Operations” on page 4-37) that overconstrains the design.

- Fixed I/O schedule is unsatisfiable (HLS-52)

The design fails to schedule in cycle-fixed I/O mode because it contains insufficient wait statements. For example, a design might contain two wait statements separating input and output operations, but the computation of the outputs from the inputs requires three clock cycles.

When one of these two errors occurs, a message similar to Example 6-1 prompts you to use BCView.

Example 6-1 HLS-52 Error Message

```
Error: Fixed IO schedule is unsatisfiable (HLS-52)
The scheduling errors can be analyzed with BCView:
  type "bc_view [-output <out_db_file>]"
```

Using the Scheduling Error Analyzer

When a scheduling error occurs and you are prompted to use BCView, you can start BCView immediately from the `dc_shell` prompt and use it to examine the causes for the error.

To review a scheduling error in the Scheduling Error Analyzer window,

1. Start BCView.

```
dc_shell> bc_view
```

2. Read the Selection Inspector window. This window usually directs you to the error.
3. Determine the two operations that bound the problem area.
4. Examine the graphical information to determine the mismatch displayed.
5. Fix the code or modify the constraints.
6. Reschedule the design.

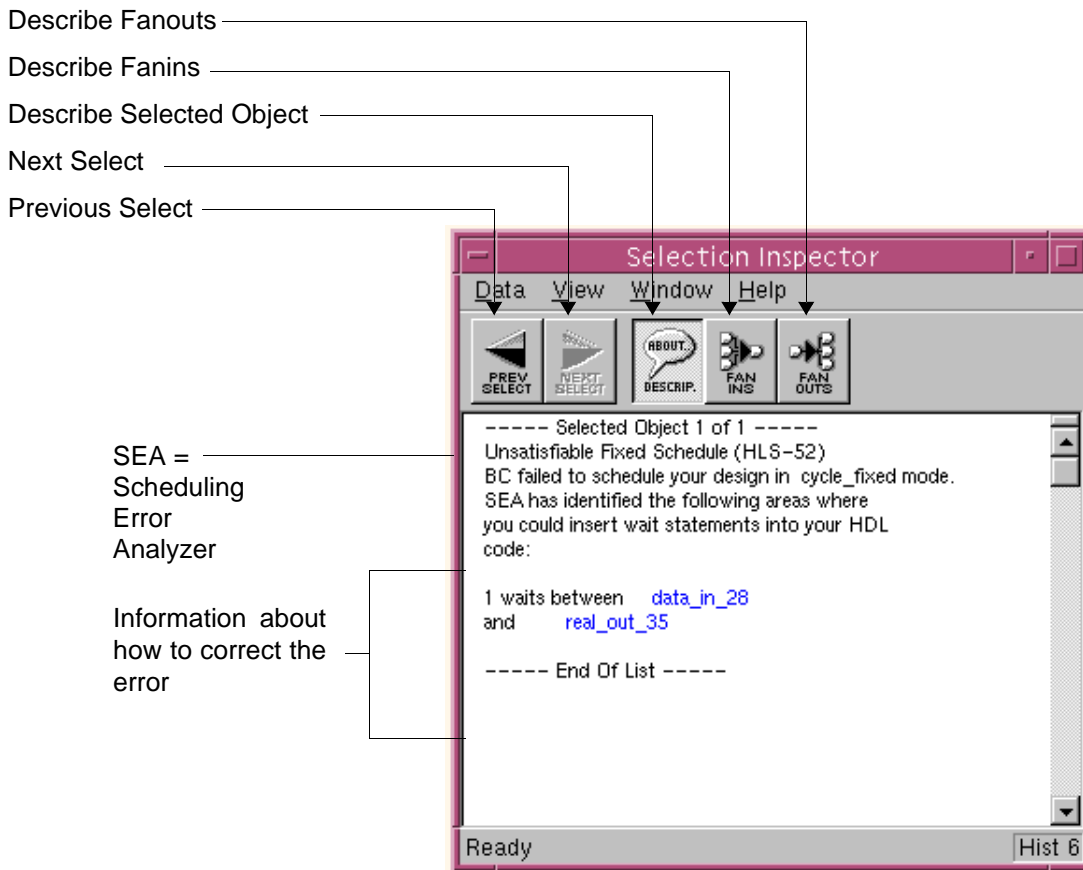
Viewing the Selection Inspector Window

The Selection Inspector window (Figure 6-3) displays detailed information about the scheduling error and about operations and constraints that are selected in the Scheduling Error Analyzer window. This window also often provides information about correcting the error.

To display the initial error message in the Selection Inspector window,

- Choose Data > Show Error Message in the Scheduling Error Analyzer window (Figure 6-3).

Figure 6-3 Selection Inspector With Error Information

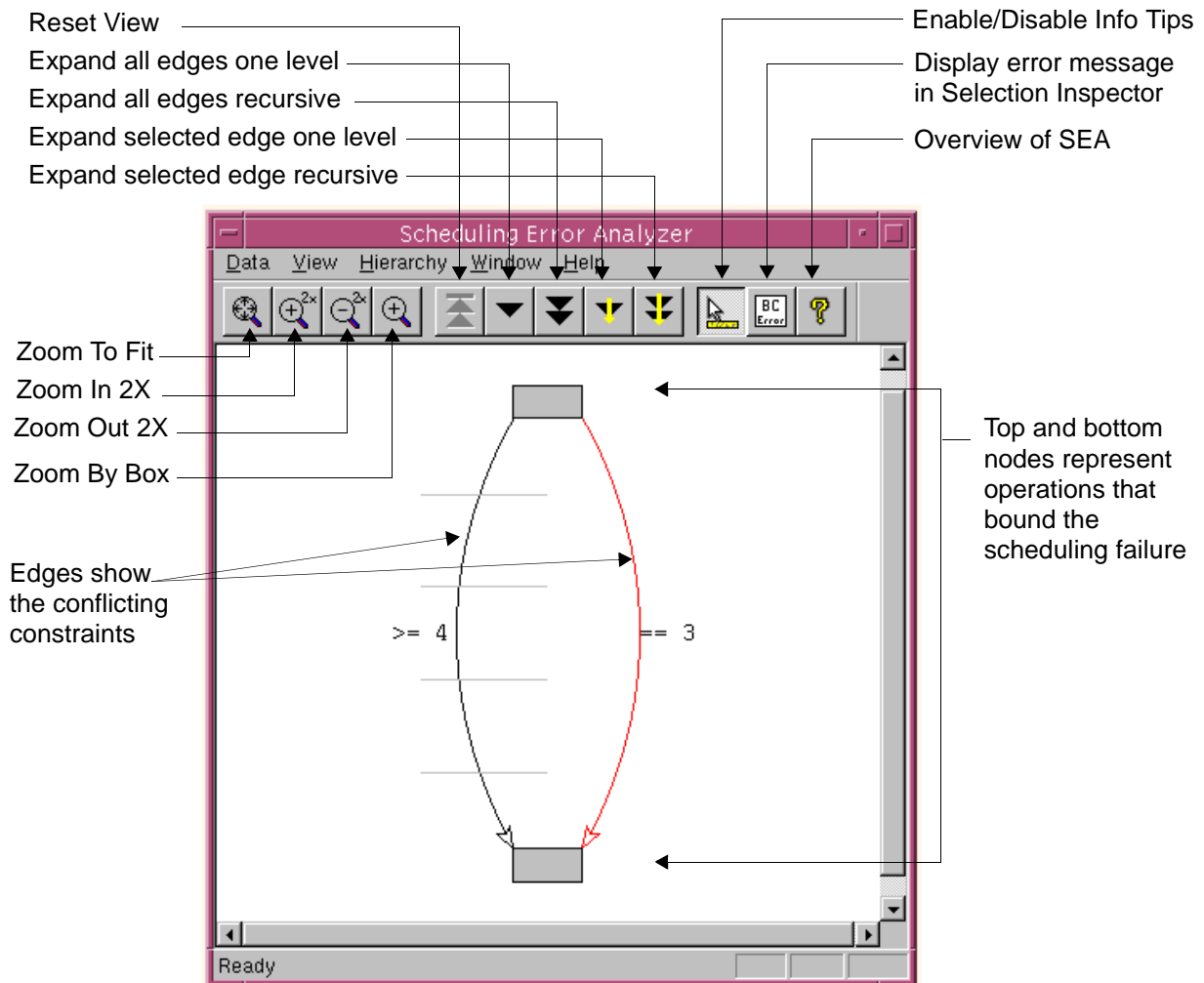


Determining the Operations That Bound the Error

In the Scheduling Error Analyzer window, a node represents a data operation (I/O, memory read/write, or arithmetic) or a control construct (loop or conditional statement). Nodes appear as bubbles, ovals, or rectangles (Figure 6-4).

The scheduling failure occurs between operations represented by the top and bottom nodes in the Scheduling Error Analyzer window.

Figure 6-4 Scheduling Error Analyzer With Bounding Operations



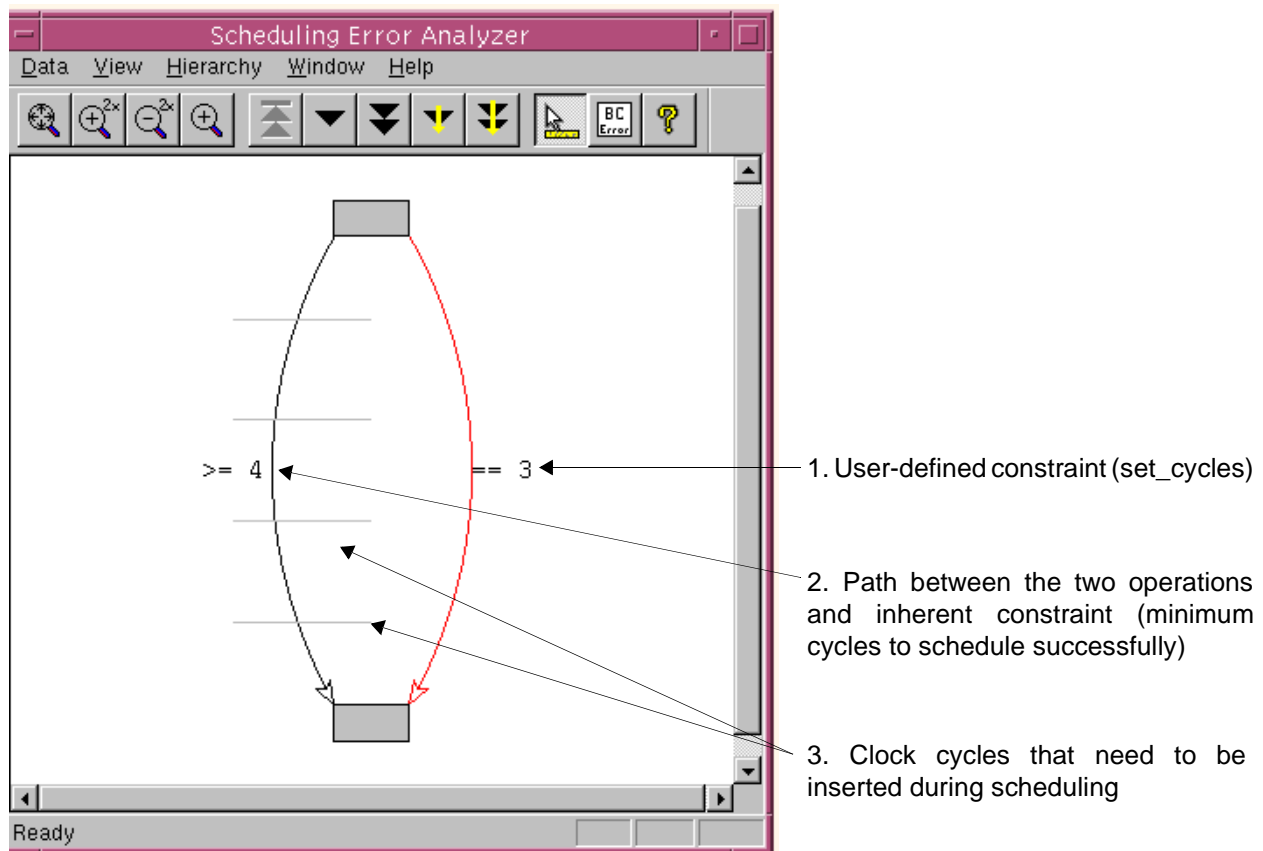
Examining the Graphic Information

Examine the information in the Scheduling Error Analyzer window (see Figure 6-5) and view the related code and detail in the Code Browser and Selection Inspector windows to determine the cause of the error. Clicking on the top and bottom rectangles will highlight the place in the code, in the Code Browser window, that is causing the problem.

Understanding the Scheduling Error Analyzer Display

The Scheduling Error Analyzer (Figure 6-5) shows a scheduling error between two operations or control constructs.

Figure 6-5 Scheduling Error Analyzer Paths and Clock Cycles



In Figure 6-5,

1. The edge on the right shows the user-defined constraint (the number of cycles specified by a SystemC Compiler constraint such as the `set_cycles` command or by the number of `wait` statements in the SystemC code between the two operations).
2. The edge on the left shows the path between the two operations formed by inherent data and control dependencies and the minimum number of cycles required by SystemC Compiler to schedule the operations successfully. You can click the curved edge to expand it and analyze the path further.
3. The horizontal lines segmenting the left edge show clock cycles (control-steps) that need to be inserted during scheduling.

The edges that represent constraints appear in BCView as described in Table 6-1.

Table 6-1 Edges Representing Constraints

Constraint	Representation
User-defined constraints	Curved red edges
Inherent constraints (data dependencies)	Straight black edges
Derived constraints	Curved black edges

Derived constraints are edges that represent a set of inherent data flow and control constraints. Derived constraints summarize the combined effect of the individual constraints in the set. By default, the Scheduling Error Analyzer window does not expand derived constraints. You can click a derived edge to expand it and show the set of inherent constraints.

Reading Labels on Edges and Nodes

The delay implied by a constraint appears as an arithmetic expression, such as ≥ 6 , next to the related edge. The label shows the minimum, maximum, or exact number of clock cycles that separate the nodes, as follows:

$\geq n$

Indicates the minimum number of control-step boundaries that must separate the nodes.

$= n$

Indicates the exact number of control-step boundaries that must separate the nodes.

$\leq n$

Indicates the maximum number of control-step boundaries that must separate the nodes.

In Figure 6-5 on page 6-13, the Scheduling Error Analyzer window indicates that the code between the two operations that bound the problem requires at least four clock cycles, but it is constrained to three cycles.

In the case of multicycle operations, the labels appear next to nodes. Labels on nodes that represent multicycle operations appear as $>n$, where n is the minimum number of clock-cycles that the operation spans. For example, a multicycle operation with a delay of two clock cycles has a label >1 .

Viewing Information About Individual Objects

You can enable the Info Tips feature, which pops up a summary of information regarding the object currently under the pointer. You can click an object to view more details about it in the Selection Inspector window.

To enable Info Tips,

- Do one of the following:
 - Choose View > Info Tips.
 - Click the Info Tip toolbar button.



A check mark appears next to the menu command when it is enabled.

Obtaining More Detailed Information

You can obtain more detailed information by

- Expanding derived edges
- Analyzing displayed information about constraints
- Reviewing the related SystemC code

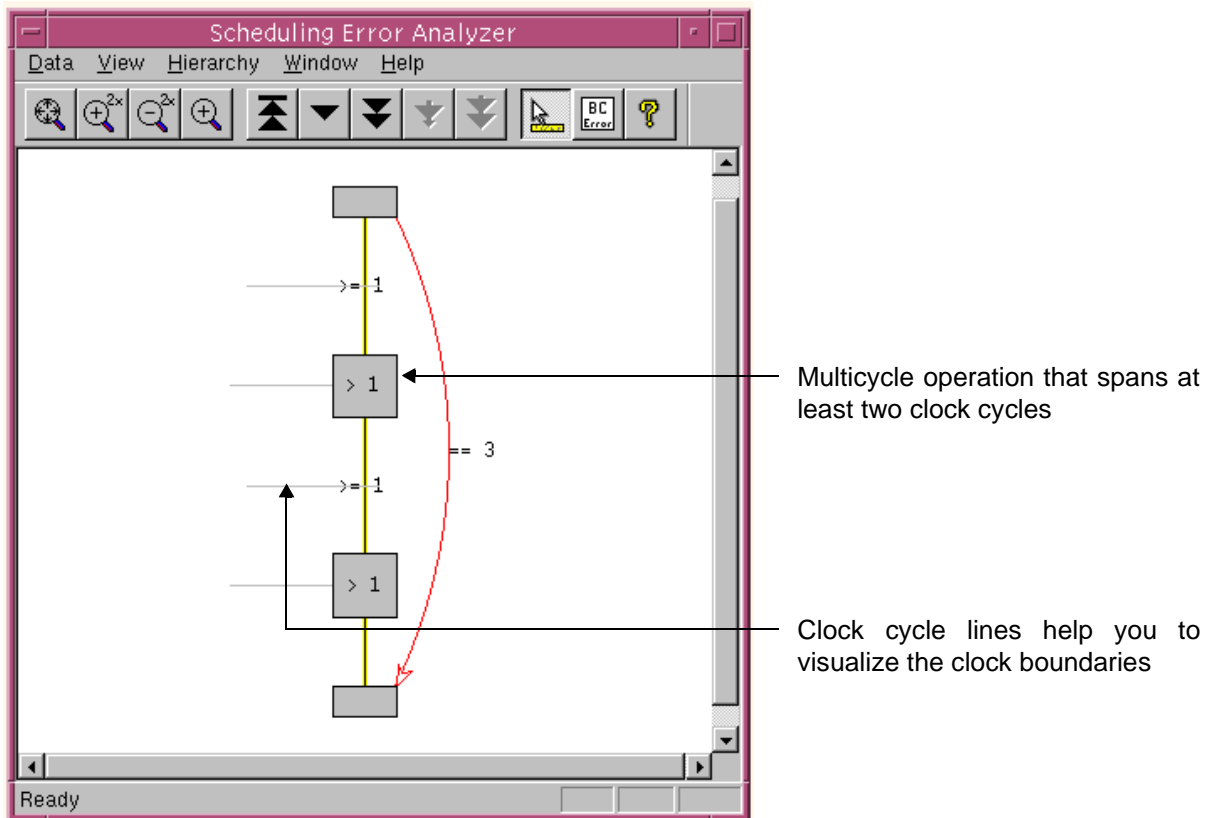
To expand a derived edge (shown as a curved, black line),

- Do one of the following:
 - Double-click the edge to expand it.
 - Select the edge, then click one of the expand toolbar buttons.



The Scheduling Error Analyzer window expands the derived edge to show all constraints and nodes contained within it (Figure 6-6). The new edges are selected.

Figure 6-6 Expanded Derived Edge



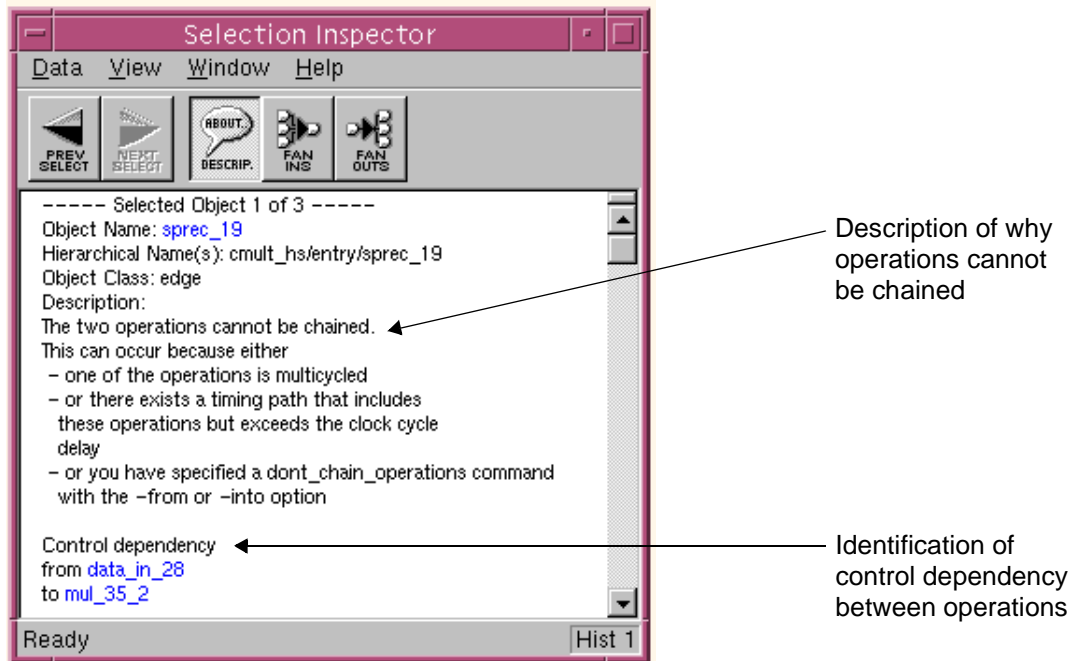
In Figure 6-6 the expanded edge shows two multicycle operations that span at least two clock cycles. Notice the clock cycle lines that show the clock boundary.

To analyze detailed information about a constraint,

1. Select an edge.
2. Read the information in the Selection Inspector window.

Figure 6-7 shows the Selection Inspector window after selection of the first edge (above the multicycle operation) in Figure 6-6.

Figure 6-7 Selection Inspector Window With Edge Information



The information in Figure 6-7 identifies the edge as having a control dependency and describes possible reasons why the two operations must be separated by a clock cycle. In this case, one of the operations is multicycled. The `data_in_28` operation supplies an input to `mul_35_2`. Because `mul_35_2` is multicycled, its inputs must be registered. This implies an inherent control dependency to prevent the two operations from chaining. Therefore, `data_in_28` must be available one clock cycle before `mul_35_2`, so the data can be registered. (For details about multicycle operations, see “Using Multicycle Operations” on page 5-19.)

To review the related SystemC code,

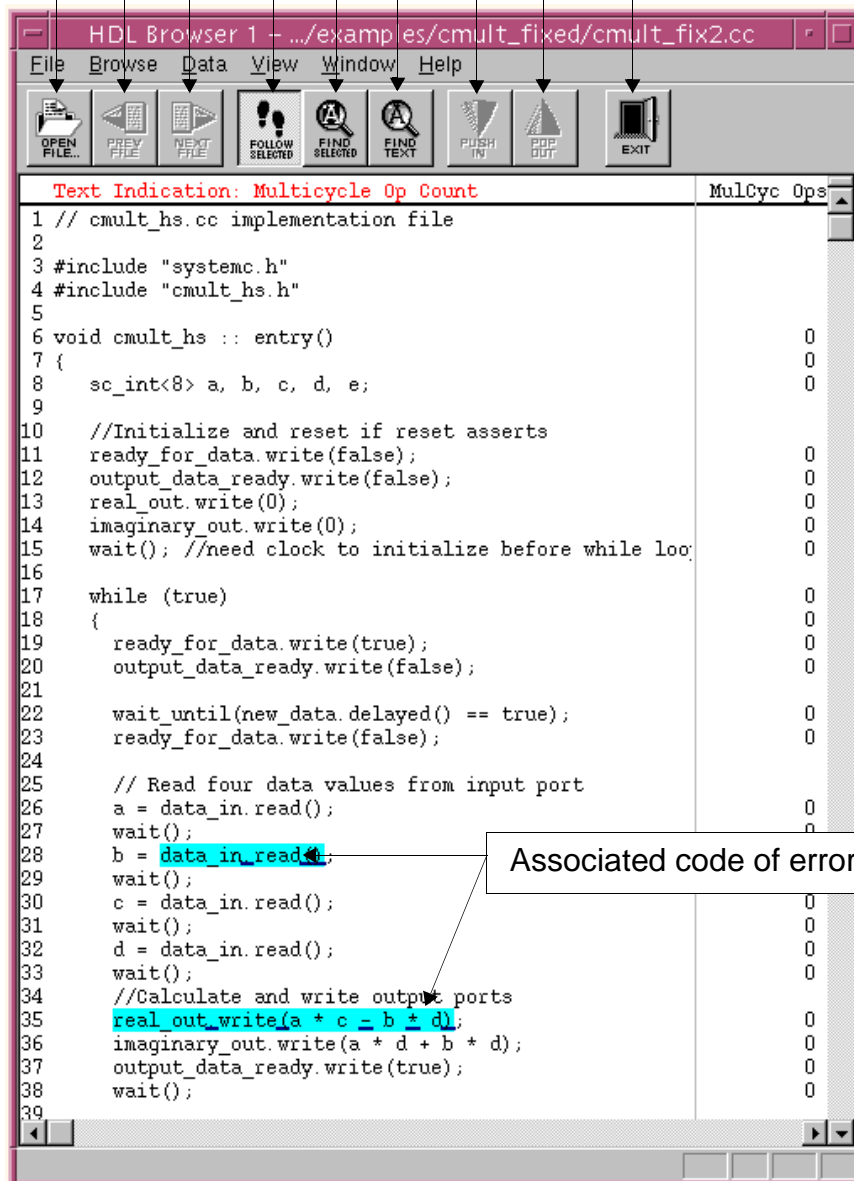
1. Click the operation in the Selection Inspection window to select it.

2. View the code in the Code Browser window.

After you select the multicycled operation in Figure 6-6, the Code Browser window shows the information in Figure 6-8.

Figure 6-8 Code Browser With Behavioral Code

- Open file
- Previous file
- Next file
- Follow selected (from other windows)
- Find selected
- Find text
- Push in a level of hierarchy
- Pop out a level of hierarchy
- Exit BCView



Fixing the Code and Rescheduling

Use a text editor to modify the source code. When you are finished, reschedule the design.

To fix the code in Figure 6-8 on page 6-20, adding an additional wait statement in the SystemC code between the `b = data_in.read()` and `real_out.write(a * c - b * d)` statements will solve the problem.

Evaluating the Architecture Generated by SystemC Compiler

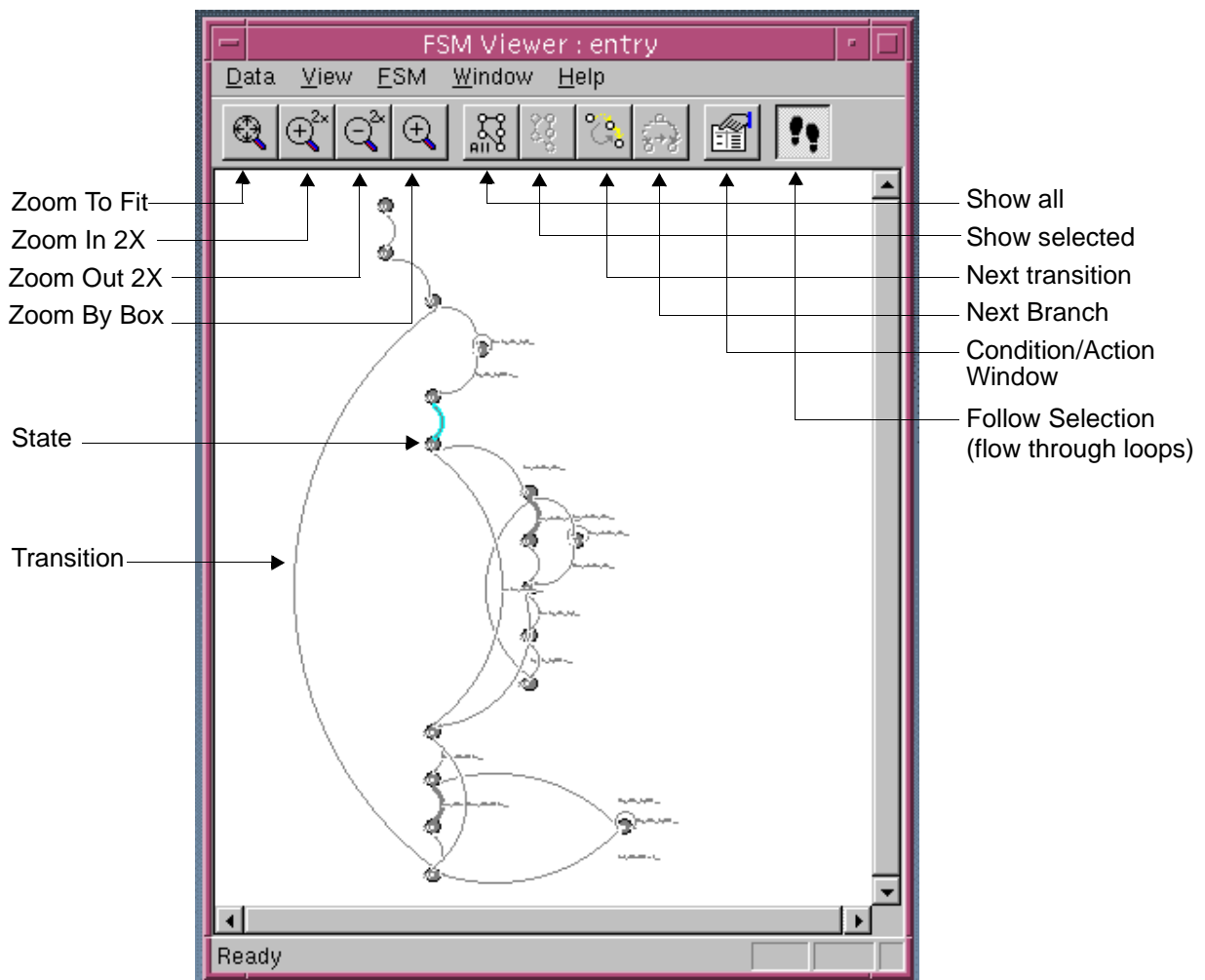
Use the FSM Viewer, Reservation Table, Code Browser, and Selection Inspector windows to review the results of a successful schedule. First review the FSM structure. You can then evaluate information about how the design is scheduled and explore ways to reduce latency and area.

When you view the design, focus on one type of information at a time to avoid the confusion that can occur if you try to evaluate all of the information for all operations at once.

Reviewing FSM Operation

SystemC Compiler generates a Mealy FSM. In BCView, bubbles represent states and arcs represent state transitions, as shown in Figure 6-9. Actions that the synthesized design executes are annotated on the state transition when they occur.

Figure 6-9 FSM Viewer With States and Transitions



Use the FSM Viewer window to review the Mealy machine by first stepping through the FSM, then reviewing state transitions and actions in detail.

Stepping Through the FSM

When you step through the state machine in the FSM Viewer window, you view the cycle-by-cycle behavior of the design and can correlate the transitions with the code highlighted in the Code Browser window.

To step through the FSM,

1. Click an arc in the FSM Viewer window to select the transition where you want to start.
2. Examine the highlighted code in the Code Browser window.
3. Press the Tab key or click the Next transition toolbar button to advance to the next transition.



4. Review the highlighted code in the Code Browser window (Figure 6-8 on page 6-20).

As you traverse through the state machine, one or more lines are highlighted in the Code Browser window because the transition may execute operations in more than one line of code.

5. To choose an alternate transition from a state that has multiple transitions, press Ctrl-Tab or click the Next branch toolbar button.



Reviewing State Transitions and Actions

Use the FSM Conditions/Actions window to analyze the details of the condition when the transition occurs and the actions performed during the transition.

To review state transitions and actions,

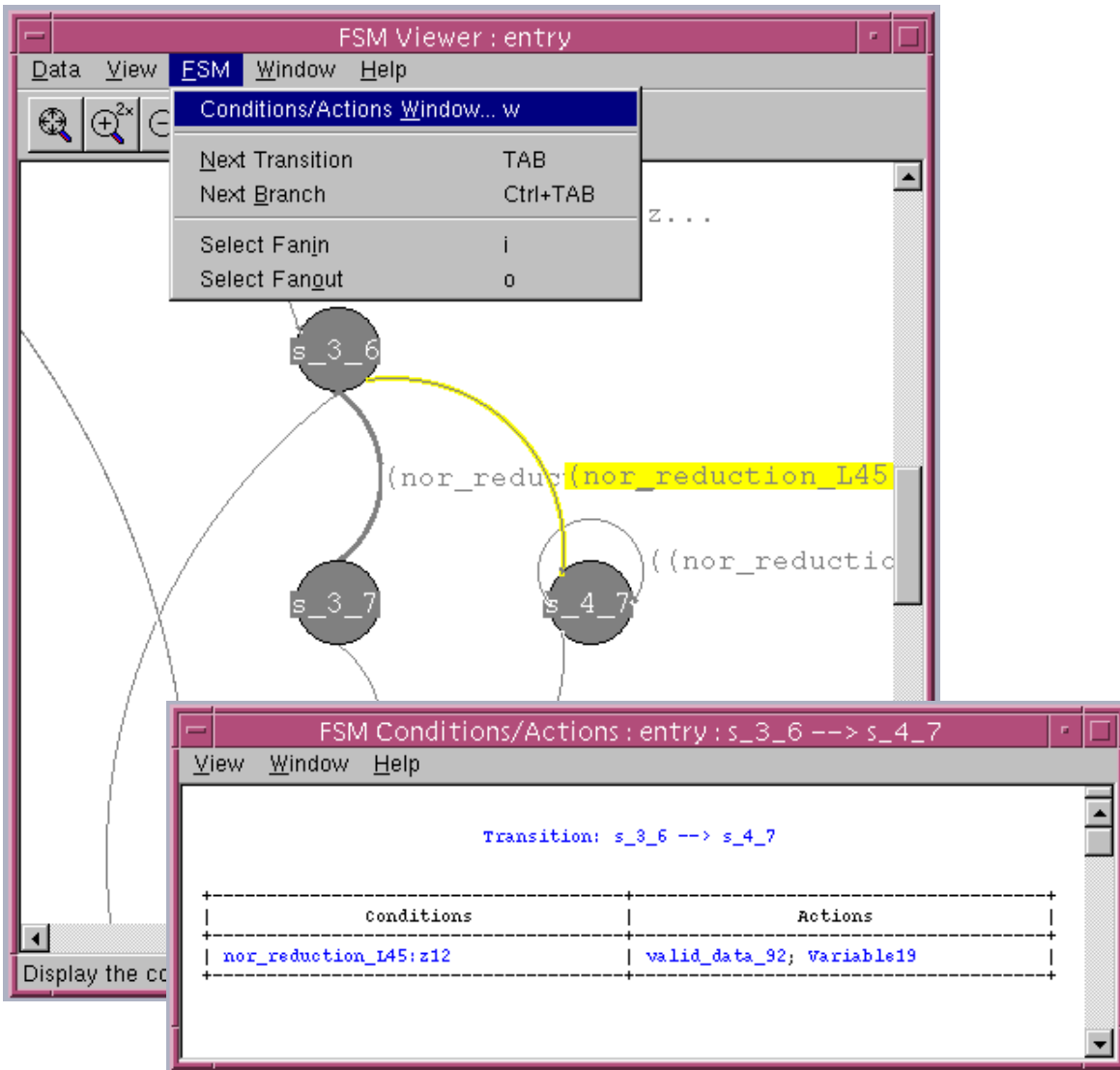
1. In the FSM Viewer window, click a transition to select it.
2. Do one of the following:
 - Choose FSM > Conditions/Actions Window.
 - Click the Conditions/Actions toolbar button.



The Conditions/Actions window is displayed, showing the conditions for the selected transition to execute and the actions that occur. Figure 6-10 shows an example of a selected transition and the corresponding Conditions/Actions window.

The information presented in the Conditions/Actions window is similar to the information presented in the Abstract FSM report generated by the `report_schedule` command. An example of this report and information about it is available in “Schedule Report of the FSM” on page 4-28.

Figure 6-10 Selected Transition With Conditions and Actions



Evaluating the Scheduled Design

Use BCView to evaluate the area and latency of a scheduled design. The Reservation Table window provides a graphical representation of the design structure and resource usage, including timing and data dependency information.

Understanding the Reservation Table Window

Using the Reservation Table window, you can analyze area, resources, latencies, operator sharing, clock-cycle utilization, chaining, combinational delays, paths in the design, registers, and loops.

The Reservation Table Window is shown in Figure 6-11

Figure 6-11 Reservation Table Window

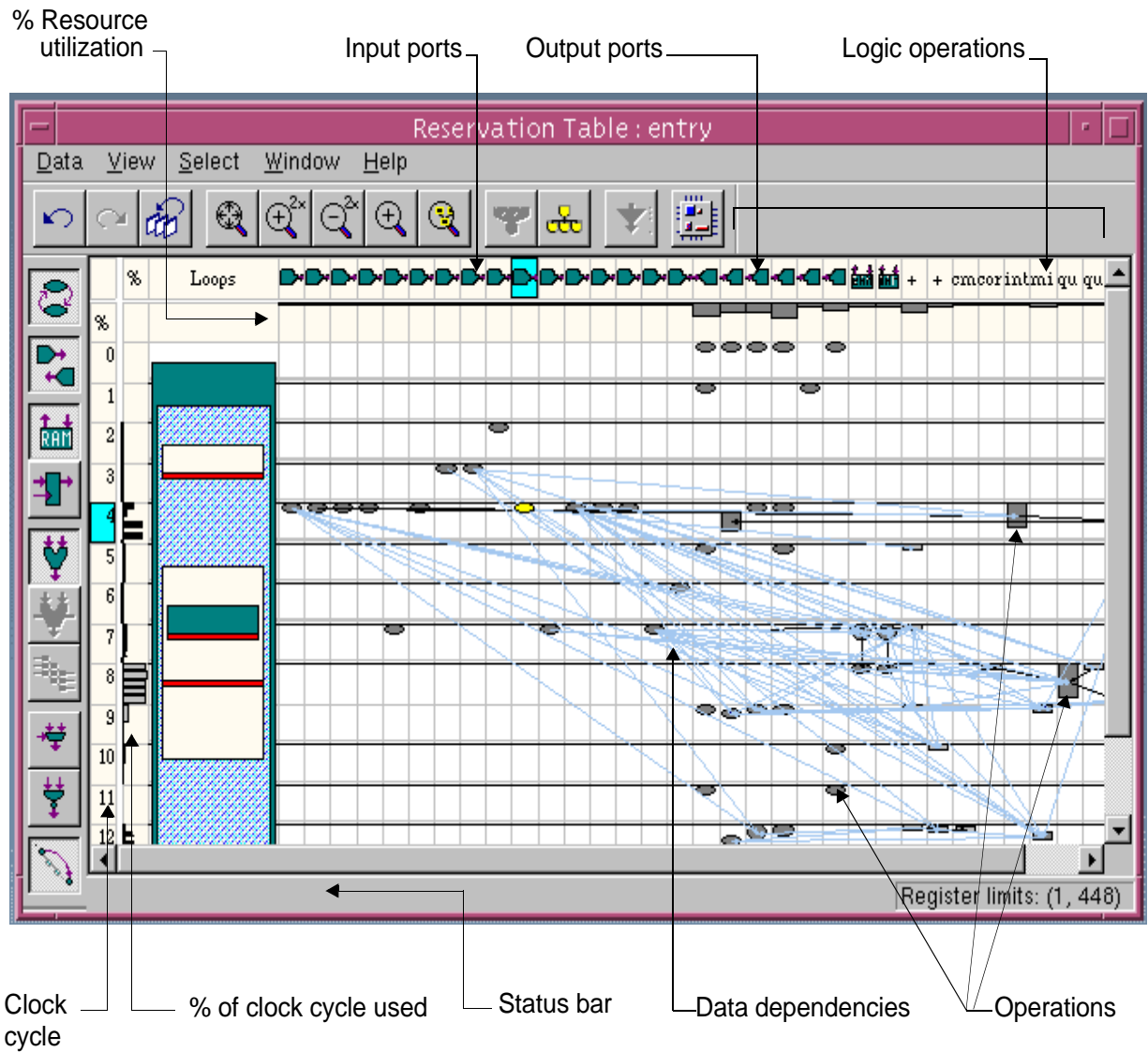









Table 6-2 briefly describes the symbols that appear in the Reservation Table window.

Table 6-2 Reservation Table Symbols

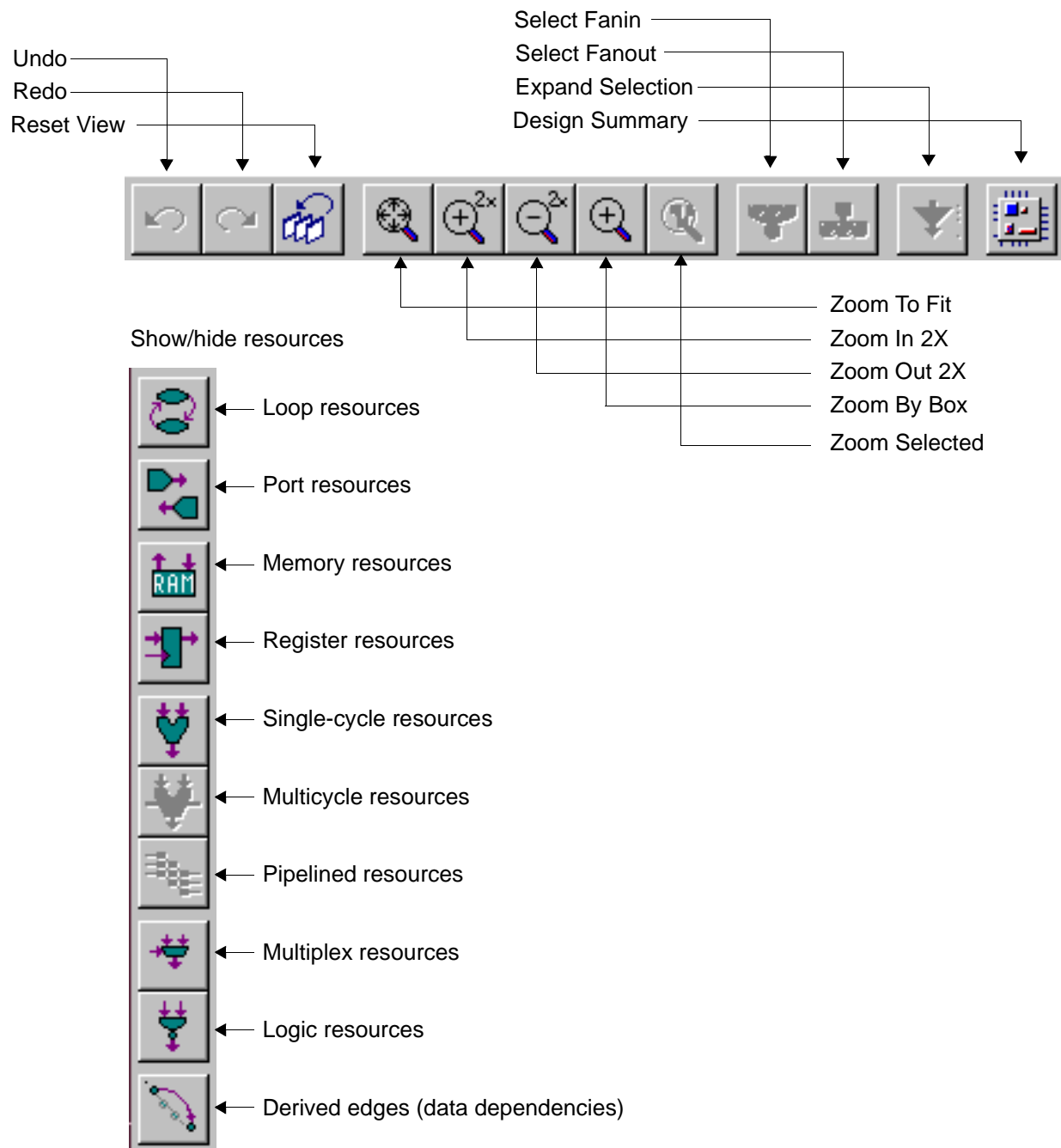
Symbol		What it represents
Gray oval		Operation with zero delay, for example a port operation.
Gray box		Operation with a combinational delay, where the height of the box is proportional to the delay.
Gray bar		Operation with combinational delay, where the length of the bar is proportional to the delay.
Arrow		Data dependency (fanin or fanout).
Arc		User or inherent constraint.
Light blue line		Derived edge. See “Examining Paths” on page 6-38.
Light grey bar (in % column)		Percentage of clock cycle used for chain delay.

In the Reservation Table window in Figure 6-11, columns represent resources, and rows represent the clock cycles. The objects in the table represent operations and other actions that the synthesized architecture performs. Each object is positioned in the column representing the resource that executes it and in the row representing the clock cycle in which it is executed. This Reservation Table displays the Inverse Quantization design from the *CoCentric SystemC Compiler Behavioral Modeling Guide*.

Information about individual objects is displayed in the status bar or in pop-up Info Tips, as described in “Viewing Information About Individual Objects” on page 6-16.

Use the Reservation Table toolbar buttons to perform the functions, which are shown in Figure 6-12.

Figure 6-12 Reservation Table Toolbar Buttons



Note: Click these icons to change the display by showing or hiding resources.

Viewing Resources, Latencies, and Operation Sharing

Resources appear in the top row of the Reservation Table window. Symbols in the column headers represent the different resources in the design such as components, input ports, output ports, memories, and logic operations. Use these columns to review resource utilization and latency, and to identify shared resources.

Showing or Hiding Resources

You can hide information in the Reservation Table to concentrate on particular data. To hide or display resources, do one of the following:

- Click the corresponding button in the vertical toolbar at the left side of the Reservation Table window (Figure 6-12 on page 6-30). Choosing a particular resource type causes that type of resource to be shown or hidden in the Reservation Table.
- Choose View > Show/Hide Resources > *Resource*.

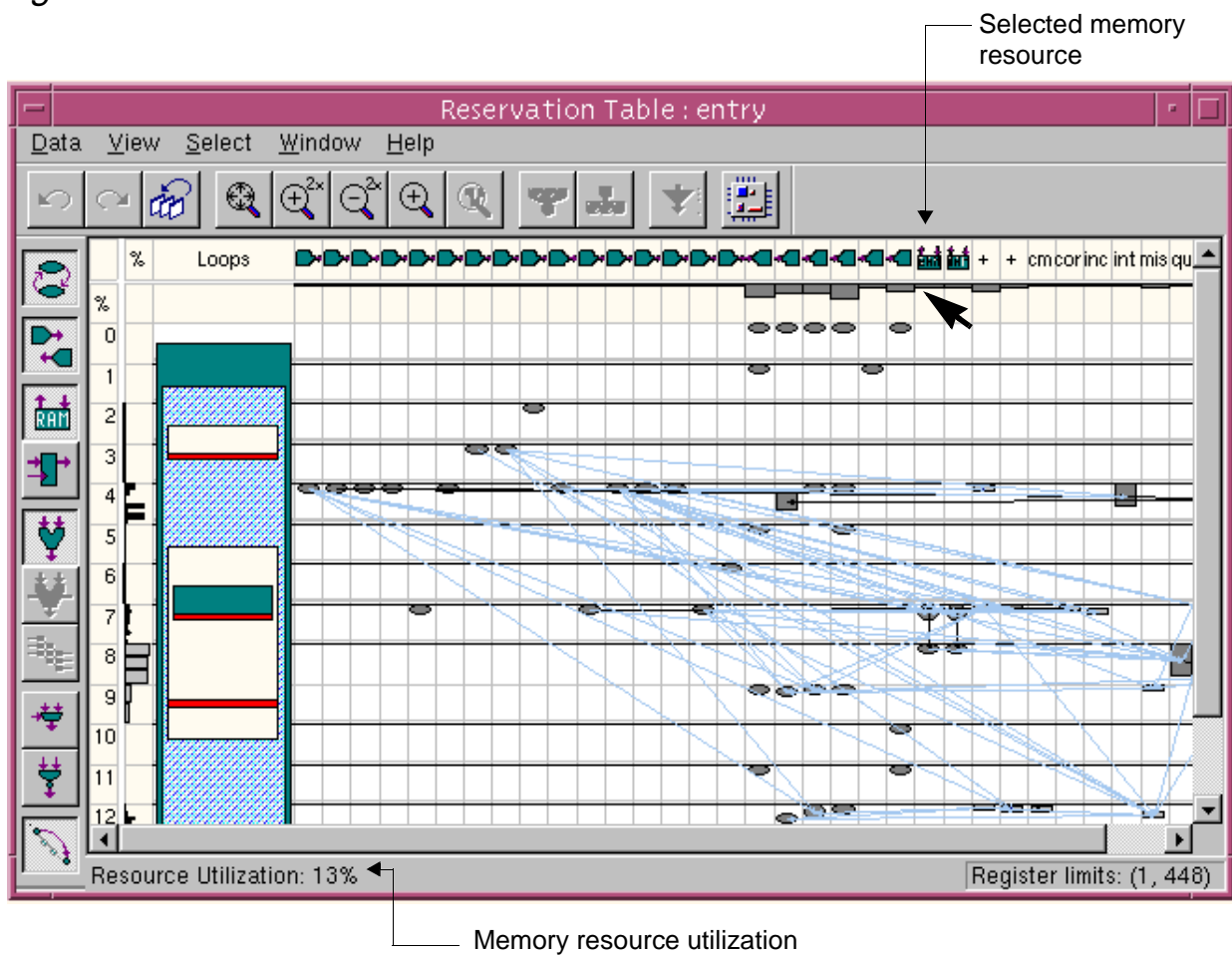
Resource is the type of resource you want to either hide or display (for example, Loops).

Resource Utilization

To review resource utilization, move the pointer over a column in the first (%) row of the window. To improve the quality of results, look for resources that are not fully used. The thickness of the bar, pointed to by the arrow in Figure 6-13, is proportional to the resource usage. If the usage is 100%, the box is filled.

The status bar displays the percentage of the total clock cycles in which that resource is active, as shown in Figure 6-13.

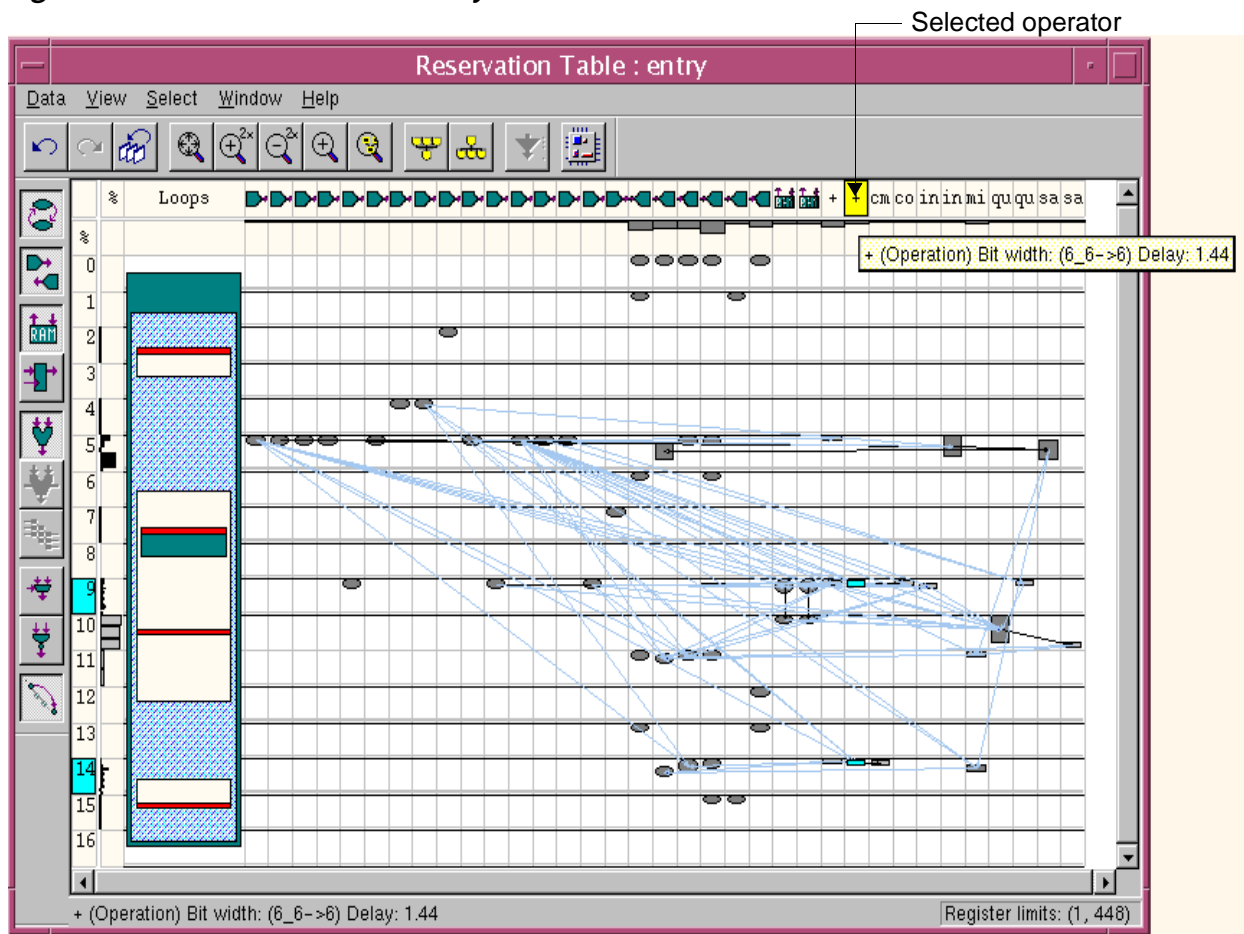
Figure 6-13 Resource Utilization in Reservation Table



Resource Delays

To determine resource delays, move the pointer across the resource column headers. The name of each resource and its delay appear in the status bar and Info Tips window as you move the pointer, as shown in Figure 6-14.

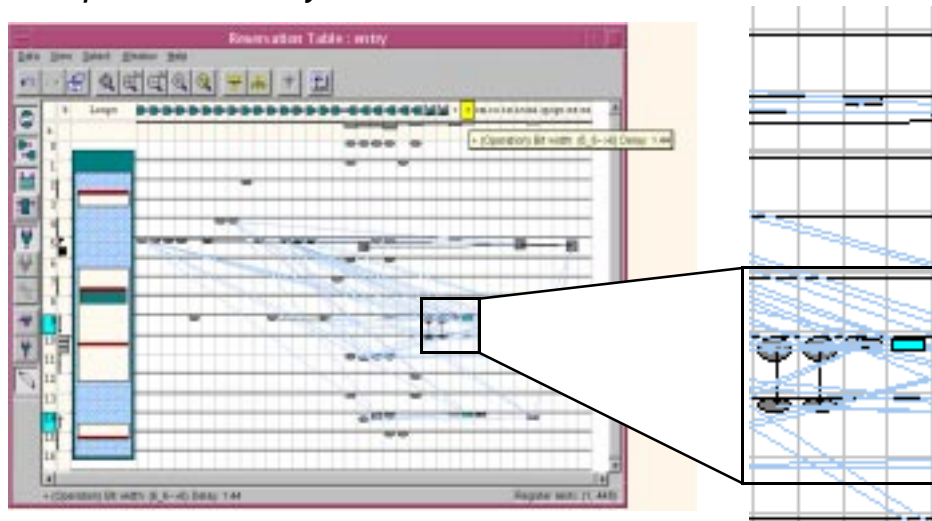
Figure 6-14 Resource Delay in Reservation Table



Operation Delay

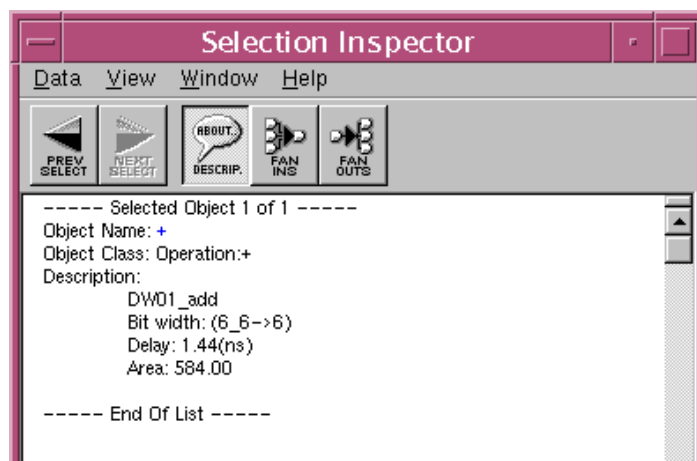
To view operation delay, observe the height of the rectangle representing an operation. Figure 6-15 shows an enlarged section of an add operation in the Reservation Table. The height represents the operation's delay.

Figure 6-15 Operation Delay in Reservation Table



Then, click an operation and read the detailed information in the Selection Inspector window, shown in Figure 6-16.

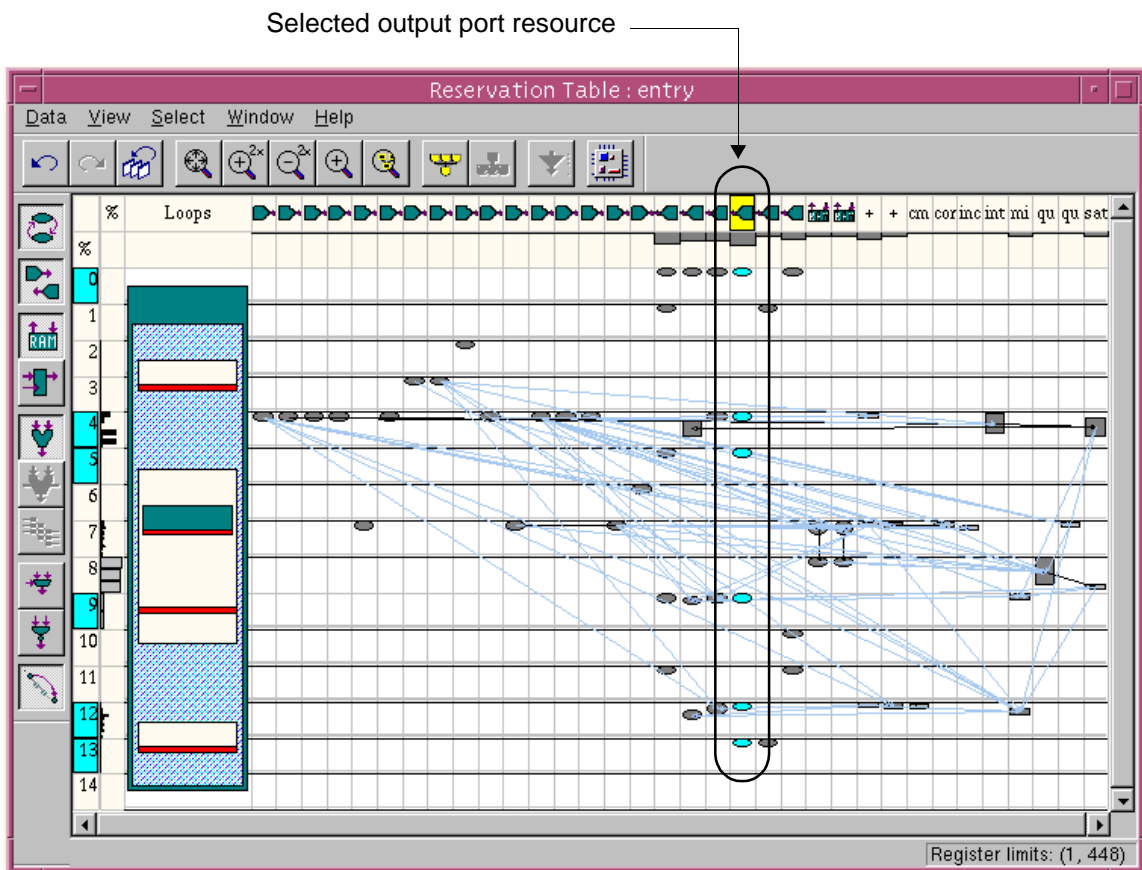
Figure 6-16 Operation Delay Detail in Selection Inspector



Shared Resources

To identify shared resources, select a resource by clicking on it, as shown in Figure 6-17. The operations allocated to that resource and the corresponding clock cycles become highlighted in blue. In this example, the resource is used in clock cycles 0, 4, 5, 9, 12, and 13. Multiple entries in the same column indicate a shared resource.

Figure 6-17 Shared Resources in Reservation Table



Viewing Clocks, Chaining, and Combinational Delay

The second column (%) of the Reservation Table window shows clock-cycle utilization, as shown in Figure 6-18 on page 6-37. The bars in a row (clock cycle) show the percentage of the clock cycle used for the delay of either a chain of operations or a single, unchained operation occurring in that specific clock cycle. Multicycle operations do not appear in the clock utilization column.

Find chained operations in the clock utilization column and review detailed information about the chains in the Selection Inspector window.

To find chained operations,

- Do one of the following:

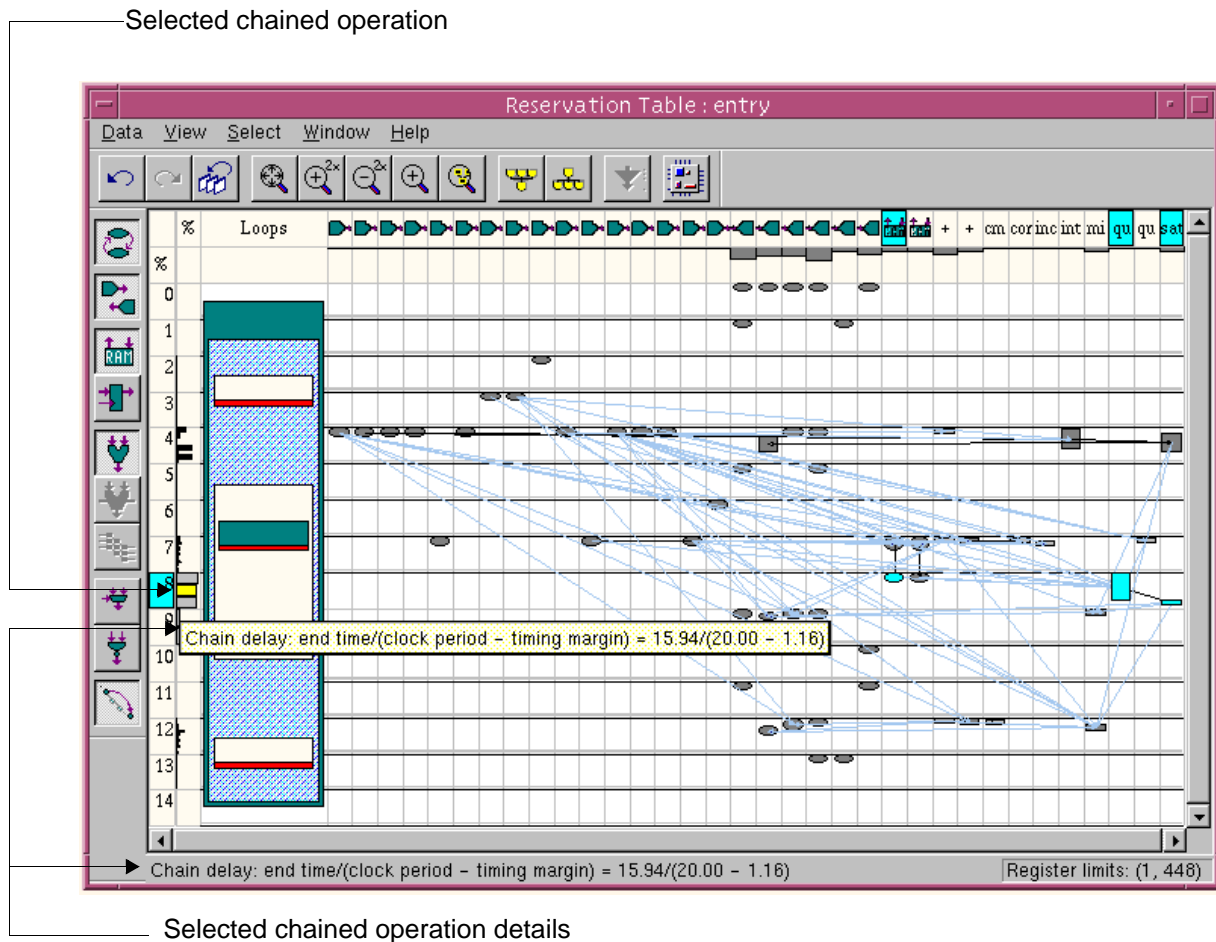
- Choose **Select > Chained Operations**.

The Reservation Table window highlights the chains in the design.

- Select the horizontal bar that represents the delay of a chain in the clock utilization column.

The corresponding operations and their resources become highlighted, as shown in Figure 6-18.

Figure 6-18 Operation Delays in Clock Cycles



The Selection Inspector window displays detailed information about the highlighted chains.

To locate and evaluate chain delay,

1. Enable Info Tips and the status bar (see “Viewing Information About Individual Objects” on page 6-16).
2. Move the pointer over a delay bar.

The status bar and Info Tips show the total delay. Figure 6-18 on page 6-37 shows an example.

Examining Paths

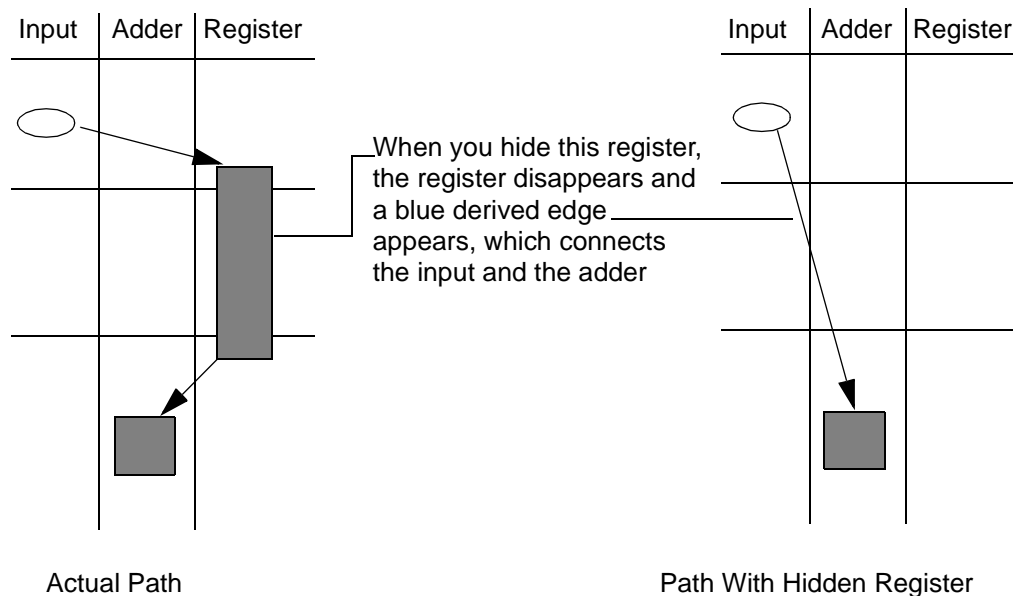
The Reservation Table uses derived edges to show the existence of a path between two operations or registers when you hide objects in the path. Displaying derived edges helps manage complexity in the Reservation Table window.

You can use derived edges to view the connectivity between objects and to narrow your review to a specific path.

Understanding Derived Edges

Derived edges appear as blue lines connecting two objects in the Reservation Table window. A derived edge appears when hidden objects exist along the path between two objects. Figure 6-19 shows an example of a derived edge.

Figure 6-19 Derived Edge Example



Derived edges are enabled by default when you open the Reservation Table window. They appear when you hide resources or operations. You can expand a derived edge to see the operations and registers it contains or hide a derived edge to further simplify the display.

To display the objects hidden by a derived edge,

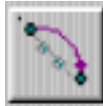
- Do one of the following:
 - Double-click the derived edge.
 - Select the derived edge and click the Expand Derived Edge toolbar button.



- Select the derived edge and choose View > Expand Selected.

To hide all derived edges,

- Do one of the following:
 - Click the Show/Hide Derived Edges toolbar button at the left side of the window.



- Choose View > Show/Hide Dependencies > Derived (expandable) Edges.

Viewing Connectivity

Use derived edges to view the connectivity between two objects, for example, input-to-output paths.

To see input-to-output paths,

1. Use the toolbar buttons at the left side of the window to hide all resources except I/O ports.
2. Observe the derived edges that result. These edges indicate the paths from input to output ports.

Viewing Isolated Paths or Objects

You can view individual paths or objects by either expanding the appropriate derived edges or zooming to isolate them.

To view a specific path using derived edges,

1. Display the derived edges between the objects of interest.
2. Select the derived edge that represents the path you want to examine and then expand it.

To view a selected object or path by zooming,

1. Select the object or path you want to examine.
2. Do one of the following:
 - Click the Zoom Selected toolbar button at the top of the window.



- Choose View > Zoom Selected.

The Reservation Table window shows only the selected objects, so you can concentrate on that specific set of objects.

To restore the previous contents of the window,

- Choose View > Undo.

To restore the initial contents of the window,

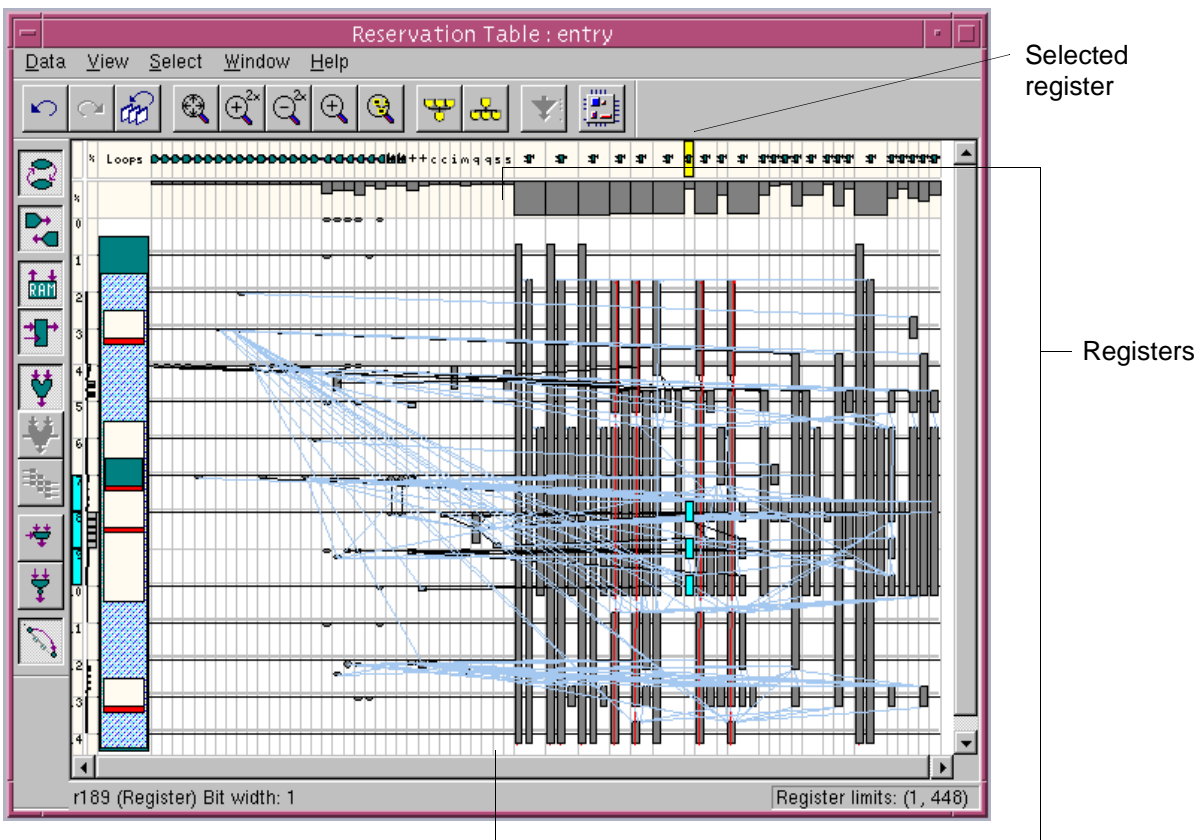
- Choose View > Reset.

Reviewing Register Use

Viewing register allocation and sharing can help you determine whether you can accomplish further area reduction.

The Reservation Table displays registers as rectangles spanning at least one clock cycle, as shown in Figure 6-20.

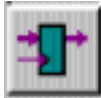
Figure 6-20 Registers in the Reservation Table



When you select a register, the clock cycles in which it is used are highlighted. The register symbol appears at the top of each column containing a register. The lower-right corner of the Reservation Table window shows the bit-width range of the displayed registers. You can limit the registers display by setting the bit-width range.

To simplify the default view of the Reservation Table window, register resources are hidden. To show registers, do one of the following:

- Choose View > Show/Hide Resources > Register.
- Click the Show/Hide Registers button in the toolbar.



To choose the size of the registers displayed,

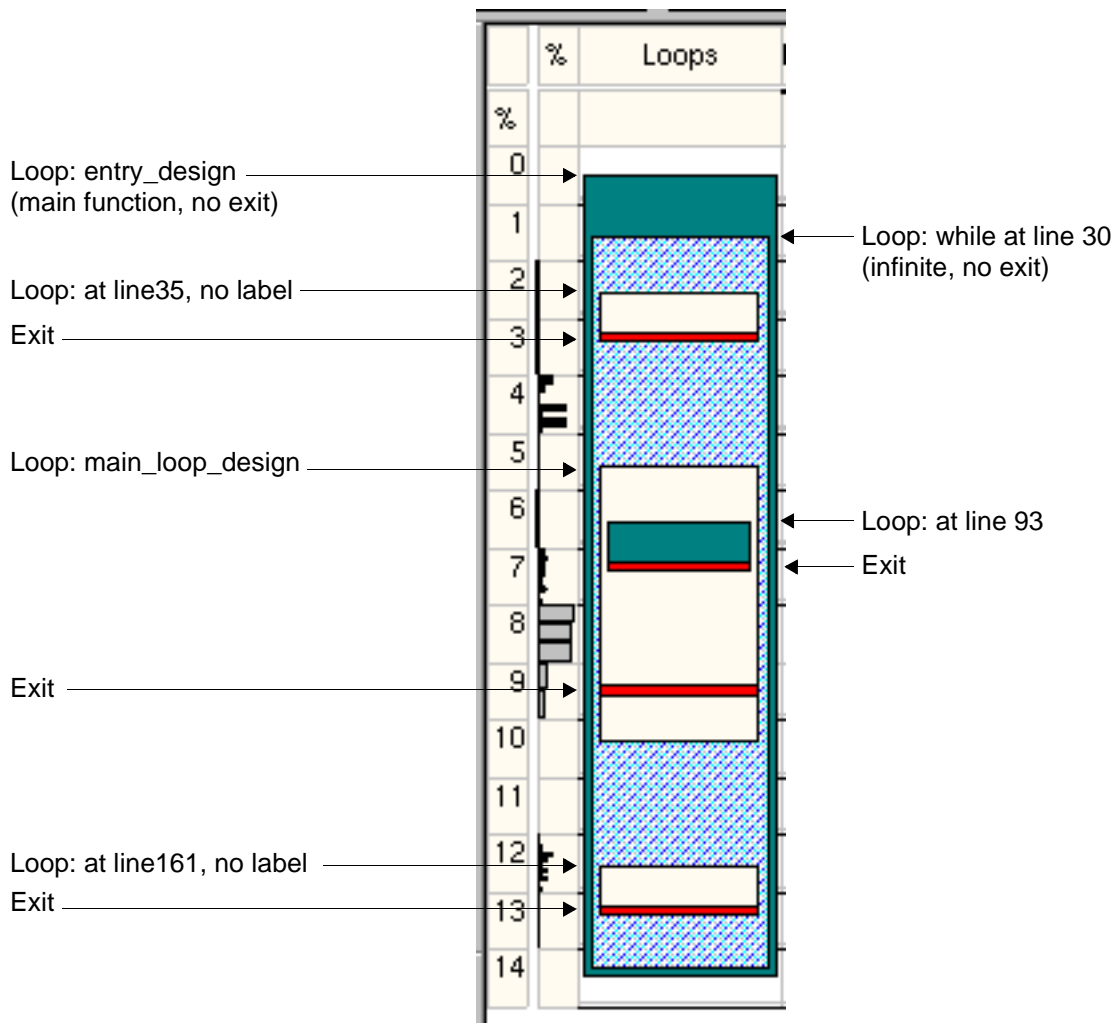
1. Choose Data > Register Bitwidth.
2. In the dialog box that appears, enter the values that represent the upper and lower limits of the bit-widths you want to display.
3. Click OK.

When you select a register in the Reservation Table window, the operation that produces the value stored in the register becomes highlighted in the Code Browser window.

Viewing Loops

Loops occupy a column on the left side of the Reservation Table window. Figure 6-26 shows just that area of the Reservation Table for the Inverse Quantization design from the *CoCentric SystemC Compiler Behavioral Modeling Guide*. Each vertical box in the Loops column represents a loop in the design. Boxes within boxes indicate nested loops, so you can view the hierarchy of the loops in your design.

Figure 6-21 Loops in the Reservation Table



An exit from a loop appears as a horizontal red line spanning the width of the loop box in the clock cycle where the exit is scheduled, as shown in Figure 6-21 on page 6-44. A single loop can have multiple exits, and the exits do not always occur at the end of a loop. For example, in a SystemC design a break statement in the code causes an exit from a loop.

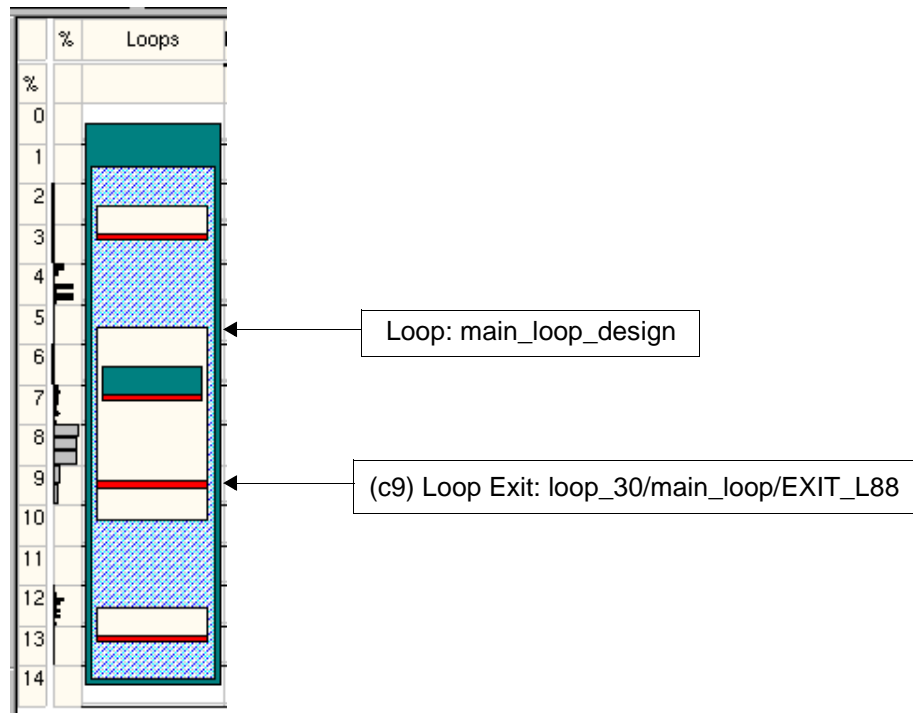
Identifying Loop Names

Use Info Tips to identify the name of the loop represented by each vertical box and to see information about the loop exits.

To use Info Tips,

1. Choose View > Display > Info Tips.
2. Move the pointer over a loop or exit to display summary information about the object. Figure 6-22 shows an example of a loop and a loop exit displayed by Info Tips.

Figure 6-22 Loop Information Tips



Viewing Loop Details

For more detailed information about a loop, such as the latency of the loop, the number of states created for the loop, and the resources used in the loop, use the Selection Inspector window.

To see detailed information about a loop,

- Click the box that represents the loop.

Detailed information appears in the Selection Inspector window. Figure 6-23 shows the first section of detailed loop data for a selected loop in Figure 6-22 on page 6-46.

Figure 6-23 Loop Details in Selection Inspector



To view information for all loops in a design,

- Choose Select > Resources (Columns) > Loop.

The Code Browser, FSM Viewer, and Selection Inspector windows display and highlight the code, states, state transitions, and detailed information related to the design loops.

Viewing Operations in a Loop

To find and review operations in a loop,

1. Select a loop by clicking the box that represents it.
2. Choose Select > Loop Operations.

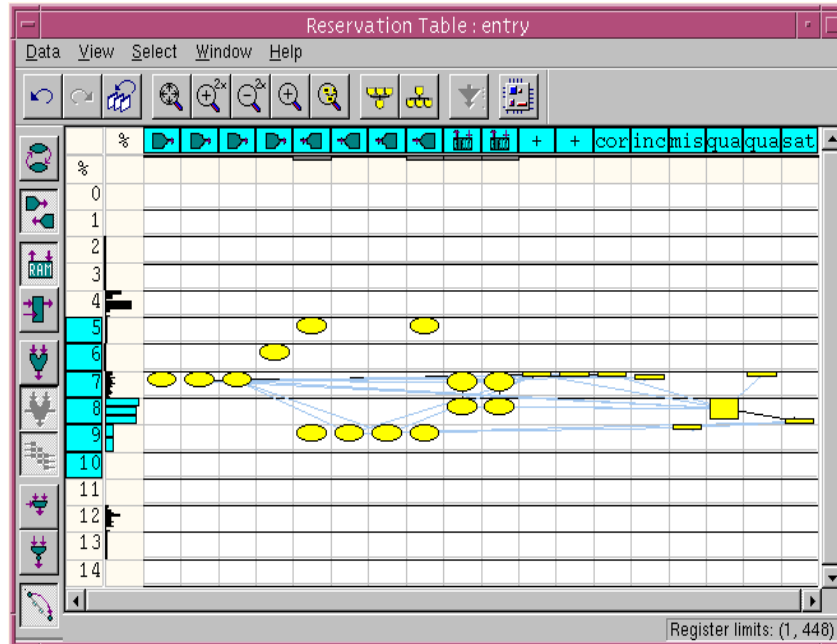
All operations in the selected loop become highlighted in both the Reservation Table window and the Code Browser window.

3. Choose View > Zoom Selected.

The Reservation Table window displays an isolated, zoomed-in view of the selected loop operations, as shown in Figure 6-24.

4. To revert to the normal Reservation Table view, click the Reset View button in the toolbar (see Figure 6-12 on page 6-30).

Figure 6-24 Loop Operations Zoomed View



Identifying Constraints and Data Dependencies

Dark lines between scheduled operations and registers in the Reservation Table window show how operations depend on data from previous operations or registers.

Arcs between operations in the Reservation Table window represent constraints. They might be specified by a designer or inherent in the code.

Viewing Data Dependencies

The fanin of an operation (an arrow going into an operation) indicates the operation's dependency on a previous operation. The fanout of an operation (an arrow going out of an operation) indicates where the operation's output data is used.

To view fanins to an operation,

1. Click the operation you are interested in to select it.
2. Do one of the following:
 - Click the fanin toolbar button at the top of the Reservation Table window.

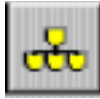


- Choose Select > Fanin.

To view fanouts from an operation,

1. Click an operation to select it.
2. Do one of the following:

- Click the fanout toolbar button at the top of the Reservation Table window.



- Choose Select > Fanout.

To view fanins to and fanouts from an operation,

1. Click an operation to select it.
2. Choose Select > Fanin/Fanout.

Viewing Constraints

By default, constraints do not appear in the Reservation Table window. When they are displayed, a check mark appears next to the type of constraints in the View menu.

To display and highlight user constraints,

- Choose View > Show/Hide Dependencies > User Constraints.

The Reservation Table window displays any user constraints.

To display and highlight inherent constraints, that is constraints present in the code.

- Choose View > Show/Hide Dependencies > Inherent Constraints.

The Reservation Table window displays any inherent constraints.

When you select (highlight) user and inherent constraints, additional information about the highlighted constraint appears in the Selection Inspector window.

Exploring Architectural Improvements

This section describes how to use BCView to identify areas where you can make architectural improvements to reduce latency or area. For more information about improving timing and area results, see Chapter 3, “Timing and Area Estimation.”

Reducing Latency

The following are methods you might use to reduce latency:

- Use the `-fastest` option with the `bc_time_design` command.
- Remove multicycle operations.
- Increase chaining.
- Use pipelined components.
- Vary the clock period.

Use BCView to identify multicycle operations, chaining opportunities, underutilized clock cycles, and to evaluate critical paths.

Identifying Multicycle Operations

Multicycle operations increase latency because they require an extra clock cycle to register the inputs to keep them stable, and they cannot be chained with other operations. Use BCView to identify multicycle operations that you can eliminate to reduce latency.

To identify multicycle operations,

- Do one of the following:

- Choose Select > Resources > Multicycle. All multicycle operations become highlighted in the Reservation Table window.
- Examine the lengths of the objects in the Reservation Table window. Any object that spans more than one row (clock cycle) is a multicycle operator.
- Select an operation that is not a register, and check the information in the Selection Inspector window. The Selection Inspector window describes the multicycle operator.

When you select multicycle components, the Code Browser, FSM Viewer, and Selection Inspector windows display and highlight the code, state transitions, and detailed information specific to the multicycle operations. By default, the Code Browser window displays the source code corresponding to multicycle components in red.

Identifying Chaining Opportunities

The maximum delay of an operation can be significantly shorter than the clock period. When this is the case, several operations can be scheduled in the same cycle. This scheduling optimization is called operation chaining. For a description of operation chaining and bit-level timing, see “Operation Chaining” on page 5-8.”

Use BCView to identify opportunities to chain operations, which can reduce latency.

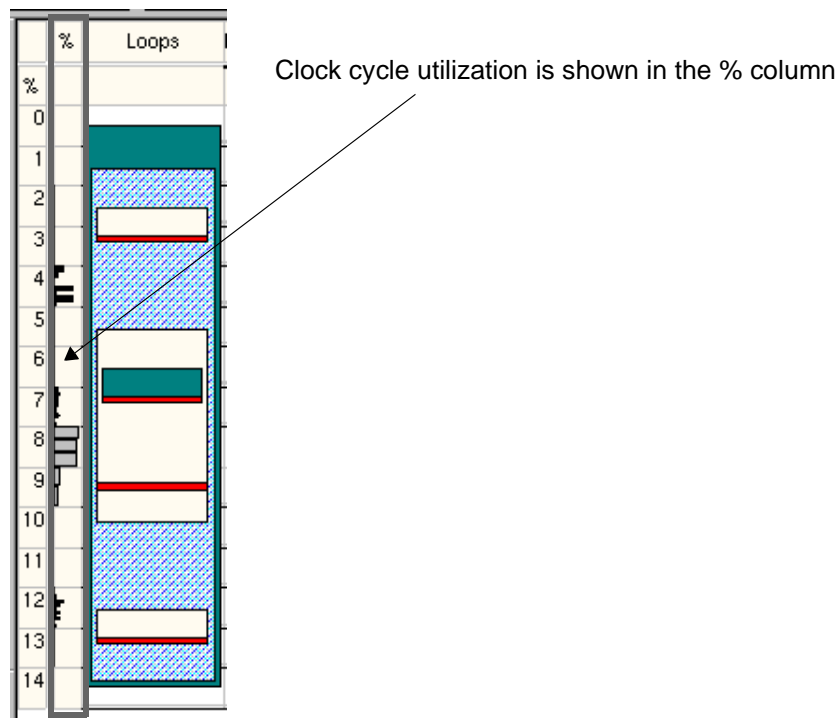
To identify chaining opportunities,

1. Look for data dependencies (edges) that cross clock cycles (rows) in the Reservation Table window. These might indicate chaining opportunities.
2. Select an operation with a data dependency.
3. View the fanins and fanouts, as described in “Viewing Data Dependencies” on page 6-49, to evaluate chaining opportunities. For information about advance chaining techniques, see Chapter 8, “Advanced Techniques.”

Viewing Clock-Cycle Utilization

The % column in the Reservation Table window is a histogram of the delays. The bars represent the amount of delay for the chained or unchained single-cycle operation in that clock cycle. Examine this histogram to determine how a design utilizes the clock cycle. White space on the right side of the column indicates that the clock cycle is not fully used and can be shortened.

Figure 6-25 Clock Cycle Utilization



Reducing Area

You can improve the area by increasing resource sharing. Use BCView to examine resource sharing in the design and to review the area detail in the design summary (described in “Viewing the Design Summary” on page 6-61).

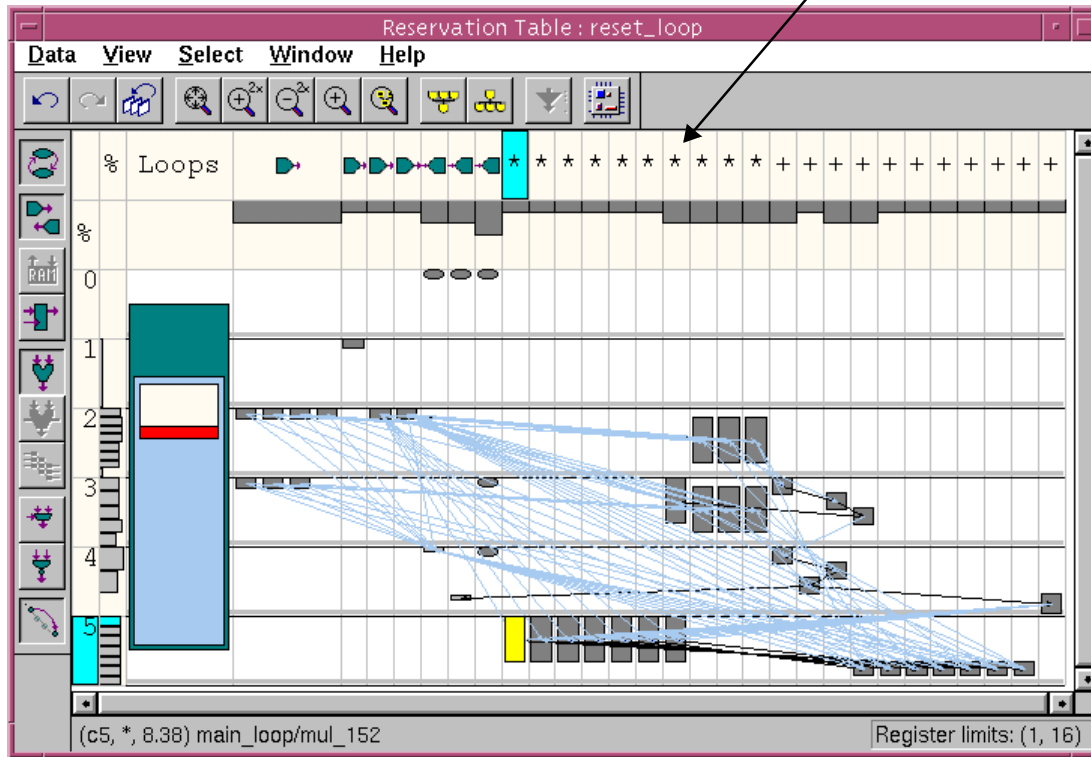
Two possible ways to increase resource sharing are to increase latency or to use SystemC Compiler commands to force sharing.

Figure 6-26 shows an example of a design with little resource sharing and the corresponding design summary. This design uses 10 multipliers and 11 adders, resulting in an area of 33,826.

Adding constraints to increase the latency of the example in Figure 6-26 increases resource sharing and reduces the area. Figure 6-27 shows the effect of stretching a loop from 6 cycles to 10 cycles using the `set_cycles` command (see “Constraining Loops and Operations” on page 4-37). This reduces the area to 14,855 by using only 3 multipliers and 3 adders.

Figure 6-26 Little Resource Sharing

This design uses 10 multipliers and 11 adders



Selection Inspector details:

- Object Name: Loop: reset_loop_design
- Object Class: Loop
- Description: Corresponds to Process reset_loop
- Clock Period: 12.50
- Scheduled in Superstate Mode

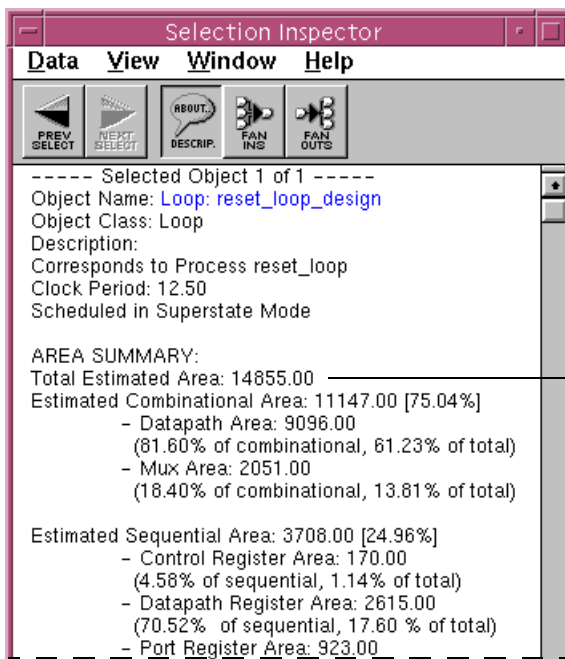
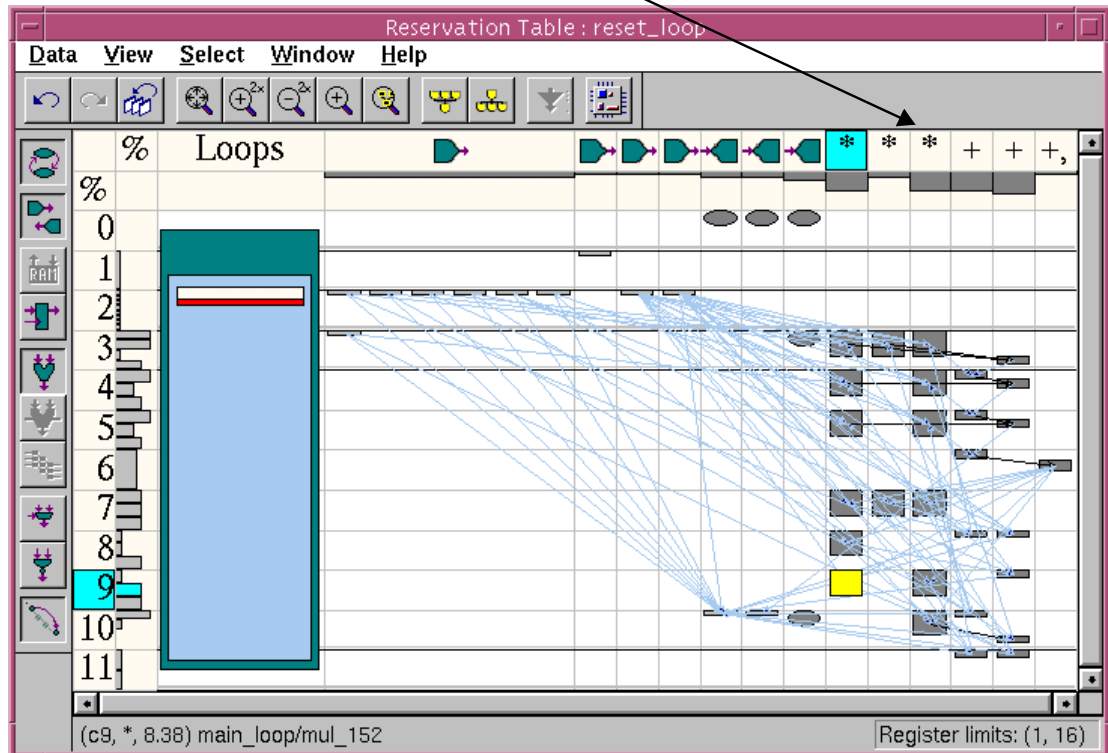
AREA SUMMARY:

- Total Estimated Area: 33826.00
- Estimated Combinational Area: 30314.00 [89.62%]
 - Datapath Area: 29191.00 (96.30% of combinational, 86.30% of total)
 - Mux Area: 1123.00 (3.70% of combinational, 3.32% of total)
- Estimated Sequential Area: 3512.00 [10.38%]
 - Control Register Area: 85.00 (2.42% of sequential, 0.25% of total)
 - Datapath Register Area: 2504.00 (71.30% of sequential, 7.40% of total)
 - Port Register Area: 923.00

Total estimated area is 33826

Figure 6-27 Shared Resources

The design now uses 3 multipliers and 3 adders



Total estimated area is 14855

Figure 6-28 Shareable Resources That Are Not Shared

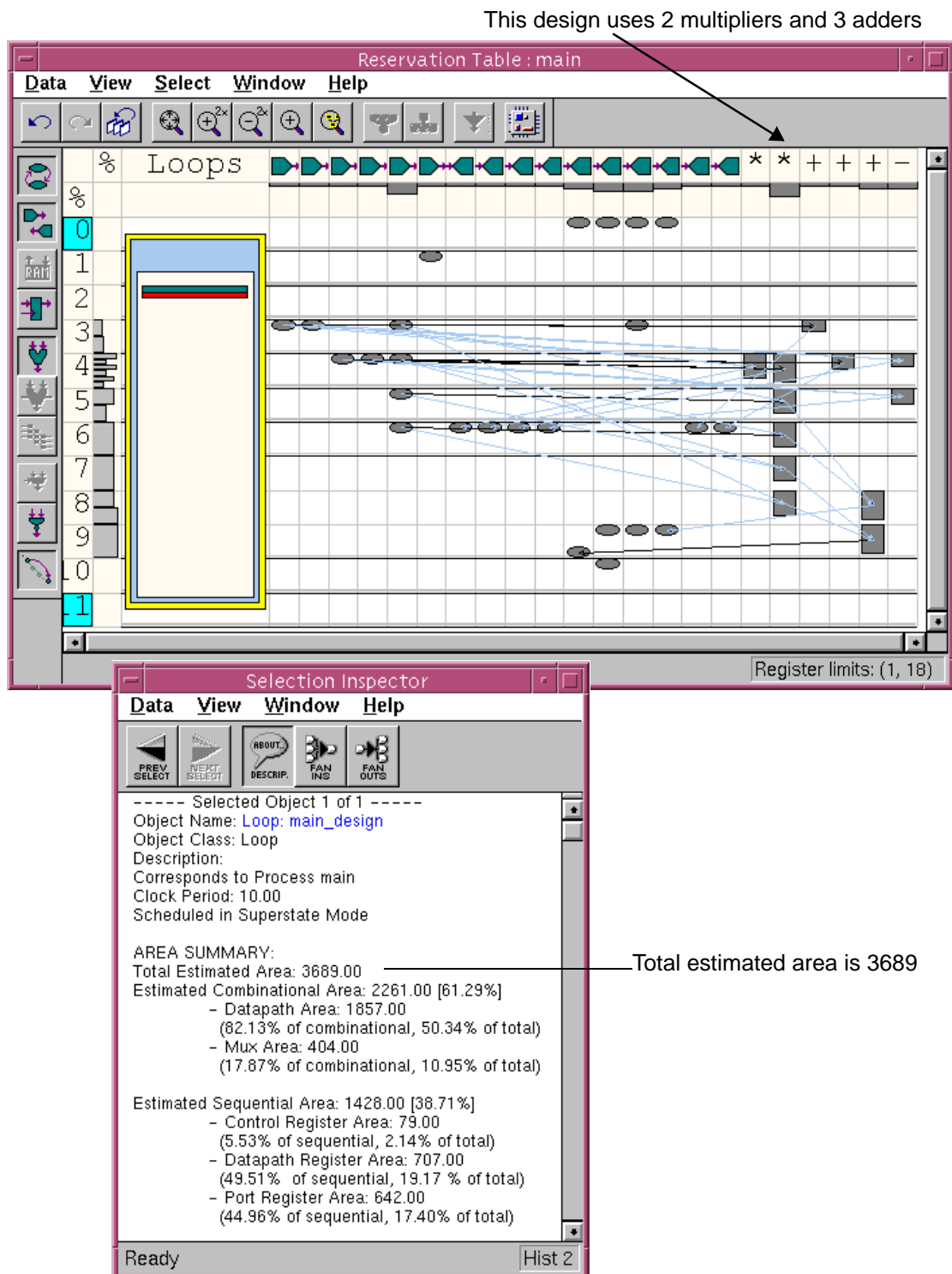
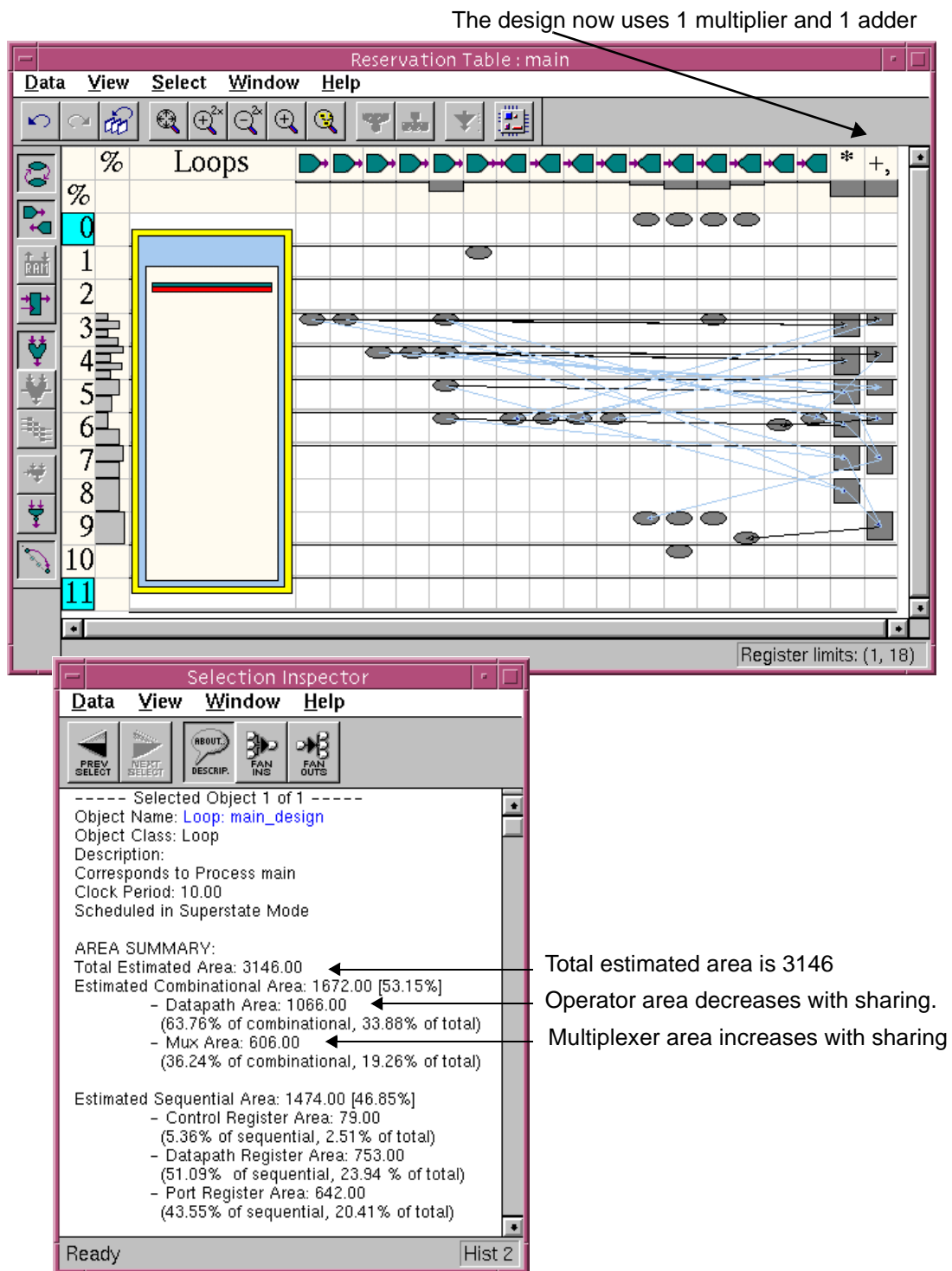


Figure 6-28 shows a case in which it appears that resources, like adders and multipliers, can be shared but they are not. You can force sharing using the `set_common_resource` command (see “Setting Common Resources” on page 4-55) with the `-max_count` and `-force_sharing` options.

Figure 6-29 shows the design after forcing resource sharing using the `set_common_resource` command.

Figure 6-29 Forced Resource Sharing



Reviewing Critical Paths

Reviewing critical paths can help identify opportunities for reducing latency or area.

To review a critical path,

1. In the Reservation Table window, click any resource, like an output port or a register to select it.
2. Choose Select > Transitive Fanin.

All paths leading into that resource become highlighted in the Reservation Table window.

Viewing the Design Summary

The design summary reports the area and resources, area breakdowns, and clock period information. Review this information to ensure your design criteria is achieved.

To display the design summary,


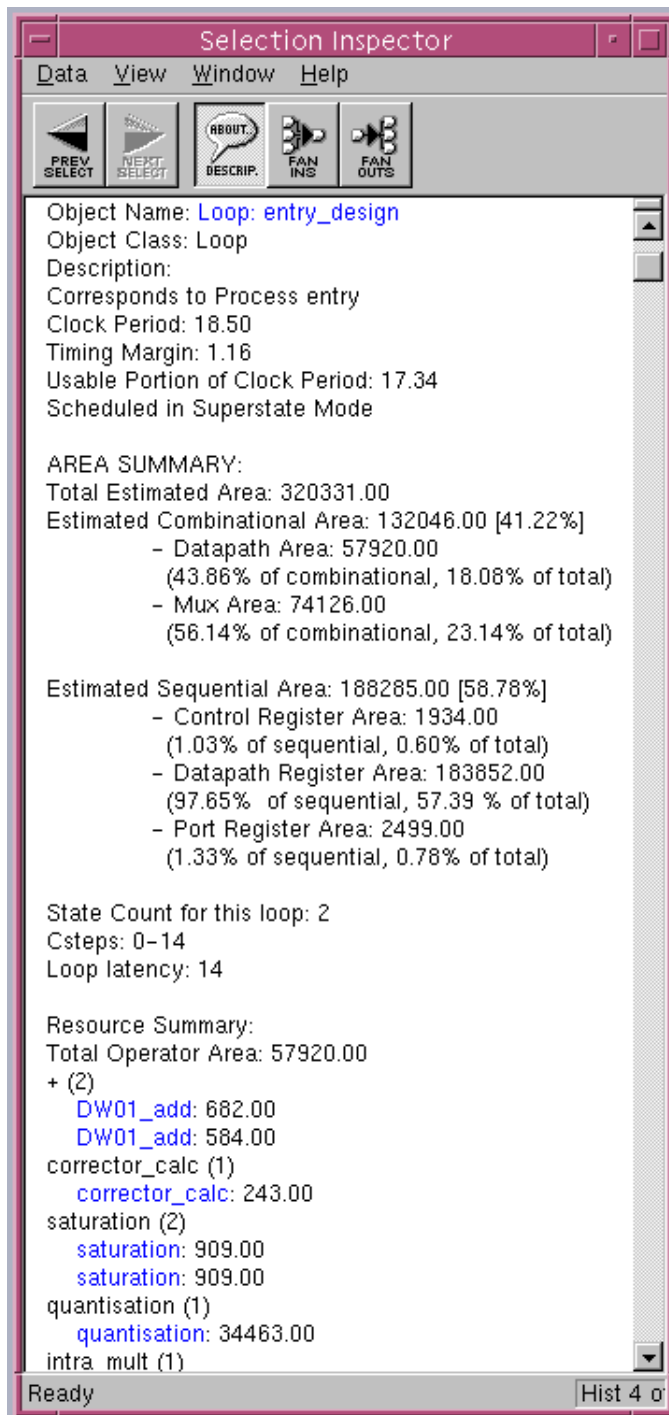
- Do one of the following in the Reservation Table window:
 - Choose Data > Design Summary.
 - Click the Design Summary toolbar button. 

Figure 6-30 shows the detailed information displayed in the Selection Inspector window.

Figure 6-30 Design Summary in Selection Inspector Window



7

Using Register Files and Memories for Arrays

This chapter describes how to implement large arrays as register files and memories to improve area, latency, and optimization time. It describes how to use, constrain, and obtain reports for register files and memories. It also describes how to generate a memory wrapper, which provides the interface to describe the memory I/O and sequential behavior of a vendor-provided memory for SystemC Compiler.

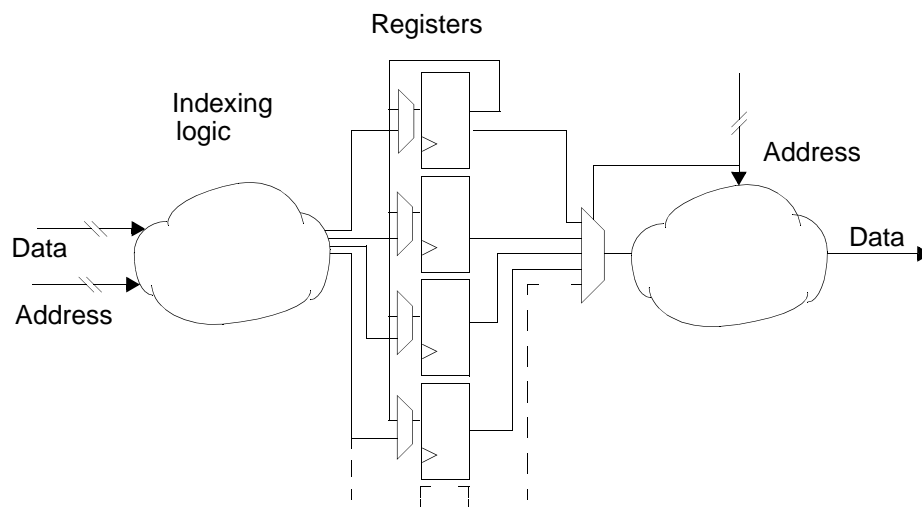
This chapter contains the following sections:

- Comparing Array Implementations
- Mapping Arrays to Register Files
- Mapping Arrays to Memory
- Generating Memory Wrappers

Comparing Array Implementations

By default, SystemC Compiler generates registers for arrays and logic for indexing into the arrays (including multidimensional arrays) in the behavioral code, as shown in Figure 7-1. SystemC Compiler generates dedicated indexing logic for each read from or write to an array. This can result in increased area for the indexing logic.

Figure 7-1 Array Generation



You can improve the quality of result for designs with 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 issues a warning because large unmapped arrays can cause long runtimes during elaboration and scheduling.

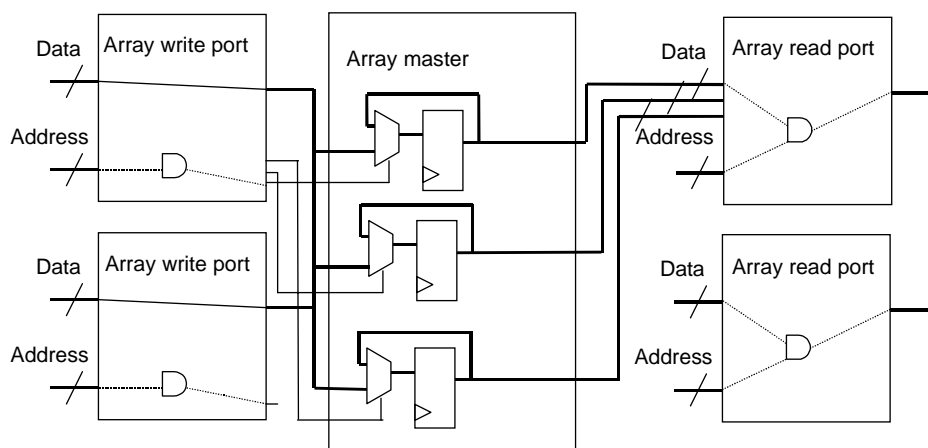
If you have large arrays (more than 1024 elements), we strongly recommend mapping them to register files or memory. If you have smaller arrays, compile the design without mapping the arrays. If you do not achieve the results you want, map all or some of the arrays to register files or to memory.

Comparing Arrays, Register Files, and Memories

It is generally more efficient to map arrays to memory than to register files because a memory uses less area than the equivalent register file. However, unless you have ready access to the appropriately sized memory and all the models you need (a vendor library for synthesis and a behavioral model for simulation), it is easier to map arrays to register files.

Register files are similar to memories except that SystemC Compiler builds the read and write ports and the register array on-the-fly. Figure 7-2 shows the architecture of a register file.

Figure 7-2 Register File Architecture



A memory (RAM) contains address decode logic that is transparent to your design. Figure 7-3 illustrates a typical dual-port memory. A memory is a single component that includes indexing and multiplexing logic as well as memory cells.

Figure 7-3 Dual-Port Memory Operations

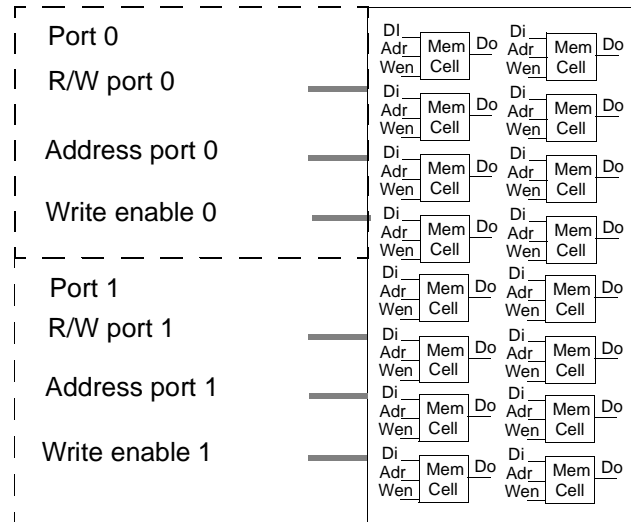


Table 7-1 shows the primary differences between implementing arrays as individual registers, register files, or memories.

Table 7-1 Comparing Arrays, Register Files, and Memories

Individual Registers	Register Files	Memories
Best for small arrays and are created by default	Best for designs that allow substantial resource sharing	More efficient and require less area than individual registers or register files
Generates registers and dedicated indexing logic for each array access	Generates shareable array read and write ports rather than dedicated indexing logic	Eliminates all indexing logic for array access. Access ports are internal to the memory
Can cause long elaboration and optimization times	Require more area than memories	Smallest area solution. Also produces the fastest SystemC Compiler runtimes
Does not extend design latency	May extend design latency if delays through the combinational address decode logic of the generated read and write ports are significant. This can happen with large arrays.	Extends design latency because each memory access requires one or more cycles to complete

Array Implementation Recommendations

When you can choose the array implementation for a design, the recommendations are:

1. Implement small arrays with individual registers. This is the default mode for SystemC Compiler.
2. Implement large arrays as memories.
3. If an appropriate memory cell is not available or the latency constraints do not allow for the use of memories, use register files.

Mapping Arrays to Register Files

Mapping arrays to register files works best for designs that allow for substantial resource sharing. This is because SystemC Compiler generates array read and write operations for each array access. These operations are allocated on shareable array read and write port resources that SystemC Compiler synthesizes. Sharing the register file read and write ports means that fewer copies of address decode logic are necessary than if one dedicated copy was made for each array access. This can happen if the array is not mapped to a register file or memory.

Mapping All Arrays to Register Files

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

```
dc_shell> bc_use_registerfiles = true
dc_shell> compile_systemc design.cc
```

Mapping Specific Arrays to Register Files

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

Example 7-1 shows a section of code that uses the `resource` compiler directive and the `map_to_registerfiles` attribute to map the `real` and `imag` arrays to two separate register files. In this example, the resources `RAM_A` and `RAM_B` are arbitrary names, the variable keyword defines the arrays that are mapped to the register file, and the `map_to_registerfiles` variable is set to `true` to indicate that the resource is a register file.

Example 7-1 Defining a Register File for a Specific Array

```
void fft::entry()
{
    // Define arrays to implement.
    sc_int<16> real[16];
    sc_int<16> imag[16];
    /* snps resource RAM_A: variables = "real",
       map_to_registerfiles="TRUE"; */
    /* snps resource RAM_B: variables = "imag",
       map_to_registerfiles="TRUE"; */
    ...
}
```

For more information about mapping specific arrays to register files and accessing them efficiently, see the *CoCentric™ SystemC Compiler Behavioral Modeling Guide*.

Understanding the Effects of Mapping to Register Files

Minimize the number of array read and write operations just as you would minimize memory accesses. For example, rather than reading an array element twice, store the content of an array element in an intermediate variable. For more information, see the *CoCentric™ SystemC Compiler Behavioral Modeling Guide*.

If SystemC Compiler determines that two array writes can access the same array element (the same location in the register file), SystemC Compiler schedules them in different clock cycles to prevent access conflicts that might corrupt data held in the register file.

Reporting Array Access Conflicts

After running the `compile_systemc` command on your design, use the `bc_report_arrays` command to report the conflicting and nonconflicting accesses to arrays mapped to register files. The command is

```
dc_shell> bc_report_arrays
```

Example 7-2 on page 7-9 shows a typical report about conflicting and nonconflicting array accesses. In this example,

- The fourth line in the conflicting accesses shows a conflicting access, `imag_read_180_2`, `imag_read_183` indicating that the second read of array `imag_read` on line 180 conflicts with the first read of array `imag_read` on line 183.
- There are no conflicting accesses across iterations of pipelined loops.

- The second line in the non-conflicting accesses shows that the `imag_read` on line 180 does not conflict with the `image_read` on line 183.
- There are no nonconflicting accesses across iterations of pipelined loops.
- The last four lines of the report indicate accesses that SystemC Compiler cannot determine if they conflict or not. By default, SystemC Compiler schedules these accesses in separate clock cycles to avoid the possibility that they might conflict.

Example 7-2 Report of Array Conflicts

Conflicting accesses in process 'entry' are as follows:

```
(imag_read_180, imag_read_183)
(imag_read_180, imag_write_186)
(imag_read_180_2, imag_read_183)
(imag_read_180_2, imag_read_183_2)
(imag_read_180_2, imag_write_183)
(imag_read_180_2, imag_write_186)
(imag_read_183, imag_write_186)
(imag_read_183_2, imag_write_183)
(imag_read_183_2, imag_write_186)
(imag_read_209, imag_write_218)
(imag_read_211, imag_write_218)
(imag_read_211, imag_write_220)
(imag_write_183, imag_write_186)
(real_read_179, real_read_182)
(real_read_179, real_write_185)
(real_read_179_2, real_read_182)
(real_read_179_2, real_read_182_2)
(real_read_179_2, real_write_182)
(real_read_179_2, real_write_185)
(real_read_182, real_write_185)
(real_read_182_2, real_write_182)
(real_read_182_2, real_write_185)
(real_read_208, real_write_217)
(real_read_210, real_write_217)
(real_read_210, real_write_219)
(real_write_182, real_write_185)
```

Conflicting accesses across iterations of pipelined loops in process 'entry' are not found.

Non_conflicting accesses in process 'entry' are as follows:

```
(imag_read_180, imag_read_180_2)
(imag_read_180, imag_read_183_2)
(imag_read_180, imag_write_183)
(imag_read_183, imag_read_183_2)
(imag_read_183, imag_write_183)
(imag_read_209, imag_read_211)
(imag_read_209, imag_write_220)
(imag_write_218, imag_write_220)
(real_read_179, real_read_179_2)
(real_read_179, real_read_182_2)
(real_read_179, real_write_182)
(real_read_182, real_read_182_2)
(real_read_182, real_write_182)
(real_read_208, real_read_210)
(real_read_208, real_write_219)
(real_write_217, real_write_219)
```

Non_conflicting accesses across iterations of pipelined loops in process 'entry' are not found.

Unable to resolve all accesses in process 'entry'.

The following accesses may conflict:

```
imag_read_254
real_read_253
```

Allowing Multiple Accesses in the Same Cycle

In some cases, SystemC Compiler cannot automatically determine whether two array accesses conflict or not, for example if the array access indices come from input reads. In such cases, SystemC Compiler schedules the two accesses in separate clock cycles by setting a precedence constraint between the two. The two accesses are scheduled in the order in which they appear in the behavioral description.

If you know that no conflicts can occur and want to schedule the two accesses in the same cycle, remove the precedence inserted by SystemC Compiler with the `ignore_array_precedences` command or the `ignore_array_loop_precedences` command. The commands are

```
dc_shell> ignore_array_precedences
          -from_set from_operations
          -to_set to_operations
```

```
dc_shell> ignore_array_loop_precedences
          operations
```

Example 7-3 shows two array reads and two array writes. The indices for the array reads are obtained from input ports of the design. SystemC Compiler cannot determine if the two array writes access the same location, and it schedules them in two separate clock cycles.

Example 7-3 Accesses That May or May Not Conflict

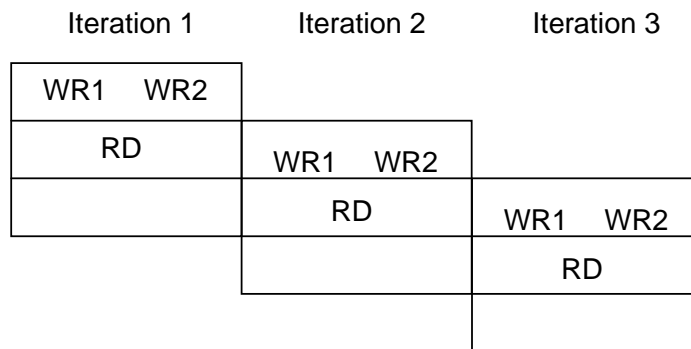
```
loop1 : while (c < 45) {
    index1 = in_a.read();
    index2 = in_b.read();
    index3 = in_c.read();
    index4 = in_d.read();
    a[index1] = x; // synopsys line_label WR1
    a[index2] = y; // synopsys line_label WR2
    wait();
    out_f = a[index3]; // synopsys line_label RD1
    c = c + 1;
    x = x + 2;
    y = y + x;
    wait();
}
```

If you are sure that the two accesses never conflict, you can inform SystemC Compiler by using the `ignore_array_precedences` command. For example,

```
dc_shell> ignore_array_precedences \
    -from_set
      { my_process/loop1/ARRAY_WRITE_RAM_A_WR1 } \
    -to_set
      { my_process/loop1/ARRAY_WRITE_RAM_A_WR2 }
```

If you pipeline *loop1* in Example 7-3 with a latency of 3 cycles and an initiation interval of 1 cycle, the array read of the first iteration needs to happen in the same clock cycle as the array writes of the second iteration, as shown in Figure 7-4.

Figure 7-4 Multiple Accesses in the Same Cycle That May Conflict



SystemC Compiler cannot determine if the indices conflict, so it prevents the accesses from being scheduled in the same cycle. This prevents *loop1* from being pipelined with an initiation interval of 1 cycle.

To inform SystemC Compiler that the accesses do not conflict, enter

```
dc_shell> ignore_array_loop_precedences \
    -from_set
      { my_process/loop1/ARRAY_WRITE_RAM_A_RD1 }
    -to_set
      { my_process/loop1/ARRAY_WRITE_RAM_A_WR1
        my_process/loop1/ARRAY_WRITE_RAM_A_WR2 }
```

This allows SystemC Compiler to pipeline *loop1* with an initiation interval of 1 clock cycle as shown in Figure 7-4.

Identifying Register File Operations

For the `ignore_array_precedences` and `ignore_array_loop_precedences` commands `-from_set` and `-to_set` arguments, you need to identify the register file operations.

When you map an array to a register file, SystemC Compiler creates three types of operations:

- An `ARR_READ` operation that represents each array read operation
- An `ARR_WRITE` operation that represents each array write operation
- An `ARR_MASTER` operation that represents the register file for the array

Each write to or read from an array creates an instance of the `ARR_WRITE` or `ARR_READ` cells, which are identified as `ARR_WRITE*` or `ARR_READ*` where the asterisk represents a unique instance number. Figure 7-2 on page 7-3 illustrates the register file architecture with these operations.

Finding Array Operation Cells

For the `ignore_array_precedences` and `ignore_array_loop_precedences` command `-from_set` and `-to_set` arguments, you can use the `find` command to locate the cells to set the precedence. For example, to find the `ARR_READ*` and `ARR_WRITE*` cells and instruct SystemC Compiler to ignore precedence between these cells, enter

```
dc_shell> op1 = find (cell -h "**ARR_READ*")
dc_shell> op2 = find (cell -h "**ARR_WRITE*")
dc_shell> ignore_array_precedences -from_set op1
        -to_set op2
dc_shell> schedule
```

Note that the `ignore_array_precedence` command is used before the `schedule` command.

Mapping Arrays to Memory

Mapping arrays to memory is more efficient than using register files or registers. A memory is a single component that includes the indexing and multiplexing logic as well as the memory cells (illustrated in Figure 7-3 on page 7-4).

Preparing to Use Memories

To use a vendor or custom memory in a design, you need to prepare interface files to incorporate the memory into your design. You need to perform the following memory preparation steps only once:

1. From the memory vendor, obtain the following memory files:
 - A vendor cell library (in .db format) that describes the boundary (ports, their names, their types, and number of bits) and electrical properties (capacitance and timing diagram information)
 - A Verilog (.v) or VHDL (.vhd) simulation model

If you do not have a vendor memory cell library, you can create an exploratory memory wrapper, described in “Creating a Memory Wrapper for an Exploratory Memory” on page 7-67.

Most memory vendors provide both a Synopsys .db file and a .lib file. If you have only the .lib file of the vendor memory library, you can convert it to a .db format by using the Synopsys Library Compiler tool.

2. Use the SystemC Compiler Memory Wrapper Generator (described in “Generating Memory Wrappers” on page 7-34) to generate the following additional files that are needed by SystemC Compiler for synthesis based on the memory cell you have chosen:

- A DesignWare synthetic library (.sl and .sldb) that describes the sequential cycle-by-cycle behavior of the memory buses and signals
- An HDL structural wrapper (.v or .vhd) interface that is used by the `compile` command to compile the wrapper to gates

3. Add the synthetic library .sldb file generated by the Memory Wrapper Generator to the `synthetic_library` variable. For example, for the memory `r6_16_wrap_6x16`, enter

```
dc_shell> synthetic_library = synthetic_library +  
r6_16_wrap_6x16.sldb
```

4. Add the synthetic library .sldb file created by the Memory Wrapper Generator and the vendor memory library .db file (if you have one) to the `link_library` variable. For example, to add the `r6_16_wrap_6x16.sldb` synthetic file and the `r6_16.db` vendor memory library, enter

```
dc_shell> link_library = link_library +  
{r6_16_wrap_6x16.sldb, r6_16.db }
```

5. When you create the wrapper files with the Memory Wrapper Generator, if you set the Design Library field of the Wrapper Properties dialog box, described in “Defining the Memory Wrapper Properties” on page 7-52, to something other than the `WORK` directory, you need to define the design library with the `define_design_lib` command.

For example, enter

```
dc_shell> define_design_lib my_design_library
-path /export/design_libraries/my_design_library
```

where *my_design_library* is the name of your design library and */export/design_libraries/my_design_library* is the name of a directory on a hard drive that will hold the library.

6. Use the `analyze` command to analyze the memory wrapper Verilog `.v` or VHDL `.vhd` source file in your design library. The `analyze` command executes quickly, so you can use it in your `dc_shell` command script without a performance penalty. Enter

```
dc_shell> analyze -f verilog r6_16_wrap_6x16.v
```

If your design library is something other than `WORK`, you need to specify the `-library` option with the `analyze` command. Enter

```
dc_shell> -library my_design_library
analyze -f verilog r6_16_wrap_6x16.v
```

Now you are ready to use the memory. Elaborate the SystemC file with the `compile_systemc` command and proceed with the rest of the SystemC Compiler flow.

Using Memory in Your Design

To use memory in your design, declare an array of variables and use the `resource` compiler directive with the `map_to_module` attribute in your code. Example 7-4 shows a local memory declaration, where the resources `RAM_A` and `RAM_B` are arbitrary names, the variable keyword defines the array variables mapped to the memory, and `map_to_module` defines the memory wrapper `r6_16_wrap_6x16` for the memory that you created with Memory Wrapper Generator.

Example 7-4 Declaring a Local Memory Resource

```
void fft::entry() {
    // Define memories to implement.
    sc_int<16> real[16];
    sc_int<16> imag[16];
    /* snps resource RAM_A: variables = "real", map_to_module = "r6_16_wrap_6x16"; */
    /* snps resource RAM_B: variables = "imag", map_to_module = "r6_16_wrap_6x16"; */
    ...
}
```

(For details about declaring local and shared arrays, resource selection, and accessing memory arrays, see the *CoCentric™ SystemC Compiler Behavioral Modeling Guide*.)

Now, you are ready to proceed with the next steps in the SystemC flow. Use the `compile_systemc` and other commands to synthesize your design.

Getting Memory and Library Information

To locate memory models in libraries and obtain information about the memories, use one or more of the following commands.

Using the `list` Command

To display and check the current definition of the `target_library`, `link_library`, `search_path`, `synthetic_library`, and other variables, use the `list` command. For example,

```
dc_shell> list synthetic_library
```

SystemC Compiler displays the current synthetic libraries, for example

```
synthetic_library = {"dw01.sldb" "r6_16_wrap_6x16.sldb"}
```

Using the report_synlib Command

To report the contents of the synthetic library, use the `report_synlib` command. Enter

```
dc_shell> report_synlib library [{module_list}]
```

The synthetic library report displays:

- A list of all operations and their pins
- A list of all modules with their pins, parameters, attributes, implementations, and bindings
- A list of all external implementations and external bindings
- A list of DesignWare subblocks declared in the library

The specified library must be already loaded into SystemC Compiler or be in the `search_path` definition.

By default, all modules are reported, or you can specify a list of modules to report.

Figure 7-5 shows a typical report of a synthetic library for a memory wrapper. For more information about this report, see the DesignWare documentation

Example 7-5 Report of Synthetic Memory Wrapper

```
*****
```

```
Report : library  
Library: r6_16_wrap_6x16.sldb  
Version: 2000.05  
Date   : Fri Sep 29 10:25:00 2000
```

```
*****
```

```
Library Type      : Synthetic  
Tool Created     : 2000.05  
Date Created     : August 01, 2000  
Library Version  : Not Specified
```

Synthetic Modules:

Module

```
r6_16_wrap      design_library: WORK
                  clocking_scheme: positive_edge
                  resource: S0_p0 (count=1)
                  resource: S1_p0 (count=1)
```

Module Pins:

Attributes:

c - clock_pin

Module	Pins	Dir	Width	Value	Stall Pin	Pin	Attributes
r6_16_wrap	dia	in	16				
	aadr	in	6				
	doa	out	16				
	wea	in	1				
	oea	in	1				
	clka	in	1			c	

Module Implementations:

Attributes/Parameters:

v - verify_only
V - verification implementation
u - dont_use
r - regular_licenses
l - limited_licenses
d - design_library
s - priority_set_id
p - priority
leg - legal

Module	Implementations	Attributes/Parameters
r6_16_wrap	wrap	

Module Bindings:

Module Binding

```
r6_16_wrap      read_port0 bound_operator: MEM_READ_SEQ_OP
                  State: 0
                  Pin Associations (module, oper):
```

Using Register Files and Memories for Arrays

```

        aadr, ADDR
        wea,"0"
        clka, CLK
    use_resource: S0_p0
        Unbound oper pin 'CLK' is bound to "1"
State: 1
    Pin Associations (module, oper):
        doa, Q
        clka, CLK
    use_resource: S1_p0
        Unbound oper pin 'CLK' is bound to "1"

    write_port0 bound_operator:
        MEM_WRITE_SEQ_OP
State: 0
    Pin Associations (module, oper):
        aadr, ADDR
        dia, D
        wea,"1"
        clka, CLK
    use_resource: S0_p0
        Unbound oper pin 'CLK' is bound to "1"

```

Using the `bc_report_memories` Command

To report specific information about the memories in the available synthetic libraries, use the `bc_report_memories` command. Enter

```
dc_shell> bc_report_memories -synthetic_libraries
```

The `-synthetic_libraries` option displays information about memories available in the synthetic libraries declared by the `synthetic_library` variable.

Figure 7-6 shows a report of a memory wrapper synthetic library. This report shows a memory (actually a memory wrapper) named `r6_16_wrap` in the current synthetic libraries.

It has one read-write port that is accessed through an address port named `aadr`. The memory read is a pipelined access that has a latency of 2 cycles with an initiation interval of 1 cycle. The memory write is a 1 cycle non-pipelined operation. The clock controlling this synchronous memory is named `clka`.

Example 7-6 Report of a Synthetic Library

```
Memory modules available in synthetic libraries
```

```
-----  
-----  
| r6_16_wrap      | Address      | MEM_READ  
=====|=====|=====|  
| Port 1  R/W    | aadr         | 2-state, 1-cyc  
|               |              | access, pipe  
-----|-----|-----  
-----  
| MEM_WRITE      | Clock        |  
=====|=====|=====|  
| 1-state, 1-cyc | clka         |  
| access, nonpipe|              |
```


To report specific information about the memories instantiated within an elaborated behavioral design, use the `bc_report_memories` command with the `-used_memories` option after executing the `compile_systemc` command. Enter

```
dc_shell> compile_systemc fft_mem.cc
dc_shell> bc_report_memories -used_memories
```

Figure 7-7 shows a report of memories used in an FFT design. This report shows two memories, RAM_A and RAM_B, of type `r6_16_wrap` instantiated in the design. Both memories have address ranges 0 to 15 and a data width of 16 bits. The read-write port accessed through address bus `aadr` is used in both memories. The behavioral process `entry` performs read and write accesses to both memories.

Example 7-7 Report of Memories Used in a Design

Memory instances used in design

```
-----
```

r6_16_wrap / RAM_B	Address	Process entry
[0 - 15] X 16		
=====		
Port 1 R/W	aadr	R, W

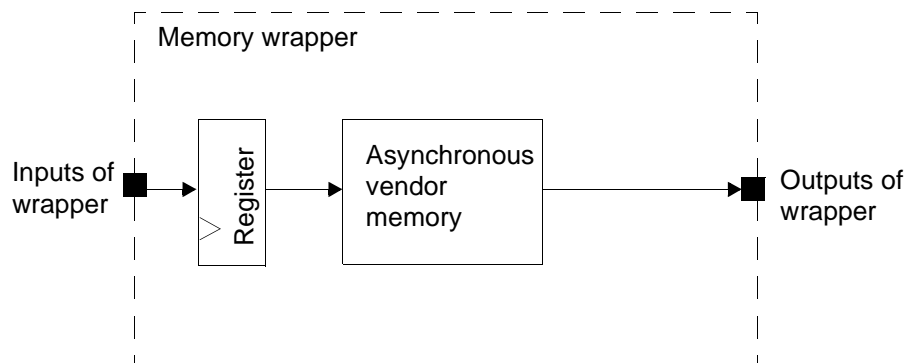
r6_16_wrap / RAM_A	Address	Process entry
[0 - 15] X 16		
=====		
Port 1 R/W	aadr	R, W

Using Asynchronous Memories

SystemC Compiler only uses memories with synchronous interfaces. This means there must be at least one register in the path from the inputs of the memory to the outputs. When you have an asynchronous memory, this is not the case. Because SystemC Compiler interfaces with the memory wrapper that you place around the memory rather than the actual memory, you can insert registers between the inputs of the memory wrapper and the inputs of the memory.

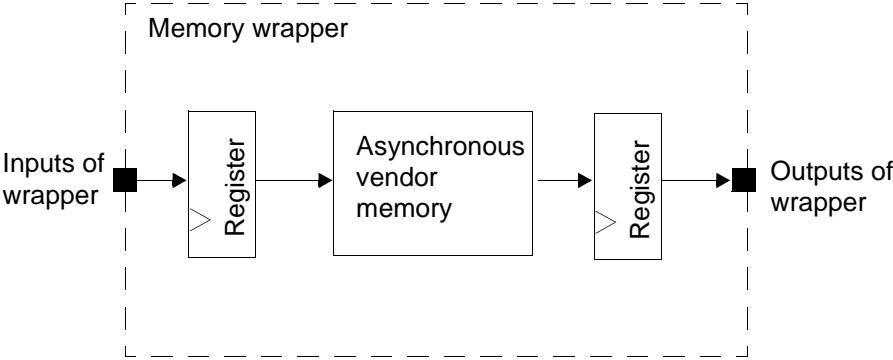
The Memory Wrapper Generator can be used to automatically insert registers in the wrapper, if you define the memory to be asynchronous (see “Defining the Memory Type and Properties” on page 7-40). Figure 7-5 illustrates the input register placement and wrapper interface the Memory Wrapper Generator tool generates for an asynchronous memory.

Figure 7-5 Asynchronous Memory With Registered Input



You can also insert registers on the outputs of an asynchronous memory using the Memory Wrapper Generator tool (see “Adding Registers to the Memory Wrapper” on page 7-58). Figure 7-6 illustrates a wrapper around an asynchronous memory with registers inserted on both the input and output.

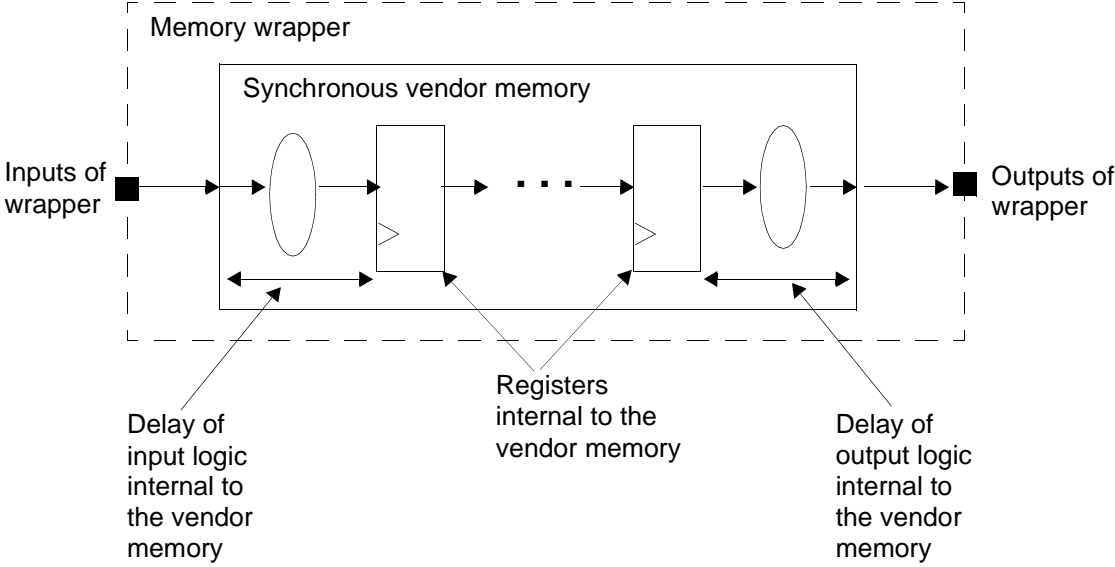
Figure 7-6 Manually Adding Registers to an Asynchronous Memory



Allowing for Vendor Memory Timing

If you are using a synchronous vendor memory, it is possible that the memory has logic between its I/O ports and its internal registers, as shown in Figure 7-7. The delay specifications for this logic are typically provided in the timing diagrams of the memory vendor’s datasheet.

Figure 7-7 Memory Access Time Specification



You need to provide the input and output delay specifications to SystemC Compiler, so it can reserve time in the clock cycle to allow for computation of this logic. Use the `set_memory_input_delay` and `set_memory_output_delay` commands to specify the vendor memory input delay and output delay. SystemC Compiler uses these delays when computing the values for the address, data, and control lines leading into the memory, and to ensure that the values arrive at the ports of the memory at the appropriate time.

Setting Memory Input Delay for Vendor Memory Timing

To set the input delay on a memory according to the vendor timing specification, use the `set_memory_input_delay` command.

```
dc_shell> set_memory_input_delay [delay_value]
          [-external ext_delay_value] [-name mem_name]
```

The *delay_value* specifies the input delay, which must be a positive number in the units of the technology library.

The `-external` option specifies the external input delay, which must be a positive number in the units of the technology library. The external input delay is applicable if the vendor memory is positioned external to the design. You can move a memory out of the design by executing the `externalize_cell` command (see “Externalize a Cell” on page 8-8). The external input delay accounts for a delay that occurs to transfer the input signal to the external memory. For example, this might account for the delay through an I/O pad taking the signal off-chip.

The `-name` option specifies the resource for one or more memories to which this command applies. The default is to apply the command to all memories in the current design. If you are specifying more than one memory name, enclose the list of names in double-quotes or braces.

Example 7-8 shows an example of the `resource` directive in the behavioral description that specifies the memory and the `set_memory_input_delay` command to set the internal input delay to 3.5 and the external input delay to 3.

Example 7-8 Set Memory Input Delay

```
/* snps resource RAM_A: variables = "real",  
   map_to_module = "r6_16_wrap"; */
```

```
dc_shell> set_memory_input_delay -name RAM_A 3.5 -ext 3
```

Setting Memory Output Delay for the Vendor Timing Specifications

To set the output delay of a memory according to the vendor timing specification, use the `set_memory_output_delay` command. The `set_memory_output_delay` command allows you to specify a delay due to logic on the outputs of a memory. You can also use this command to specify an additional delay to access an external, off-chip memory.

```
dc_shell> set_memory_output_delay [delay_value] [-external  
ext_delay_value] [-name mem_name]
```

The *delay_value* specifies the output delay, which is the output delay of the memory. The *delay_value* must be a positive number in the units of the technology library.

The `-external` option specifies the external output delay. The external output delay is the delay caused by positioning the memory external to the design. You can move a memory out of the design by executing the `externalize_cell` command (see “Externalize a Cell” on page 8-8). The external delay can represent delay elements such as I/O pad and off-chip delays.

SystemC Compiler reserves time in the clock cycle for the total of the internal and external memory delays, and it only schedules operations that use the output data of the memory that can fit in the remainder of the clock cycle.

The `-name` option specifies one or more memories to which this command applies. The default is to apply the command to all memories in the current design. If you are specifying more than one memory name, enclose the list of names in double-quotes or braces.

Constraining Read and Write Operations on Memory

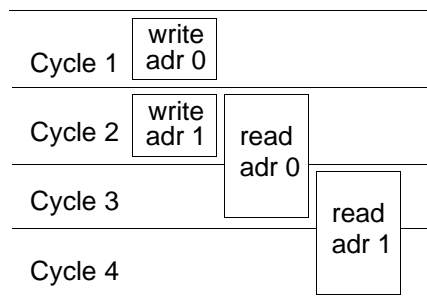
The number of memory read or write operations that can access the same memory simultaneously depends on the type of memory and the number of ports it has.

- Single port memories perform only one read or write in each clock cycle.
- Multiple port memories can perform several memory read and write operations at the same time, depending on the configuration of the ports, such as read ports, write ports, and read/write ports.
- Memories that allow pipelined accesses can overlap memory read and write operations, where a second memory access can be initiated before the first memory access is completed.

The number of cycles required for a sequence of memory read or write operations depends on the type of memories and the access patterns on the memories. Typical memories perform a read in two clock cycles and a write in one clock cycle.

SystemC Compiler automatically pipelines memory accesses if the memory allows it. Figure 7-8 illustrates pipelined memory accesses, where two reads happen in three cycles and two writes happen in two cycles. Notice the pipelining where the second read begins in the same cycle as the second stage of the first read.

Figure 7-8 Pipelined Memory Accesses



Reporting Conflicting Memory Accesses

If the memory being used has multiple ports and allows for multiple simultaneous memory accesses, SystemC Compiler can schedule several memory accesses in the same cycle. However, it will not schedule two memory accesses to the same memory location in the same cycle. SystemC Compiler tries to determine if two memory accesses can conflict, that is access the same memory location. If they never conflict, SystemC Compiler allows them to be scheduled in the same cycle.

To generate a report about which memory accesses conflict and which do not conflict, use the `bc_report_memories` command with the `-conflicting` or `-non_conflicting` options. For example, to list the nonconflicting pairs, enter

```
dc_shell> bc_report_memories -non_conflicting
```

Example 7-9 shows a typical memory report of nonconflicting accesses.

Example 7-9 Report Nonconflicting Memory Accesses

Non_conflicting accesses in process 'entry' are as follows:

```
(imag_read_180, imag_read_180_2)
(imag_read_180, imag_read_183_2)
(imag_read_180, imag_write_183)
(imag_read_183, imag_read_183_2)
(imag_read_183, imag_write_183)
(imag_read_209, imag_read_211)
(imag_read_209, imag_write_220)
(imag_write_218, imag_write_220)
(real_read_179, real_read_179_2)
(real_read_179, real_read_182_2)
(real_read_179, real_write_182)
(real_read_182, real_read_182_2)
(real_read_182, real_write_182)
(real_read_208, real_read_210)
(real_read_208, real_write_219)
(real_write_217, real_write_219)
```

Non_conflicting accesses across iterations of pipelined loops in process 'entry' are not found.

Unable to resolve all accesses in process 'entry'.

The following accesses may conflict:

```
imag_read_254
real_read_253
```


Using the `ignore_memory_precedences` Command

If SystemC Compiler determines that two memory accesses can conflict (access the same memory location), it sets a precedence constraint so that they do not execute in the same clock cycle. It also ensures that they execute in the order in which they appear in the behavioral description.

In certain situations, SystemC Compiler cannot make a static determination about whether two memory accesses conflict. This can happen when the index expressions for the memory accesses depend on runtime information such as the value on input ports of the design. In these situations, SystemC Compiler considers it possible for these memory accesses to access the same locations at the same time, and it inserts precedence constraints as if they were conflicting accesses.

If you are sure that the accesses do not conflict, use the `ignore_memory_precedences` command so SystemC Compiler is allowed to schedule the two accesses in the same clock cycle. This results in a smaller latency for the design. Enter

```
dc_shell> ignore_memory_precedences [-process process_name]
-from_set from_operations -to_set to_operations
```

The `-process` option applies this command to only the process `process_name`. If this option is not specified, this command applies to all behavioral processes.

The `-from_set` and `-to_set` options remove all precedence constraints inserted from memory accesses in the `from_operations` set to those in the `to_operations`.

For example, to remove precedence conflicts between memory accesses to arrays, read and imag, enter

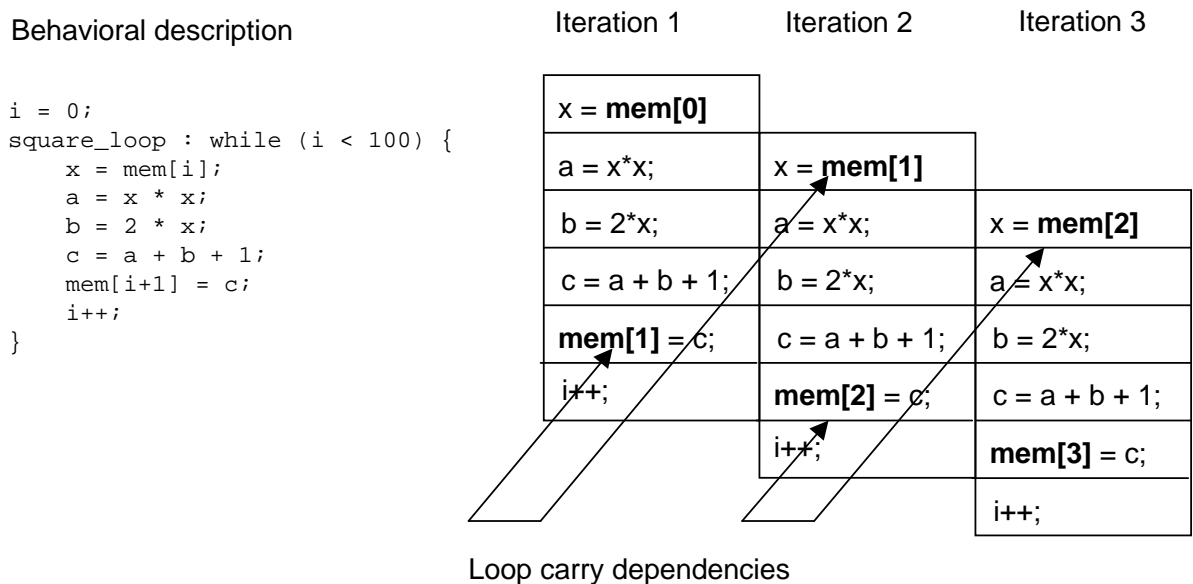
```
dc_shell> op1 = find (cell -h "*read_read*")
dc_shell> op2 = find (cell -h "*imag_read*")
dc_shell> ignore_memory_precedences -from_set op1
      -to_set op2
dc_shell> schedule
```

Note that the `ignore_memory_precedences` command is used before the `schedule` command.

Using the `ignore_memory_loop_precedences` Command

When a loop containing memory accesses is pipelined, it is possible that memory writes in one loop iteration introduce dependencies with memory reads in subsequent iterations, as illustrated in Figure 7-9.

Figure 7-9 Invalid Schedule With Loop Carry Dependency



SystemC Compiler automatically inserts loop-carry dependencies to prevent these violations. If your design has pipelined loops, the reports generated by the `bc_report_memories` command with the `-conflicting` and `-non_conflicting` options has a section that lists pairs of memory accesses that are determined to conflict (or not conflict) across iterations of the pipelined loop.

If SystemC Compiler determines that a pair of memory accesses across loop iterations can access the same memory location, or if it cannot determine that they do not access the same memory location, then SystemC Compiler automatically inserts a precedence constraint to ensure that the loop-carry dependency is not violated.

If you are sure that the two accesses never access the same location, use the `ignore_memory_loop_precedences` command to direct SystemC Compiler to remove the precedence constraint.

```
dc_shell> ignore_memory_loop_precedences  
          [-process process_name] {operations}
```

The *operations* option defines the memory access operations that you allow SystemC Compiler to assume do not conflict across iterations of the pipelined loop that contains them. For example,

```
dc_shell> op1 = find (cell -h "*read_read*")  
dc_shell> op2 = find (cell -h "*imag_read*")  
dc_shell> ignore_memory_loop_precedences { op1, op2 }  
dc_shell> schedule
```

Note that the `ignore_memory_loop_precedences` command is used before the `schedule` command.

Generating Memory Wrappers

This section describes how to generate memory wrappers using the Memory Wrapper Generator graphical user interface (GUI) tool included with SystemC Compiler.

Understanding the Memory Wrapper Generator Tool

SystemC Compiler requires that the memories used in the design be described as synthetic DesignWare components. Most memory vendors do not provide memories as synthetic DesignWare components, therefore the Memory Wrapper Generator tool is provided with SystemC Compiler to encapsulate vendor memories as DesignWare components. This tool generates a memory wrapper (in Verilog or VHDL format) and a synthetic library description (in the Synopsys .sldb format) for use with SystemC Compiler.

SystemC Compiler uses the Verilog/VHDL wrapper file to instantiate the memory in the design during elaboration and the .sldb file to determine the characteristics of the memory.

The Memory Wrapper Generator tool also enables you to insert custom logic in the memory wrapper, if it is required to interface with the memory.

You can use the Memory Wrapper Generator tool in two ways:

- To encapsulate an existing memory model from a memory vendor, described in “Creating a Memory Wrapper for a Vendor Memory” on page 7-39.

- To create an exploratory memory interface to experiment with different memory architectures that you might want to use, described in “Creating a Memory Wrapper for an Exploratory Memory” on page 7-67.

Using the Memory Wrapper Generator Tool

Start the Memory Wrapper Generator tool from a directory where you have write permission. You will need to save your memory wrapper files in this directory.

For example, to change to the directory `my_design` and start the Memory Wrapper Generator tool,

- Enter the following at the UNIX prompt:

```
unix% cd my_design
unix% memwrap &
```

When the Memory Wrapper Generator window is initially displayed it is empty, as shown in Figure 7-10.

Note:

If the Memory Wrapper Generator window does not display, see “Starting the Memory Wrapper Tool” on page A-14.”

Figure 7-10 Empty Memory Wrapper Window

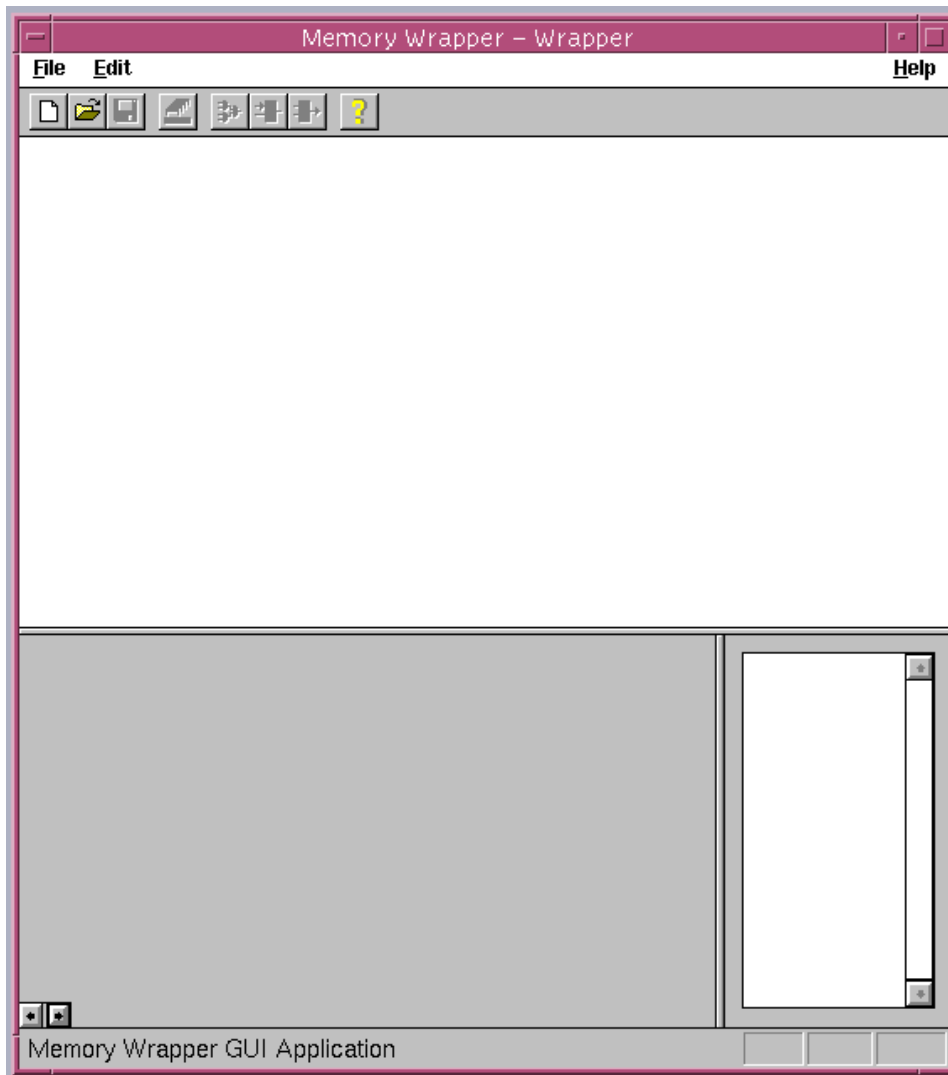
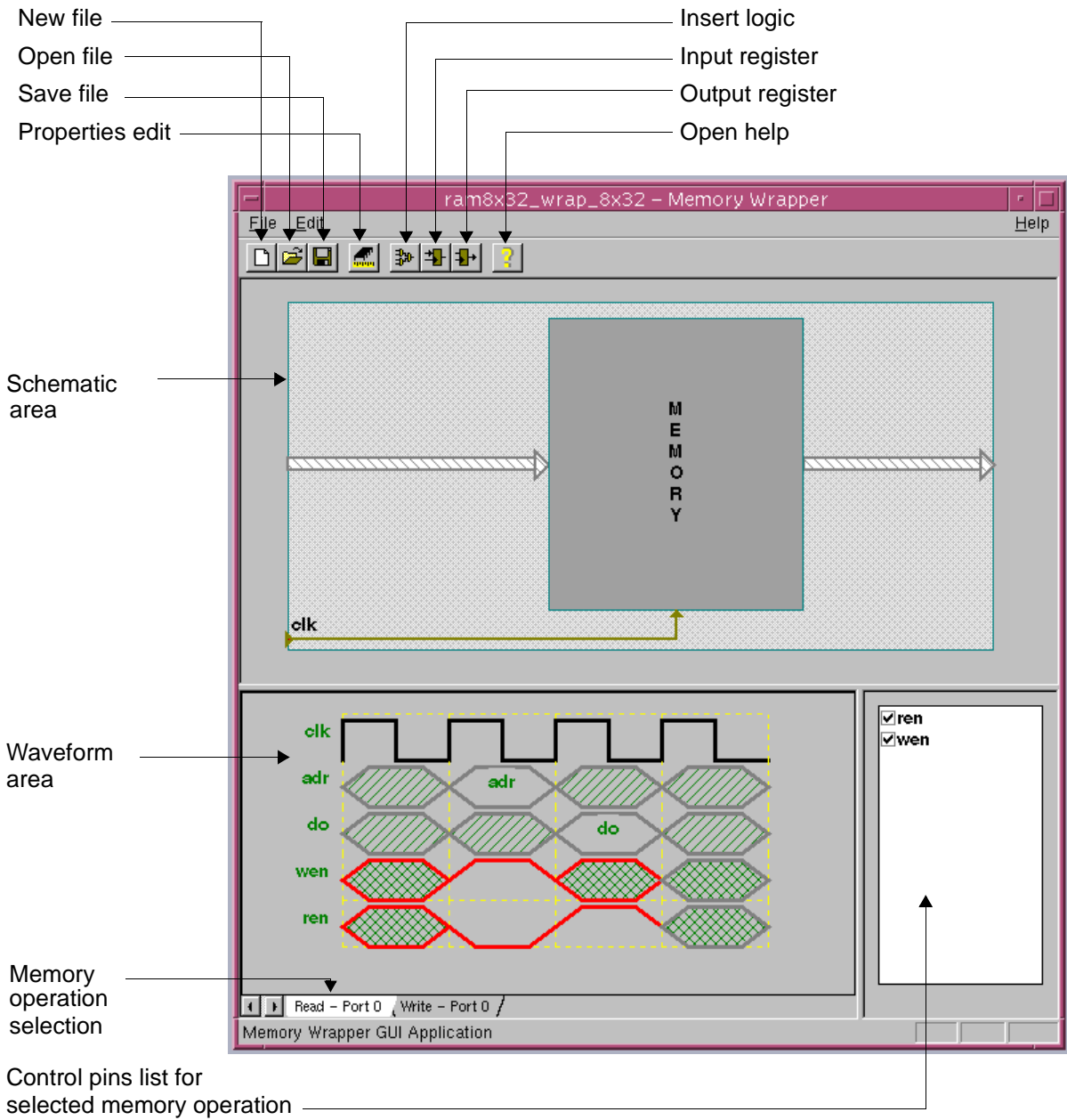


Figure 7-11 shows an example of the Memory Wrapper window after a wrapper is created.

Figure 7-11 Completed Memory Wrapper



The three display areas in the Memory Wrapper window show the following information after you create a wrapper:

Schematic Area

The top display area shows a schematic of the memory wrapper.

Waveform Area

The lower-left display area shows the waveforms for each of the memory operations of the memory wrapper.

Control Pins List

The lower-right display area shows the control pins list with items like a chip enable or write enable that are associated with a wrapper memory operation you select in the Waveform Area.

Logical Port

The Memory Wrapper Generator groups vendor memory pins into physical ports and connects them to logical ports in the memory wrapper. Logical ports refer to ports in the memory wrapper, and pins and physical ports refer to the vendor memory.

One memory wrapper logical port includes the address bus, data bus, and control signals to access one physical port of the vendor memory.

Creating a Memory Wrapper for a Vendor Memory

The Memory Wrapper Generator saves the current memory wrapper specification in a .wrap file. This file encapsulates all of the information necessary to generate the synthetic library file (.sldb) and the HDL wrapper file needed by SystemC Compiler. You can read in and modify a .wrap file that was previously created using the Memory Wrapper Generator.

The general steps for creating a memory wrapper for an existing vendor memory are

1. Define the type and properties of the memory.
2. Assign vendor memory pins to the logical ports of the memory wrapper.
3. Define properties of the wrapper and add control pins to the wrapper, if necessary.
4. Edit the wrapper by adding or deleting logic or registers and specifying the waveforms.
5. Save the wrapper files.

Defining the Memory Type and Properties

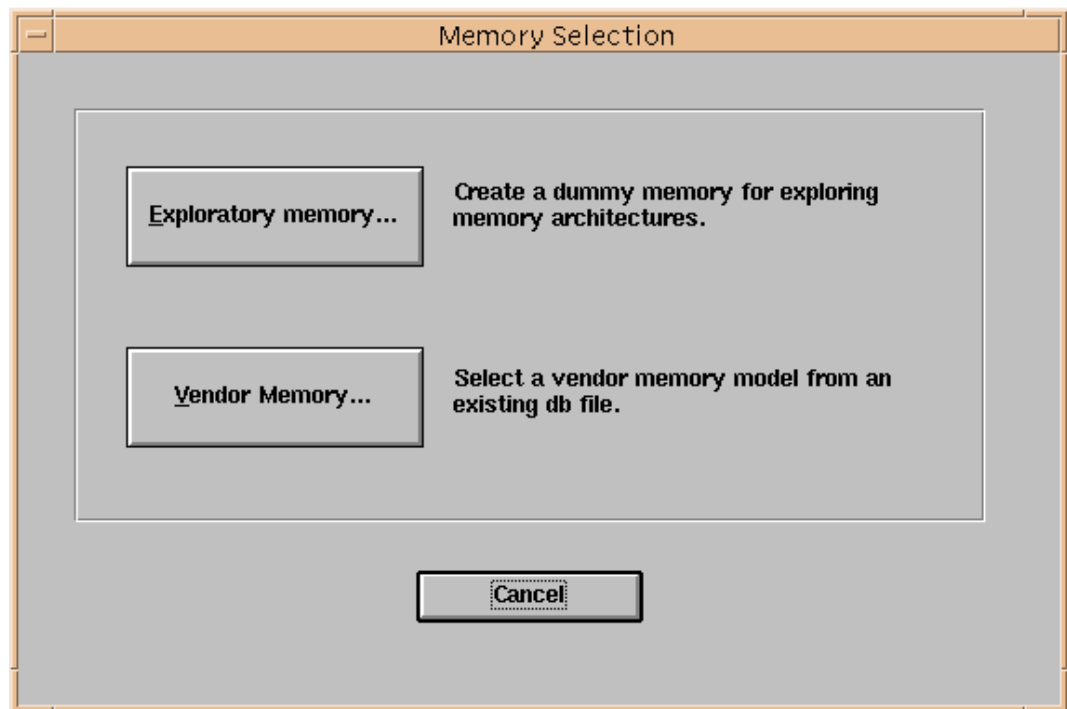
To define the type and properties of the memory,

1. Choose File > New in the Memory Wrapper window.

(To open a previously saved .wrap file, choose File > Open.)

The Memory Selection dialog box is displayed (Figure 7-12).

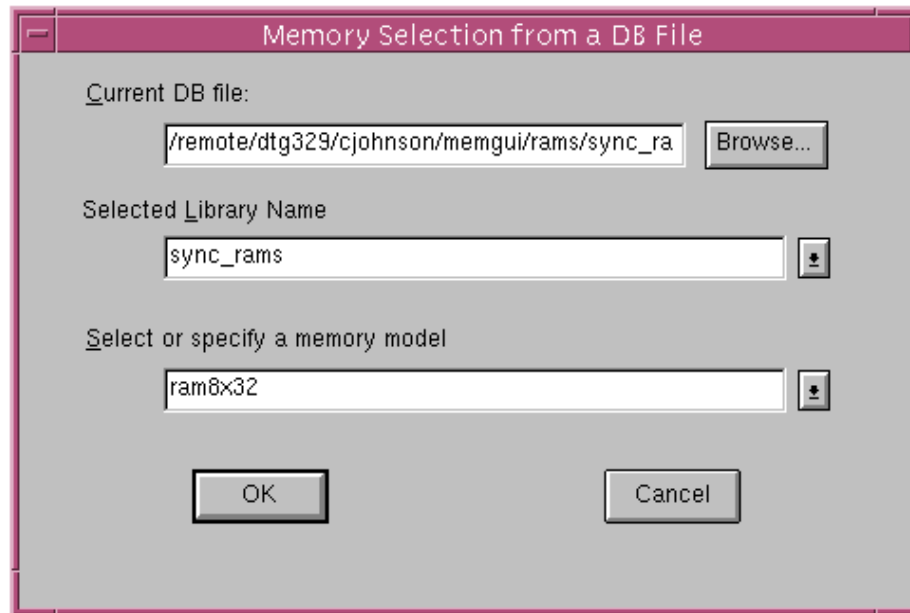
Figure 7-12 Memory Selection Dialog Box



2. Click Vendor Memory.

The Memory Selection from a DB File dialog box (Figure 7-13) is displayed.

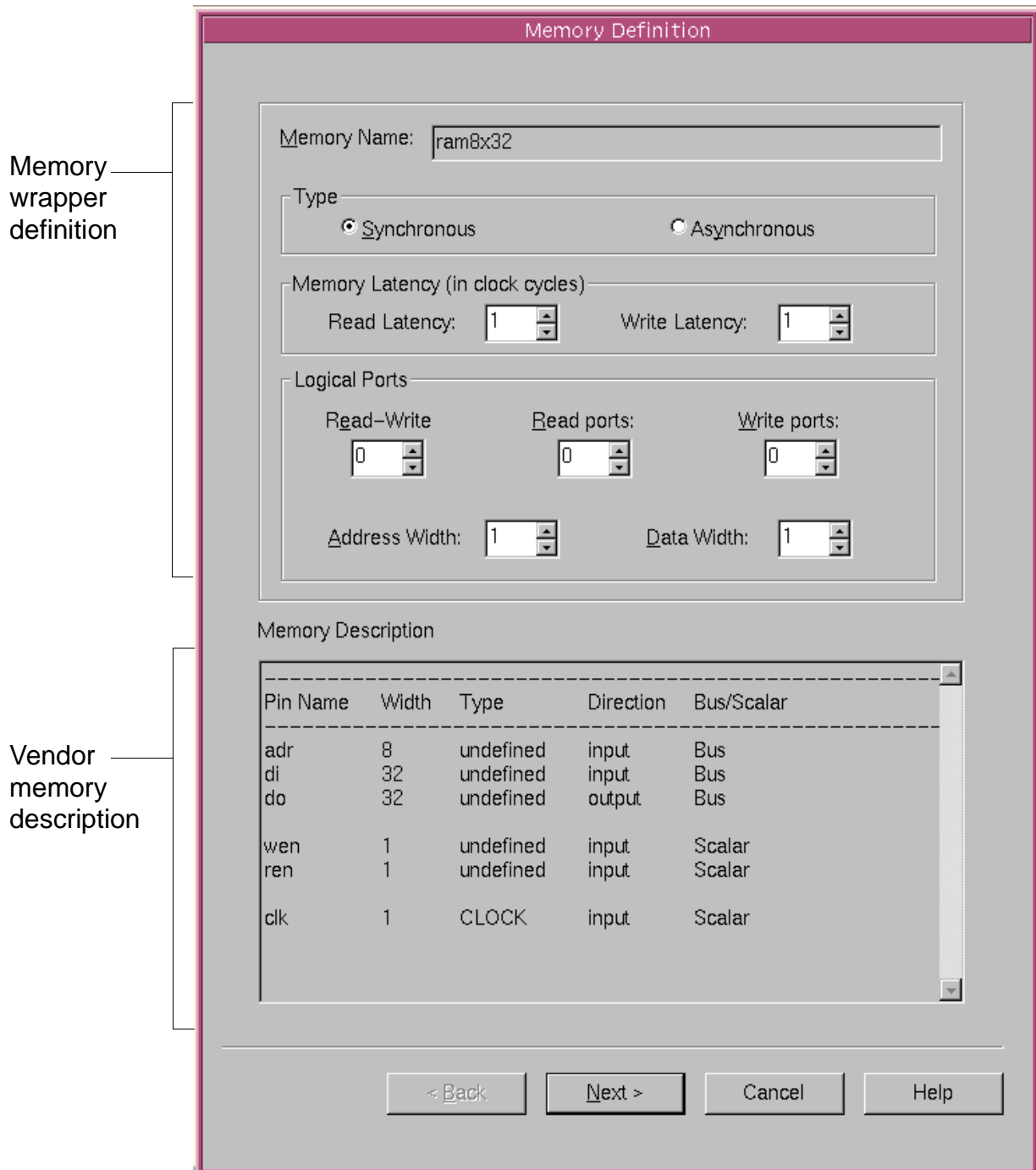
Figure 7-13 Memory Selection from a DB File Dialog Box



- a. Enter a .db file name in the Current DB File text field, or click on Browse to select a .db file.
- b. Check that the correct vendor library is selected in the Selected Library Names field.
- c. Enter or select a memory model name from the vendor library. For example, the *ram8x32* memory model is selected in Figure 7-13 from the *sync_rams* vendor library.
- d. Click OK.

The Memory Definition dialog box (Figure 7-14) is displayed for the selected vendor memory. To list the contents of a library and other related commands, see “Getting Memory and Library Information” on page 7-18.

Figure 7-14 Memory Definition Dialog Box



3. The memory model name you selected from the vendor library is displayed in the Memory Name field.
 - a. Select the Synchronous type if the vendor memory has a clock. Otherwise, select Asynchronous. (See “Using Asynchronous Memories” on page 7-24.)
 - b. Enter or choose the number of clock cycles for the memory read and write latency. Memory latency depends on the type of memory, and you can obtain it from the memory vendor datasheet. For synchronous memories, enter the number of clock cycles the vendor memory model requires to complete one read and one write operation.
 - c. In the Logical Ports section, enter the number of logical ports in the memory wrapper for each of following types:

Read-Write ports A read-write port is a logical port that you can use to perform either a memory read or a memory write, but not both at the same time. The physical ports associated with a read-write logical port are typically an address bus, a data in bus (for the memory write), a data out bus (for the memory read), and the control lines.

Read ports A read port is a logical port that you can use only to perform a memory read. The physical ports that it typically connects to are an address bus, a data out bus (for the data being read out of memory), and the control lines.

Write ports

A write port is a logical port that you can use only to perform a memory write. The physical ports that it typically connects to are an address bus, a data in bus (for the data being written to memory), and the control lines.

- d. In the Logical Ports section, enter the address bit-width and the data bit-width. The address and data bit-widths are common to all ports.

Result: Figure 7-15 shows the completed memory definition.

Figure 7-15 Completed Memory Definition

Memory Definition

Memory Name:

Type
 Synchronous Asynchronous

Memory Latency (in clock cycles)
Read Latency: Write Latency:

Logical Ports
Read-Write: Read ports: Write ports:
Address Width: Data Width:

Memory Description

Pin Name	Width	Type	Direction	Bus/Scalar
adr	8	undefined	input	Bus
di	32	undefined	input	Bus
do	32	undefined	output	Bus
wen	1	undefined	input	Scalar
ren	1	undefined	input	Scalar
clk	1	CLOCK	input	Scalar

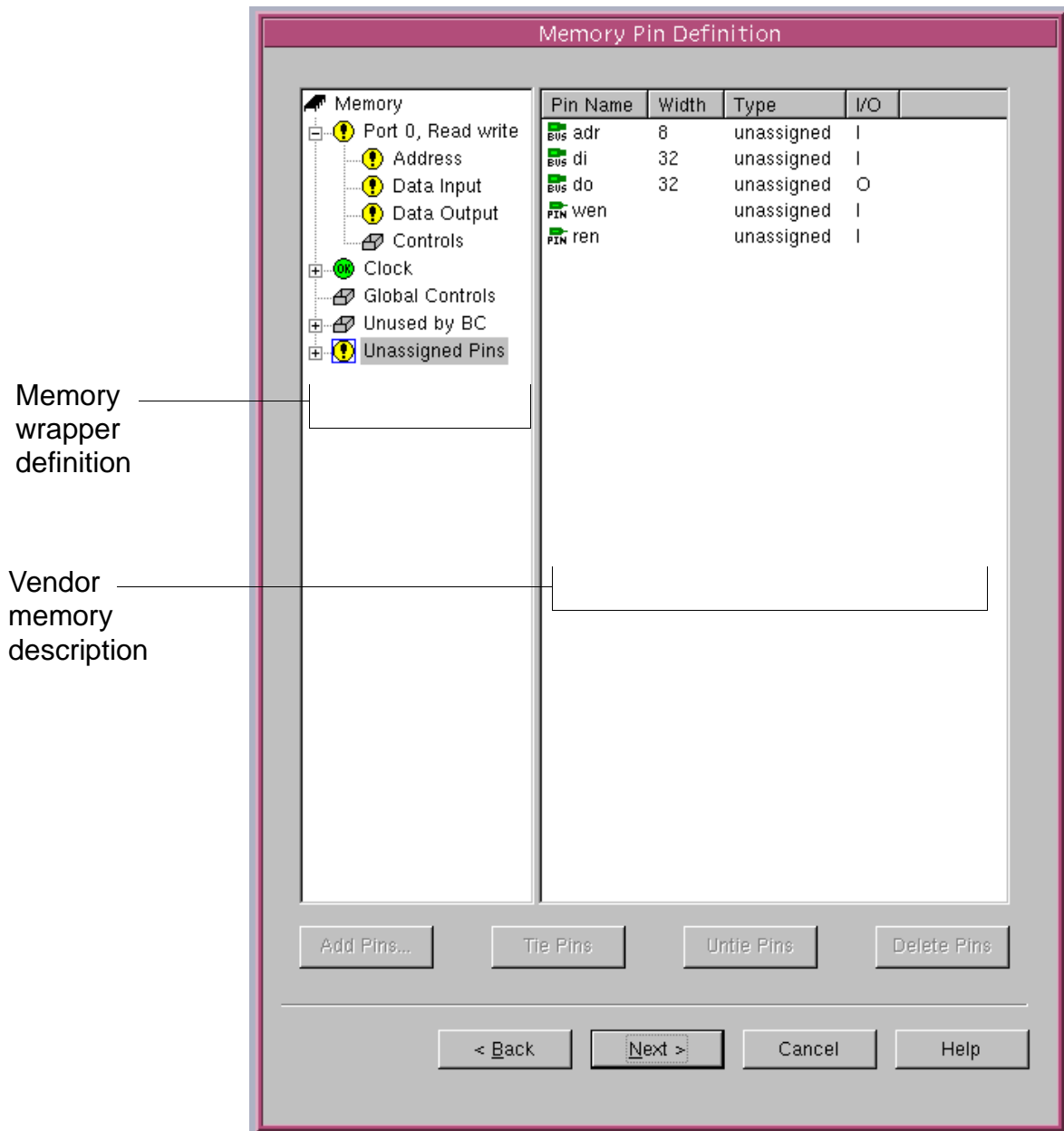
< Back Next > Cancel Help

4. Click the Next button to display the Memory Pin Definition dialog box (Figure 7-16).

Assigning Memory Pins to the Wrapper Logical Ports

The Memory Pin Definition dialog box (Figure 7-16) shows the assignment of the vendor memory pins to the logical ports in the memory wrapper. Use the Memory Pin Definition dialog box to assign or reassign vendor memory pins to the correct memory wrapper logical ports.

Figure 7-16 Memory Pin Definition Dialog Box



When you are creating a wrapper from a memory in a .db file, all the pins are initially unassigned (as in Figure 7-16).

To assign pins for a memory in a .db file,

1. Assign pins to the correct categories by dragging unassigned vendor memory pins (on the right side of the Memory Pin Definition dialog box) to the appropriate memory wrapper logical port categories (on the left side of the dialog box).
 - a. Select and drag all vendor address pins to the memory wrapper logical port address category.
 - b. Select and drag all vendor data input pins to the logical port data input category.
 - c. Select and drag all vendor data output pins to the logical port data output category.
 - d. Select and drag each vendor control pin such as write enable (wen) and read enable (ren) to the logical port controls category. When assigning control pin assigned to a logical port are signals that are used only for accesses to that specific physical port.
 - e. Select and drag any signals that are used for all memory accesses and are not associated with a single port to the Global Controls wrapper category. Global Controls are pins that are relevant for accesses to all the physical ports.
 - f. Select and drag any other signals such as test pins that are not used by SystemC Compiler to the Unused by BC category. You can expand the Unused by BC category hierarchy, and choose the appropriate category for the unused pins.

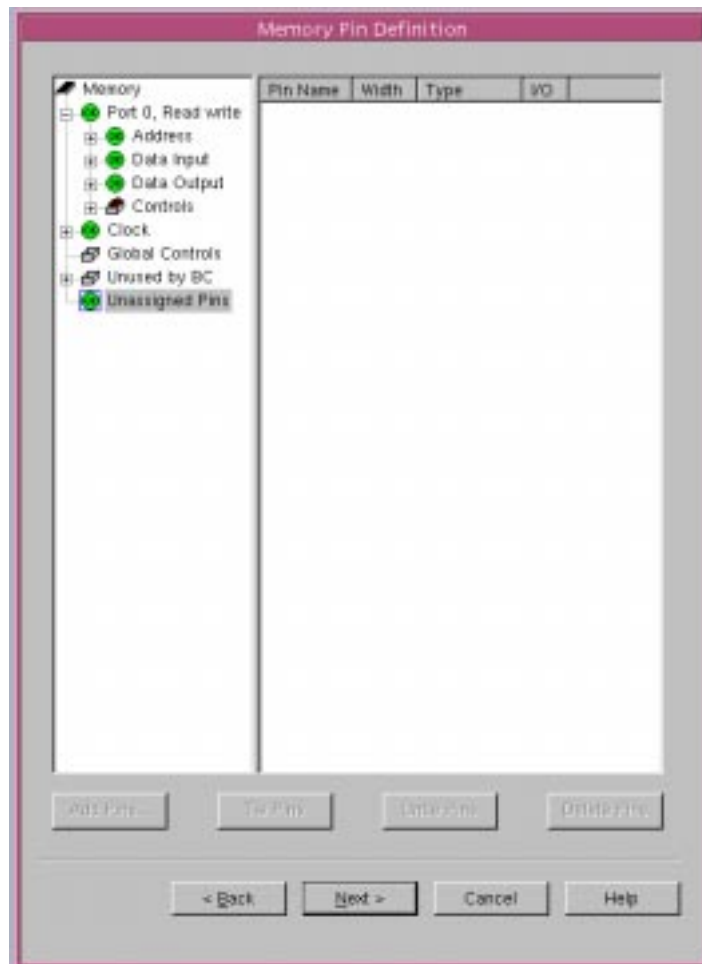
Memory Wrapper Generator extracts the clock name from the vendor memory .db file.

Shortcut:

While assigning or reassigning pins, you can select multiple pins using the Shift-click method (for consecutive selection) or the Control-click method (for nonconsecutive selection), or by drawing a rectangle around the pins with the mouse.

Result: When all vendor pins are assigned to a logical port wrapper category, a green circle with an OK is displayed next to the Port Address, Data Input, Data Output, Clock, and Unassigned Pins wrapper categories. After you assign all vendor pins, the vendor pins field is empty. Figure 7-17 shows a completed memory pin definition.

Figure 7-17 Completed Memory Pin Definition



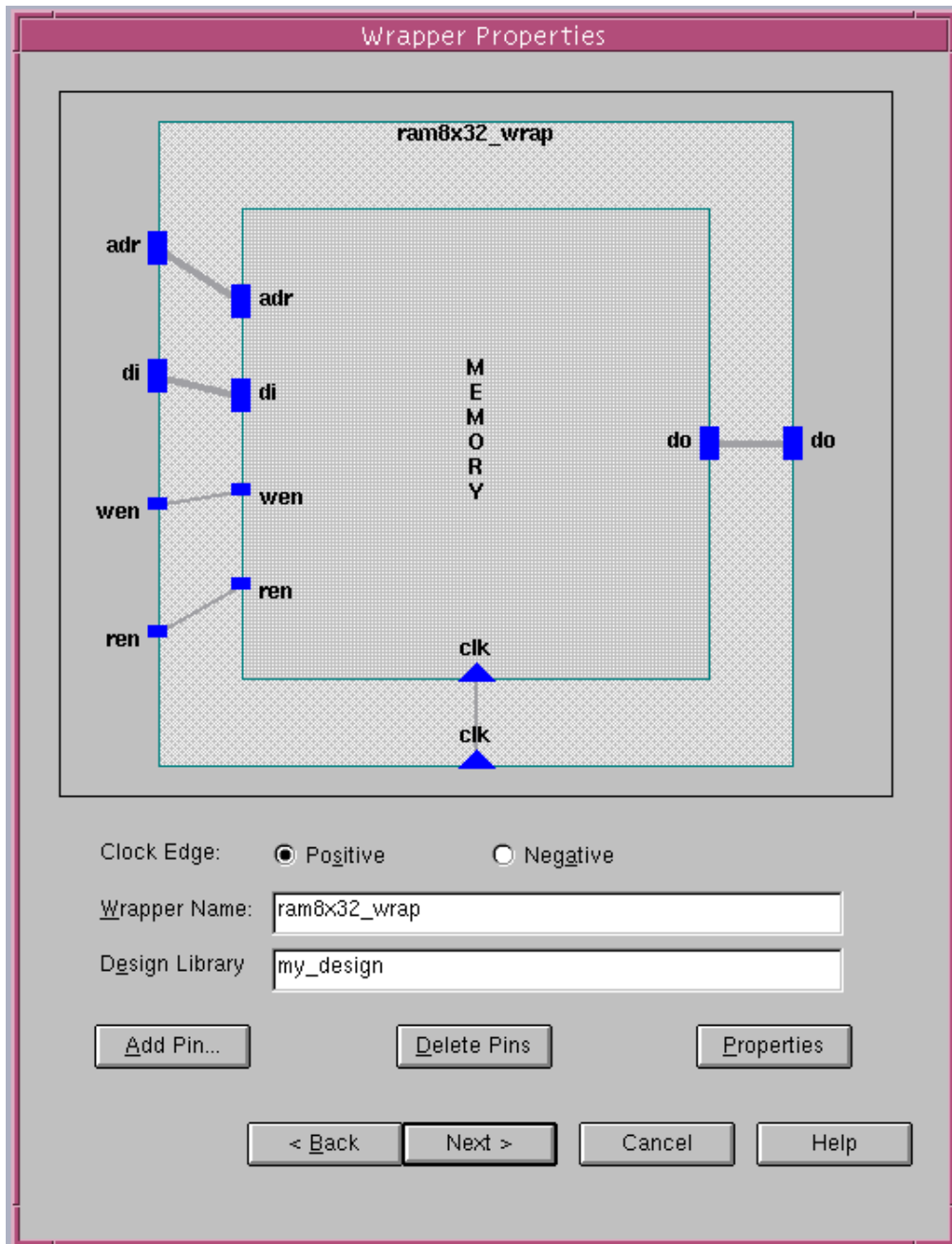
2. In some cases, the vendor memory's pins are not explicitly grouped into buses in the memory library file. In step 1, you assigned these individual pins to logical port categories, and now you need to specify the order (MSB to LSB) of these pins with respect to pins of the corresponding logical port.

For example, if the vendor memory has pins ADDR_X, ADDR_Y, ADDR_Z, and ADDR_W, and you placed these in the address category of logical port 0, you need to specify the order in which these pins map to the bits of the logical port's address bus.

To specify the ordering of the address and data pins (MSB to LSB),

- a. Select the appropriate category (Address, Data Input, or Data Output).
 - b. Change the order of the vendor pins by dragging the pins within the vendor pin list or by clicking the Pin Name column header. Clicking the Pin Name column header reverses the order of the pins from MSB to LSB, or from LSB to MSB.
3. Click on the Next button to display the Wrapper Properties dialog box (Figure 7-18).

Figure 7-18 Completed Wrapper Properties Dialog Box



Defining the Memory Wrapper Properties

The Wrapper Properties dialog box (Figure 7-18) shows the connections between the memory vendor cell and the wrapper. You can use the Wrapper Properties dialog box to further specify the wrapper interface and see how the wrapper pins are connected to the memory cell.

Initially, a wrapper pin or bus is created and connected to the memory pin or bus for each memory pin or bus that is not located in the Unused by BC category in the Memory Pin Definition dialog box.

You can also use the Wrapper Properties dialog box to add extra control pins to the wrapper. These extra control pins can be for test logic or for adding extra control or decode logic at a later time.

To define the wrapper properties,

1. Select positive or negative to define the clock edge used by the memory. This must be the same as the active clock edge that you use in your behavioral description.
2. In the Wrapper Name text field, enter the name of your wrapper module. By default, the Memory Wrapper Generator uses the Memory Name you defined as the memory name (Figure 7-15 on page 7-45) with an additional *_wrap* as the name.
3. In the Design Library field, enter the *design_library* name representing the design library where SystemC Compiler places the design for your wrapper when you analyze the wrapper VHDL (or Verilog) files. This is also the design library in which the synthetic library (.sldb) that contains your wrapper expects to find the design for your wrapper. Designate the WORK library unless you have a particular library where you want the files written. You can use any name except a reserved Synopsys library name such as "DW01."

The *design_library* name is the logical name of a library where you want the files written. You can map this logical library name to a physical UNIX directory with the `define_design_lib` command before you analyze the wrapper VHDL or Verilog files during synthesis. For example,

```
dc_shell> define_design_lib my_design_library
-path /export/design_libraries/my_design_library
```

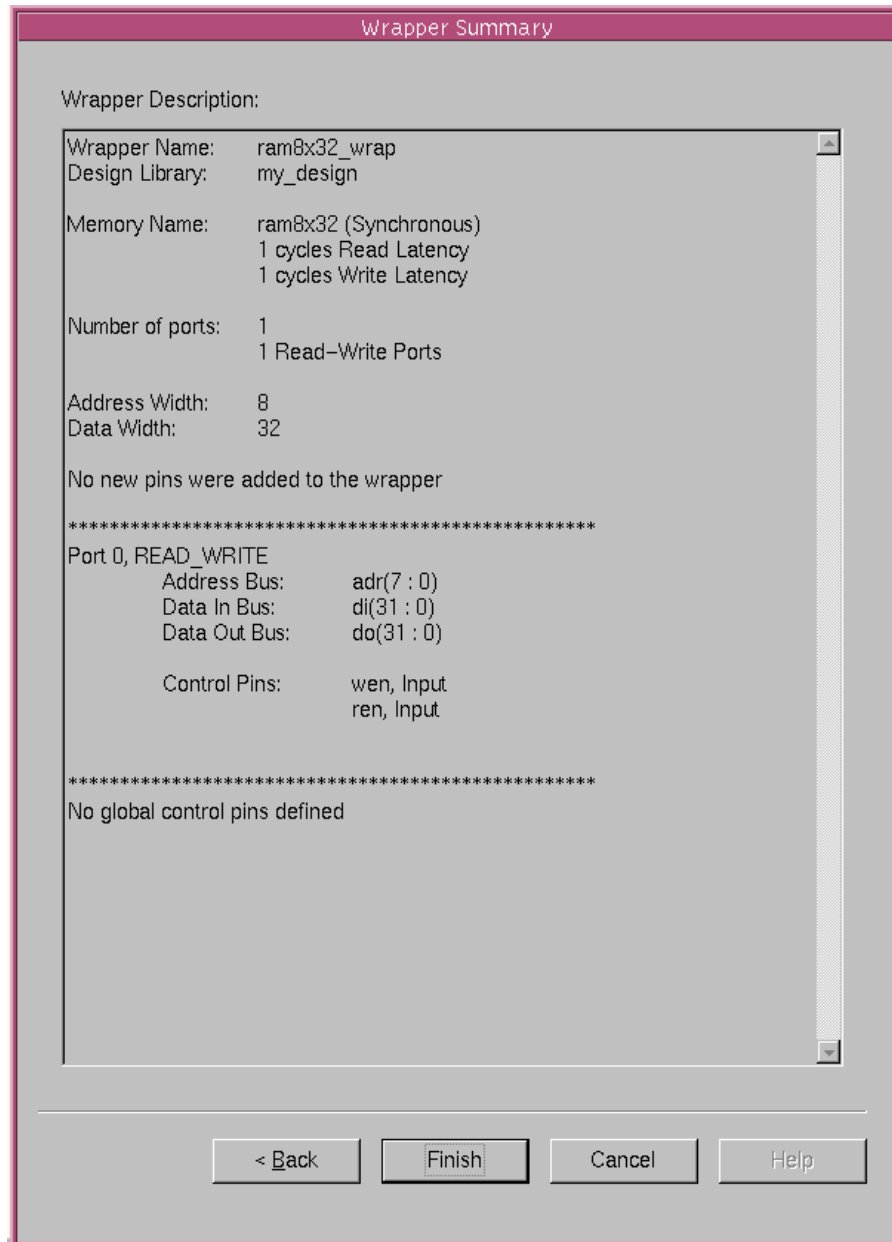
Result: Figure 7-18 shows a completed wrapper properties.

4. Add pins as necessary. (You can add only global control pins to the wrapper. You cannot add pins to the vendor memory. For details about global control pins, see “Assigning Memory Pins to the Wrapper Logical Ports” on page 7-46.)
 - a. Click the Add Pin button.
 - b. Fill out the displayed dialog box and click OK.
5. Delete pins as necessary (you can only delete wrapper pins).
 - a. Select the pins you want to delete.
 - b. Click the Delete Pins button.
6. Change the default control pin connections as necessary (you can change only the connections of control pins).
 - a. Click a wrapper pin.
 - b. Drag the pin to the memory pin with which you want to connect it.
A new connection removes any previous connections and connects the two pins.

When you connect a wrapper control pin to a memory control pin, the wrapper pin inherits the properties of the memory pin, for example the memory port that it is associated with and whether it is an input pin or an output pin.

7. Disconnect wrapper pins, as necessary, by clicking them and dragging them to an open area.
8. Click the Next button to display a Wrapper Summary dialog box (Figure 7-19).

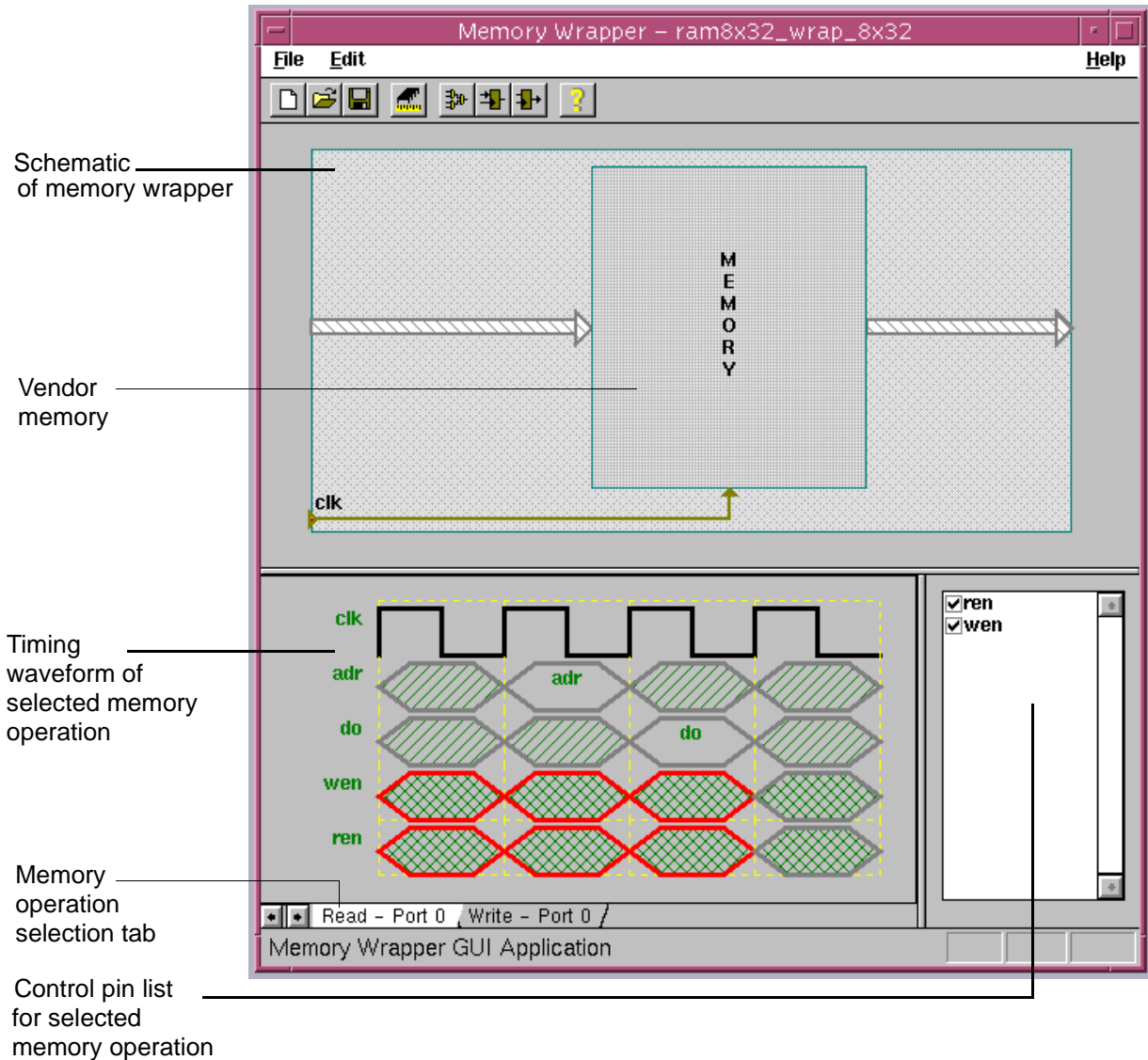
Figure 7-19 Wrapper Summary



9. If the displayed summary information is correct, click Finish.

When you are finished, the wrapper is displayed in the Memory Wrapper main window (Figure 7-20).

Figure 7-20 Memory Wrapper Displayed in Main Window



Reviewing the Memory Wrapper

The upper display area (the schematic in Figure 7-20) shows how the vendor memory is embedded in the wrapper.

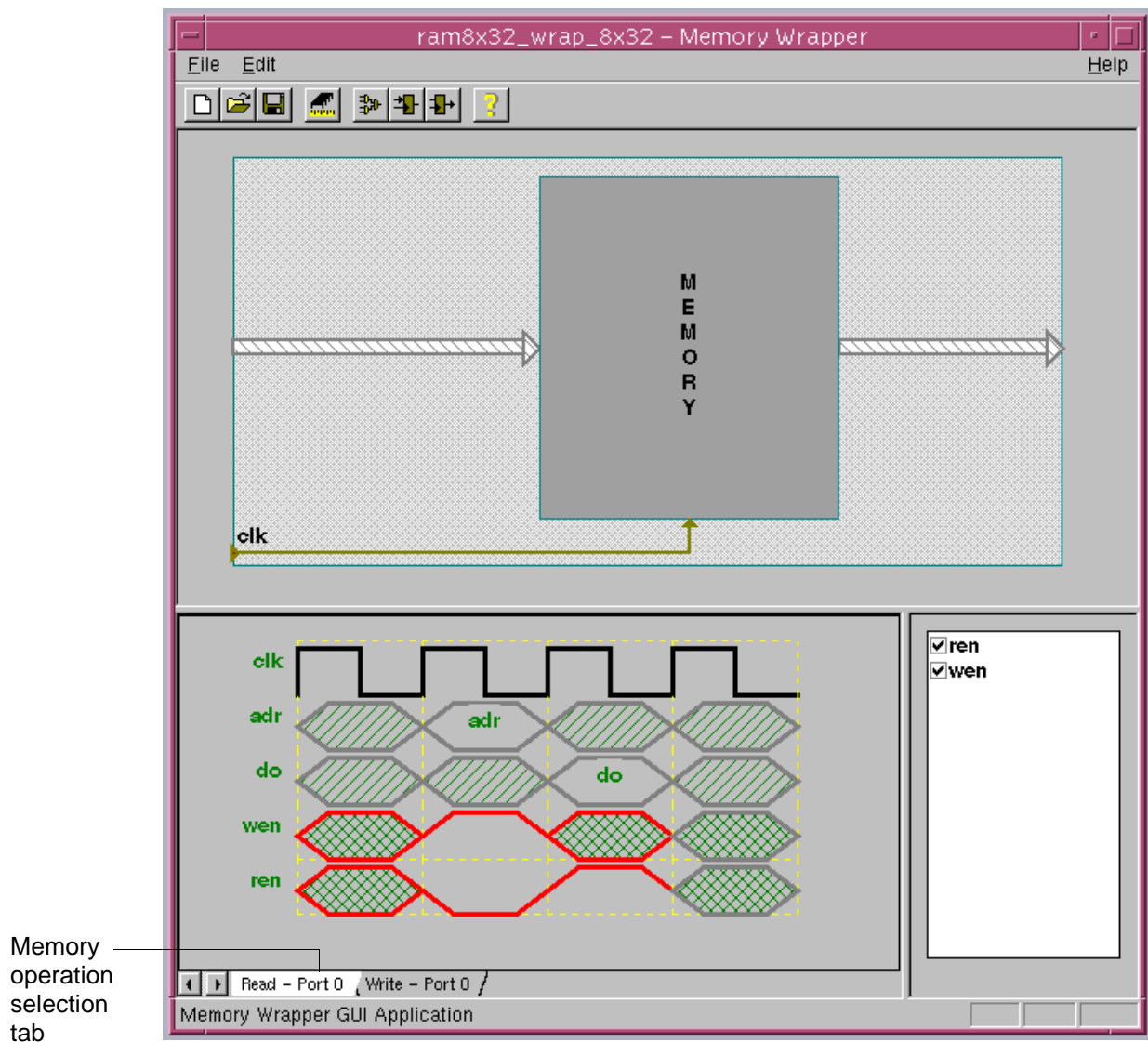
The lower-left display area shows the timing waveforms of the selected memory operations, for example a memory read from port 0. You can click on the Memory Operation Selection tabs to select another memory operation. All control pin names and waveforms refer to the wrapper, not the pins on the vendor memory. All information you enter in the waveform display area applies to the wrapper.

The lower-right display area (the port control pin list) displays the list of available control pins for the memory operation that you select and is shown in the waveform to the left. Initially all control pins are marked with a check mark (displayed in the waveform). To deselect a control pin, click the box next to it to remove the check mark.

Editing the Waveform Values

The waveform display shows the cycle-by-cycle protocols for a memory read or write operation on a logical port. For example, Figure 7-21 shows the timing protocol for a memory read from the logical port 0. The waveform shows that the *adr*, *do*, *wen*, and *ren* signals are involved in a memory read. The address of memory to be read is asserted on the *adr* bus in the second cycle. At the same time, *wen* is asserted and *ren* is deasserted. This completes the memory read request and the data is placed on the *do* bus in the next clock cycle.

Figure 7-21 Read Port Protocol Waveforms



The first and last cycles of the waveforms show the values of the various signals when there are no requests. These are called the inactive values and can be low, high, or don't care. Editable waveforms are drawn in red.

To edit the waveform state,

1. Double-click on the pin's waveform to change its state from don't care to high or low.

Each double-click changes the state from don't care to high to low in a circular fashion.

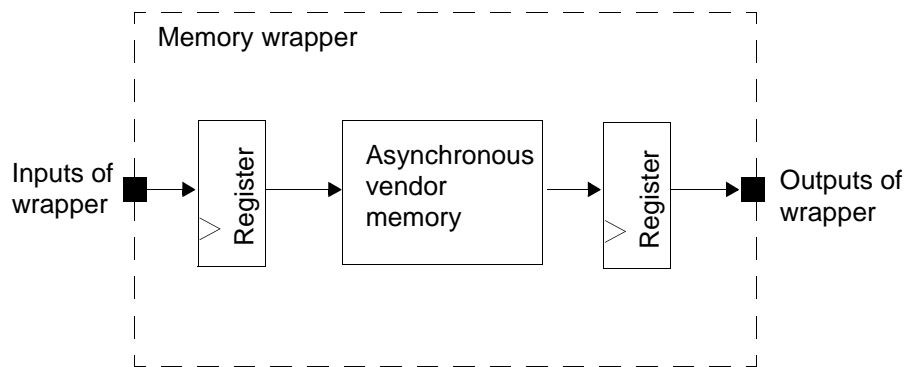
2. Click on the Read or Write Port tab to select another operation on the same port or an operation on a different port.

Adding Registers to the Memory Wrapper

You can add registers within the wrapper to the input or output of the memory. Adding these registers is required for an asynchronous memory to convert it to a synchronous memory. (See "Using Asynchronous Memories" on page 7-24.)

Adding registers increases the latency of memory accesses. For example, in the asynchronous memory shown in Figure 7-20 with registered inputs and outputs, the latency of a memory read is 2 clock cycles: one to register the inputs and the second to register the output data from the memory read. The output data is available at the data output port of the memory wrapper in the third clock cycle.

Figure 7-22 Manually Adding Registers to an Asynchronous Memory



To add or delete input registers or output registers between the wrapper and the memory, do one of the following:

1. Choose Edit > Register Type
2. Click the Input Register button or the Output Register button on the toolbar



Input Register



Output Register

The added registers are functional registers. Do not use them as testability registers.

Adding Custom Logic to the Memory Wrapper

To insert custom logic between the wrapper and the memory,

1. Do one of the following:
 - Choose Edit > Insert Logic

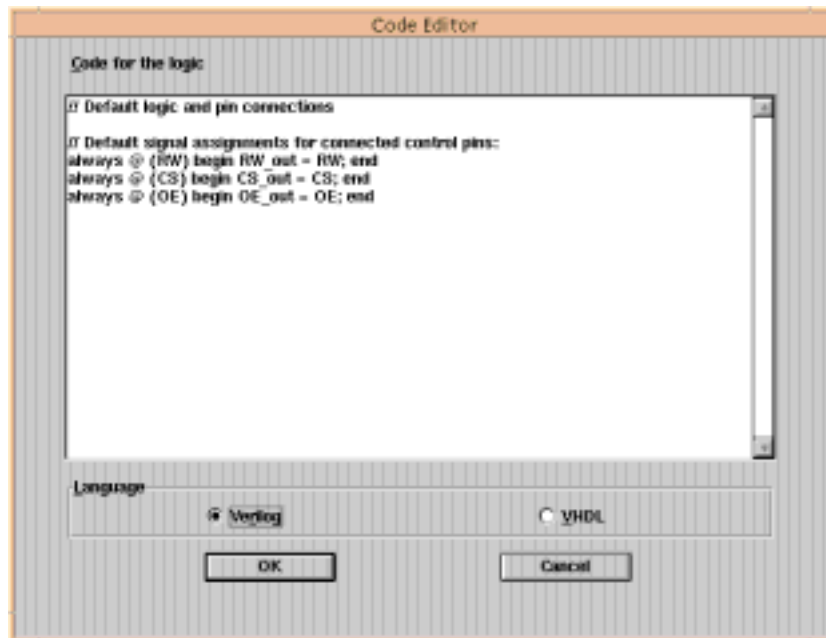
- Click the Insert Input Logic button on the toolbar.



The Code Editor dialog box is displayed with default code (Figure 7-23).

2. In the Code Editor dialog box, select the Verilog or VHDL language. The Memory Wrapper Generator displays the default code. The default code lists the connection between each wrapper port and the corresponding memory port. To insert logic, write it as you would write RTL. Note, however, the code must be Verilog or VHDL. Figure 7-23 shows an example of default Verilog code.

Figure 7-23 Code Editor Dialog Box With Default Code



3. Enter Verilog or VHDL code for the custom logic.

Use this Code Editor dialog box to edit the default code and combinational logic.

4. Click OK to complete the logic insertion. A box labeled as Logic is inserted in the Schematic window to indicate the custom logic.

To remove custom logic between the wrapper and the memory,

- Choose Edit > Insert Logic. The Insert Logic command becomes unchecked (it toggles), and the logic is removed.

To change custom logic between the wrapper and the memory,

1. Double-click on the Logic box in the Schematic window.

The Code Editor dialog box is displayed.

2. Edit the contents and click OK.

If you are adding registers or custom logic into the wrapper, you must ensure that the latency of the read/write operations and the waveform specified match the behavior of the memory with the wrapper. If they do not match, you must go back and edit the latency or the waveforms.

Viewing and Editing the Wrapper Properties

If you want to review or edit any of the memory wrapper dialog boxes, choose the Edit > Edit Properties command or click on the Properties button in the tool bar.



Properties

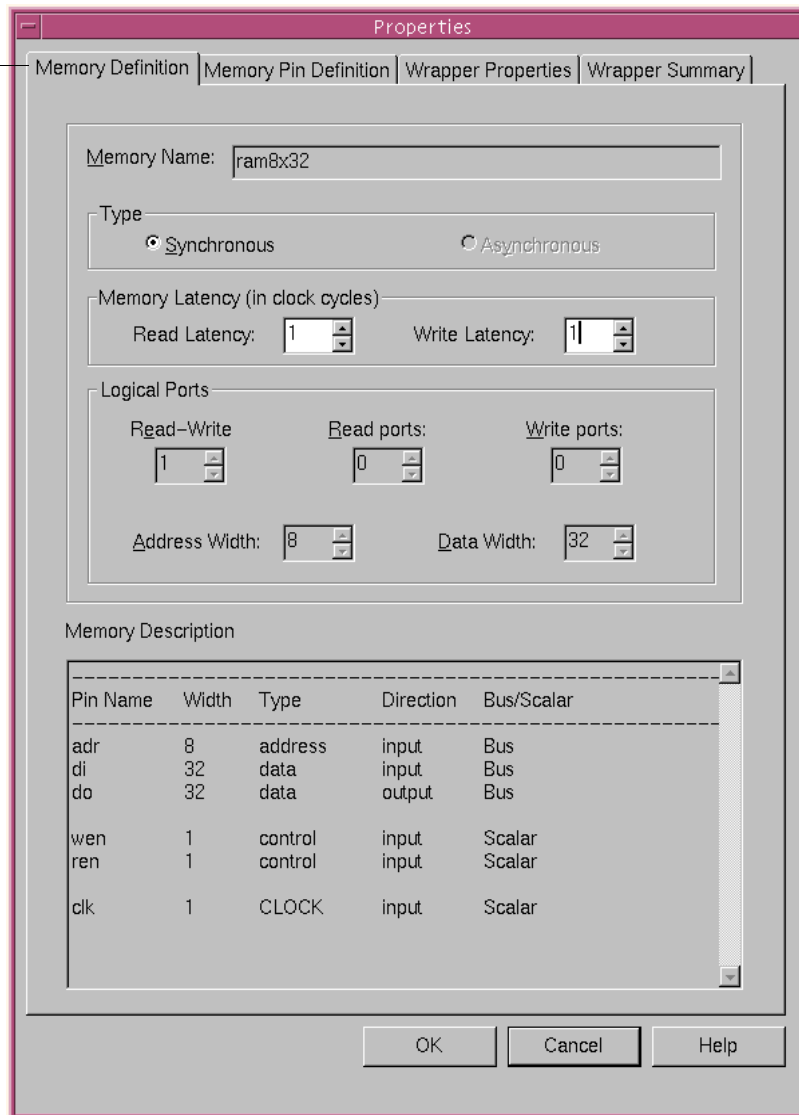
Figure 7-30 shows the Properties window with the Memory Definition dialog box displayed. You can select the Memory Definition, Memory Pin Definition, Wrapper Properties, and Wrapper Summary dialog boxes by clicking on the tabs at top.

The properties are described in

- “Defining the Memory Type and Properties” on page 7-40
- “Assigning Memory Pins to the Wrapper Logical Ports” on page 7-46
- “Defining the Memory Wrapper Properties” on page 7-52
- “Reviewing the Memory Wrapper” on page 7-56

Figure 7-24 Properties Dialog Boxes

Selection tabs



Saving the Memory Wrapper Files

After specifying the waveforms, save the wrapper (.wrap) file, the associated synthetic library (.sldb) file, and the Verilog (.v) file. For details about file types, see “Preparing to Use Memories” on page 7-15.

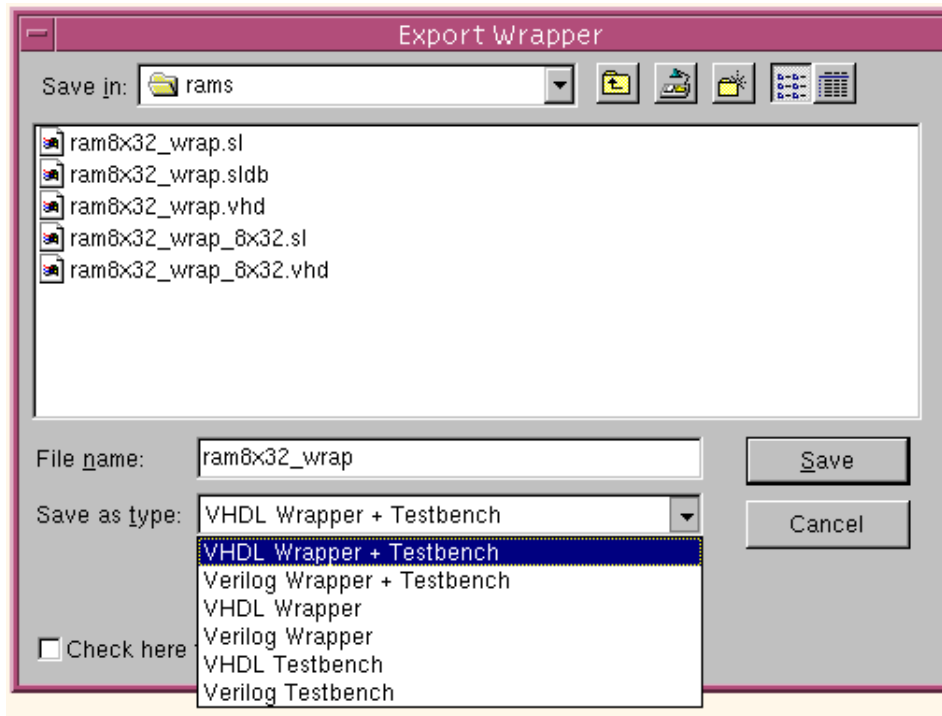
You can save just the .wrap file and read it into the Memory Wrapper Generator tool at a later time to create the .sldb, .sl, or .v files.

By default, the Memory Wrapper Generator tool uses the memory name with `_wrap` for the file names, for example, `memory_wrap.sldb` and `memory_wrap.v`

To save the wrapper and associated files,

1. To save just the `memory_wrap.wrap` file, choose File > Save. You can also save the .wrap file using the File > Save As command.
2. To save the `memory_wrap.sl`, `memory_wrap.sldb`, and `memory_wrap.v` files, choose File > Export Wrapper. Figure 7-25 shows the Export Wrapper dialog box.

Figure 7-25 Export Wrapper Dialog Box



3. In the Export Wrapper dialog box, select the Save as type to be Verilog Wrapper or VHDL Wrapper.

You can also use this command to save a testbench for the memory wrapper, which is described in “Generating a Memory Wrapper Testbench” on page 7-79.

Note:

This command saves a synthetic library .sl file. This file contains the DesignWare Developer source code that was used to generate the .sldb file. It is included for reference only. It is not used by SystemC Compiler, and you can delete it.

Using Generated Vendor Memory Wrappers With SystemC Compiler

To run SystemC Compiler using a vendor memory,

1. In the source code, use the `resource` compiler directive to map an array to the memory wrapper. Example 7-10 shows declaration of a resource named `RAM_A`, accessed by the array variable `mema`, and its wrapper module is `ram8x32_wrap`. For details about declaring and accessing memories, see the *CoCentric™ SystemC Compiler Behavioral Modeling Guide*.

Example 7-10 Memory Array Definition

```
//SystemC code fragment
// Declare the memory access array
sc_int<32> mema[256];
/* synopsys resource RAM_A :
variables = "mema",
map_to_module = "ram8x32_wrap"; */
```

2. Map the design library name you designated when creating the memory wrapper files to a physical UNIX directory. For example,

```
dc_shell> define_design_lib my_design_library
-path /export/design_libraries/my_design_library
```

3. Run SystemC Compiler `analyze` command to analyze and elaborate the memory wrapper file into the design library you specified in step 2.

```
dc_shell> analyze -f verilog ram8x32_wrap.v
-library my_design_library
```

4. Add the memory `.sldb` file generated by Memory Wrapper Generator to the `synthetic_library` variable. For example,

```
dc_shell> synthetic_library = synthetic_library +  
ram8x32_wrap.sldb
```

5. Add the memory library with the .db files to the `link_library` variable.

```
dc_shell> link_library = link_library + ram8x32.db
```

6. You are now ready to run `compile_systemc` on your design.

Creating a Memory Wrapper for an Exploratory Memory

The Memory Wrapper Generator creates a unique file format called a .wrap file. This file encapsulates all of the information necessary to generate a synthetic library (.sldb) for an exploratory memory. You can use this .sldb file for architectural exploration of memories with single or dual ports, a read/write versus a separate read and write port, and various address and data bit widths.

The general steps for creating a memory wrapper are

1. Define the type and properties of the memory.
2. Define properties of the wrapper and add control pins to the wrapper, if necessary.
3. Edit the wrapper by adding or deleting logic or registers and specifying the waveforms.
4. Save the wrapper.

Defining the Memory Type and Properties

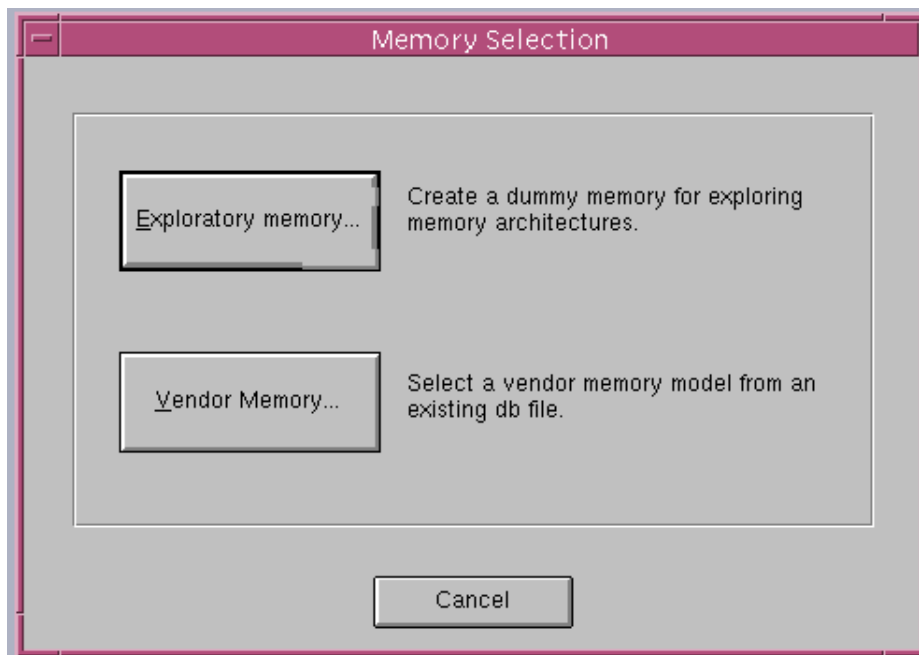
To define the type and properties of the exploratory memory wrapper,

1. Choose File > New.

(To open a previously saved .wrap file, choose File > Open.)

The Memory Selection dialog box is displayed (Figure 7-12).

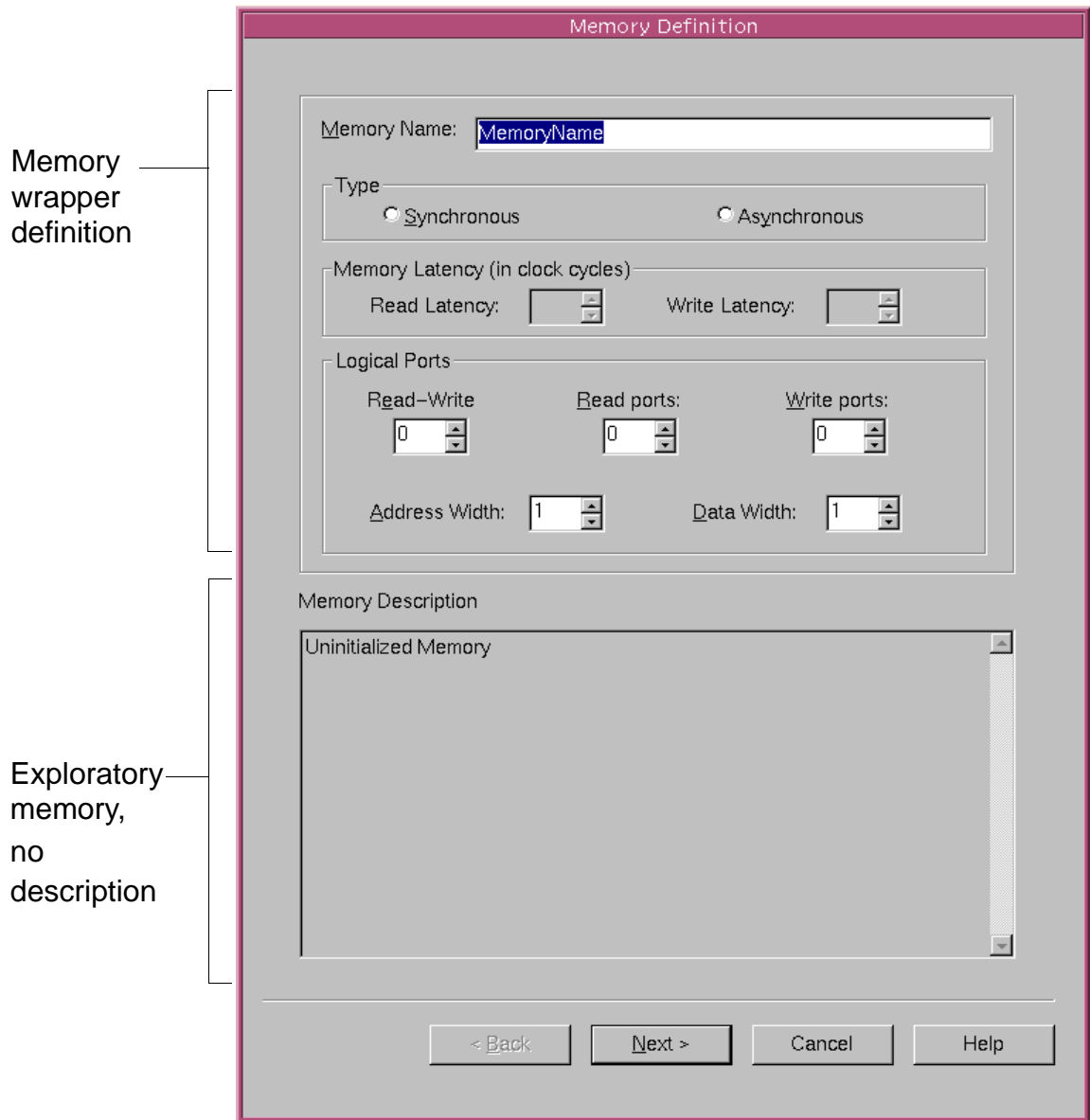
Figure 7-26 Exploratory Memory Selection Dialog Box



2. Click the Exploratory Memory button.

The Memory Definition dialog box (Figure 7-27) is displayed for an exploratory memory. Because a physical memory does not actually exist, the Memory Description field displays “Uninitialized Memory” to indicate no information is available.

Figure 7-27 Exploratory Memory Definition Dialog Box



3. Enter the properties for the wrapper interface that SystemC Compiler uses.
 - a. Enter a name in the Memory Name text field.

- b. Select the Synchronous type if the wrapper has a clock. Otherwise, select Asynchronous.
- c. Enter the number of clock cycles the wrapper requires to complete one read operation and one write operation.
- d. In the Logical Ports section, enter the number of logical ports (for example, for a single-port or dual-port memory). The following briefly describes the port information:

Read-Write ports A read-write port is a logical port that you can use to perform either a memory read or a memory write, but not both at the same time. The physical ports associated with a read-write logical port are typically an address bus, a data in bus (for the memory write), a data out bus (for the memory read), and the control lines.

Read ports A read port is a logical port that you can use only to perform a memory read. The physical ports that it typically connects to are an address bus, a data out bus (for the data being read out of memory), and the control lines.

Write ports A write port is a logical port that you can use only to perform a memory write. The physical ports that it typically connects to are an address bus, a data in bus (for the data being written to memory), and the control lines.

- e. In the Logical Ports section, enter the address bit-width and the data bit-width. The address and data bit-widths are common to all ports.

Result: Figure 7-28 shows the completed Memory Definition example.

Figure 7-28 Completed Exploratory Memory Definition

Memory Definition

Memory Name:

Type

Synchronous Asynchronous

Memory Latency (in clock cycles)

Read Latency: Write Latency:

Logical Ports

Read-Write: Read ports: Write ports:

Address Width: Data Width:

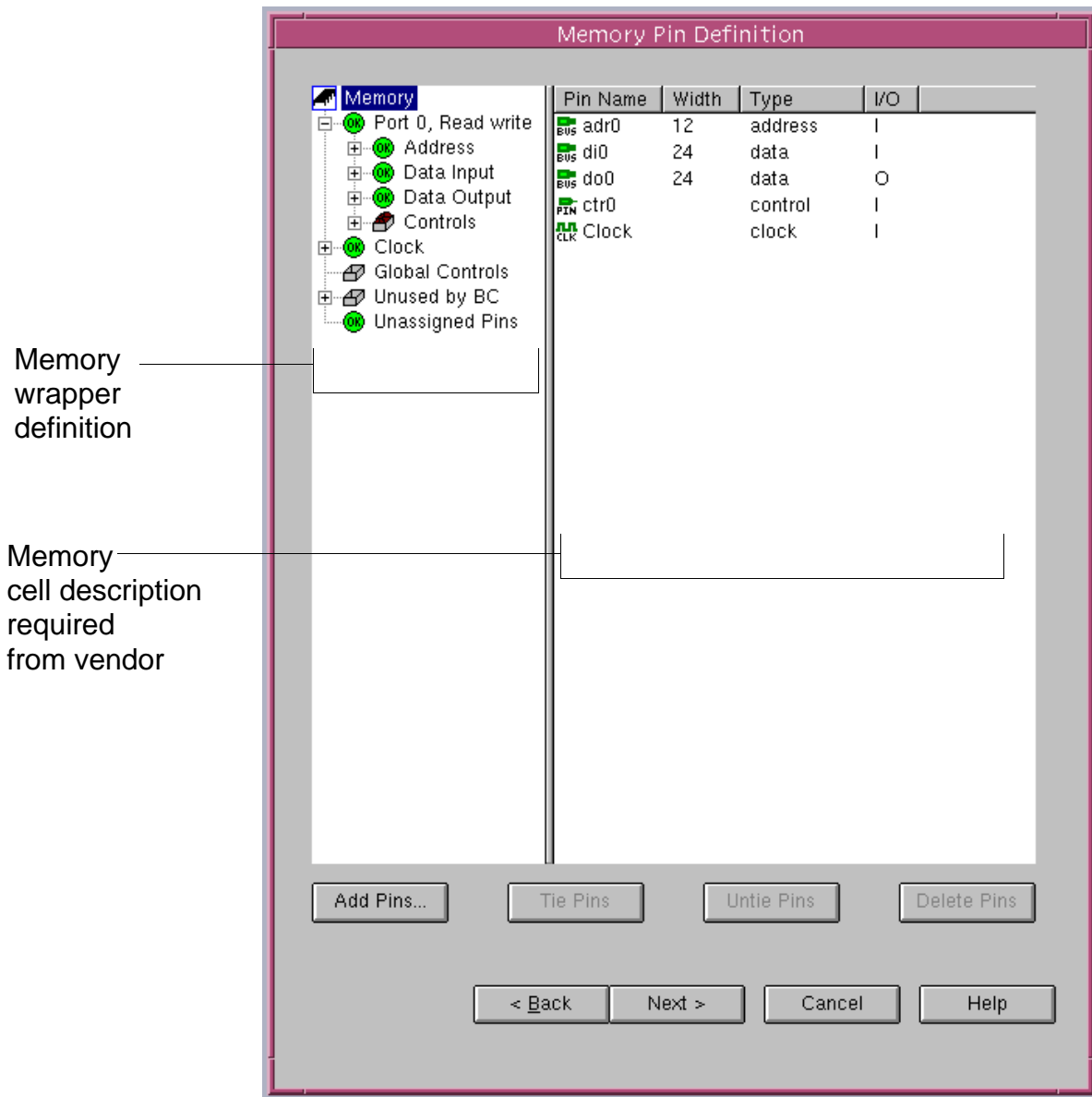
Memory Description

Uninitialized Memory

< Back Next > Cancel Help

4. Click the Next button to display the Memory Pin Definition dialog box (Figure 7-16).

Figure 7-29 Exploratory Memory Pin Definition Dialog Box



Based on the input and output ports you defined, the Memory Wrapper Generator tool generates a pin definition list required from the memory vendor for a matching memory cell.

Assigning Pins to the Memory Logical Ports

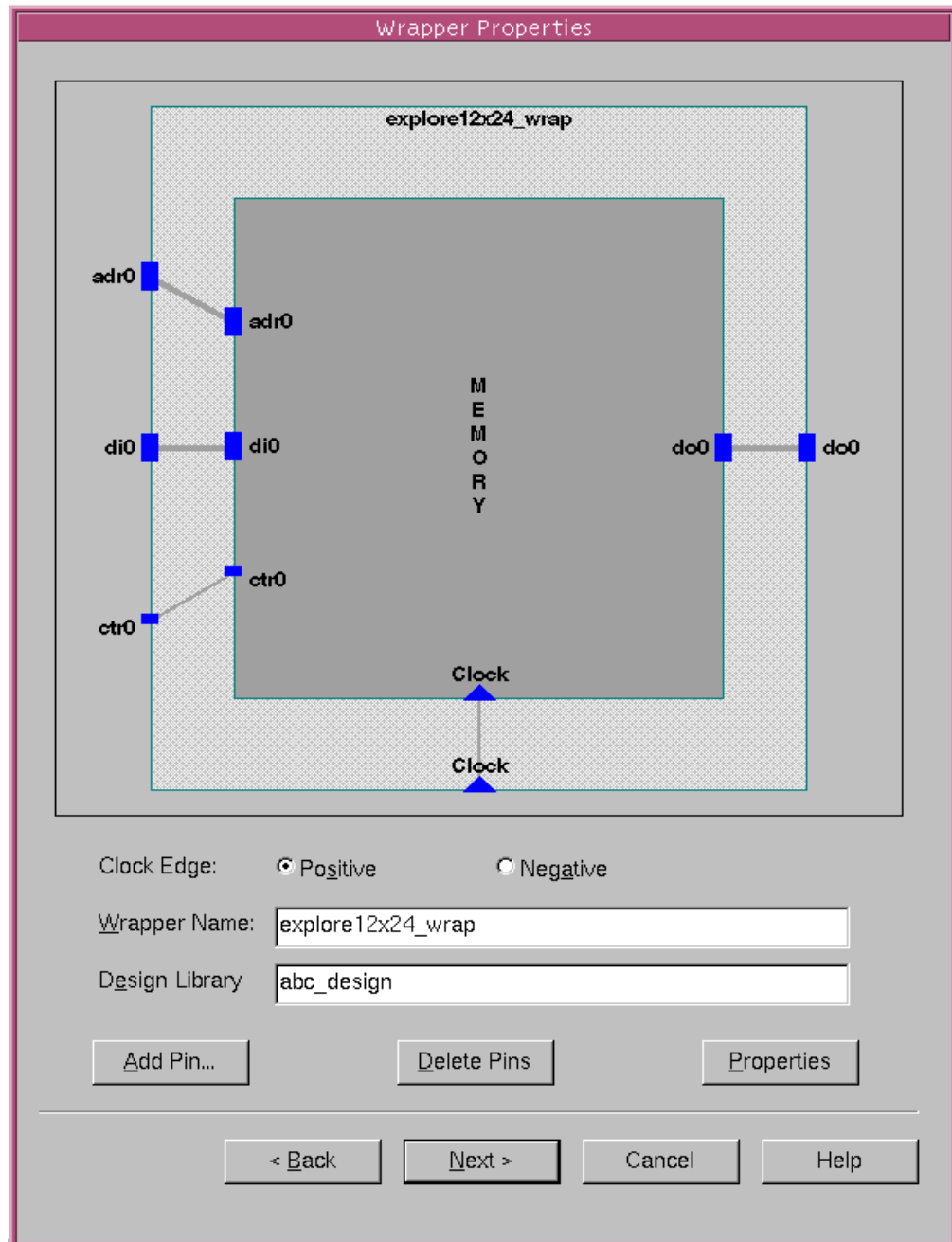
Default pins and buses are initially created and assigned automatically. The Memory Pin Definition dialog box shows the ports and control pins used in the memory wrapper, and it shows the pins that would be required in a vendor memory to match the wrapper interface.

A green circle with an OK is displayed next to the Port Address, Data Input, Data Output, Clock, and Unassigned Pins categories in the wrapper define. This means there is no pin assignment action required.

Click the Next button to display the Wrapper Properties dialog box (Figure 7-18).

If you need to add additional pins, click on the Add Pins button.

Figure 7-30 Exploratory Wrapper Properties Dialog Box



Defining the Exploratory Memory Wrapper Properties

The Wrapper Properties dialog box shows the connections between the wrapper and a memory of this type.

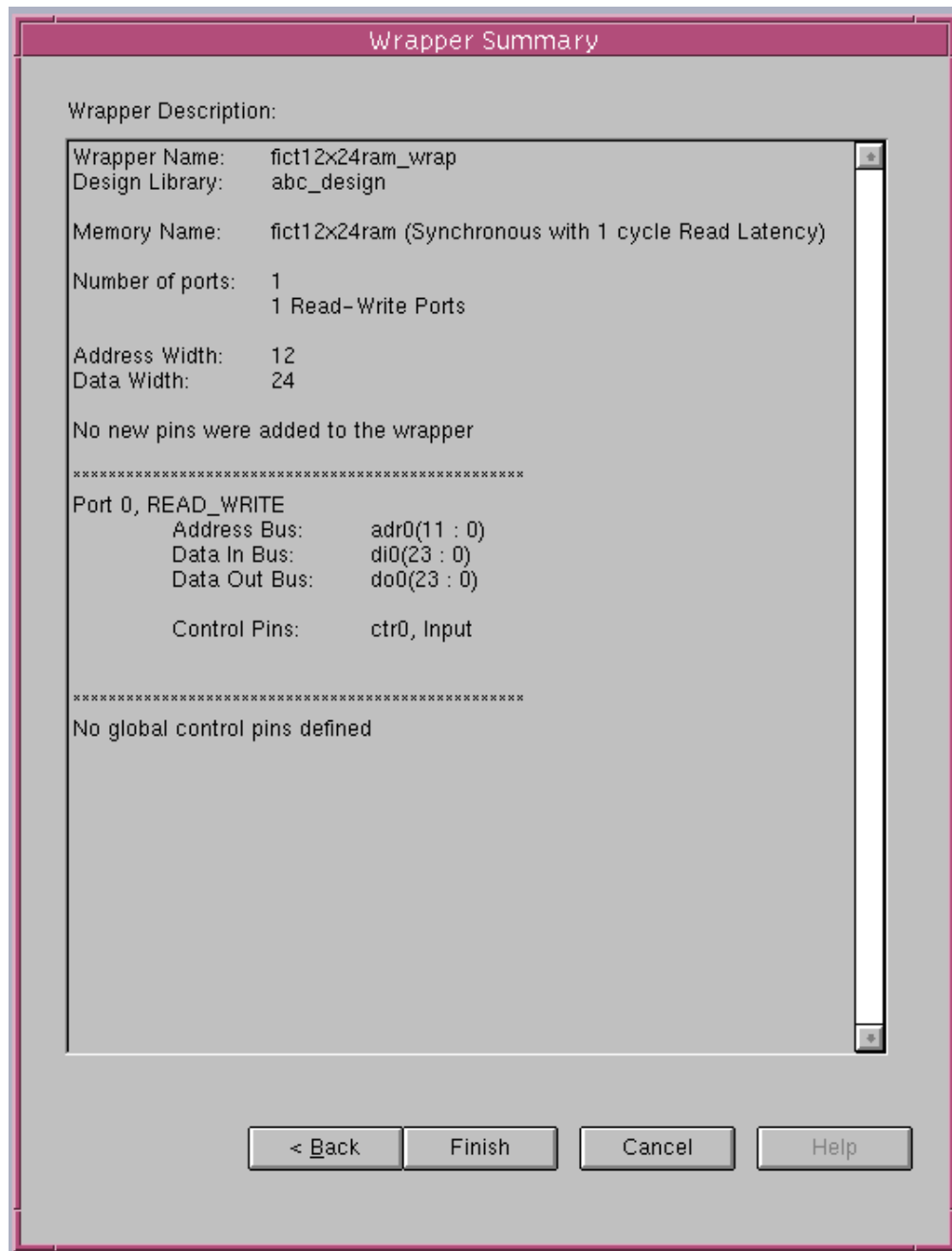
To define the wrapper properties,

1. Select positive or negative to define the clock edge used by the exploratory memory, which should be the same as your design.
2. In the Wrapper Name text field, the name of the wrapper is displayed.
3. In the Design Library field, enter a name representing the design library in which the wrapper design will reside.

The name is a design library that contains the analyzed structural wrapper. Enter the WORK design library, if you do not have a designated library name. You can use any name except a reserved Synopsys library name such as “DW01.”

4. Add pins as necessary (you can add only global control wrapper pins, not memory pins). This step is not usually needed for exploratory memories.
5. Click the Next button to display a Wrapper Summary dialog box (Figure 7-31).

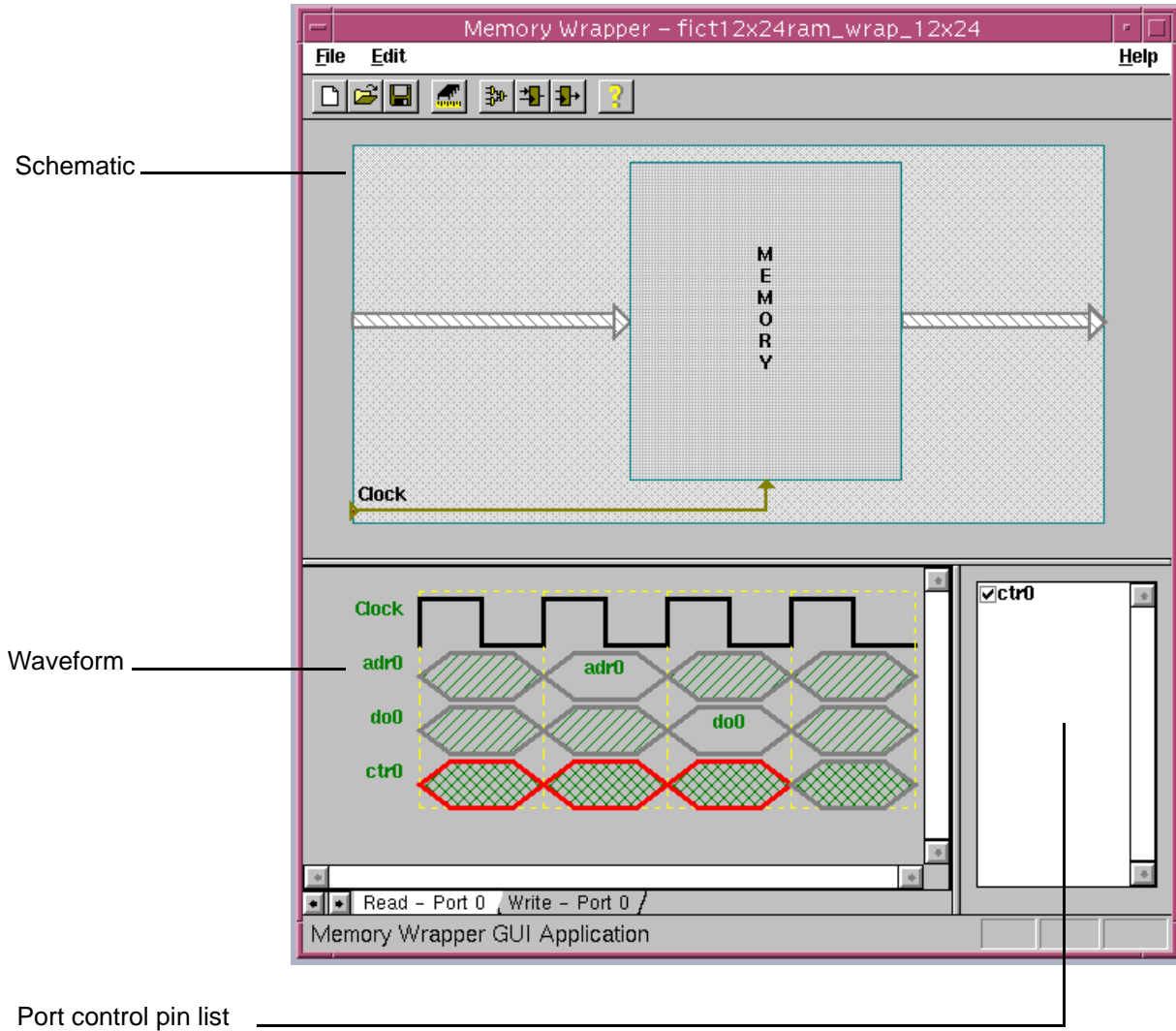
Figure 7-31 Exploratory Memory Wrapper Summary



If the displayed summary information is correct, click Finish.

When you are finished, the wrapper is displayed in the Memory Wrapper main window (Figure 7-20).

Figure 7-32 Exploratory Memory Wrapper in Main Window



Reviewing and Editing the Exploratory Memory Wrapper

You can review and edit the memory wrapper for an exploratory memory using the commands in “Viewing and Editing the Wrapper Properties” on page 7-61.

Saving the Exploratory Memory Wrapper Files

After specifying the waveforms, save the .wrap and .sldb files.

To save the wrapper,

1. Choose File > Save, which saves the .wrap file.
2. Choose File > Export Wrapper, and select the Save as type to be Verilog Wrapper or VHDL Wrapper, which saves the .sldb file and the Verilog or VHDL wrapper description file. “Saving the Memory Wrapper Files” on page 7-64 shows this dialog box and provides additional details about exporting wrapper files.

After the exploratory memory is generated, you can use it the same way as a vendor memory, described in “Using Generated Vendor Memory Wrappers With SystemC Compiler” on page 7-66.

Generating a Memory Wrapper Testbench

After you generate the memory wrapper, it is important to verify that it meets the memory vendor's specifications. Checking the wrapper after you generate it can save verification and redesign time later in the design cycle.

The Memory Wrapper Generator tool can automatically create a self-checking simulation testbench and test case to verify that the memory wrapper is correctly specified and generated. The testbench performs a sequence of writes to and reads from the memory. Generating the memory wrapper testbench requires that you have a Verilog or VHDL memory simulation model, which is typically supplied by the memory vendor.

To verify your memory wrapper, use the automatically created test design and testbench to run behavioral simulation of your memory wrapper.

The memory wrapper testbench generated files use your specified memory wrapper name with an additional `_D` as the file names, for example `memory_wrap_D.v`.

To test your wrapper with a memory wrapper testbench,

1. Choose File > Export Wrapper (Figure 7-25 on page 7-65), select the Save as type to be Verilog Testbench or VHDL Testbench, which saves the following files:

- A sample design containing memory operations, `memory_wrap_D.v` or `memory_wrap_D.vhd`

This file contains a behavioral design that reads data from its input port, writes the data into the memory, reads it back from the memory, and writes it to its output port.

- A self-checking simulation testbench, *memory_wrap_D_tb.v* or *memory_wrap_D_tb.vhd*

This file writes a number of values to the sample design and reads values back from the sample design. It compares each value read from the sample design to the expected value.

To create a very simple test case that writes and reads only one value, enable the “Check here to create simple test only” option in the Export Wrapper dialog box, and save the testbench files again.

- A dc_shell script file, *memory_wrap_D_v.scr* or *memory_wrap_D_vhd.scr*

This script contains the dc_shell commands to synthesize the sample design. You can customize the script.

2. Run the *memory_wrap_D_v.scr* script in dc_shell.

This synthesizes the sample design and generates a structural RTL design.

3. Simulate the RTL design with the generated wrapper file and the memory simulation model. For example, to simulate a Verilog RTL design with VCS, read in the *_D_tb.v, *_D.v, and the *_D.scr files.

This simulation reveals any problems with the memory wrapper specification. If problems are detected, you need to correct the memory wrapper definition and generate a revised memory wrapper to produce a valid memory testbench simulation.

8

Advanced Techniques

This chapter explains advanced features and techniques you can use to further improve scheduling and the quality of results.

This chapter contains the following sections:

- Using Multiple Files to Describe a Design
- Speculative Execution
- Setting a Specific Implementation for Components
- Externalize a Cell

Using Multiple Files to Describe a Design

If your design has multiple modules that are defined in separate files, you can use either the `#include` directive or preserved functions to bring the external files into the primary design.

Using `#include`

To bring separate files in a primary design file, you can use the `#include` directive. This is useful if you want to include the same files in several designs. For example, you may have IP that is used in many designs. However, this is not a recommended C programming style, because using the `#include` directive increases the size of your program.

Using Precompiled Netlists

External precompiled netlists in the form of `.db` files can be brought into your primary design with the preserved function capability.

To bring the precompiled netlist into the primary design,

1. Add the `preserve_function` compiler directive to a function in your behavioral description that represents the external precompiled netlist.
2. Elaborate the design with the `compile_systemc` command.
Enter

```
dc_shell> compile_systemc primary_design.cc
```

3. Read in the precompiled netlists with the `read_preserved_function` command. This command maps the precompiled netlist to the preserved function.

Enter

```
dc_shell> read_preserved_function_netlist
           module1_elab.db
dc_shell> read_preserved_function_netlist
           module2_elab.db
```

4. Use the `link` command to link the design. Enter

```
dc_shell> link
```

If your precompiled netlist is in a directory that is not defined as a design library, use the `define_design_lib` command to map the directory to a design library before using the `read_preserved_function_netlist` command.

For example, to map `module1.db` and `module2.db` files in the `library1` directory to the `library_name1` design library, enter

```
dc_shell> define_design_lib library_name1
           -path /remote/design_libraries/library1
dc_shell> compile_systemc primary_design.cc
dc_shell> read_preserved_function_netlist module1_elab.db
           -design_library library_name1
dc_shell> read_preserved_function_netlist module2_elab.db
           -design_library library_name1
dc_shell> link
```

How to create and use preserved functions is described in “Using Preserved Functions” on page 5-23. Also see the coding guidelines in the *CoCentric™ SystemC Compiler Behavioral Modeling Guide*.

Speculative Execution

You can reduce the length of critical paths that contain conditional operations by allowing SystemC Compiler to perform speculative execution. The `bc_enable_speculative_execution` variable is set to `false` by default. To enable speculative execution, set this variable to `true` before executing the `schedule` command. Enter

```
dc_shell> bc_enable_speculative_execution = "true"
dc_shell> schedule -io_mode superstate_fixed
```

When speculative execution is enabled, conditional operations are precomputed before the results of the conditional branches are known. Results of branches that are not executed are ignored. This applies only to data path operations, such as

```
if (condition)
    y = z + q;
else
    y = z - q;
```

The original behavioral description, for example, could contain reading of an input port, an add, a subtract, and writing the results to an output port that must occur in the same clock cycle, as shown in Example 8-1.

Example 8-1 Executing Without Speculative Execution

```
...
wait();
wait();
cond_bool = in_port.read() + b;
if (cond_bool)
    z = x - y;
else
    z = y - x;
out_port.write(z);
wait();
...
```

Synthesis of the code in Example 8-1 requires control chaining. Control chaining is when the condition controlling the selection of a branch execution happens in the same cycle as operations in that branch. This happens in Example 8-1 because the following occur in the same clock cycle:

- The read of `in_port`
- The computation of `cond_bool`
- The selection of the appropriate branch
- The execution of the appropriate subtraction
- The write to the output

Enabling speculative execution allows SystemC Compiler to restructure the code internally similar to Example 8-2, where the changes are shown in bold. When you enable speculative execution, control chaining is preempted and the length of the critical paths are reduced.

Example 8-2 Executing With Speculative Execution

```
...
x1 = x - y;
wait();
x2 = y - x;
wait();
cond_bool = in_port.read() + b;
if (cond_bool)
    z = x1;
else
    z = x2;
out_port.write(z);
wait();
...
```

Setting a Specific Implementation for Components

Use the `bc_set_implementation` command to define a specific implementation for an operation before executing the `bc_time_design` command. This overrides the `bc_time_design` command default implementation selection for that operation. For example, to specify the `DW01_add`, `cla` implementation for the `add` operation on line 114, enter

```
dc_shell> bc_set_implementation entry/main_loop/add_114
           -module DW01_add
           -implementation cla
dc_shell> bc_time_design
```

The `-module` option specifies the synthetic component from a DesignWare library, and the `-implementation` option specifies the specific implementation of that component. The implementation is used for the `add_114` operation.

You can use the `bc_set_implementation` command to list all possible implementations for an operation. For example, to list the possible implementations for the `add_114` operation, enter

```
dc_shell> bc_set_implementation entry/main_loop/add_114
          -list_valid
```

You may inadvertently restrict sharing of multifunctional units when you use the `bc_set_implementation` command. For example, if you specify an `DW01_add` implementation for an add operation, you are not allowing sharing of the `DW01_addsub` implementation for that add operation with another subtraction operation.

You can also use the `bc_set_implementation` command to define an implementation for all operations of the same type. Use the `find` command to extract the full path to the operations. For example, to find all the add operations and specify that the `DW01_add`, `cla` implementation is used, enter

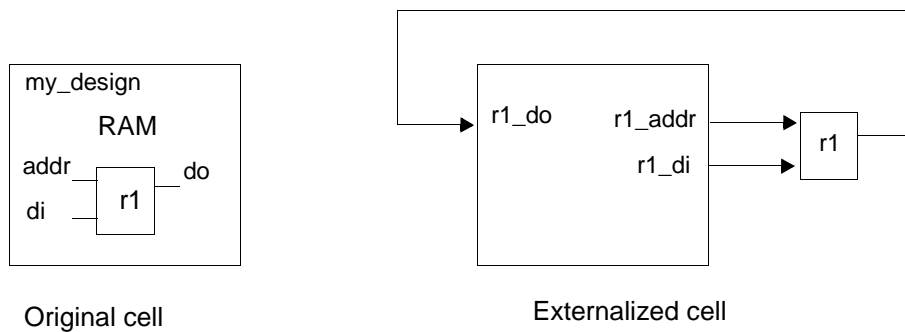
```
dc_shell> bc_set_implementation find (cell {*add*} -hier)
          -module DW01_add
          -implementation cla
```

Externalize a Cell

Externalizing a cell in your design means to make it an external model, for example a memory model.

The external cell outputs become inputs to your design, and the external cell inputs become outputs of your design. Figure 8-1 illustrates the r1 cell before and after making it external.

Figure 8-1 Externalize a Cell



To externalize a RAM cell,

1. Modify your behavioral description to map an array to a RAM.
2. Schedule the design.
3. Use the `compile` command to compile the design into gates.
4. Use the `externalize_cell` command to externalize the RAM cell.

For example,

```
dc_shell> schedule -io_mode super  
dc_shell> compile  
dc_shell> externalize_cell RAM_A
```

Advanced Techniques

8-10

A

Setting Up SystemC Compiler

This appendix describes the basic information and commands you need to know to set up and start SystemC Compiler in the following sections:

- Defining Environment Variables and Paths
- Defining Libraries and Other Variables
- Starting the SystemC Compiler Command Interface
- Issuing SystemC Compiler Commands
- Using Scripts
- Using `compile_systemc` Command Preprocessor Options
- Starting BCView
- Starting the Memory Wrapper Tool

- Getting Command, Variable, and Error Help

Defining Environment Variables and Paths

Before you can start SystemC Compiler on your workstation, you need to define the SYNOPSIS, SNPSLM_LICENSE_FILE, and SYSTEMC_CPP environment variables. Define the SYNOPSIS environment variables as the path to the SystemC Compiler installation.

```
setenv SYNOPSIS path
```

where *path* is the Synopsys synthesis products installation (for example, on your system it could be /usr/releases/bin/synopsys)

```
set path = ($SYNOPSIS/sparcOS5/syn/bin $path)
```

```
setenv SNPSLM_LICENSE_FILE your_license_key_path
```

```
setenv SYSTEMC_CPP "path options -Iinclude_path"
```

where *path options* is one of the following:

- the GNU C++ Compiler version 2.95.2 or later compiler options are:

```
-trigraphs -E -C -U__GNUG__  
-U__GNYG__ -Wp, -no-gcc, -pedantic
```

or

- the Sun SparcWorks C++ Compiler version 5.0 or later compiler options are:

```
-E -xCC -Xc
```

The *include_path* is the path to the SystemC include directory with the *systemc.h* header file (part of the SystemC Class Library installation).

Defining Libraries and Other Variables

SystemC Compiler uses the same variables as Design Compiler to set up synthesis. The following table shows the important variables that define a synthesis setup.

Variable	Description	Example Definition
<code>target_library</code>	Technology Library	<code>{"tc6a_cbacore.db"}</code>
<code>synthetic_library</code>	DesignWare Library	<code>{"dw01.sldb" "dw02.sldb"}</code>
<code>link_library</code>	Link Library	<code>{"*" "}" + target_library + synthetic_library</code>
<code>search_path</code>	Search Path	<code>search_path + ./CC + ./DB + ./</code> (By default, <code>search_path</code> includes the current working directory and the <code>\$SYNOPTSYS/libraries/syn path.</code>)

You can define SystemC Compiler variables in a `.synopsys_dc.setup` file, from the command interface shell, or in your command script.

SystemC Compiler reads the .synopsys_dc.setup files in the following order:

Search Path	Typical Variable Definitions
`\${SYNOPTSYS}/admin/setup/.synopsys_dc.setup	Site setup definitions
~/.synopsys_dc.setup	User setup definitions
./synopsys_dc.setup	Project setup definitions

For details about these variables, the .synopsys_dc.setup file, choosing a target technology, a wire load model, and operating conditions, see the *Design Compiler Command-Line Interface Guide* and the Synopsys man pages.

Starting the SystemC Compiler Command Interface

To invoke the SystemC Compiler command interface, change to the working directory containing your design source and enter the dc_shell command at the system prompt:

```
unix% dc_shell
```

This command launches the command interface, reads the .synopsys_dc.setup files, and displays a dc_shell prompt.

Creating a command.log File

SystemC Compiler records the commands you enter at the command prompt in the command log file. A new command log file is created each time you launch dc_shell. Remember to save each log file with a different file name before starting another command interface shell, otherwise it will be overwritten

By default, the site-wide `.synopsys_dc.setup` file defines the `command_log_file` as

```
command_log_file = "./command.log"
```

After you run an interactive session of SystemC Compiler, you can use the `command.log` file to create and customize a command script file (see “Using Scripts” on page A-6.)

Recording Your Command Session

To record the commands you issue and the system responses in a log file for later evaluation, at the UNIX prompt, pipe the output of the `dc_shell` command to the UNIX `tee` command. The `tee` command copies the output of your screen to the designated log file. For example,

```
unix% dc_shell | tee filename.log
```

Issuing SystemC Compiler Commands

After you launch the SystemC Compiler tool, you enter commands at the `dc_shell` prompt, for example

```
dc_shell> compile_systemc my_design.cc
```

If SystemC Compiler is able to execute the command successfully, the system response is 1. However, if SystemC Compiler cannot execute the command, the system response is 0 and an error message informing you of the problem is displayed.

Listing SystemC Compiler Variables

You can list all of the SystemC Compiler variables for behavioral synthesis and their current settings by using the following command at the `dc_shell` prompt:

```
dc_shell> list -variables bc
```

Using Scripts

A script file, also called a command script, is a sequence of `dc_shell` commands in a text file. Command scripts enable you to execute `dc_shell` commands automatically.

Creating Scripts

To create a command script, create a text file of the commands you want to enter at the `dc_shell` prompt. Or, you can use a saved `command.log` file and modify the commands for your current synthesis run.

Script Example

Example A-1 shows an example command script you can use as a guide to create a script that suits your requirements.

Example A-1 SystemC Compiler Command Script

```
/* define variables */
bc_enable_analysis_info    = "true"

target_library             = {"tc6a_cbacore.db"}
synthetic_library         = {"dw01.sldb" "dw02.sldb"}
link_library               = {"*"} + target_library + synthetic_library;
search_path                = search_path;

clock_name                 = "clk"
clock_period               = 20

read dw01.sldb
read standard.sldb

/* parse and elaborate SystemC code */
compile_systemc cmult.cc

/* write elaborated db file */
write -f db -hier -o cmult_elab.db

/* set constraints on the chip */
create_clock clock_name -p clock_period

/* estimate timing and report estimates */
bc_time_design
report_resource_estimates

write -hier -o cmult_timed.db

/* check design for coding style */
bc_check_design

/* display time stamp to see scheduling time */
sh date

/* schedule and report scheduling */
schedule -io super -effort medium
report_schedule
sh date
```

```
/* write design database and rtl code */
write -hier -f verilog -o cmult_rtl.v
write -hier -f db -o cmult_rtl.db

/* compile to gates & write gate-level database */
sh date
compile
sh date
write -hier -f db -o cmult_gate.db

/* report timing, resources, and area */

report_timing
report_resources
report_area

quit
```

Using the Script

You can provide a command script when you start `dc_shell` or use the `include` command from `dc_shell`.

To provide a command script named `command.scr` when you start `dc_shell`, enter the following at a UNIX prompt:

```
unix% dc_shell -f command.scr
```

To run the same command script from a `dc_shell` prompt, enter

```
dc_shell> include oommand.scr
```

For more information about creating and using commands scripts, see the *Design Compiler Command-Line Interface Guide*

Using UNIX Shell Commands

You can include a UNIX shell command in a SystemC Compiler script by preceding it with `sh`, for example

```
sh date
```

Note:

The UNIX shell command response is 0 for success or 1 to indicate an error, which is the reverse of SystemC Compiler responses.

Using `compile_systemc` Command Preprocessor Options

The `compile_systemc` command uses a standard C++ compiler's C++ preprocessor. The preprocessor is defined by the `SYSTEMC_CPP` environment variable, which is set to the GCC compiler by default (see “Defining Environment Variables and Paths” on page A-2).

You can also define preprocessor options with the following `compile_systemc` command options:

```
dc_shell> compile_systemc  
          [-cpp cpp_program]  
          [-cpp_options options]  
          design.cc
```

Use the `-cpp` option to specify a C++ preprocessor to use with the `compile_systemc` command other than the default.

This option directs the `compile_systemc` command to use the specified C++ preprocessor. Otherwise, it uses the C++ preprocessor that is defined by the `SYSTEMC_CPP` environment variable.

You can also specify `compile_systemc` command preprocessor options with the `-cpp_options` argument. For example, the options can be used as

```
dc_shell> compile_systemc -cpp
  "/usr/local/bin/g++ -trigraphs -E -C -U__GNUC__ -U__GNYG__
  -Wp,-no-gcc,-pedantic" cmult.cc
```

where `usr/local/bin` is the path to the `g++` executable and the other terms within the double-quotes are `g++` preprocessor options. The preprocessor options may be different for your system. Enclose the `cpp_program` specification in double-quotes.

You can also use the `-cpp_options` option to specify C++ preprocessor arguments you want the `compile_systemc` command to pass to the C++ preprocessor.

For example, to pass arguments to the C preprocessor without changing the designated preprocessor, enter

```
dc_shell> compile_systemc -cpp_options
  "-DMACX -I/u/systemc/include" cmult.cc
```

where `DMACX` defines a C preprocessor macro and instructs the C preprocessor to look for header files in the directory `/u/systemc/include`.

For information about GCC or SparcWorks compiler options, consult the compiler documentation.

Starting BCView

You can start BCView from within the SystemC Compiler environment or from a UNIX shell. Before using the SystemC Compiler `compile_systemc` command, set the `bc_enable_analysis` variable to true so SystemC Compiler generates the additional analysis information used by BCView. See “Preparing Designs for BCView” on page 6-2 for details.

If you are running BCView on a remote system, set the display to your system, for example

```
unix% setenv DISPLAY hostname:0.0
```

Starting BCView From dc_shell

Start BCView from within `dc_shell` either after a successful schedule or after scheduling errors occur.

To start BCView from `dc_shell`,

- Enter the following at the `dc_shell` prompt:

```
dc_shell> bc_view  
[-output out_db_file]  
[-project_file project_file_name]  
[-dont_start]
```

This generates a project settings file, opens an xterm window, and starts BCView. Use the options if you want to also specify the project file name or create the `.proj` file without starting BCView.

For more information about the `bc_view` command and its options, see the Synopsys man pages.

Starting BCView From a UNIX Shell

You can start BCView from a UNIX shell if the project settings file is available.

To start BCView from a UNIX shell,

1. Produce a .db file with analysis information at appropriate stages in the design flow (for example, after the `schedule` command).
2. Create a project settings file (.proj file).

To do this from `dc_shell`, enter

```
dc_shell> bc_view -dont_start
```

3. Issue the following command:

```
unix% bc_view -f file.proj
```

Using BCView in Your Script

You can incorporate BCView directly into your design flow by including it in your script. Example A-2 shows part of a script that includes BCView in the flow. The BCView related lines are bold.

Example A-2 Using BCView in a Script

```
bc_enable_analysis_info = "true"  
compile_systemc design.cc  
...  
schedule -io superstate  
...  
...(successful schedule)  
bc_view
```

Opening BCView Windows

When you start BCView for post-scheduling analysis, it automatically displays the FSM Viewer, HDL Browser, Selection Inspector, and Reservation Table windows.

When you start BCView in error analysis mode, it automatically displays the Selection Inspector, Code Browser, and Scheduling Error Analyzer windows. To open BCView in error analysis mode, see “Using the Scheduling Error Analyzer” on page 6-10.

To open a BCView window during a session,

- Choose Window > Create > *window_name* in any currently open BCView window.

where *window_name* is the name of the window you want to open.

Starting the Memory Wrapper Tool

Before using the Memory Wrapper tool, set your display environment variable. Then change to the directory that contains your memory files to start the program.

To set the display environment variable,

- Enter the following at the UNIX prompt:

```
unix% setenv DISPLAY hostname:0.0
```

To change directories and start the Memory Wrapper tool,

- Enter the following at the UNIX prompt:

```
unix% cd my_memory_design  
unix% memwrap &
```

See “Generating Memory Wrappers” on page 7-34 for details about using the Memory Wrapper tool.

Getting Command, Variable, and Error Help

The *CoCentric™ SystemC Compiler Quick Reference* provides a list of frequently used commands and variables.

You can also find information about the SystemC Compiler, Behavioral Compiler, and Design Compiler commands, variables, and errors in the online man pages, which are available using the various access methods described in the following sections.

System Prompt

You can access online man pages for SystemC Compiler commands, variables, and errors from the UNIX prompt by entering the following command:

```
unix% man command_name / variable_name / error_name
```

Note: You need to add the `/${SYNOPSIS}/doc/syn/man` path to your `MANPATH` environment variable.

SystemC Compiler Command Prompt

You can access online man pages for SystemC Compiler commands, variables, and errors from the SystemC Compiler system prompt by entering the following command:

```
dc_shell > help [command_name / variable_name / error_name]
```


B

Complex Number Multiplier Example Files

This appendix shows the source code, command script, and examples of reports generated by SystemC Compiler for the complex number multiplier example. It contains the following sections:

- Complex Number Multiplier Source Code
- Command Script
- Reports Created During Synthesis

Complex Number Multiplier Source Code

Example B-1 shows the source code for the complex number multiplier, which uses two-way handshaking.

Example B-1 Complex Multiplier Source Code

```
/*  
*****  
*/  
  
// cmult.h header file  
SC_MODULE(cmult_hs) {  
    // Declare ports  
    sc_in<bool> reset;  
    sc_in<bool> new_data;  
    sc_in<sc_bv<8> > data_in;  
    sc_in_clk clk;  
    sc_out<bool> ready_for_data;  
    sc_out<bool> output_data_ready;  
    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);  
    }  
};
```

```

/*****/

// cmult.cc implementation file

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

void cmult_hs :: entry()
{
    sc_int<8> a, b, c, d;

    //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(); //required clock before while loop

    while (true)
    {
        ready_for_data.write(true);
        output_data_ready.write(false);

        wait_until(new_data.delayed() == true);
        ready_for_data.write(false);

        // 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 * c);
        output_data_ready.write(true);
        wait();
    }
}

```

Command Script

Example B-2 shows the command script to synthesize and compile the complex number multiplier to gates. It uses the commands and writes files according to the steps recommended in Chapter 2, “Using SystemC Compiler.”

Example B-2 Command Script for Complex Number Multiplier

```
/******run_cmult.scr script*****/
/* define variables */
bc_enable_analysis_info    = "true"

target_library            = {"tc6a_cbacore.db"}
synthetic_library         = {"dw01.sldb" "dw02.sldb"}
link_library              = {"*"} + target_library + synthetic_library;
search_path               = search_path;

clock_name                = "clk"
clock_period              = 20

read dw01.sldb
read standard.sldb

/* parse and elaborate SystemC code */
compile_systemc cmult.cc

/* write elaborated db file */
write -f db -hier -o cmult_elab.db

/* set constraints on the chip */
create_clock clock_name -p clock_period

/* estimate timing and report estimates */
bc_time_design
report_resource_estimates

write -hier -o cmult_timed.db

/* check design for coding style */
bc_check_design

/* display time stamp to see scheduling time */
sh date
```

Complex Number Multiplier Example Files


```
/* schedule and report scheduling */
schedule -io super -effort medium
report_schedule
sh date

/* write design database and rtl code */
write -hier -f verilog -o cmult_rtl.v
write -hier -f db -o cmult_rtl.db

/* compile to gates & write gate-level database */
sh date
compile
sh date
write -hier -f db -o cmult_gate.db

/* report timing, resources, and area */

report_timing
report_resources
report_area
```

Reports Created During Synthesis

You can create various reports during a typical synthesis session. Examples of the reports created with the commands in Chapter 2, “Using SystemC Compiler,” are shown in the following sections.

Estimated Resources

Example B-3 shows the report of resource estimates generated by the `report_resource_estimates` command.

Example B-3 Report Resource Estimates

```
/****report_resource_estimates*****/

Cumulative delay starting at new_data_22:
  new_data_22 = 0.000000
  neq_L22     = 0.067000

Cumulative delay starting at neq_L22:
  neq_L22     = 0.067000

Cumulative delay starting at data_in_30:
  data_in_30  = 0.000000
  mul_35     = 6.340029
  sub_35     = 10.138293
  real_out_35 = 10.138293

Cumulative delay starting at real_out_35:
  real_out_35 = 0.000000

Cumulative delay starting at mul_35:
  mul_35     = 6.357016
  sub_35     = 10.150984
  real_out_35 = 10.150984

Area for processors that can implement mul_35 (* = used for timing):
  *DW02_mult(nbw) = 2750.742432

Cumulative delay starting at mul_35_2:
  mul_35_2   = 6.357016
  sub_35     = 10.430123
  real_out_35 = 10.430123
```

Complex Number Multiplier Example Files

Area for processors that can implement mul_35_2 (* = used for timing):
*DW02_mult(nbw) = 2750.742432

Cumulative delay starting at imaginary_out_36:
imaginary_out_36 = 0.000000

Cumulative delay starting at data_in_32:
data_in_32 = 0.000000
mul_35_2 = 6.340029
mul_36_2 = 6.340029
mul_36 = 6.340029
add_36 = 10.138293
imaginary_out_36 = 10.138293
sub_35 = 10.417433
real_out_35 = 10.417433

Cumulative delay starting at mul_36:
mul_36 = 6.357016
add_36 = 10.150984
imaginary_out_36 = 10.150984

Area for processors that can implement mul_36 (* = used for timing):
*DW02_mult(nbw) = 2750.742432

Cumulative delay starting at add_36:
add_36 = 8.784022
imaginary_out_36 = 8.784022

Area for processors that can implement add_36 (* = used for timing):
*DW01_add(rpl) = 94.239998
DW01_addsub(rpl) = 503.678986

Cumulative delay starting at data_in_26:
data_in_26 = 0.000000
mul_35 = 6.357016
mul_36 = 6.357016
add_36 = 10.150984
sub_35 = 10.150984
imaginary_out_36 = 10.150984
real_out_35 = 10.150984

Cumulative delay starting at data_in_28:
data_in_28 = 0.000000
mul_36_2 = 6.357016
mul_35_2 = 6.357016
add_36 = 10.097484
imaginary_out_36 = 10.097484

```
sub_35 = 10.430123
real_out_35 = 10.430123
```

Cumulative delay starting at sub_35:

```
sub_35 = 9.158618
real_out_35 = 9.158618
```

Area for processors that can implement sub_35 (* = used for timing):

```
DW01_sub(rpl) = 410.551453
*DW01_addsub(rpl) = 503.678986
```

Cumulative delay starting at mul_36_2:

```
mul_36_2 = 6.357016
add_36 = 10.097484
imaginary_out_36 = 10.097484
```

Area for processors that can implement mul_36_2 (* = used for timing):

```
*DW02_mult(nbw) = 2750.742432
```

```
Cycle Margin      : 2.86 (Default)
  FSM              : 0.55
  MUX              : 1.21
  FF               : 1.11
Clock Uncertainty : 0.00
```

Schedule Report

Example B-4 shows the schedule report generated by the `report_schedule` command.

Example B-4 Schedule Report

```
/******report_schedule*****/

*****
          Date       : Wed Nov  8 13:18:51 2000
          Version    : 2000.11-PROD
          Design     : cmult_hs
*****

*****
* Summary report for process entry: *
*****

-----
          Timing Summary
-----

Clock period 20.00
Loop timing information:
  entry.....8 cycles (cycles 0 - 8)
    loop_17.....7 cycles (cycles 1 - 8)
      loop_22.....1 cycle (cycles 2 - 3)
        (continue) skip_short_branch_1..... (cycle 3)
        (exit) EXIT_L22_1..... (cycle 2)

-----

          Area Summary
-----

Estimated combinational area    6127
Estimated sequential area      1734
TOTAL                          7861

9 control states
11 basic transitions
2 control inputs
7 control outputs

-----

          Resource types
-----
```

Register Types

=====
8-bit register.....3
16-bit register.....1

Operator Types

=====
(8_8->16)-bit DW02_mult.....2
(16_16->16)-bit DW01_add.....1
(16_16->16)-bit DW01_sub.....1

I/O Ports

=====
1-bit input port.....1
1-bit registered output port.....2
8-bit input port.....1
16-bit registered output port.....2

Area Report

Example B-5 shows the report of area generated by the `report_area` command.

Example B-5 Report Area

```
/*****report_area*****/
```

```
*****  
Report : area  
Design : cmult_hs  
Version: 2000.11-PROD  
Date   : Wed Nov  8 13:19:25 2000  
*****
```

Library(s) Used:

```
    cba_core (File: /u/bcp/IMAGES/rhei_dcshell_PROD/libraries/syn/  
tc6a_cbacore.db)
```

```
Number of ports:          45  
Number of nets:           303  
Number of cells:          232  
Number of references:     31
```

```
Combinational area:      1303.439941  
Noncombinational area:   371.780029  
Net Interconnect area:  5836.250000
```

```
Total cell area:        1675.219971  
Total area:              7511.469727
```

Timing Report

Example B-6 shows the report of timing generated by the `report_timing` command.

Example B-6 Report Timing

```
*****report_timing*****/
```

```
Information: Updating design information... (UID-85)
```

```
*****
```

```
Report : timing
        -path full
        -delay max
        -max_paths 1
```

```
Design : cmult_hs
Version: 2000.11-PROD
Date   : Wed Nov  8 13:19:22 2000
```

```
*****
```

```
Operating Conditions:
Wire Load Model Mode: top
```

```
Startpoint: fsm_block_cell/entry_ctl_state/entry_ctl_state[4]
             (rising edge-triggered flip-flop clocked by clk)
```

```
Endpoint:  imaginary_out_reg/imaginary_out_reg[15]
             (rising edge-triggered flip-flop clocked by clk)
```

```
Path Group: clk
Path Type:  max
```

Des/Clust/Port	Wire Load Model	Library
cmult_hs	tc6a120m2	cba_core

Point	Incr	Path
clock clk (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
fsm_block_cell/entry_ctl_state/entry_ctl_state[4]/CLK (fdm5a2)	0.00	0.00 r
fsm_block_cell/entry_ctl_state/entry_ctl_state[4]/Q (fdm5a2)	1.28	1.28 r
U79/Y (inv1a0)	0.63	1.91 f
U69/Y (or2c1)	0.77	2.68 r
U67/Y (buf1a2)	0.74	3.43 r

U4/Y (ao4a2)	0.85	4.28 r
r141/A[4] (cmult_hs_DW02_mult_8_8_1)	0.00	4.28 r
r141/U57/Y (and2a2)	0.51	4.78 r
r141/U1/U3140_1_4_0/CO (fala0)	0.66	5.44 r
r141/U1/U3140_2_5_0/S (fala0)	0.86	6.30 r
r141/U1/U3140_3_5_0/S (fala0)	0.78	7.09 f
r141/U1/U3220_4_5/S (hala0)	0.60	7.69 f
r141/U1/U9720/A[0] (cmult_hs_DW01_add_11_1)	0.00	7.69 f
r141/U1/U9720/U0_1_0/Y (or2c0)	0.36	8.05 r
[7m--More--[m[A		
[K r141/U1/U9720/U0_3_0/Y (and2b1)	0.47	8.51 r
r141/U1/U9720/U0_5_0/Y (xnor2a2)	0.59	9.10 f
r141/U1/U9720/SUM[0] (cmult_hs_DW01_add_11_1)	0.00	9.10 f
r141/PRODUCT[5] (cmult_hs_DW02_mult_8_8_1)	0.00	9.10 f
add_36/B[5] (cmult_hs_DW01_add_16_0)	0.00	9.10 f
add_36/U1_5/CO (fala0)	0.73	9.83 f
add_36/U1_6/CO (fala0)	0.64	10.47 f
add_36/U1_7/CO (fala0)	0.64	11.11 f
add_36/U1_8/CO (fala0)	0.64	11.74 f
add_36/U1_9/CO (fala0)	0.64	12.38 f
add_36/U1_10/CO (fala0)	0.64	13.02 f
add_36/U1_11/CO (fala0)	0.64	13.65 f
add_36/U1_12/CO (fala0)	0.64	14.29 f
add_36/U1_13/CO (fala0)	0.64	14.93 f
add_36/U1_14/CO (fala0)	0.67	15.60 f
add_36/U1_15/Y (xor3a2)	0.51	16.10 f
add_36/SUM[15] (cmult_hs_DW01_add_16_0)	0.00	16.10 f
U36/Y (mx2a2)	0.34	16.45 f
U118/Y (and2b1)	0.27	16.72 f
imaginary_out_reg/imaginary_out_reg[15]/D (fd1a1)	0.00	16.72 f
data arrival time		16.72

clock clk (rise edge)	20.00	20.00
clock network delay (ideal)	0.00	20.00
imaginary_out_reg/imaginary_out_reg[15]/CLK (fd1a1)	0.00	20.00 r
library setup time	-0.27	19.73
data required time		19.73

data required time		19.73
data arrival time		-16.72

slack (MET)		3.01

Report Resource

Example B-7 shows the report of resources generated by the `report_resource` command after executing the `compile` command to do logic synthesis.

Example B-7 Report Resources

```
/******report_resources*****/
```

```
*****  
Report : resources  
Design : cmult_hs  
Version: 2000.11-PROD  
Date   : Wed Nov  8 13:19:24 2000  
*****
```

Resource Sharing Report for design cmult_hs in file cmult.cc

Resource	Module	Parameters	Contained Resources	Contained Operations
r139	DW01_add	width=16		add_36
r141	DW02_mult	B_width=8 A_width=8		mul_35 mul_35_2 mul_36_2
r143	DW02_mult	B_width=8 A_width=8		mul_36
r146	DW01_sub	width=16		sub_35

Implementation Report

Cell	Module	Current Implementation	Set Implementation
add_36	DW01_add	rpl	
mul_36	DW02_mult	nbw	
r141	DW02_mult	nbw	
sub_35	DW01_sub	rpl	

No multiplexors to report

Complex Number Multiplier Example Files

Index

Symbols

#include directive 8-2

A

all_inputs function 3-9

allocating,overview 1-9

analyze command 7-66

architectural exploration 5-2

 evaluation 6-21

 examples 5-2

 guidelines 5-6

area 3-21

 constraints 3-1

 estimates 3-16, 3-17

array, large 7-2

asynchronous memories 7-24

attribute

 map_to_modules 7-17

 map_to_registerfiles 7-7

B

bc_chain_read_into_mem variable 5-12

bc_chain_read_into_oper variable 5-12

bc_check_design command 2-10

bc_dont_register_input_port command 5-18

bc_enable_analysis_info variable 2-5, 6-2

bc_enable_chaining variable 5-11

bc_enable_speculative_execution variable 8-4

bc_margin command 5-15, 5-16

bc_report_arrays command 7-8

bc_report_memories command 7-22, 7-30

bc_set_implementation command 8-6

bc_time_design command 2-11, 5-12

bc_use_registerfiles variable 7-6

bc_view command 6-3

BCView 6-1

 analyzing scheduling errors 2-15

 design evaluation 2-5

 error analysis mode A-13

 finding scheduling errors 2-15

 overview 1-15

 recommended flow 6-8

 Reservation Table

 chained operations 6-36

 description 6-26

 reading columns and rows 6-28

 reviewing results 6-2

 starting A-11

 windows 6-5

 FSM Viewer 6-6

 HDL Browser 6-6

 Reservation Table 6-6

- Scheduling Error Analyzer 6-7
- Selection Inspector 6-6, 6-11
- behavioral description
 - clock cycle and I/O 3-3
- bit-level timing 3-13
- bitwise timing 5-8

C

- calculating margin 5-12
- cells to constrain 4-31
- chain_operations command 5-11
- chained
 - operations 5-11, 6-36
- changing design name 2-7
- clock
 - creation 2-8
 - period setting 2-8
- clock cycle
 - code 4-10
 - compared to I/O 3-3
 - margin 3-14, 5-12
 - relation to operation delay 3-2
 - utilization 6-54
- code browser, BCView 6-6
- Code Editor, memory 7-61
- combinational logic
 - definition 3-2
 - delay 3-2
- command
 - log file A-4
- command flow 2-4
 - overview 2-3
 - preserved functions 5-32
- command script
 - complex multiplier B-4
- commands
 - analyze 7-66
 - bc_check_design 2-10
 - bc_dont_register_input_port 5-18
 - bc_margin 5-15, 5-16
 - bc_report_arrays 7-8
 - bc_report_memories 7-22, 7-30
 - bc_set_implementation 8-6
 - bc_time_design 2-11, 5-12
 - bc_view 6-3
 - chain_operations 5-11
 - compile 2-24, 2-26
 - compile_preserved_functions 5-30
 - compile_systemc 2-6
 - create_clock 2-8, 3-7
 - dc_shell 2-3
 - define_design_lib 5-29
 - dont_chain_operations 5-11
 - elaborate 2-7
 - entry interface A-5
 - externalize_cell 7-26, 8-8
 - find 4-33
 - free 2-16
 - ignore_array_loop_precedences 7-11
 - ignore_array_precedences 7-11
 - ignore_memory_loop_precedences 7-33
 - ignore_memory_precedences 7-31
 - include A-8
 - link 2-7
 - list 7-18
 - list -libs 3-9, 5-35
 - order 2-4
 - pipeline_loop 4-45
 - read 2-20
 - read_lib 7-66
 - read_preserved_function_netlist 5-28
 - recording in a log A-5
 - remove_analysis_info 2-6, 6-3
 - remove_attribute 5-8
 - remove_clock 3-8
 - remove_design 2-16
 - remove_scheduling_constraints 4-54
 - report_area 2-16
 - report_clock 3-7
 - report_hierarchy 4-34
 - report_lib 3-10
 - report_multicycles 5-20

- report_resource_estimates 2-12, 3-17, 5-15
- report_resources 2-16
- report_schedule 2-15, 4-20
 - abstract FSM 4-28
 - operations 4-22
- report_schedule variables 4-26
- report_scheduling_constraints 4-53
- report_synlib 5-35, 7-19
- report_timing 2-16
- response 2-6
- schedule 2-13, 2-14, 4-18, 4-31, 5-11
 - extend_latency 2-13
- script entry A-6
- set_common_resource 4-55
- set_cycles 4-37, 4-42
- set_dont_use 5-7, 5-15
- set_exclusive_use 4-57
- set_input_delay 3-8
- set_max_cycles 4-37, 4-42
- set_memory_input_delay 7-26
- set_memory_output_delay 7-26, 7-27
- set_min_cycles 4-37, 4-42
- set_operating_conditions 3-9
- set_wire_load 3-12
- UNIX shell commands A-9
- UNIX shell entry A-9
- write A-10
 - RTL .db 2-19
 - timed design 2-12
- write_lib 7-66
- write_rtl 2-20
- compile_preserved_functions command 5-30
- compile_systemc command 2-6
- compiler directives
 - line_label 4-32
 - map_to_operator 5-34
 - preserve_function 5-24
 - resource 7-7, 7-17, 7-66
- compiling gate-level netlist 2-24, 2-26
- complex multiplier
 - command script B-4

- source code B-2
- components
 - sharing 4-7
- constraints
 - environmental conditions 2-9
 - identifying cells 4-31
 - initial 2-8, 2-9, 3-1
 - latency 4-37
 - loop 4-38
 - memory 7-28
 - nested loops 4-39
 - removing 4-54
 - schedule 2-14, 4-18, 4-31
 - viewing 6-50
- control constructs 6-12
- control steps, in Scheduling Error Analyzer 6-15
- create_clock command 2-8, 3-7
- creating cells 4-31
- critical path 3-19
 - determining 1-6
 - reviewing 6-61
- cycle-fixed mode
 - description 4-13
 - description of 4-13
 - I/O scheduling 4-12

D

- data path
 - creation 1-10
 - in a circuit diagram 1-5
- dc_shell A-4
- dc_shell command entry 2-3
- define environmental variables A-2
- delay of an operation 6-53
- deleting analysis information 6-3
- design
 - checking 2-11
 - command order 2-4
 - constraints 2-9
 - flow

- resuming 2-3
- resuming from RTL .db 2-17
- SystemC Compiler 2-1
- name changing 2-7
- reducing complexity 5-23
- summary reports 6-61
- synchronous 3-2
- timing 2-8
- DesignWare
 - components 5-23
 - using 5-34
 - pipelined components 5-37
- dont_chain_operations command 5-11
- dont_start option A-12

E

- elaborate command 2-7
- entering commands 2-3
- environmental variables
 - defining A-2
- error analysis mode, BCView A-13
- error messages A-15
- errors
 - HLS-51 6-9
 - HLS-52 6-9
 - messages 6-10
 - scheduling 6-9
- estimating time and area 2-11
- example designs
 - file location 1-13
- explore architecture 5-2
 - examples 5-2
 - guidelines 5-6
- extend latency 4-19, 4-55
- externalize_cell command 7-26, 8-8

F

- fanouts, edges in Reservation Table 6-49

- find command 4-33
- finding
 - array operation cells 7-14
 - cells 4-33
 - inherent constraints 6-50
 - Reservation Table objects 6-28
 - scheduling errors 6-9
 - user constraints 6-50
- flip-flop
 - margin 3-15
- flow recommended for BCView 6-8
- for loops 4-41
- free command 2-16
- FSM
 - Conditions/Actions window 6-24
 - definition 4-8
 - in a circuit diagram 1-5
 - margin 3-16
- FSM creation 1-10
- FSM Viewer 6-6
 - usage 6-23
- functions
 - all_inputs 3-9
 - preserve 5-23

G

- gate-level netlist, writing out 2-24, 2-26
- global margin value 5-16
- goals, resource 4-55

H

- handshake
 - pipelining loops 4-50
- hardware allocation 1-9
- HDL Browser 6-6
- HDL file 2-20, 2-21
- help A-15
- hierarchy 5-23

HLS-51 error 6-9

HLS-52 error 6-9

I

I/O

- define 3-3

- mode selection 2-13

- operations 4-10

ignore_array_loop_precedences command 7-11

ignore_array_precedences command 7-11

ignore_memory_loop_precedences command 7-33

ignore_memory_precedences command 7-31

implementation selection 5-7

include command A-8

inferring memories 7-15

inferring registers 4-5

infinite loops 4-41

Info Tips

- finding scheduling errors 6-16

- loop names 6-45

initial constraints, specifying 2-8

initial interval

- pipelining, clock cycles 4-45, 4-46, 4-47

initiation interval 4-47

inputs to SystemC Compiler 1-12

invoking BCView A-11

L

labeling source code lines 4-32

large array 7-2

latency 4-57

- definition 4-18

- optimize 5-1

lifetime analysis of variables 4-6

line_label compiler directive 4-32

link command 2-7

link_library variable 2-5, A-3

list command 7-18

- libraries 3-9, 5-35

log, creating file of commands A-4, A-5

loops

- boundaries 4-41, 4-43

- carry dependencies 4-48

- constraining 4-37, 4-38

- details 6-46

- exits 6-45

- for 4-41

- infinite 4-41

- names 6-45

- nested

 - constraining 4-39

- operations 6-48

- pipelining 4-44

 - exit 4-52

 - restrictions 4-47

 - with handshake 4-50

 - with memory and I/O 4-49

- viewing 6-44

- while 4-41, 4-52

M

man pages A-15

map_to_modules attribute 7-17

map_to_operator compiler directive 5-34

map_to_registerfiles attribute 7-7

margin

- calculation 5-12

- global value 5-16

maximum delay of a processor 6-53

memory

- access 7-29

 - strategy 7-15

- and registers 7-5

- architecture exploring 7-35

- assigning pins to wrapper logical ports 7-46

- asynchronous 7-24

- basics 7-15
- Code Editor 7-61
- exploratory 7-35
- exploratory wrapper 7-67
- in a circuit diagram 1-5
- inferring 1-11, 7-15
- latency 7-43
- pipelined accesses 7-29
- pipelining loops 4-49
- resources
 - constraining 7-28
- starting wrapper GUI A-14
- timing 7-25
- vendor 7-34
- vendor library 7-34
- vendor wrapper 7-39
- wrapper properties 7-52
- wrapper testbench
 - Memory Wrapper Generator
 - testbench generation 7-79
- Memory Wrapper Generator 7-16
 - tool description 7-34
- methodology of SystemC Compiler 2-1
- multicycle components
 - replacing 5-23
- multicycle operations
 - definition 5-19
 - identifying 6-51
 - implementation 5-19
 - increased latency 5-21
 - latency increase 5-37
- multiple modules 8-2
- multiplexer
 - margin 3-16, 5-14

N

- naming conventions
 - cells 4-31
- netlists
 - levelized 2-21
 - precompiled 8-2

- nodes in SEA 6-12

O

- object_list 5-7
- operations
 - chaining
 - bitwise 5-8
 - description 5-11
 - delay 3-2, 6-34
 - in Scheduling Error Analyzer 6-12
 - overview 3-5
 - selecting 5-7
- operator
 - chaining
 - definition 5-8
- optimize
 - latency and area 5-1
- options
 - dont_start A-12
 - preprocessor A-9
- outputs of SystemC Compiler 1-12
- overview
 - SystemC Compiler
 - output 1-14

P

- parallel paths 3-19
- physical synthesis, preparation 2-25
- pipeline
 - components 5-37
 - constraints
 - scheduling 4-45
 - loop 4-44
 - exit 4-52
 - overview 1-11
 - restrictions 4-47
- pipeline_loop command 4-45
- place and route, preparation 2-24
- precedence

- relations 7-31
- precompiled netlist, creating 5-30
- Precompiled netlists 8-2
- preprocessor options A-9
- preserve_function
 - compiler directive 5-23
 - restrictions 5-33
 - using 5-24
- preserved functions 5-23
 - adding external files to primary design 2-8
 - bit-width restrictions 5-33
 - command
 - flow 5-32
 - compiling 5-30
 - creating 5-25
 - flow for using 5-27
 - identifying what to preserve 5-24
 - restrictions 5-33
- processor chaining
 - definition 6-53
- project settings file A-12

R

- read command 2-20
- read_lib command 7-66
- read_preserved_function_netlist command 5-28
- reducing runtimes 7-2
- register file
 - inferring
 - overview 1-11
- registers
 - allocation 4-6, 4-7
 - and memories 7-6
 - and RAM 7-5
 - bit width 6-42
 - dedicated 4-6
 - exclusive 4-57
 - file operators 7-13
 - inferred 4-5
 - margin 3-15, 5-13
 - removing unnecessary 5-18
 - sharing 4-6, 4-7
 - use 6-42
- remove_analysis_info command 2-6, 6-3
- remove_attribute command 5-8
- remove_clock command 3-8
- remove_design command 2-16
- remove_scheduling_constraints command 4-54
- removing scheduling constraints 4-54
- removing unnecessary registers 5-18
- report_area command 2-16
- report_clock command 3-7
- report_hierarchy command 4-34
- report_lib command 3-10
- report_multicycles command 5-20
- report_resource_estimates command 2-12, 3-17, 5-15
- report_resources command 2-16
- report_schedule command 2-15, 4-20
 - abstract FSM 4-28
 - operations 4-22
 - variables 4-26
- report_scheduling_constraints command 4-53
- report_synlib command 5-35, 7-19
- report_timing command 2-16
- reports
 - array conflicts 7-9
 - clock margin 5-14
 - design summary 6-61
 - hierarchy 4-34
 - multicycle 5-21
 - nonconflicting memory accesses 7-30
 - pipelined loop timing summary 4-46
 - resource estimates 3-17
 - resource estimation, chained operations 5-10
 - schedule of FSM 4-28
 - schedule of operations 4-22
 - schedule of variables 4-26

- schedule summary 4-20
- synthesis B-6
- synthetic memory wrapper 7-19
- Reservation Table 6-6
 - chained operations 6-36
 - clock-cycle utilization 6-54
 - derived edges 6-38
 - connectivity 6-40
 - description 6-26
 - hiding/showing resources 6-31
 - paths 6-38
 - reading columns and rows 6-28
 - register bit-width 6-42
 - register use 6-42
 - shared resources 6-35
 - viewing loop details 6-46
 - viewing loop exits 6-45
 - viewing loops 6-44
 - viewing operations in loops 6-48
- reservation table 4-4
- resource
 - delays 6-33
 - estimates report 5-10
 - setting goals 4-55
 - shared 6-35
 - sharing 4-4, 6-55, 6-59
 - utilization 6-32
- resource compiler directive 7-7, 7-17, 7-66
- resource estimate 3-17
- resource-constrained scheduling 4-19
- resource-driven scheduling 4-55
- restrictions
 - pipelining loops 4-47
 - preserved functions 5-33
- reviewing results with BCView 6-2
- RTL
 - writing the .db file 2-19
 - writing the simulation file 2-21

S

- schedule command 2-13, 2-14, 4-18, 4-31
 - extend latency 4-19, 4-55
 - processor chaining 5-11
- scheduling
 - constraints 2-14, 4-18, 4-31
 - removing 4-54
 - cycle fixed 1-8, 2-13, 4-11, 4-13
 - default 4-18
 - effort level 2-14
 - errors
 - finding 6-9
 - errors, using BCView 2-15
 - extend_latency 4-19, 4-55
 - I/O 4-10
 - I/O cycle-fixed mode 4-12
 - minimizing latency 4-3
 - objectives 4-18
 - operation 4-3
 - overview 1-7, 4-1, 4-2
 - performing 4-18
 - pipeline
 - constraints 4-45
 - resource sharing 4-4
 - resource-constrained 4-19
 - set_common_resource command 4-57
 - smallest area priority 2-13
 - summary report 4-20
 - superstate fixed 1-8, 2-13, 4-11, 4-14, 4-15, 4-16
 - extend_latency 2-13
 - superstates, definition 4-16
 - timing-constrained 4-18
- Scheduling Error Analyzer 6-7, 6-10
 - description 6-9
 - determining control steps 6-15
 - finding scheduling errors 6-16
 - nodes 6-12
- scripts A-6
- search_path variable 2-5, A-3
- Selection Inspector 6-6

- Selection Inspector Window 6-11, 6-37
- set_common_resource command 4-55
- set_cycles command 4-37, 4-42
- set_dont_use command 5-7, 5-15
- set_exclusive_use command 4-57
- set_input_delay command 3-8
- set_max_cycles command 4-37, 4-42
- set_memory_input_delay command 7-26
- set_memory_output_delay command 7-26, 7-27
- set_min_cycles command 4-37, 4-42
- set_operating_conditions command 3-9
- set_wire_load command 3-12
- setup variables A-3
- sharing components 4-7
- sharing registers 4-7
- simulation
 - cycle-accurate leveled HDL netlist 2-21
 - writing out gate-level netlist 2-24, 2-25, 2-26
- source code, labeling lines 4-32
- source browser 6-6
- source code
 - complex multiplier B-2
- speculative execution 8-4
- starting
 - BCView A-11
 - memory wrapper GUI A-14
 - SystemC Compiler A-4
- superstate-fixed mode
 - description of 4-16
 - I/O scheduling 4-14
- superstates, definition of 4-16
- .synopsys_dc.setup file A-3
- synthesis
 - controlling 4-10
 - preserved functions 5-31
 - reports B-6
- synthesis flow 1-3
- synthesizable RTL out
 - write command to generate 2-20

- synthetic library
 - define 2-5
 - location 1-14
- synthetic_library
 - variable 2-5
- synthetic_library variable A-3
- SystemC Compiler
 - design flow 2-1
 - I/O operations 4-11
 - output
 - RT-level 2-18, 2-19
 - timed .db file 2-12

T

- target_library variable 2-5, A-3
- technology library 1-13
 - location 1-13
- time, units of measure 3-21
- timed .db file
 - writing out 2-12
- timing 1-6
 - bit level 3-13
 - constraints 3-1, 4-57
 - estimates 3-17
 - estimation 5-7
- timing of memories 7-25
- timing-constrained scheduling 4-18
- tracing constraints 6-15

U

- UNIX shell commands A-9
- user constraints 6-14
- using scripts 2-3

V

- variables
 - bc_chain_read_into_mem 5-12
 - bc_chain_read_into_oper 5-12

- bc_enable_analysis_info 2-5, 6-2
- bc_enable_chaining 5-11
- bc_enable_speculative_execution 8-4
- bc_use_registerfiles 7-6
- environmental
 - defining A-2
- lifetime analysis 4-6
- link_library 2-5, A-3
- search_path 2-5, A-3
- setup A-3
- .synopsys_dc.setup A-3
- synthetic_library 2-5, A-3
- target_library 2-5, A-3
- vendor
 - technology library 1-13
- verification, comparison of scheduling modes 4-16

W

- while loops 4-41
- wrapper
 - exploratory memory 7-67
 - vendor memory 7-39
- write
 - gate-level netlist for simulation 2-24, 2-25, 2-26
 - gate-level netlist for synthesis 2-24, 2-26
 - HDL file 2-20, 2-21
 - RTL .db file 2-19
 - timed .db file 2-12, A-10
 - timed design 2-12
- write_lib command 7-66
- write_rtl command 2-20