Synphony C Compiler Reference Manual

Version K-2015.06, June 2015

SYNOPSYS®

Copyright Notice and Proprietary Information

© 2015 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.

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.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at http://www.synopsys.com/Company/Pages/Trademarks.aspx.
All other product or company names may be trademarks of their respective owners.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Contents

About This Manual xx

Customer Support. xxii

1. Introduction

About Synphony C Compiler. 1-2
Synphony C Compiler Products. 1-3

Getting Started 1-4
Starting the Tool. 1-4
Getting Help. 1-6
Technical Support 1-6

Related Synopsys Tools 1-7
Design Compiler Tool for ASIC Synthesis 1-7
Synplify Products for FPGA Synthesis. 1-7
VCS Tool for Simulation. 1-7

2. The Graphical User Interface

Understanding the Graphical User Interface 2-2
Menus 2-4
The Tool Windows 2-9
Analysis Views. 2-17

3. Synphony C Compiler Commands

About the Command Line Interface 3-2
Command-Line Entry. 3-2

Tool Startup Commands 3-2
Commands by Type 3-4
Alphabetical List of Commands 3-5

Command Descriptions 3-6
analyze 3-6
build 3-6
close_project 3-7
delete_implementation. 3-7
delete_project 3-8
estimate_power 3-8
export_implementation 3-9
get_defaults 3-10
get_implementations 3-11
get_params 3-11
help 3-11
link_implementation 3-12
lock_implementation 3-12
new_implementation 3-13
new_project 3-13
open_project 3-14
quit 3-14
report_implementation 3-14
reset_defaults 3-17
reset_rtl_package 3-18
rtl_synthesize (ASIC) 3-18
rtl_synthesize (FPGA) 3-19
select_implementation 3-20
set_params 3-20
show 3-28
unlock_implementation 3-29
verify 3-30
view_waveform 3-30

Additional SCC Options 3-31
Schedule Auxiliary Option 3-31
Synthesize Auxiliary Options 3-31

Synphony C Compiler EV Commands 3-38
Synphony C Compiler EV Launch Command 3-39
Synphony C Compiler EV Supported Commands 3-39

4. Projects and Implementations

Projects and Implementations 4-2
Projects 4-2
Implementations 4-3

Directory Structure 4-4
Structure of a Project Directory 4-4
Structure of an Implementation Directory 4-5

Project and Implementation Commands 4-6
Project Setup Commands 4-7
Example of Design Command Sequence 4-7

5. Programming Model Basics

Overview of the Programming Model 5-3

Synthesizable Code and Testbench 5-3

Task Processing in PPAs 5-4
Shared Resource Handling 5-5
PPA Hardware 5-6
Task Overlap Scenarios 5-8
Handshake Event Types 5-8
Continuous Processing 5-8
Low-Level Control Using synthesize_auxopts 5-9

Loop Nests and Processing Arrays 5-11

Compiler Transformations 5-13
Perfectization 5-16
Rate Matching 5-25

Performance Metric 5-26
Effect of II on Generated Hardware 5-29
Factors Affecting II 5-31
Specified Performance Versus Delivered Performance 5-32
Common Performance Bottlenecks 5-32

Feedback Paths 5-33

Data Structures and Hardware Mapping 5-34
Scalar Variables 5-34
Arrays 5-35

Communicating with the External Environment 5-39
Memory Mapping 5-41
Loop Dependence Rules Across Tasks 5-41

Communication Between PA Modules 5-43
Loop Sequentialization Rules Within a Task 5-44
Streaming Data 5-46

Hardware Sharing 5-48
Sharing Across Cycles in an II>1 Design 5-50
Sharing of Mutually-Exclusive Operators 5-51
Hardware Sharing Coding Strategy 5-52
Sharing Larger Blocks 5-53

Parallelism 5-55

Task Overlaps 5-57
Resource Allocation for Task Overlap 5-57
Maximum Achievable Task Overlap 5-58
Invariant Properties Under Task Overlap 5-58
Hardware-Based Task Overlap Scenarios 5-60

Implications of Coding Styles and Syntax 5-62
Variable Scoping 5-62
Registers and Wires 5-62
Input Code and the Generated Hardware 5-64
Variable Trip Count Loops and Early Exits 5-65
Perfectization Choices 5-66
Trade-off Between Cost and Performance 5-68
Affine Expressions of Loop Indexes 5-68
Specifying Bounds on Loop Iterations 5-69
Loop Flattening 5-70

Good Hardware Techniques for the Compiler 5-71
Avoiding Natural Termination of Loop in the Presence of Breaks 5-71
Exposing Loop Exit Conditions to the Compiler for Timing 5-71

Outer Loops for Getting the Desired Results 5-72
Loop Merging 5-75
Reducing II 5-77
Concatenating Memories 5-79
Shifters 5-79
Scalarization Versus Internal Fast Memory 5-80

6. C/C++ Synthesizable Syntax

Synthesizable Code and Testbench Code 6-2

Type Syntax 6-3
File Syntax 6-3
Base Type 6-4
Modifiers 6-5
Enum and Typedef 6-5
Parameterized Integer Data Types 6-6
Arrays. 6-9
Pointers 6-10
Static Variables and State Encapsulation 6-11
Class and Structures 6-12
Constructors and Destructors 6-13
Unsupported C++ Syntax 6-15

Procedure Syntax 6-16
Procedure Parameters and Return Values 6-16

Statement and Expression Syntax 6-17
Initializations in Declarations 6-19

Testbench Code 6-21
Testbench Code Examples 6-22

ANSI Support 6-23
Unsupported ANSI C Elements 6-24
ANSI C Elements Supported with Specific Semantics 6-25
Implementation-Defined ANSI C Elements 6-25

7. Pragmas and Constraints

User-Specified Constraints 7-2
Tool Interface Constraints 7-2

Code Pragmas 7-2
$SCC Labeling. 7-3

Tcl Constraint Commands 7-6
Cycle-Level Scheduling 7-10

Overview of Pragmas 7-12
Pragma Specification. 7-13
Pragmas by Function. 7-14

Pragma Descriptions 7-18
bitsize Pragma 7-20
boundary_register and Other Boundary Register Pragmas 7-20
Boundary Register Specification for Streams 7-22
Boundary Register Specification for Memories 7-23
bus_name Pragma. 7-24
bus_standard Pragma 7-24
bus_type Pragma. 7-25
stable Pragma 7-25
data_name, ready_name, request_name Pragmas. 7-26
dont_care_code Pragma 7-27
fifo_length Pragma. 7-27
flush Pragma 7-28
force_constant_addressing Pragma 7-28
forward_path_external_delay and Other External Delay Pragmas 7-29
hold_register Pragma 7-29
host_memory_access Pragma 7-30
host_scalar_access Pragma 7-31
ii Pragma 7-31
internal_blockram Pragma. 7-32
internal_fast Pragma 7-33
localize_access Pragma 7-33
max_schedule_length Pragma 7-35
memory_structure Pragma 7-36
memory_access_latency Pragma 7-36
multi_buffer Pragma. 7-37
no_dependence Pragma 7-37
no_inter_loop_memory_analysis Pragma 7-41

no_inter_loop_stream_analysis Pragma 7-42
no_inter_task_memory_analysis Pragma 7-45
no_inter_task_stream_analysis Pragma 7-46
no_inter_tcab_subtask_memory_analysis Pragma 7-48
not_input, not_output Pragmas 7-49
num_iterations Pragma 7-50
packed Pragma 7-53
preserve_array Pragma 7-57
read_write_ports Pragma 7-58
rom Pragma 7-60
signal_name_prefix, signal_name_identifier, and signal_name_suffix Pragmas 7-60
signal_polarity Pragma 7-67
signal_production Pragma 7-67
scalarize Pragma 7-68
unroll Pragma 7-70
user_supplied, user_supplied_fpga Pragmas 7-72

8. Synthesizable Code

Non-Standard Variable Widths 8-2
bitsize Pragma Examples 8-2
Bitsize for Data Types 8-3
Struct Bitsize 8-3

Memory Support 8-6
Array Mapping 8-6
Local Memory 8-9
ROMs. 8-10
Packed and Unpacked Memories 8-11
Memory Arbitration 8-14
Banking and Multi-Buffering of Memories 8-15
Scalarization 8-21

Stream Support. 8-24
Class-Based Stream Definition 8-25
Function-Based Stream Definition 8-29
External Streams 8-30
Internal Streams 8-31
Passthru and Non-Passthru FIFOs 8-31

User Controls for Streaming Interfaces 8-32

Mutual Exclusion. 8-35
Parallel Hardware 8-35
Switch Case Syntax. 8-36
Full-Case Syntax 8-38
The scc_assert(0) Construct 8-40

Pipeline Flushing 8-40

Multi-Level Break Statements 8-43
Syntax and Semantics 8-43
Outer Loop Controller 8-43
Custom Opcodes 8-44

Pre-Defined Custom Operations. 8-45
Arithmetic Operations 8-46
Pipelined Arithmetic. 8-50
Single-Cycle Delay. 8-50

9. Fixed-Point Library

Fixed-Point Library Support 9-2

Quantization and Overflow Modes 9-3

Fixed-Point Arithmetic Library Usage 9-7
Examples 9-7

Fixed-Point Division 9-9

Fixed-Point Library Classes 9-9
Arithmetic Operators 9-10
Relational Operators 9-11
Logical Operators 9-11
Bitwise Operators 9-11
TypeCast Operators. 9-12
Constructors in Fixed-Point Library 9-13
Operator Support in Non-Synthesizable Code 9-13

SystemC s_fixed / s_ufixed Conversion 9-13

Limitations of the Fixed-Point Library 9-14

10. Hierarchical Synthesis

Hierarchical Block Overview 10-3

Hierarchical Block Operations. 10-4
- Building Hierarchical Blocks. 10-4
- Importing Hierarchical Blocks 10-4
- Verifying Hierarchical Blocks 10-5

Specifying Constraints for Hierarchical Blocks 10-5

Sharing Hierarchical Blocks 10-6

Importing Hierarchical Blocks 10-9

Physical Instantiation of Hierarchical Blocks. 10-11

Overlapped Execution of TCABs and IPPAs. 10-12

Specifying the Properties of TCAB and IPPA 10-13
- Details of TCAB Property 10-14
- Details of IPPA Property 10-16
- TCAB Pipelinability 10-16

Target Performance Specification for TCABs 10-18
- Architectural Pipelinability Specification 10-19
- Input Sample Time Specification 10-19
- Output Production Time Specification 10-20
- Resource First Use Time Specification 10-21
- Resource Last Use Time Specification 10-21
- Example 10-22

Target Performance Specification for IPPAs 10-23

TCAB Input/Output Controls 10-24

Uses of TCABs 10-26
- Enhanced Datapath Sharing 10-26
- Reconfigurable Pipeline of Logic 10-26
- Performance-Based Logic Partitioning 10-27
- Power Savings 10-27
- Hide Complexity of Lower Blocks at Higher Levels 10-28
- Outer Loop Control 10-28

TCAB Sharing and Inlining 10-29

TCAB-Specific Parameters 10-31

Classifications of TCABs 10-32
Combinational and Sequential TCABs 10-32
Pipelined and Nonpipelined TCABs. 10-32
Shared Stall Domain and Independent Stall Domain TCABs 10-33

11. Channel Synthesis

Introduction 11-2

Predefined SCC Classes for Channel. 11-2
scc_channel_if 11-2
scc_port and scc_in 11-3
scc_instream_if and scc_outstream_if. 11-3
scc_stream_with_get_trigger and scc_stream_with_put_trigger. 11-4

Channel Examples 11-4
Example: Basic Channel With PPA 11-5
Example: Class With Channel Member as Parameter to PPA/TCAB 11-7

Channel Simulation. 11-9

Channel Build Flow 11-10
Building the Channel 11-10
Importing the Channel Into a PPA 11-10
Example: run.tcl File 11-11

Implementation Parameter Support 11-12

Limitations. 11-13

12. Synphony C Compiler Linting

Overview of Linting 12-2

Compile-Time Syntax Checking 12-2

Simulation-Based Semantic Checking 12-3
Bitsize Pragma Checks 12-3
Uninitialized Variable Checks. 12-5
Out-of-Bounds Array Access Checks 12-6
Array Usage Distance Specification. 12-8

13. Analysis Views and Reports

Overview of Generated Reports 13-2
- Alphabetical List of Reports 13-2
- Organization of the Reports 13-3

Design Summary Reports 13-4
- report_constraints.txt 13-4
- report_summary.txt 13-6

User Code Reports 13-19
- report_original_loop_to_line_number.txt 13-19
- report_perfectized_loop_info.txt 13-19
- report_rate_match.txt 13-20
- PPA Loop Graph, Sequentialization Loop Graph 13-22
- PPA FIFO Report 13-25
- Timing Analysis Reports 13-26

Internal Timing Report 13-27
- report_feedback_path.txt 13-28
- report_pseudo_code.txt 13-37
- report_false_paths.txt 13-40
- Example of a Successful Schedule 13-40
- Example of a Failing Schedule 13-43

Integration Reports 13-47
- report_address_map.txt 13-47
- report_interface.txt 13-48
- report_interface_timing.txt 13-55

QoR Analysis Reports 13-56
- report_cost.txt 13-57
- report_register.txt 13-61
- report_hierarchy.pdf and report_hierarchy_detailed.pdf 13-64

Performance Analysis Reports 13-67
- Dynamic Performance Graph 13-67
- report_performance.txt 13-67
- report_verify_rtl_performance.txt 13-73
- Support for Performance Charting 13-74

C to RTL Mapping 13-74

Power Estimation Reports 13-76
report_power_theater_offline.txt 13-76
report_power_theater_sv.txt 13-76
report_spyglass_offline.txt 13-76
Verilog Simulation Analysis Reports 13-78
report_memory_access_trace.txt 13-78

14. System Integration and Verification

Overview of the Hardware Interface 14-3
System Interface 14-4
Handshake Interface 14-5
Scalar Interface 14-8
Stream Interfaces 14-9
Input Stream Interface 14-10
Output Stream Interface 14-10
Stream Interface Customization 14-11
Local Memory Interfaces 14-12
Direct Port Interface to Memories in PPA 14-12
Memory-Mapped PPA Components 14-14
Specification to Memory-Map a PPA Component 14-14
Address Map Input 14-15
Address Map File 14-15
Address Map Output 14-16
Address Restrictions 14-16
Holes in the Address Map 14-17
Host Interface Bus 14-17
SCC Native Host Interface 14-18
OCP Interface 14-19
Memory-Mapped PPA Control (PSW) 14-19
Persistent start and done 14-21
Memory-Mapped Scalar Interface 14-22
PPA Interface Port Naming and Polarity 14-24

Interface Timing 14-25

Deployment of Memory-Mapped PPA. 14-26
API for Memory-Mapped PPA Deployment 14-26
Procedure for Memory-Mapped PPA Deployment. 14-27

Pragmas for EDK support (FPGA Only) 14-29
Bus Name 14-30
Bus Standard. 14-30
Bus type. 14-31
Naming the pcore Ports of Synphony C Compiler stream 14-31

TCAB Integration 14-34
TCAB Interface and Integration Guidelines 14-34
Customization of TCAB Interface Ports 14-37

Verification 14-38
C-Simulation 14-38
Post-Netlist-Synthesis Simulation Support 14-41

SystemVerilog (SV) Universal Verification Methodology. 14-42
Online Documentation for Testbench. 14-42
Simulation Reporting 14-43
Application Coding Guidelines. 14-43
SV Simulation Features 14-44
Limitations 14-44
Compiler and Execution Options of verify -rtl Command. 14-44
Interfacing to SCC SystemC Transaction Level Models 14-45
Comparison Strategy Overview. 14-46
SystemVerilog Verification Guidelines 14-46
Streamproc and Driver Creation Guidelines 14-47
Other SystemVerilog Guidelines 14-47

15. Synphony C Compiler Output

Overview of Synphony C Compiler Output 15-2

Hierarchy of Generated RTL. 15-3

Package Directory Structure. 15-4

Directory Structure to Support SV Simulation 15-7

Corner Case Simulation Support 15-8
Assertions 15-9
Monitors and Checkers 15-10
RTL Features 15-10
Overall Hardware Strategy 15-11
Registers 15-11
Clocks 15-11
Resets 15-12
RTL Timing Closure 15-12
Timing Closure and False Timing Paths 15-13
Synthesis Assumptions 15-13
Design-for-Test 15-13
Abort 15-14
Error Signaling 15-15
Autostart 15-16
Infinite Run 15-17
Localization 15-17
FIFO Hierarchy Priority 15-18
Feature Interactions 15-18
Clock Gating (ASIC Only) 15-19
Memory Arbitration 15-21
FIFO Implementation (FPGA Only) 15-23
STARC Rule Support 15-23

16. Hardware Architecture

Introduction 16-2
Pipeline of Processing Arrays 16-2
Tightly Coupled Accelerator Block 16-5
Processing Array (PA) 16-5
Core Processor 16-6
Interconnect 16-7
Selector Wiring at the Input of the Functional Unit 16-8
Registers 16-9

Merge Registers 16-9
Predicate Merger 16-10

17. Scripts and Utilities

Introduction 17-2
Script for Debussy and nLint Tools 17-2
Script for Verplex Conformal LEC Tool 17-3
Abort Points 17-3
Script for Synopsys Virsim Tool 17-4

18. ASIC Macrocell Characterization

Overview of Synphony C Compiler Novo 18-2
The Tool Flow 18-2
Novo Macrocell Characterization 18-2
Novo Command Syntax 18-3
Characterization Configuration File 18-4
Sample Configuration File 18-4
Configuration File Parameters 18-5
Novo Input and Output Files 18-9
Input Files 18-9
Output Files 18-9

19. DSP Mapping for FPGAs

Overview of DSP Mapping 19-2
Why Use the DSP Mapping API? 19-2
What Does the DSP Mapping API Do? 19-3
DSP Mapping Usage Overview 19-4
Verilog Simulation 19-5
Limitations 19-5
Recommendations 19-6
DSP Resource Sharing 19-6
Altera DSP Mapping 19-8
Static Control 19-9

Dynamic Control 19-12
Synphony C Compiler Class for Altera DSP 19-13
Using DSP Mapping on Altera 19-15
Altera Coding Examples 19-16
Xilinx DSP Mapping 19-19
Static Control 19-20
Dynamic Control 19-21
Synphony C Compiler Class for Xilinx DSP 19-22
Using DSP Mapping on Xilinx 19-23
Xilinx Coding Examples 19-24
APIs for Commonly Used Functions 19-26

Index

Preface

This preface includes the following sections:

- About This Manual
- Customer Support

About This Manual

The Synphony C Compiler tool generates RTL design and testbench descriptions from high-level C/C++ models, thereby providing an RTL synthesis solution for algorithm-based designs. This book provides detailed reference information for Synphony C Compiler users.

Audience

This reference manual is for design engineers and C/C++ programmers who use Synphony C Compiler to implement designs. Readers should have prior experience with programming and RTL synthesis.

Related Publications

For additional information about the Synphony C Compiler tool, see the documentation on the Synopsys SolvNet® online support site at the following address:

https://solvnet.synopsys.com/DocsOnWeb

You might also want to see the documentation for the following related Synopsys products:

- Design Compiler®
- Synplify®
- VCS ®

Release Notes

Information about new features, enhancements, changes, known limitations, and resolved Synopsys Technical Action Requests (STARs) is available in the *Synphony C Compiler Release Notes* on the SolvNet site.

To see the *Synphony C Compiler Release Notes*,

1. Go to the SolvNet Download Center located at the following address:

   https://solvnet.synopsys.com/DownloadCenter

2. Select Synphony C Compiler, and then select a release in the list that appears.

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates syntax, such as `write_file`.
`Courier italic`	Indicates a user-defined value in syntax, such as `write_file` *`design_list`*.
`Courier bold`	Indicates user input—text you type verbatim—in examples, such as `prompt>` **`write_file top`**
[ ]	Denotes optional arguments in syntax, such as `write_file [-format` *`fmt`*`]`
...	Indicates that arguments can be repeated as many times as needed, such as *`pin1 pin2 ... pinN`*
|	Indicates a choice among alternatives, such as `low | medium | high`
Ctrl+C	Indicates a keyboard combination, such as holding down the Ctrl 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 SolvNet online customer support and through contacting the Synopsys Technical Support Center.

Accessing SolvNet

The SolvNet site includes a knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. The SolvNet site also gives you access to a wide range of Synopsys online services including software downloads, documentation, and technical support.

To access the SolvNet site, go to the following address:

https://solvnet.synopsys.com

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to sign up for an account.

If you need help using the SolvNet site, click HELP in the top-right menu bar.

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 support case to your local support center online by signing in to the SolvNet site at https://solvnet.synopsys.com, clicking Support, and then clicking “Open A Support Case.”
- Send an e-mail message to your local support center.
  - E-mail support_center@synopsys.com from within North America.
  - Find other local support center e-mail addresses at

    http://www.synopsys.com/Support/GlobalSupportCenters/Pages
- Telephone your local support center.
  - Call (800) 245-8005 from within North America.
  - Find other local support center telephone numbers at

    http://www.synopsys.com/Support/GlobalSupportCenters/Pages

1

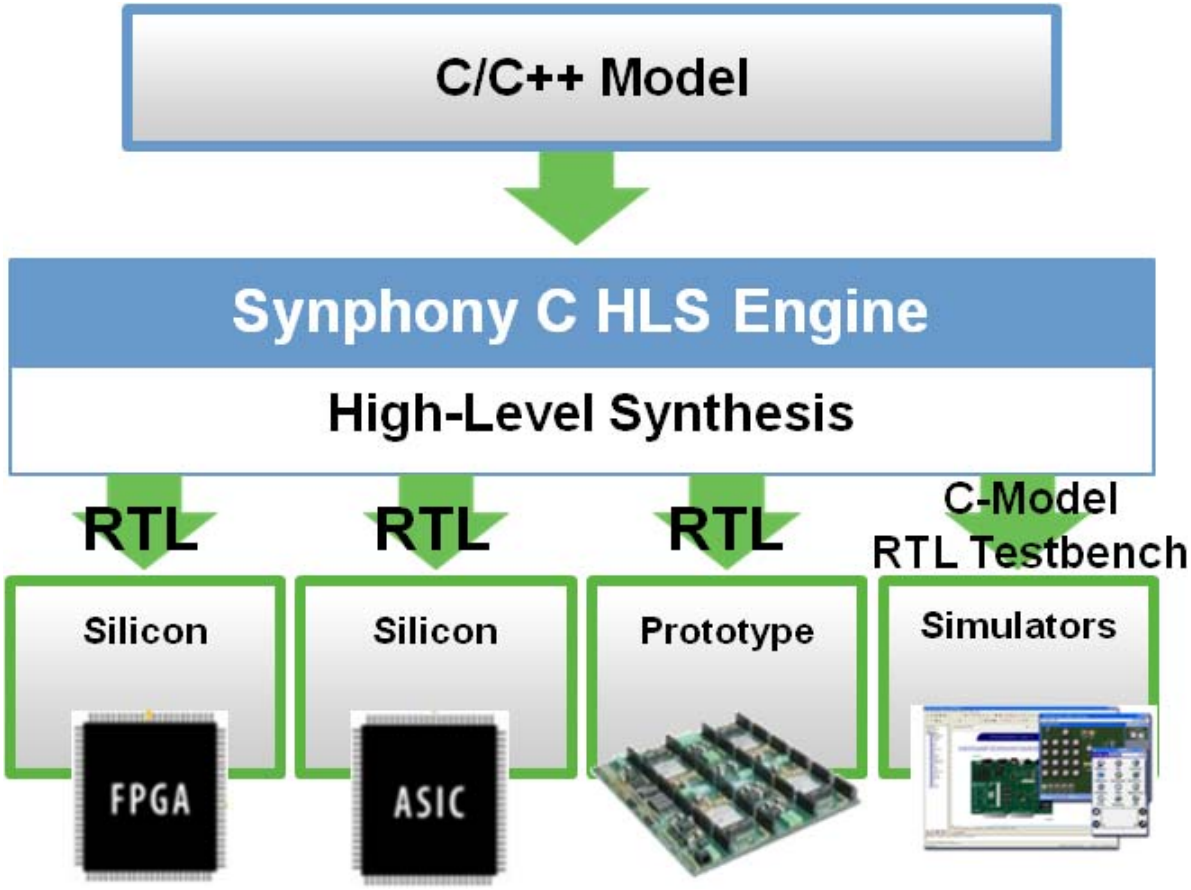
Introduction

This chapter introduces you to the Synphony C Compiler (SCC) tool:

- About Synphony C Compiler
- Getting Started
- Related Synopsys Tools

About Synphony C Compiler

The Synopsys® Synphony C Compiler (SCC) tool helps fulfill the demand for embedded digital systems for special purposes, by providing functionality for rapid, accurate, and efficient system design development. As the following figure shows, the Synphony C Compiler tool provides an algorithm-to-tapeout synthesis solution for algorithm-based designs. It lets you start with high-level C/C++ models and generate RTL that is optimized for either ASIC or FPGA targets.



The SCC tool creates a customized design while reducing the traditional obstacles of design time, designer availability, and design cost. This tool offers the following advantages:

- Automated design flows that start with higher-level C/C++

  The Synphony C Compiler technology redefines system-on-chip (SoC) design by shifting the design focus from designing at the RTL level to designing at the algorithmic level. It

saves development time and reduces costs by creating hardware from the algorithms that implement the reference standard. The SCC software supports both C and C++ for synthesizable code.

- Rapid architecture exploration

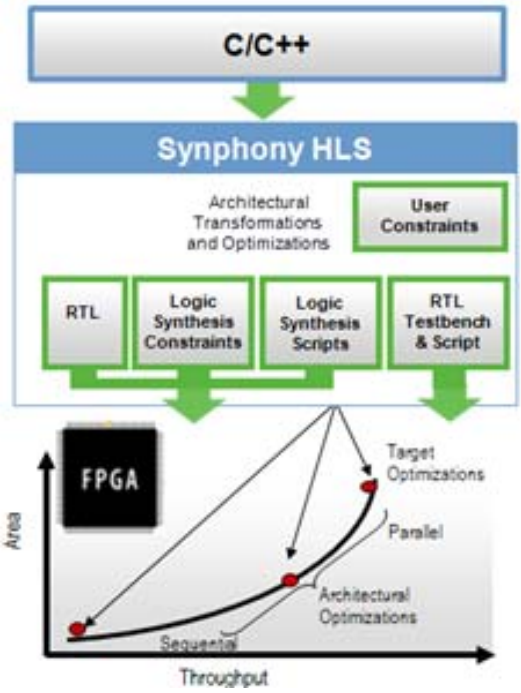
  The design space exploration and configuration tools like the Architecture Analyzer automatically generate a series of pareto-optimal design alternatives. Having configurable IP ensures that the resulting RTL is compatible with existing RTL-to-netlist tools for ASICs and FPGAs. This enables fast and efficient timing closure and physical design. The SCC product is essential for designers who have tight deadlines and stringent performance or area targets. It allows designers to explore alternatives and find the best implementation to reuse IP.

- Unified verification with reduced effort and lower risk

  The SCC tool lets designers improve quality with its right-by-construction synthesis and architectural schema that reuses preverified components. It includes comprehensive reporting, analysis, and verification tools that simplify the verification process.

Synphony C Compiler Products

There are two versions of the tool, which are optimized for either ASIC or FPGA development.



Unless specifically stated otherwise, the terms *Synphony C Compiler* and *SCC* refer to all variations of the tool. Similarly, features and coding styles described in this document apply to all the products unless explicitly stated otherwise.

Synphony C Compiler ASIC

The ASIC version of the tool generates RTL that is optimized for ASICs. This version includes some features that are not available in the FPGA version, like clock gating and power estimation. The defaults for certain parameters are optimized for ASIC designs.

Synphony C Compiler FPGA

The FPGA version of the tool is optimized for FPGA targets. It takes C input and generates device-specific RTL for Altera or Xilinx FPGAs. You can get multi-vendor, Altera-specific, or Xilinx-specific versions. The default settings for this tool are optimized for FPGA designs.

For a complete list of supported devices and speed grades in each family, open the FPGA version of the Synphony C Compiler GUI and choose Implementation > Configure New or Configure Current. In the dialog box, under HW Settings, use the pull-down menus to select the desired Family, Part, Package, and Speed Grade from the available choices.

You can also find a list of the currently supported FPGA devices in the following text file in the Synphony C Compiler FPGA product installation:

```
${SCC_HOME}/scc/picofe/bin/supported_fpgas.txt
```

Getting Started

See the following sections for information about getting started with the tool:

- Starting the Tool
- Getting Help
- Technical Support

Starting the Tool

The following summarizes how to start the tool.

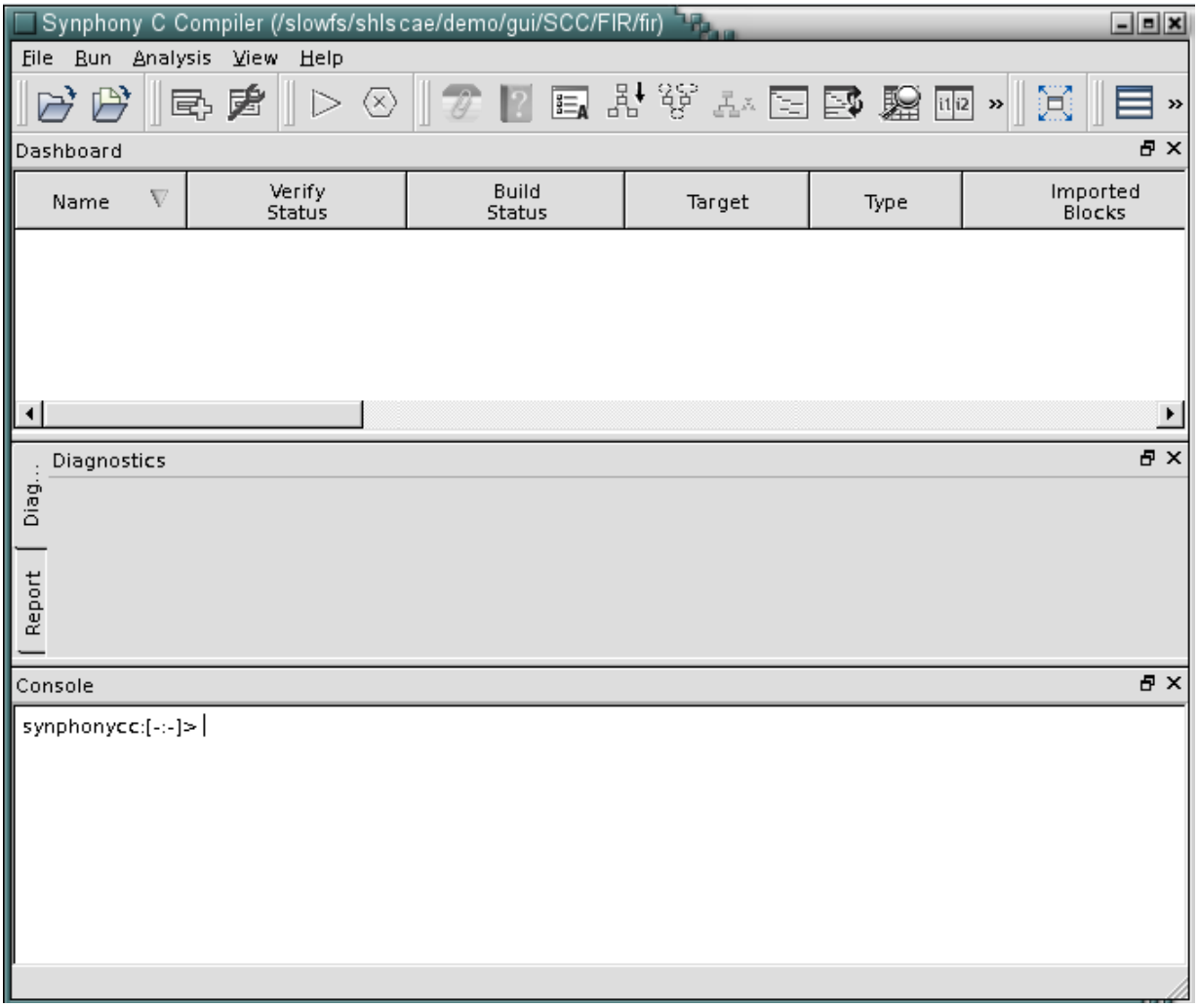
1. Install the Synphony C Compiler tool according to the installation instructions.

2. Launch the GUI from a UNIX platform, using the appropriate command for your software:

`synphonycc`	Starts the SCC ASIC tool.
`synphonycc_fpga`	Starts the SCC FPGA tool.
`synphonycc_fpga -xilinx`	Starts the SCC FPGA tool with the Xilinx-specific license.

The command starts the tool and opens the main GUI main window. If you have previously run the tool, the window corresponding to the previous project is displayed.

The following figure shows the main GUI window when you first start the tool.



Getting Help

Before you call Synopsys Support, look through the documented information.

- Access the online documentation from the Help menu. This table shows where to locate different kinds of information:

For help with...	Refer to the...
Using the tool	*Synphony C Compiler User Guide*
How to...	*Synphony C Compiler User Guide*
Flow information	*Synphony C Compiler User Guide*
Working with other tools	Application Notes
Licensing	Synopsys SolvNet website
GUI and feature descriptions	*Synphony C Compiler Reference Manual*
Language and syntax	*Synphony C Compiler Reference Manual*
Command descriptions	*Synphony C Compiler Reference Manual*

- Use the command line to access help for commands:
  - To get a list of available commands, type `help` at the command line.
  - To display help for a command, type `<`*command_name*`> -help` at the command line. The tool prints details about the command.
- Refer to the product information available on Solvnet.

Technical Support

To request assistance from Technical Support, use the SolvNet Online Support utility. This online web-based interface is the preferred mechanism for contacting Technical Support and may facilitate a faster response than requesting support through email. You can access SolvNet Online Support from the link on this web page: `http://www.solvnet.com`

Related Synopsys Tools

The following describe some recommended Synopsys tools that you can use in conjunction with the Synphony C Compiler tool to take your C code to a hardware implementation. For information about the supported and recommended tool versions, see the Synphony C Compiler Release Notes, available on SolvNet under “Downloads.”

Design Compiler Tool for ASIC Synthesis

The Synopsys Design Compiler® tool offers a suite of RTL synthesis and test solutions for ASIC designs. It provides guidance to the back-end place-and-route tools. Use the RTL generated by the Synphony C Compiler tool as input to the Design Compiler tool.

Synplify Products for FPGA Synthesis

The Synopsys Synplify® suite (Synplify, Synplify Pro®, and Synplify Premier) consists of logic and physical synthesis tools to synthesize FPGA designs. The Synplify and Synplify Pro tools are logic synthesis tools and the Synplify Premier tool includes physical synthesis as well as logic synthesis. All these tools let you target a specific FPGA technology and optimize your design for it. They also provide guidance to the back-end place-and-route tools.

VCS Tool for Simulation

Use the Synopsys VCS® tool to simulate the Synphony C Compiler RTL output. The VCS tool lets you verify and debug your design early in the design cycle. It uses multicore technology to run design, testbench, assertions, coverage, and debug in parallel. Its powerful debug and visualization environment lessens the turnaround time to find and fix bugs.

2

The Graphical User Interface

The graphical user interface (GUI) for the Synphony C Compiler tool includes commands that let you interact with the tool. It also includes analysis views that help you understand the QoR, timing, and performance aspects of your design. This chapter describes the GUI and its features:

- Understanding the Graphical User Interface

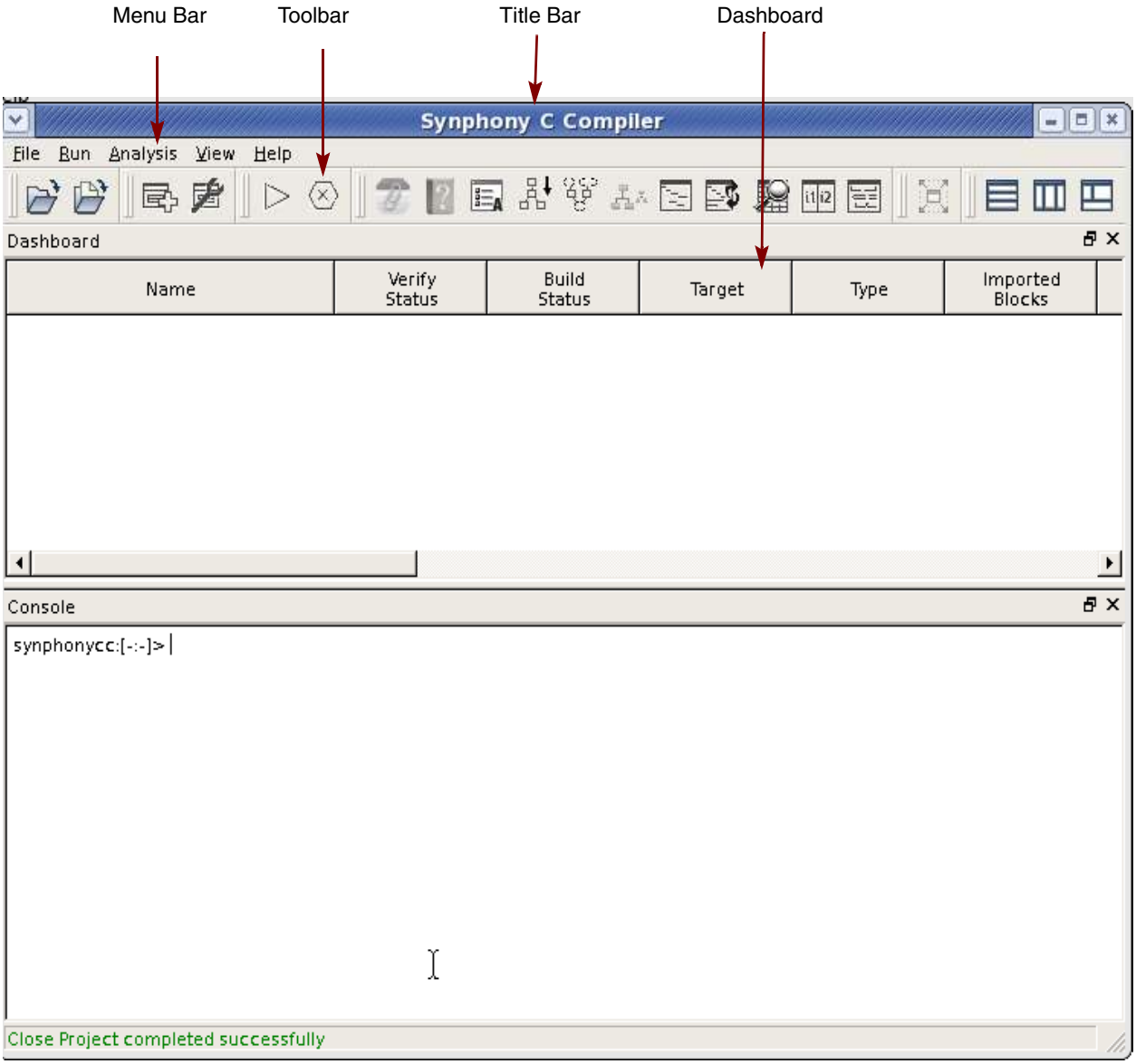
Understanding the Graphical User Interface

The Synphony C Compiler software provides two ways for you to interact and explore the implementations in the tool.

- Graphical User Interface (GUI)
- Command Line Interface

This chapter describes the GUI; see Chapter 3, “Synphony C Compiler Commands for a description of the command line interface.

The following figure shows the Synphony C Compiler GUI when opened for the first time.



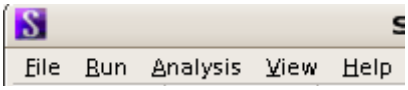
The GUI consists of the following:

Item	Description
Title bar	Displays the full path of the project directory.
Menu bar	Provides access to the tool commands. See “Menus” on page 2-4 for details.

Item	Description
Toolbar	Displays various icons that are shortcuts to commands.
Dashboard	Displays the summary properties of all implementations in one place.
Console Window	Lets you manually type in Synphony C Compiler commands. This window also shows the output of the commands from all modes of command execution.
The Status bar	Displays the status of the last executed command run. A successful completion is indicated in green, and a failing command result is shown in red.

Menus

The GUI groups the different commands into pull-down menus, as shown:



The individual menus, their commands, and the associated dialog boxes are described in the following sections:

- File Menu
- Run Menu
- Analysis Menu
- View Menu
- Help Menu

File Menu

Use the File menu to open a project or a file and to save the project. The File menu commands are:

Command	Description
New Project	Creates a new project.

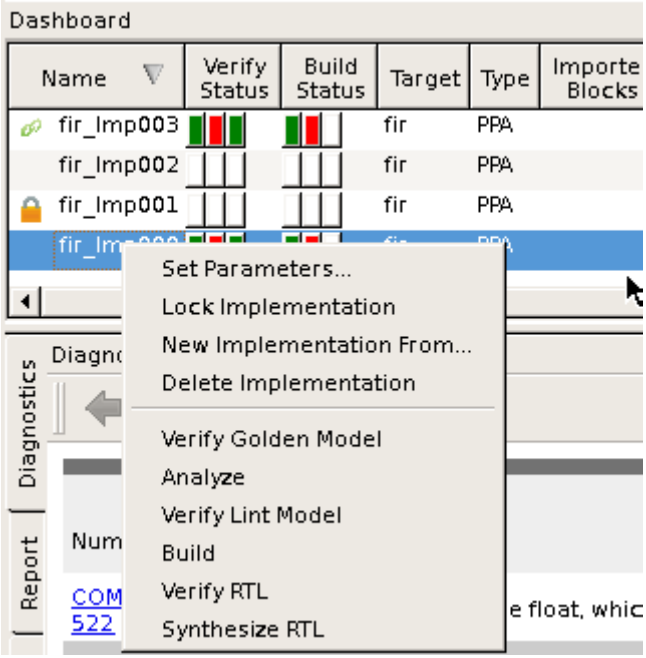
Command	Description
Open Project	Opens a project. The Analysis view opens when you open a project.
Close Project	Closes a project. When you close a project, the Analysis view gets hidden.
Delete Project	Deletes a project from the Dashboard window.
Refresh Projects	Refreshes the Dashboard window, and any opened project for the active implementation.
Recent Projects	Displays a list of recently opened projects.
New Implementation	Creates a new implementation.
New Implementation From...	Creates a new implementation with inherited parameters from a source implementation
Link Implementation	Links an existing implementation into the current project.
Lock Implementation	Locks an implementation.
Export Implementation	Exports an implementation. It exports an archive (source, header, and Tcl files) of the selected implementation.
Delete Implementation	Deletes an implementation.
Set Parameters	Opens a separate window where you can specify the parameters for your implementation. You can also override the settings specified in the Set Default Parameters for an implementation.
Set Default Parameters	Sets the default parameters for all the implementations in the project. Additionally, if you do not specify any parameters in Set Parameters window, it takes the default settings for all the implementations in the project.
Open File	Opens a file browser from which you select a file.
Exit	Exits the Synphony C Compiler tool.

Run Menu

Use the Run menu to run individual commands and to analyze the outcome of each step. This table lists the Run menu commands. For details about using these commands, see the "Building and Verifying the Design" chapter in the *Synphony C Compiler User Guide*.

Commands	Description
Run	Runs the selected steps on the active implementation.
Stop	Stops the Run step.
Verify Golden Model	Performs C simulation.
Analyze	Compiles the files that you specify and generates intermediate files that are used by subsequent `build` and `verify -lint` commands.
Verify Lint Model	Performs dynamic linting and additional compliance checks for bitwidth ranges, array bounds, and uninitialized variables.
Build	Performs high-level program transformation such as function inlining and static checks, for accepted C Syntax. The command generates code for lint simulation.
Verify RTL	Performs loop optimization and loop scheduling on the high-level program.
Synthesize RTL	Synthesizes the RTL from the Synphony C Compiler tool to netlist using backend tools.
Reset RTL Package	Creates rtl_package folder by overwriting the previously created rtl_package folder.
Export Tcl Script	Creates Tcl script for the selected implementation which can be used to regenerate the selected implementation.
Run Tcl Script	Runs the specified Tcl script.

The Run steps, starting with the Verify Golden Model command to the Synthesize RTL command is one complete flow. You can also select individual steps to run. Alternatively, you can right-click the implementation name and choose to run an individual step.



Analysis Menu

Use the Analysis menu commands to analyze an implementation graphically based on features after the implementation is run. The Analyze menu commands:

Command	Description	Prerequisite
Report	Displays the report view	Completion of one or more flow steps
Diagnostics	Presents a table of error and warning messages in HTML format, organized by error type.	Completion of one or more flow steps
Code Tree View	Displays the code tree with function calls and loops.	Successful analyze
Sequentialization Graph	Helps you understand if the loop nests in your code execute simultaneously or sequentially with respect to each other.	Successful build
PPA Graph	Helps you understand the resource usage in your design.	Successful build

Command	Description	Prerequisite
Hierarchy Graph	Helps you understand the hierarchy blocks in your design and their instantiation.	Successful build
Operation Schedule Viewer	Helps you understand the scheduling of different operations in a loop nest.	Successful build
Feedback Path Viewer	Helps you understand the critical feedback paths present in each loop.	synthesize completion
Resource Browser	Provides cost breakdown to macrocells in each PA. Also contains information about the different phases in which the macrocells are used. Shows information about TCAB sharing across PAs.	Successful build
Relative Cost Viewer	Compares implementations.	Successful build
Dynamic Performance Graph	Shows the performance of the design in the SystemC thread, RTL simulation, and RTL simulation.	Performance simulations such as `verify -rtl`.
C-to-RTL Mapping	Shows the correspondence between the C and RTL code.	Successful Synthesize RTL
Send to Analysis View	Sends some parts of the views in the Report view window to the Analysis view window. The views that can be sent to the Analysis view window are the ones in the Analysis menu, namely Code Tree view, Sequentialization Graph, PPA Graph, Hierarchy Graph, Operation Schedule Viewer, Feedback Path Viewer, Resource Browser, and so on. In the Analysis view window, you can view the graphs from different implementations and compare.	Successful synthesize

View Menu

Use View menu commands to control the display of specific views and windows:

Command	Description
Dashboard	Toggles the display of the Dashboard window.
Diagnostics	Toggles the display of the Diagnostics window.
Report	Toggles the display of the Report window.
Analysis	Toggles the display of the Analysis window.
Console	Toggles the display of the Console window.
Document View	Toggles the display of file Document View window on the screen. When you open a file, the file opens in the Document View window.
Full Screen	Toggles the size of the Analysis view window between full screen and the normal size.
Tile Windows	Displays all the current windows in tiled manner in the Analysis view only.
Cascade Windows	Displays all the current windows in a cascaded manner in the Analysis view only.
Close All Windows	Closes all the current windows in the Analysis view only.
Preferences...	Use this to configure the Style, Font, Dashboard, Toolbars, Layout, Doc Viewer settings for your project,

Help Menu

Use the Help menu to view the online documentation in PDF format. The About command displays information about the current Synphony C Compiler version and the copyright.

The Tool Windows

The main tool window is made up of subsidiary windows, which are described here:

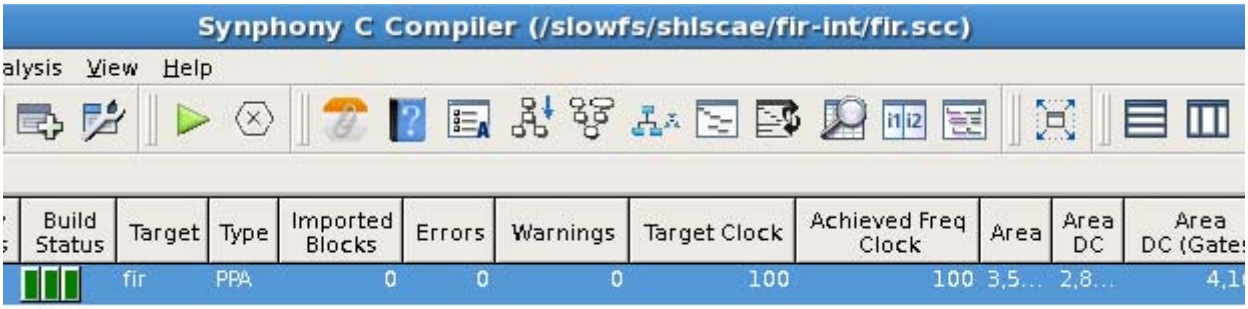
- The Dashboard Window
- The Console Window

- The Document View Window

For information about the analysis view windows, see “Analysis Views” on page 2-17.

The Dashboard Window

The Dashboard window displays the summary of the properties of all implementations in one place. The window is automatically updated when the GUI refreshes after a command completes. This is an example of this window:

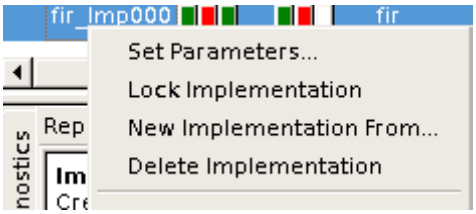


The window shows various implementation properties, which you can use for analysis. Note that the Achieved Clock Frequency differs from the Input Clock Frequency value if the tool cannot schedule operations at the desired input frequency.

In the Dashboard window, you can see six widgets representing the status of an implementation: three widgets for Build status, and three widgets for Verify status. You can also know the status of an implementation with the color coding of the individual widgets. The color coding is represented by grey, red, green, orange, and white, with each color indicative of a specific status.

Color Code	Implementation Status
Red	Failed / Hard_constraint_violation
Grey	Imported TCAB /Channel is not built or is rebuilt
Orange	Soft constraint violation
Green	Completed successfully
White	Not run

You can also see implementation specific options when you right-click at an implementation in the Dashboard window.

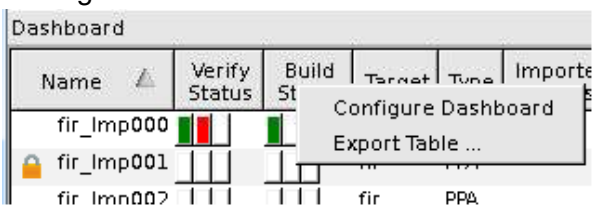


You can do the following operations in this window to manipulate the information and the design:

- You can determine what properties are displayed with the View command.
- By default, the Dashboard window shows parameter values for all existing implementations from the project, but you can change this behavior with the Configure Dashboard Window command.
- You can also use the Dashboard window to make an implementation active by double-clicking its name in the first column of the watch window.

Configuring the Dashboard Window

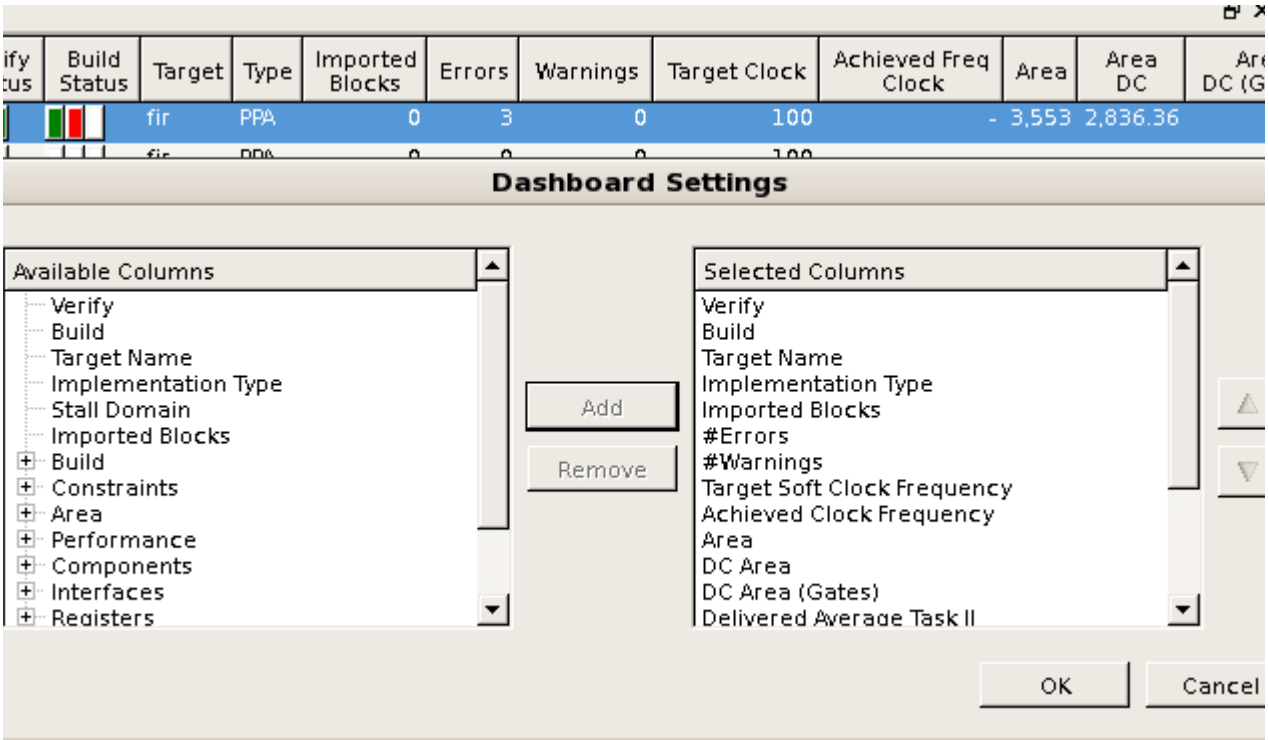
Right-click on the header in the Dashboard window, and select Configure Dashboard option.



When you select the Configure Dashboard option, the Dashboard Settings window opens. You can configure your dashboard by choosing any item listed in the Available Columns panel, then click Add. The selected item appears in the Selected Columns panel. Click OK to exit the configuration window. Use the up and down arrows on the right-hand side, to reorder the items according to which you want to view in the Dashboard window.

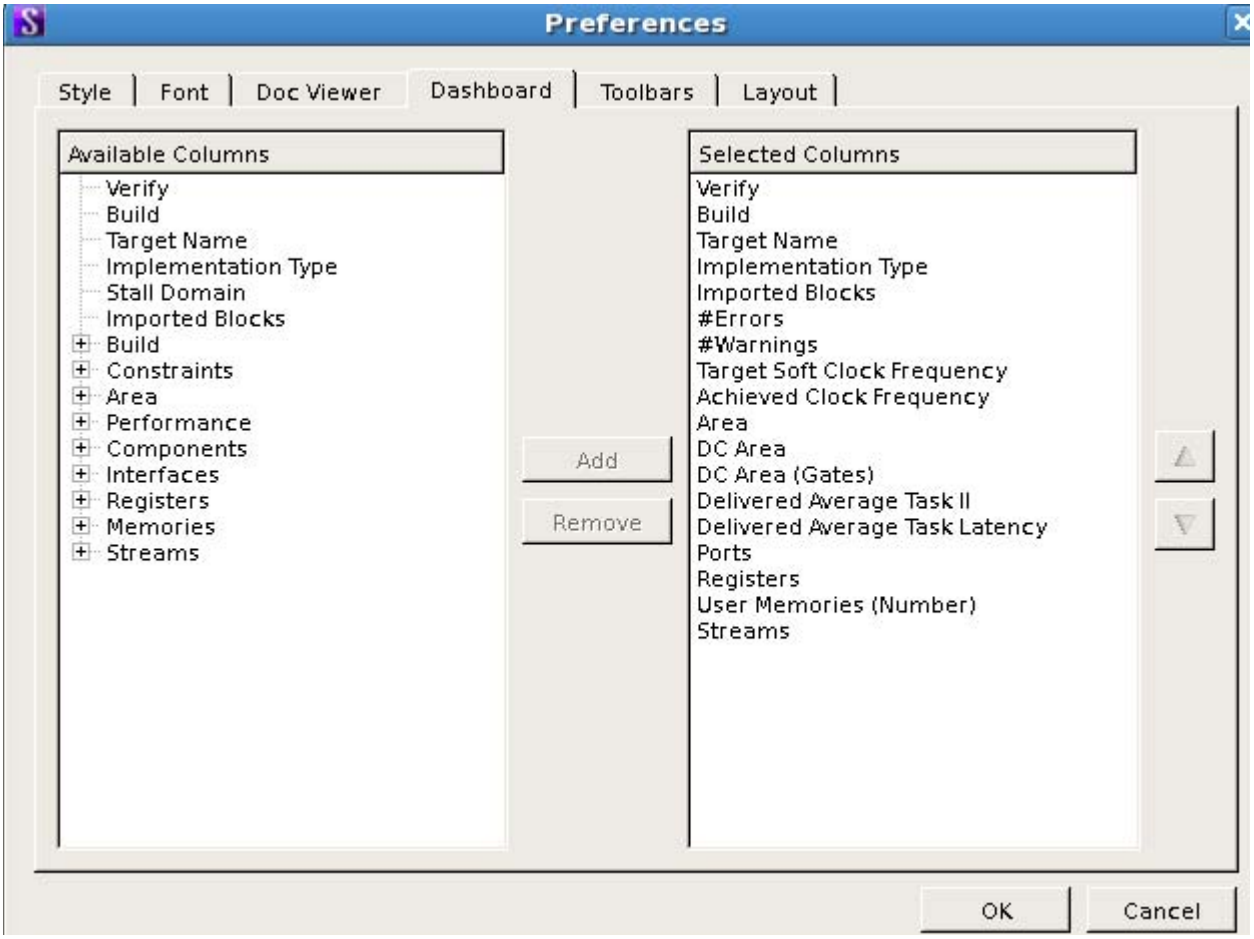
In the Dashboard, you can resize the column width. When you open a new session of the GUI, the column width setting is retained.

You can see that the Dashboard view contains columns as per the items you configured.



Alternatively, you can also select View > Preferences > Dashboard to configure your

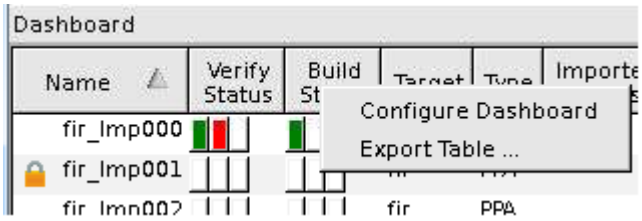
Dashboard settings.

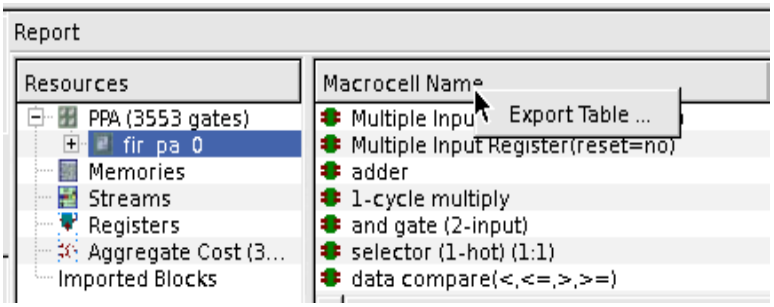


Exporting Implementation Data to CSV Format in GUI

When you right-click at the header of the Dashboard, you can see the Export Table option. Select this option to export the Dashboard contents into a file in the .csv format from the GUI.

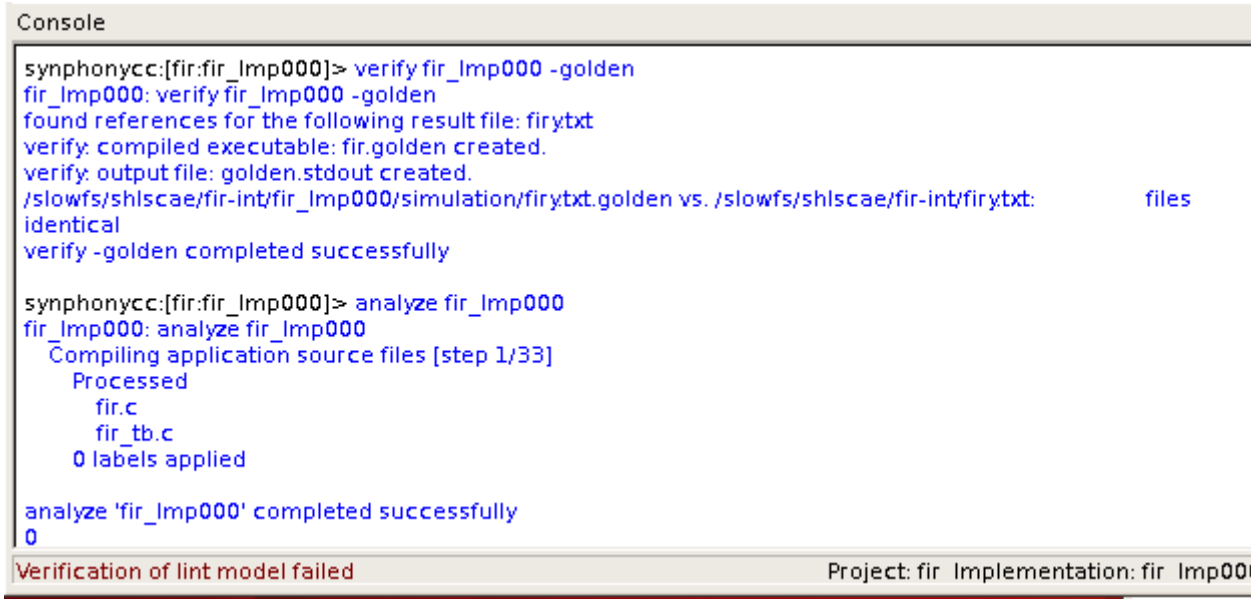
You can also export the hardware resources into a file in the .csv format, when you right-click on the Resources tab in the GUI, as shown in the following figures:





The Console Window

This is a shell window that provides a Tcl console for entering commands. You can execute any Tcl command supported by the Synphony C Compiler tool in this window. When you open a project or execute any of the Implementation, Group, or Run menu commands, the corresponding Tcl command is displayed in the shell window in blue.



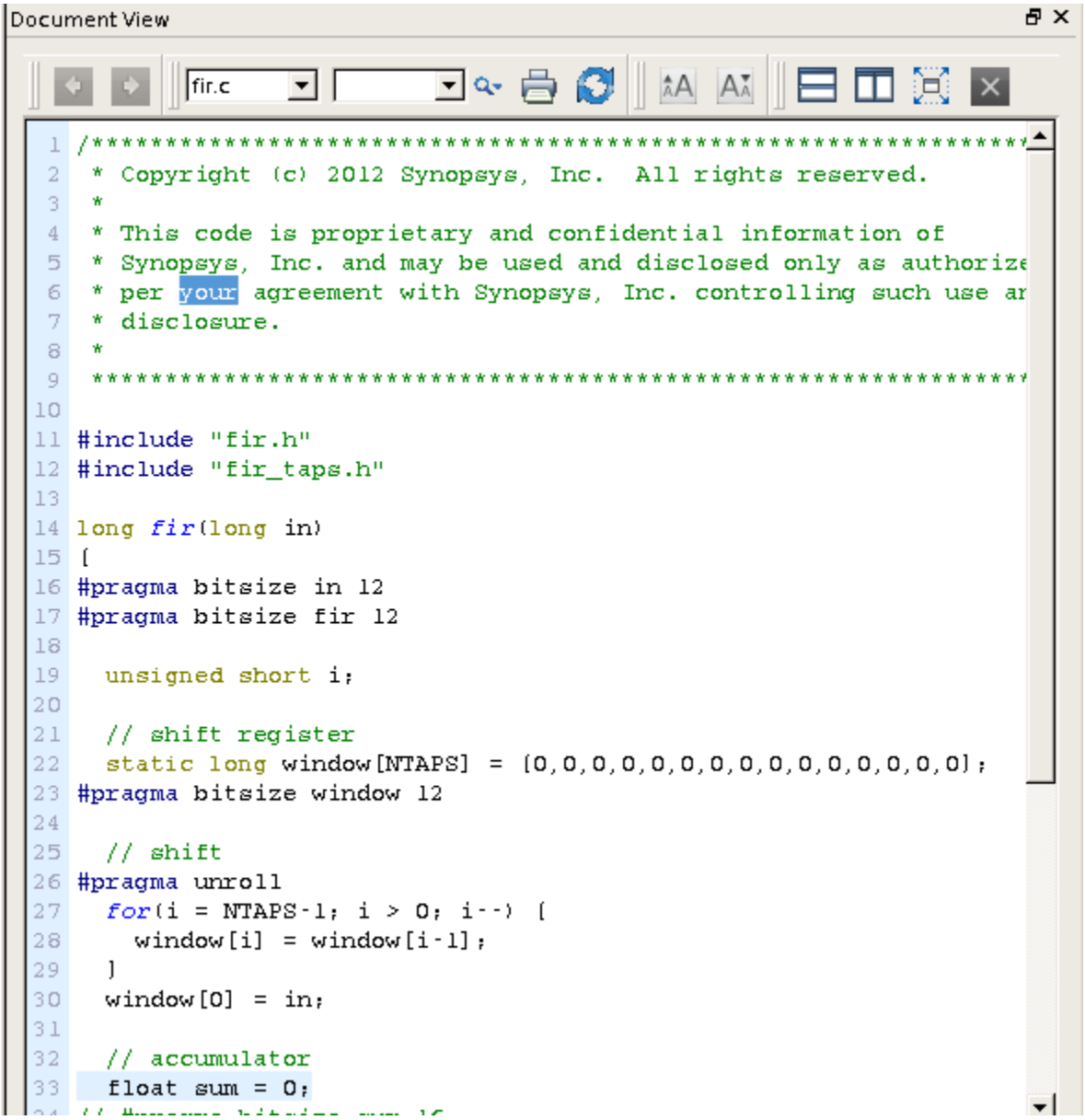
The console window lets you do the following to navigate and manipulate text in the window:

Delete key Backspace key	Deletes characters at the prompt.

Home key	Moves to the beginning of the prompt.
End key	Moves to the end of the command typed at the current prompt.
Cut and paste	Select an executed command using mouse and then double click middle button of mouse to paste that command at the prompt.
Ctrl+C	Interrupts execution of a command.
clear command	Clears the text in the window.

The Document View Window

The document view window is a text file browser and viewer. To remove or restore the Document View window, toggle the View > Document View menu option.



Using the pull-down menu or arrow buttons, you can select or scroll through the ASCII text files associated with a project such as C program, RTL, and report files. The window has buttons to print and refresh the current file, to tile multiple files vertically or horizontally, and to expand or close the view of the currently selected file. The currently selected text file has its line numbers highlighted in blue.

To set the editor characteristics such as line numbering, font, and recognized file name extensions, use the View > Editor Options menu command.

Analysis Views

The Synphony C Compiler tool provides several analysis views to analyze the QoR, scheduling of operations, and performance of an implementation.

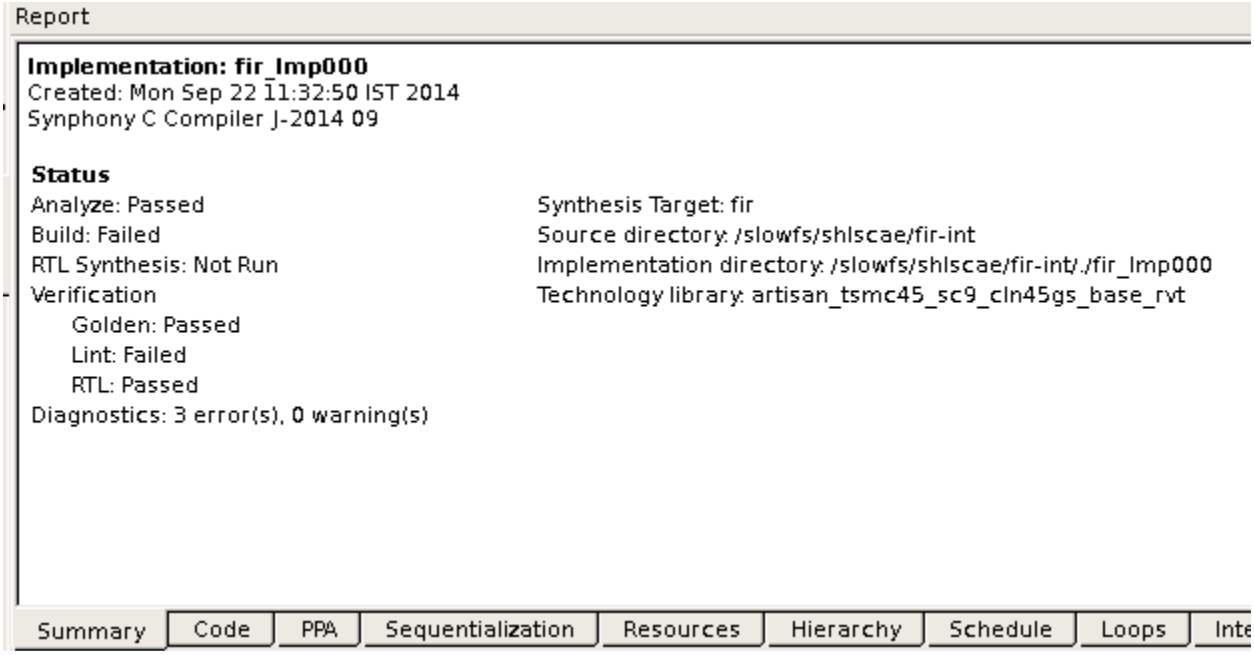
This section describes the GUI analysis views.

- Report View
- Diagnostics
- Code Tree View
- Sequentialization Graph
- PPA Graph
- Hierarchy Graph
- Operation Schedule Viewer
- Feedback Path Viewer
- Resource Browser
- Relative Cost Viewer
- Dynamic Performance Graph
- C to RTL Mapping

For information about accessing these views, see “Analysis Menu” on page 2-7.

Report View

The Report view presents an overview of a wide range of reports for an implementation and also the views for each implementation. To launch this view, select the implementation and then choose Analysis > Report or click the icon.



The report file shows the synthesis decisions made by the tool that affect performance characteristics such as clock rate, throughput, and latency. You can use the information in the report to determine the performance characteristics of the implementation and to set the performance constraints for subsequent synthesis runs.

The report is organized hierarchically by topic:

- Summary Header – shows basic information such as the application function name, file name, clock frequency, and technology library
- Implementation Status – shows the run status for each synthesis and simulation phase: Passed, Not Run, or Failed; with links to detailed reports
- Interface Summary – displays the TCAB properties, port interface
- Block Diagram – links to PPA Graph, Sequentialization Graph, Hierarchy Graph, ppa_all_fifos.pdf report, and ppa_sync_fifos.pdf report
- HW Architecture – shows the architectural features, timing, and area
- Dynamic Performance – links to the Dynamic Performance Graph and performance reports with the `verify -rtl` command performance reports
- Code Structure – links to the operation schedule pseudo code report file
- Relaxed Constraints – shows any constraints such as frequency or II that are relaxed by the tool to complete synthesis

- Reports – links to the design summary report files and reports the user code, internal timing, integration, QoR analysis, and performance analysis

The window also contains the following Views for each implementation—Summary, Code, PPA, Sequentialization, Resources, Hierarchy, Schedule, Loops, Interfaces, Performance, Implementation, Reports, as described in the following table:

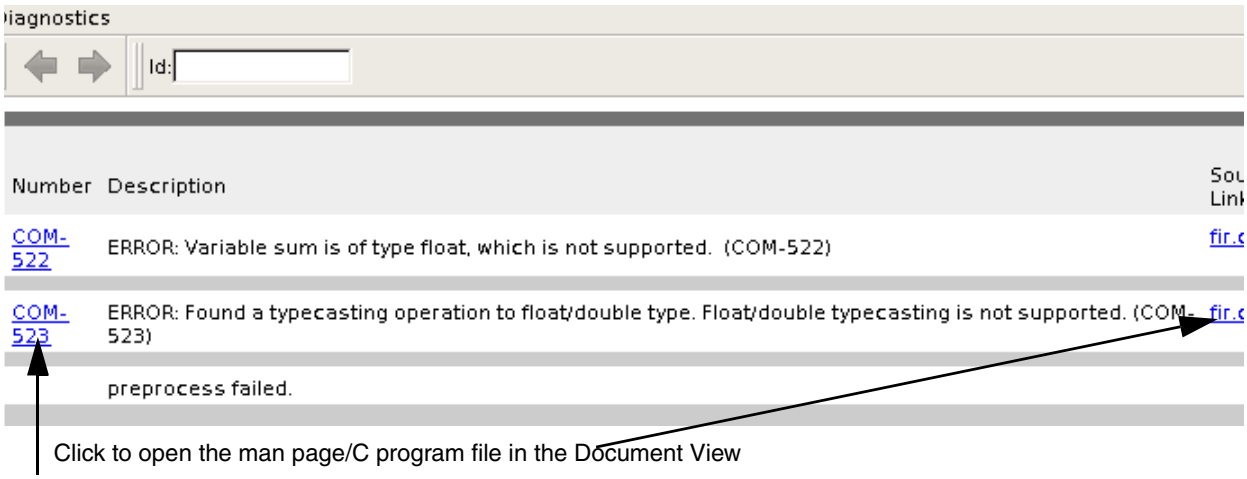
Report View Tab	Description
Summary Tab	Shows the results of the build and verification flow.
Code Tab	Contains a tree view of the C/C++ structure and hierarchy of functions, conditions, loops, and function calls in your code.
PPA Tab	Shows the PPA graph. Helps you understand the resource usage in your design.
Sequentialization Tab	Shows the Sequentialization graph. Helps you understand if the loop nests in your code execute simultaneously or sequentially with respect to each other.
Resources Tab	Displays the hierarchy of the logic components for the Verilog RTL. The view also reports the number of instances and area count, and provides the link to the C/C++ code.
Hierarchy Tab	Displays the design hierarchy, in which one implementation uses other implementations.
Schedule Tab	Displays the Gantt chart representation of the operations in your C/C++ code. The chart shows the operation dependencies and schedules.
Loops Tab	Contains a listing of the top-level loops in your C/C++ code and their properties. The table lists the pipelining and latency for the top-level loops.
Interfaces Tab	Contains a table explaining the I/O ports at the top level of the generated RTL. This tab helps you understand the interface between the RTL module generated by the Synphony C Compiler and the rest of your RTL design.
Performances Tab	Displays the performance of the design during the RTL simulation. The start cycle, stop cycle, and the number of stalled cycles for each top-level loop are reported as table and graphical format.
Implementation Tab	Shows a file browser for the directories and files of the implementation. The directory path of the implementation is listed at the top.

Report View Tab	Description
Reports Tab	Contains a listing of all the reports in text format shown in the Document Viewer generated during synthesis and allows you to display them in the GUI by clicking the report name.

Diagnostics

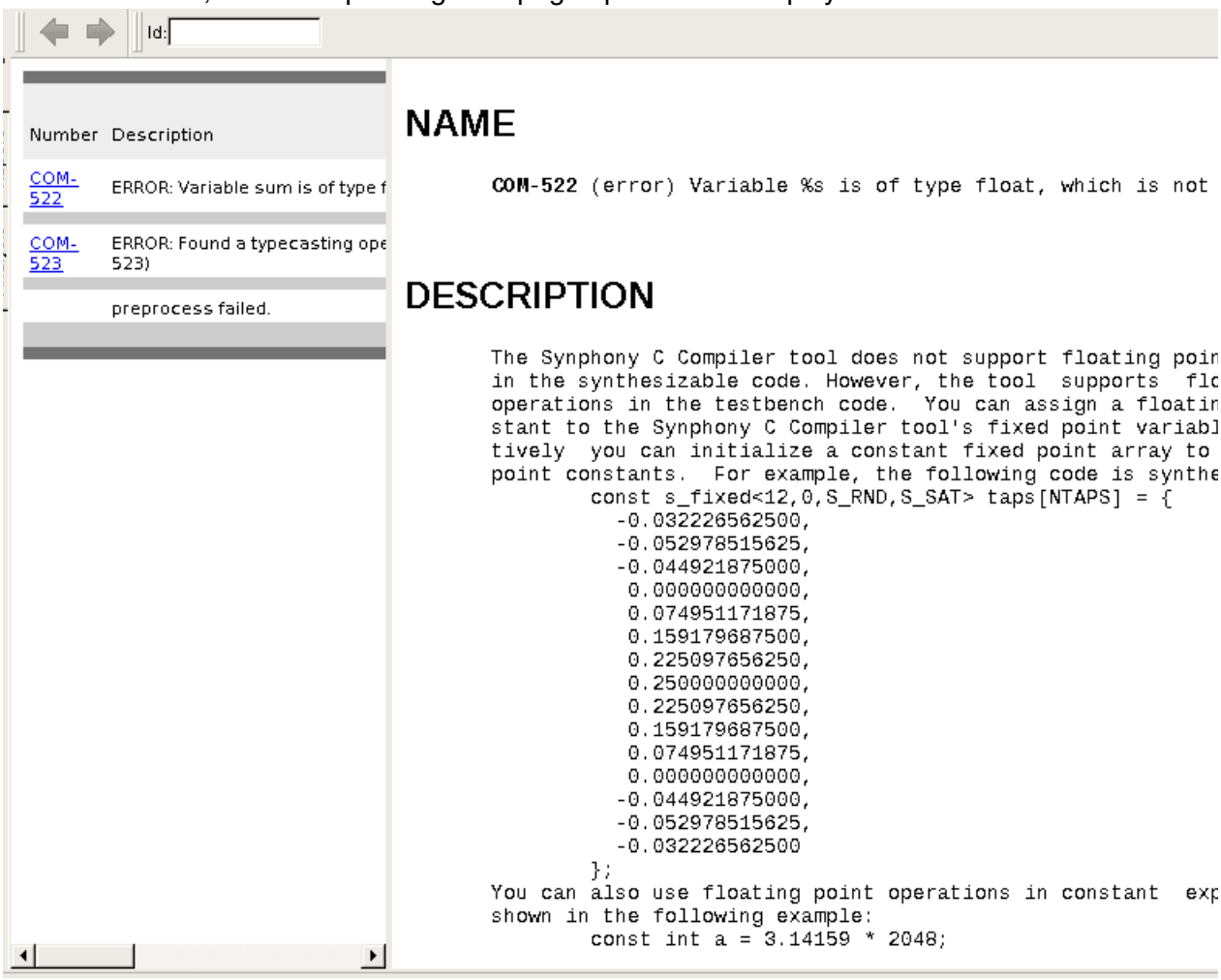
The Diagnostics View presents the errors and warning messages of the most recent run in an organized format. Multiple errors of the same type are reported together. You can expand and collapse groups of messages and jump to a man page or display the source file associated with a given message.

To open the Diagnostics View, choose Analysis > Diagnostics or click the question mark icon in the toolbar. The following figure shows a typical Diagnostics View.



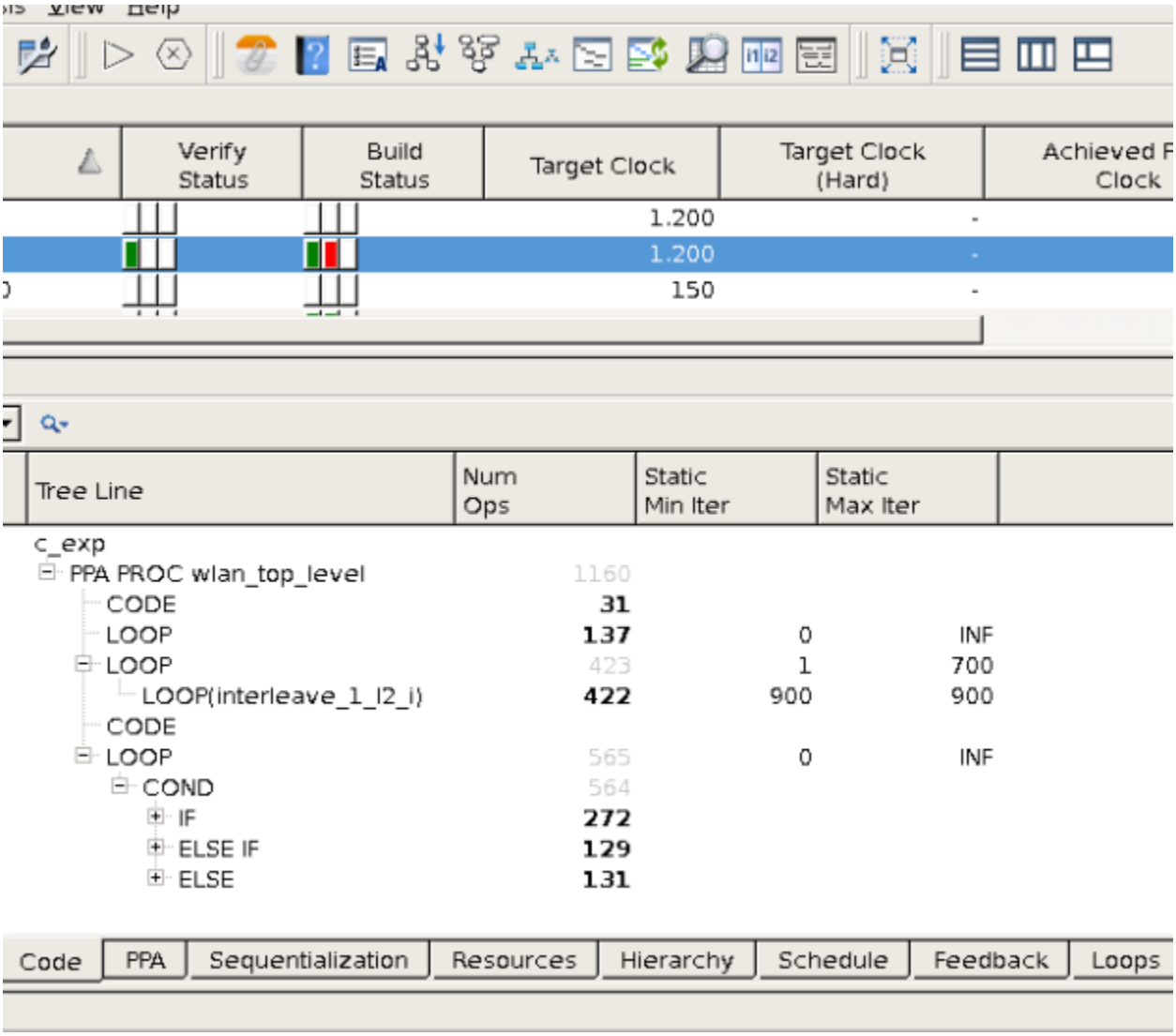
You can click the blue error number to display the man page of the error, or click a blue file name to open the file in a Document View window. The triangle buttons at the top let you navigate forward and backward through various views, as in a web browser. To show or hide errors, warnings, and notes, click the context menu header. Only errors and warnings are shown by default.

To view a man page of a command, type the command in the ID field. When you click on the ID number, the corresponding man page opens in the display window as shown:



Code Tree View

The Code Tree view provides detailed information about the loops, procedure calls, and conditions. This is an example of this view:



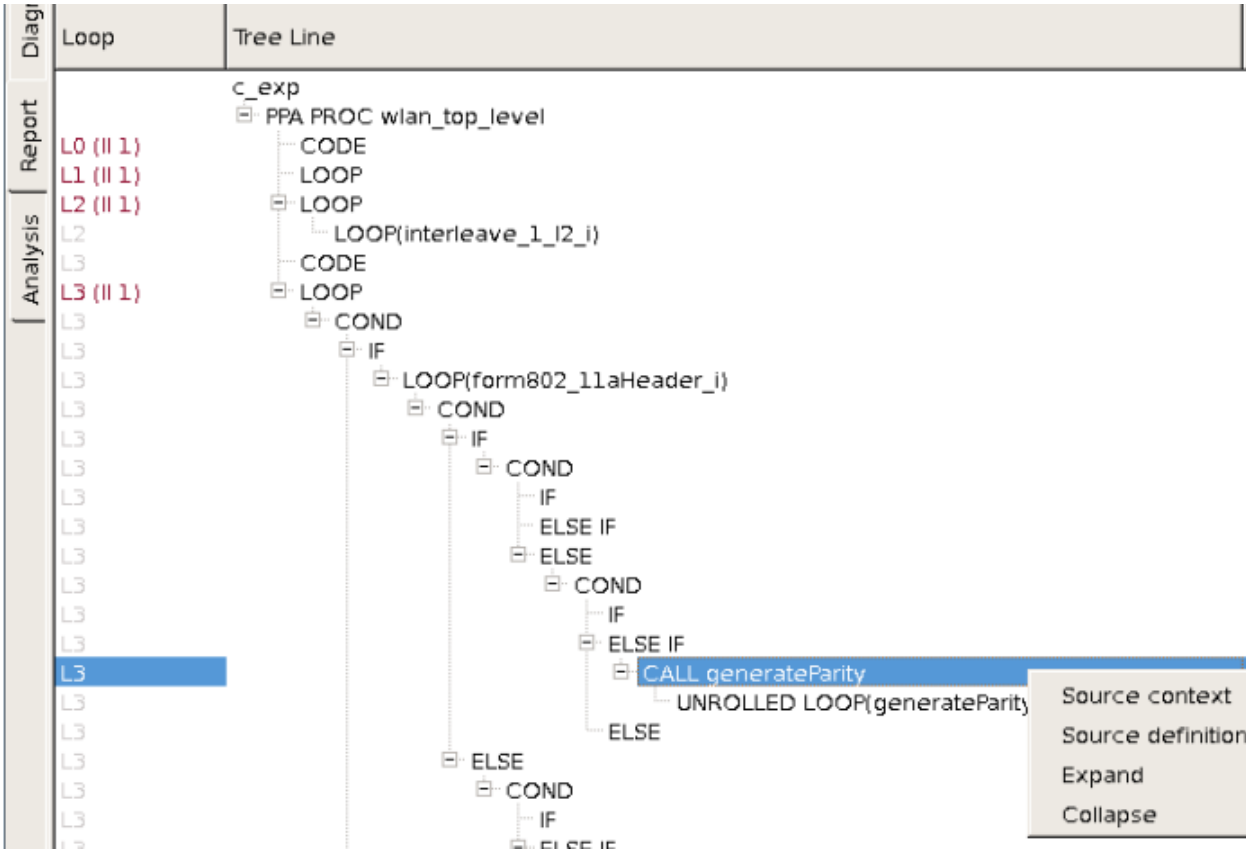
This view consists of the following columns:

Header	Description
Tree Line	Displays the procedure, TCAB or PPA and condition branches within each loop in a tree lined fashion.

Header	Description
Loop	Displays the loop numbers for each item and the II of the loop nest shown in the Tree view.
Num Ops	Displays the total number of operations in the logic cone at this level. This information is gathered after static analysis of your code.
Static Minimum Iteration	If the loop nests in your code run for a fixed number of iterations always, this column, displays the statically specified or determined lower bound on the number of times this loop would execute.
Static Maximum Iteration	If the loop nests in your code run for a fixed number of iterations always, this column displays the statically specified or determined upper bound on the number of times this loop would execute.

Context-Sensitive Node Menus

Right-click the loop, code segments, and function calls to get context-sensitive menu. The following figure shows some of the context-sensitive menus.

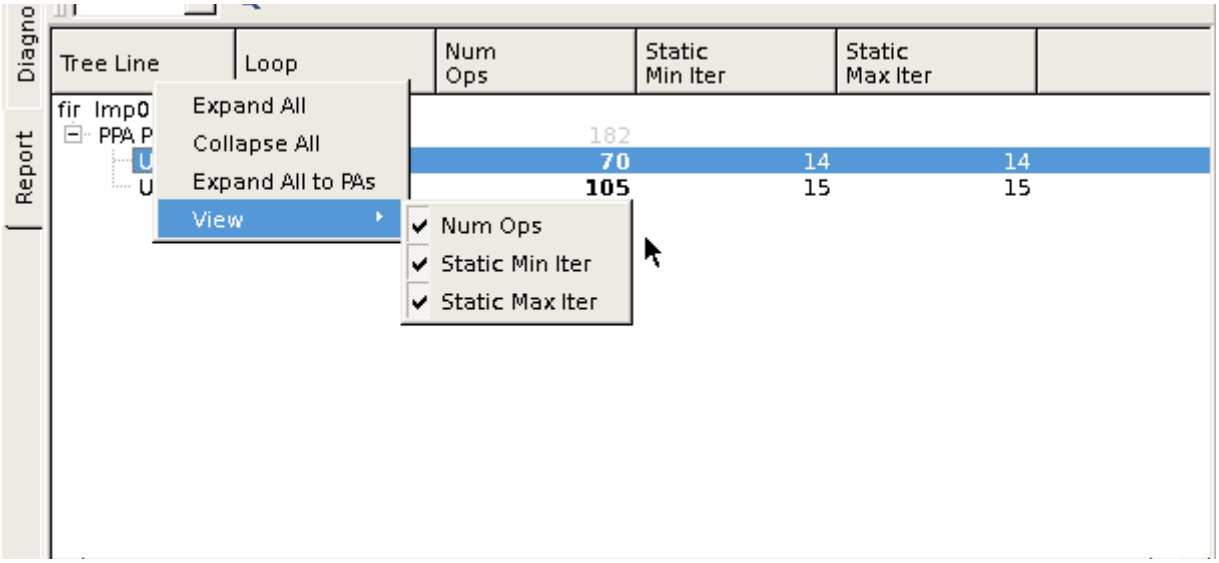


The Node menu commands are:.

Commands	Description
Source context	Displays the corresponding code where this node is called in the source file.
Source definition	Displays the definition of the selected function (only if the function corresponds to a *call*). This is enabled for items with CALL label in the code tree.
Expand	Expands the entire tree view at this level.
Collapse	Collapses the entire tree view at this level.

Tree Line Header Menu Commands

Right-click the Tree Line header to bring up expansion options that apply to the tree view.

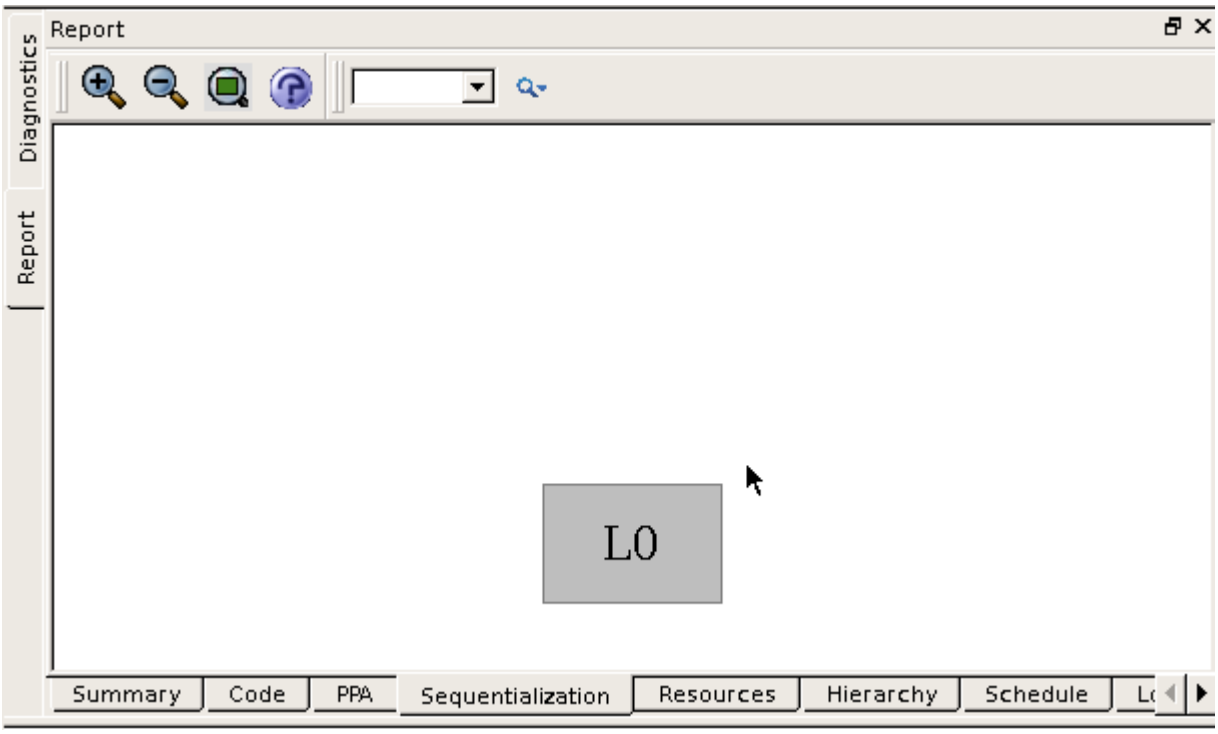


The Line Header menu commands are:

Command	Description
Expand All	Expands the entire tree view.
Collapse All	Collapses the entire tree view to display only the major nodes (Loops).
Expand All to PAs	This option consolidates the tree view to show only the nodes where new PAs start
View	Displays the selected columns in the Analysis window. Select the items from the list to display. The sub-options displayed are: Num Ops, Static Min Iter, and Static Max Iter.

Sequentialization Graph

Select Analysis > Sequentialization Graph or click the Sequentialization Graph icon ( ) to display the graph.



The sequentialization graph indicates loop sequentialization and resource dependencies with colored arrows. The direction of the arrow indicates the order of execution.

Blue arrows	Resource dependence because of shared memories
Green arrows	Resource dependence because of shared stream ports
Yellow arrows	Data dependence where the scalar output of one loop is consumed by another
Red arrows	Communication across loops through FIFOs
Rectangles	Loop nests

You can also click the question mark icon to display a key for the symbols used.

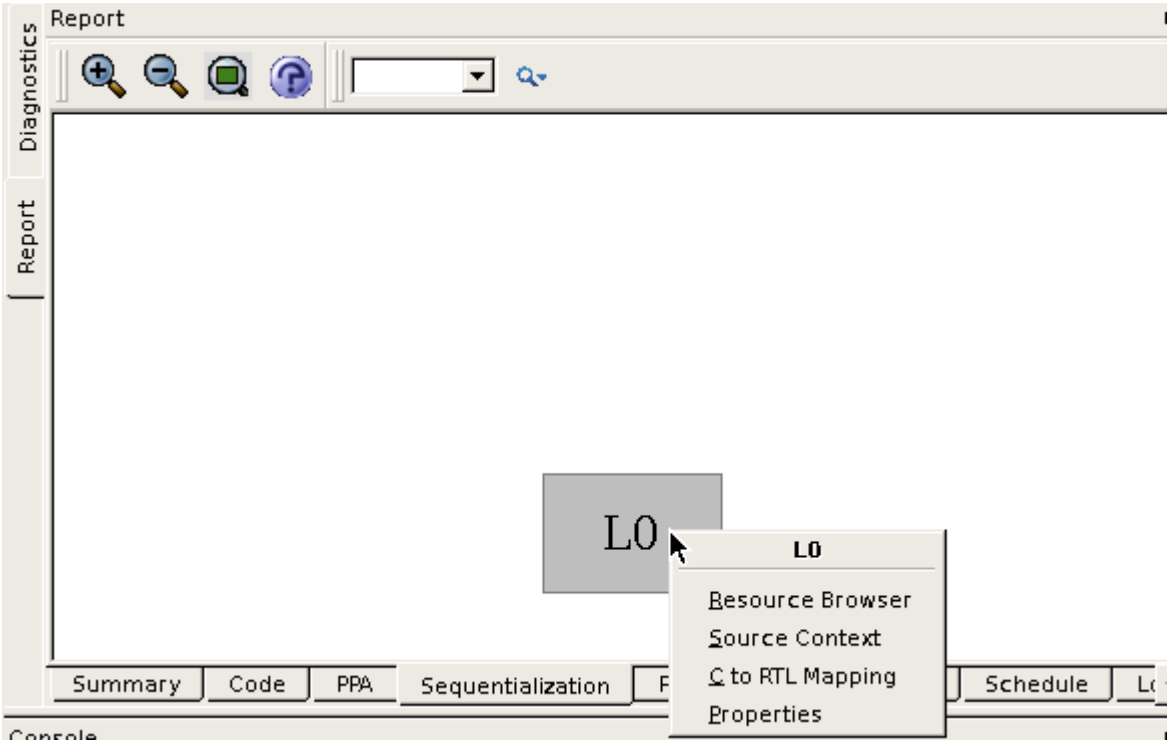
Toolbar Icons

The icons in the toolbar are:

Icons	Description
Zoom In	Increases the size of all elements by 20% in the x-direction.
Zoom Out	Decreases the size of all elements by 20% in the x-direction.
Zoom Fit	Expands or shrinks the window contents to fit all the contents in the viewable area.
Graph Key	Displays the Sequentialization Graph Key, which describes the symbols used in the window.

Context-Sensitive Information

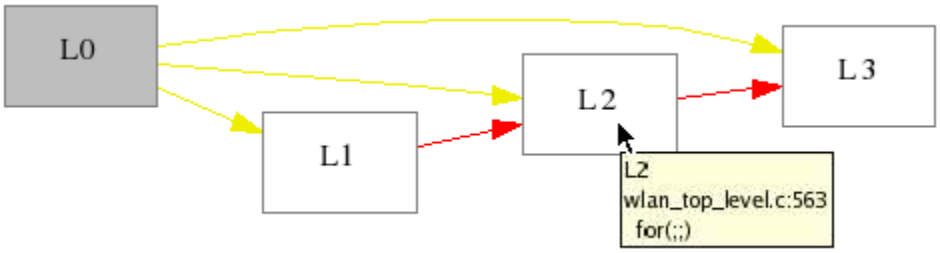
Right-click an object in the graph to see the context-sensitive menu for the object. The following figure shows the menu for loops.



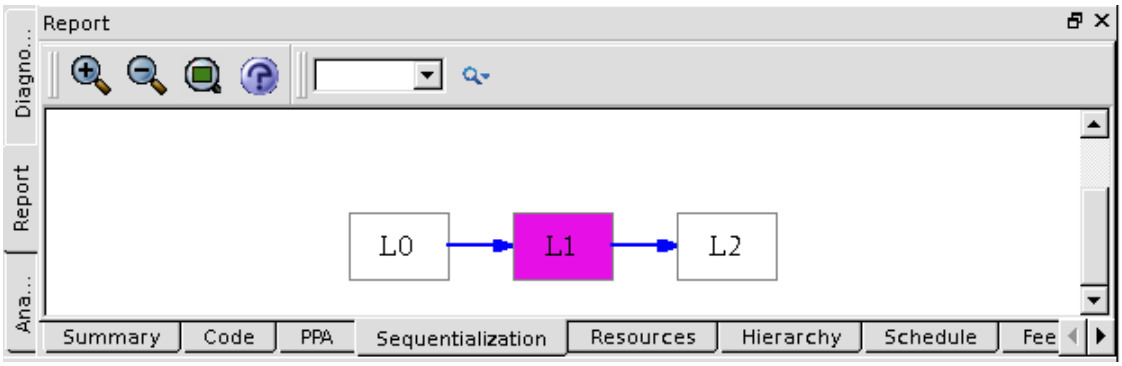
This table describes the commands:

Context Menu Items	Description
Resource Browser	Displays the hardware resource for the selected element in the Resource Browser Analysis view. See "Resource Browser" on page 2-57.
Source context	Opens a split text editor window with the C or C++ source code, with the cursor at the line and column for the selected element.
C to RTL Mapping	Enables C to RTL mapping in a text editor. See "C to RTL Mapping" on page 2-67.
Properties	Displays loop information for the selected element. See "Object Properties" on page 2-29.

Hovering over an object shows the ToolTip with the label.

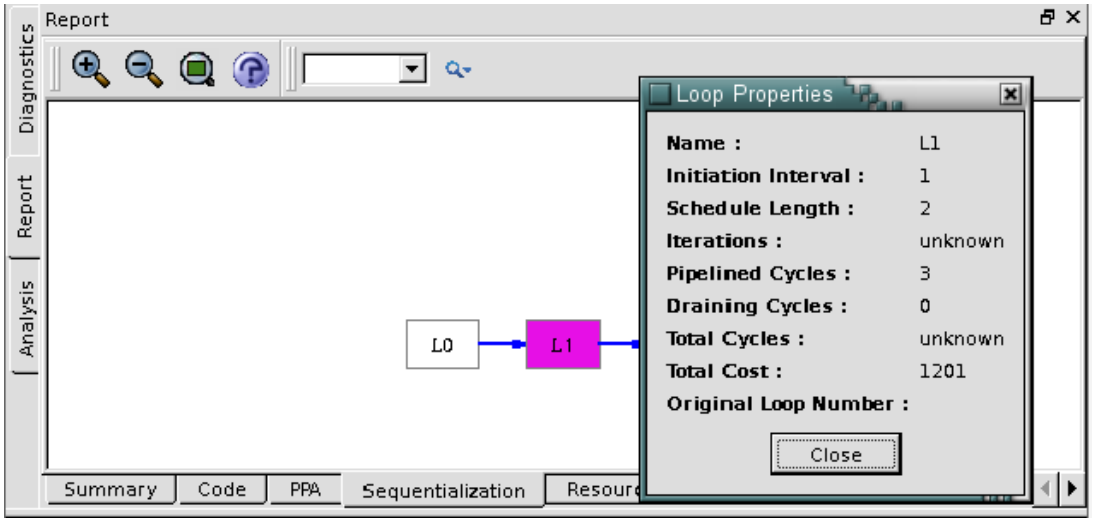


Clicking an object highlights it and makes all its incoming and outgoing edges bold. Clicking on the text highlights the text and opens the source context.



Object Properties

The Properties command displays the properties for the selected element in the Sequentialization Graph. Right click the selected loop and select Properties.

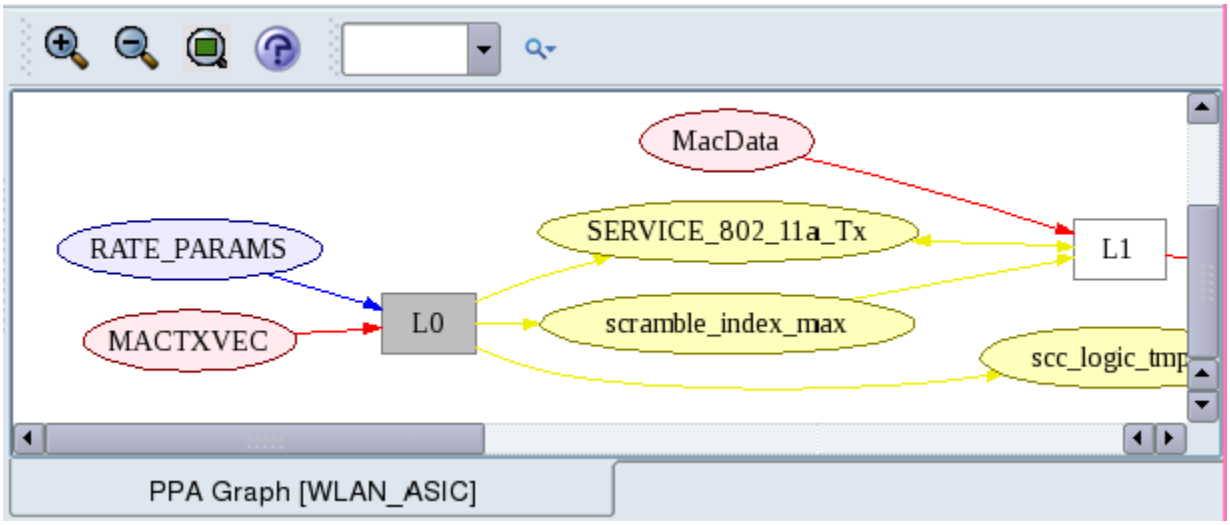


The Loop Properties window displays the following information of the selected element:

Property	Description
Name	Specifies the name of the loop.
Initiation Interval	Specifies the time gap between two successive runs of the loop.
Schedule Length	Specifies the length of the instruction schedule in clock cycles.
Iterations	Specifies the number of iterations in the loop nest.
Pipelined Cycles	The number of cycles where the current task iterations are still being executed
Draining Cycles	The number of cycles to complete the execution of this loop after all iterations of the current task are processed.
Total Cycles	Specifies total cycles taken to execute the loop nest excluding the effects of stalling and early exits.
Total Cost	Specifies the number of gates used to implement the loop.
Original Loop Number	New loops created by perfectization might have different loop numbers in the generated hardware than the original loops you created. This property helps you match a loop to its original loop number.

PPA Graph

Invoke this view by selecting Analyze > PPA Graph or by clicking the PPA Graph icon ( ). The PPA Graph is resource-centric. It displays information about the resources used by each loop.



Rectangles	Loops.
Blue ovals	Memories and arrays.
Red ovals	Streams and FIFOs.
Yellow ovals	Scalars.
Green ovals	FIFOs that have more than one reader or inter-loop dependency.
Arrow direction	From a loop node to a resource node: indicates that the loop writes to the resource. From a resource node to a loop node: indicates that the loop reads from the resource. Bidirectional arrow: indicates that the loop both reads to and writes from the resource.
Green arrows	Shared stream ports. Arrow direction indicates the sequential order enforced by the sharing.

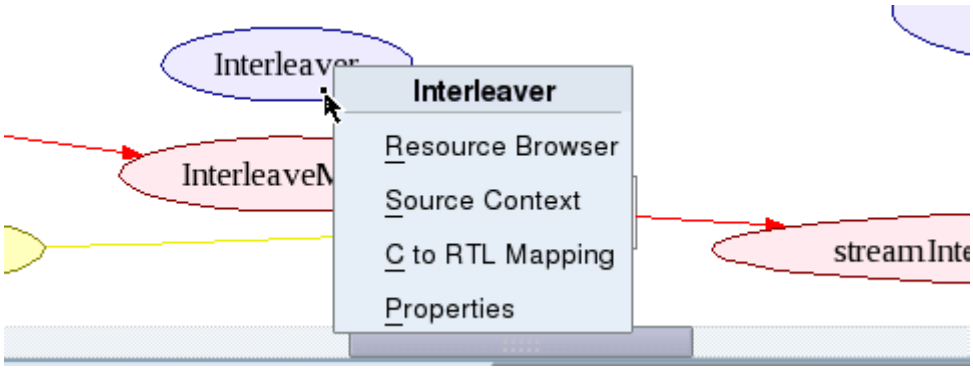
Red arrows	FIFO communication.
Blue arrows	Stream communication.
Yellow arrows	Scalar communication.

See the following for more information about the PPA Graph:

- "Toolbar Icons" on page 2-27
- "Context-Sensitive Information" on page 2-32
- "Loop, Stream, Memory, and Scalar Properties" on page 2-34

Context-Sensitive Information

Right-click an object to bring up the context-sensitive menu for that object. This menu is available for ovals and rectangles: loops, streams, memories, and scalars. See the example in the following figure.

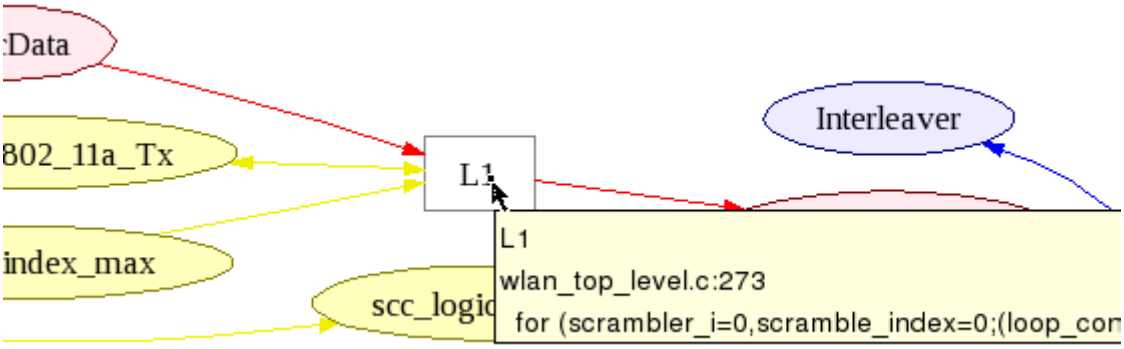


The following table describes the commands in the menu:

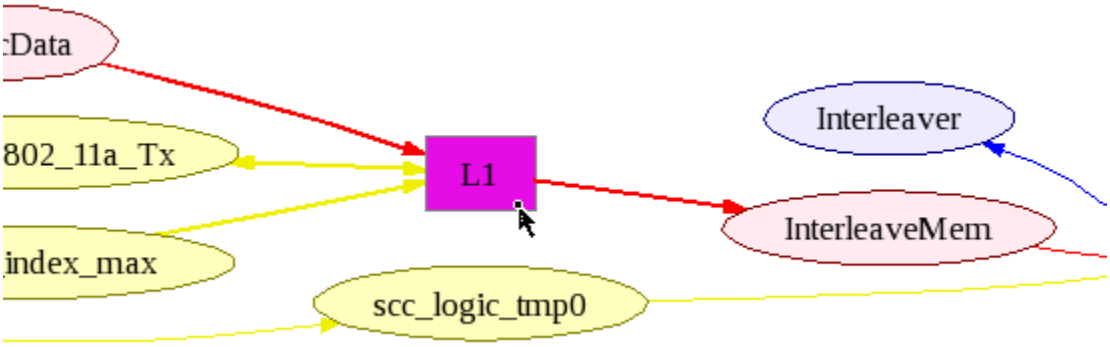
Commands	Description
Resource Browser	Displays the hardware resource corresponding to the selected element in the Resource Browser. See "Resource Browser for PPA Graph" on page 2-60.
Source context	Opens a split text editor window with C or C++ source code. The cursor is placed on corresponding source code line and column of the selected element.

Commands	Description
C to RTL Mapping	Enables C to RTL mapping in a text editor.
Properties	Displays the following information about the selected element. See "Loop, Stream, Memory, and Scalar Properties" on page 2-34 for details.

Hovering over an object shows the ToolTip with the label:

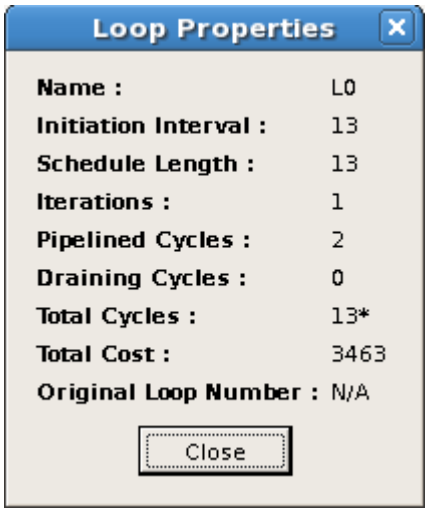


Clicking on an object highlights it and all its incoming and outgoing edges become bold. Clicking on the text highlights the text boundary and opens the source context.



Loop, Stream, Memory, and Scalar Properties

The Properties command displays the properties of the selected element in the Sequentialization Graph. Right-click the selected loop and select Properties.

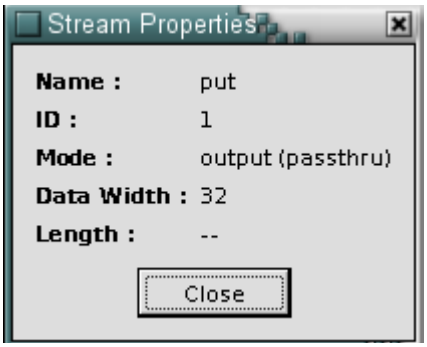


The Loop Properties window displays the following information of the selected element:

Properties	Description
Name	Specifies the name of the loop.
Initiation Interval	Specifies the time gap between two successive runs of the loop.
Schedule Length	Specifies the length of the instruction schedule in clock cycles.
Iterations	Specifies the number of iterations in the loop nest.
Pipelined Cycles	The number of cycles where the current task iterations are still being executed.
Draining Cycles	The number of cycles to complete the execution of this loop after all iterations of the current task are processed.
Total Cycles	Specifies total cycles taken to execute the loop nest excluding the effects of stalling and early exits.
Total Cost	Specifies the number of gates used to implement the loop.

Properties	Description
Original Loop Number	If any new loops were created due to perfectization, then the loop numbers in the generated hardware might be different from original loops written by you. This property helps user track the original loop number of a loop before any new loops got introduced due to perfectization.

Stream Properties



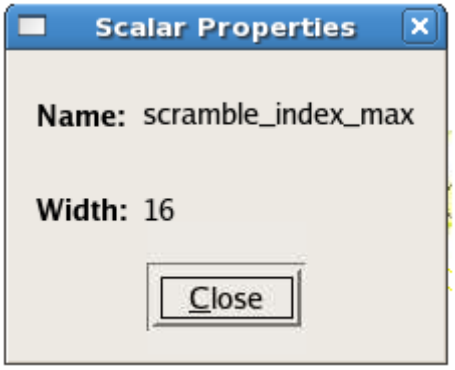
Properties	Description
Name	Specifies the name of the loop.
Identification Number	Specifies the time gap between two successive runs of the loop.
Mode	Specifies the length of the instruction schedule in clock cycles.
Data Width	Specifies the number of iterations in the loop nest.
Length	Specifies the length of the stream.

Memory Properties

Property	Description
Name	Reports the name of the memory.
ID	Reports the identification number of the memory.
Parameters	Reports the organization of words, width, number of buffers, and their latency.

Property	Description
Description	Reports the physical implementation of the memory. • Provider Synthesized (S) or User supplied (U) • Instantiation Specifies the physical location of the memory. It has the following two options: E - External to the design boundary I - Internal to the design boundary
Indexed Ports	Reports the number of Read-Only, Write-Only, and Read-Write ports for each buffer.
Arbitration Information Requests	Reports the number of Read-Only, Write-Only, and Read-Write requestors.
Arbitration Information Ports	Reports the number of Read-Only, Write-Only, and Read-Write ports.
Boundary Info Boundary Registers	Reports the presence or absence of forward and return boundary registers.
Boundary Info External Delays	Reports the amount of external delay of the forward and return boundary registers.

Scalar Properties

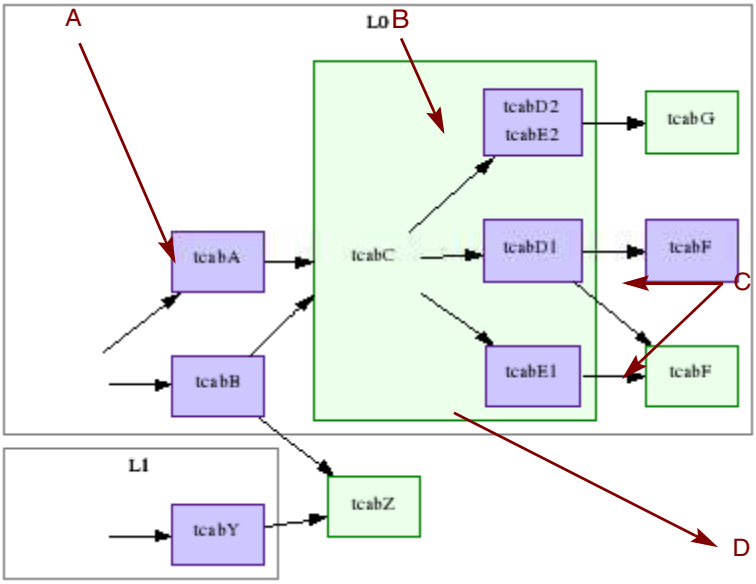


Property	Description
Name	Specifies the name of the scalar.
Width	Specifies the width of the scalar.

Hierarchy Graph

Select Analysis > Hierarchy Graph or click the Hierarchy Graph icon ( ) to display the graph. The graph shows the different modules imported into the design and the hierarchy of the modules.

The graph consists of rectangular boxes and arrows. A TCAB instance is hierarchically shared only if it has multiple callers in the graph. Green boxes are shared hierarchically; blue boxes are not shared. The graph does not represent other ways of TCAB sharing within a hierarchical level.

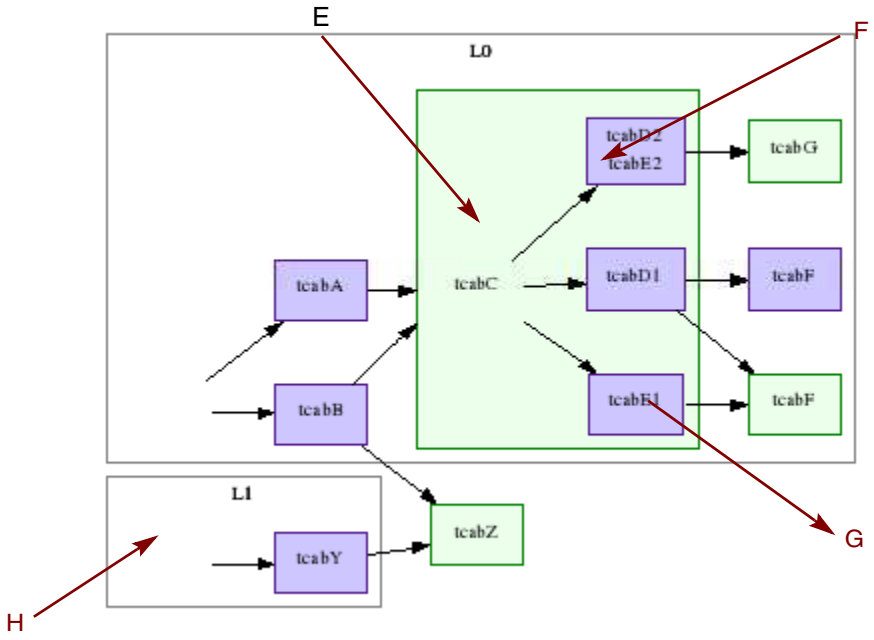


A A string in a box
The string is the name of the instantiated TCAB function or loop.

B Multiple strings in a box
Implies that all instances listed in the box have the same predecessors and successors (therefore they are merged into one).

C	Repeated strings in different boxes Repeated strings represent different instances of the same TCAB.
D	Boxes within boxes Instances represent the physical hierarchy in the generated RTL.

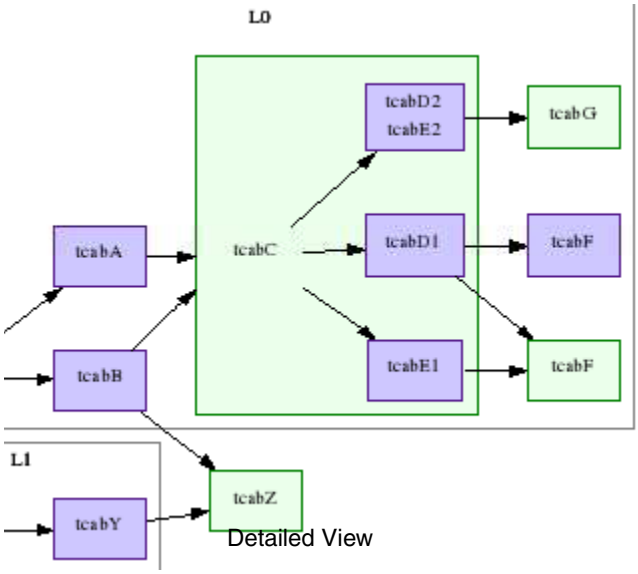
The following figure and table describes the arrows in the Hierarchy Graph.



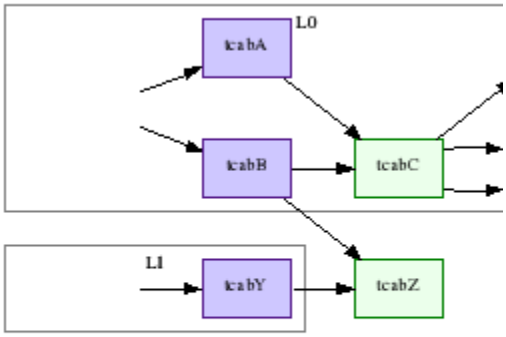
E	Arrow The start of the arrow indicates the caller and its end indicates what is being called.
F	Start of an arrow connected to a box The callers are all the TCAB instances or loops associated with that box.
G	End of an arrow connected to a box The instances being called are the TCAB instances associated with that box.
H	Unconnected arrow If the arrow is contained within a box, the callers are all the TCAB instances or loops associated with the container box.

There are two Hierarchy Graph views:

Views	Description
Detailed view	Displays the complete hierarchy, with child-TCABs inside the parent-TCABs across several levels of hierarchy.
Top-level view	Displays only the first-level TCABs and hides all child-TCABs.



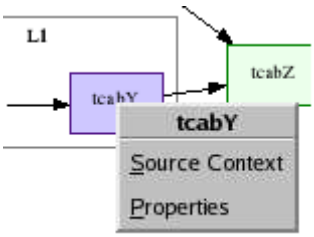
Detailed View



Top-level View

Context-Sensitive Menu

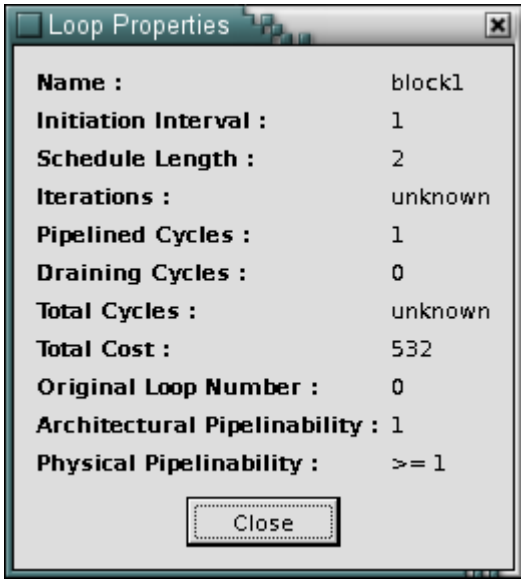
Right-click text to see the properties and source for that particular TCAB instance.



The command directly takes you to the source position. If there are multiple source positions, the Source Context command lists the options and you can select from them.

Properties

The Properties command displays information about the selected object.

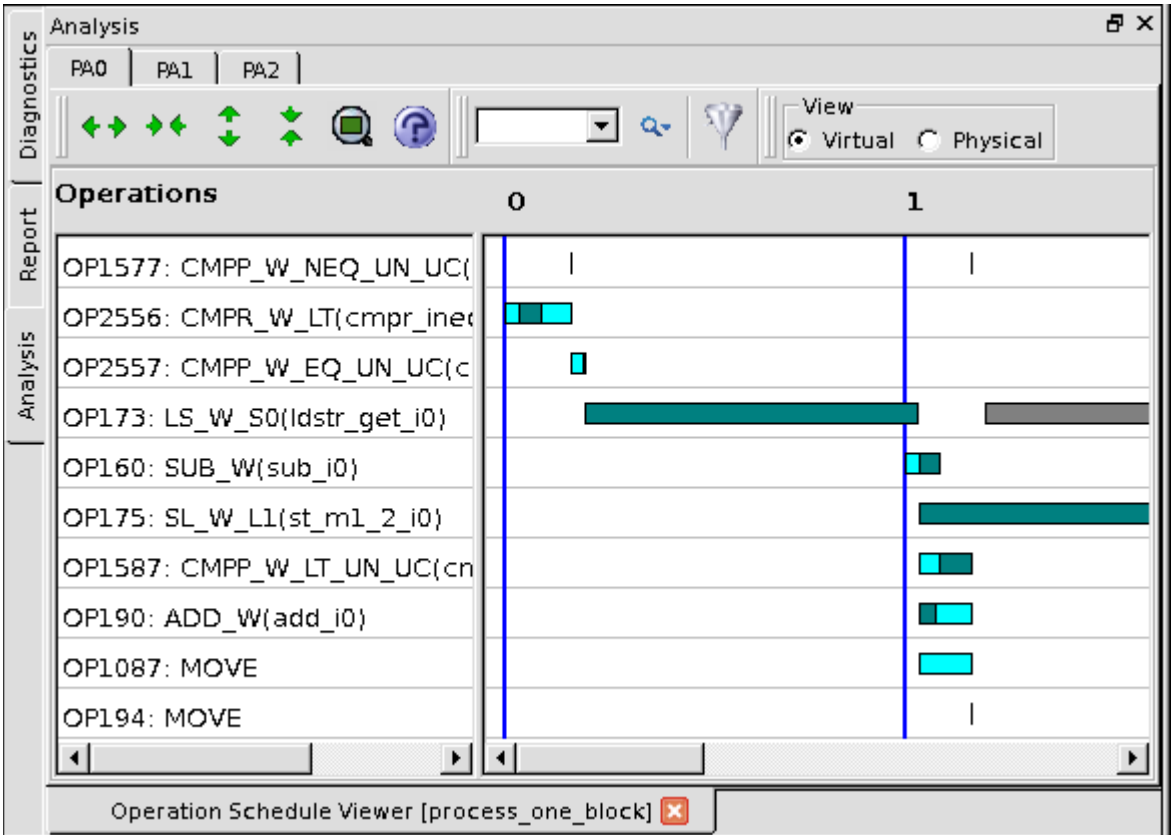


Operation Schedule Viewer

Invoke the Operation Schedule view by selecting Analysis > Operation Schedule Viewer or clicking the Operation Schedule Viewer icon ( ).

The Operation Schedule Viewer displays the operation schedule on a PA-by-PA basis. Each PA is shown in a different tab and each tab shows the operations scheduled for that PA. The operations are shown in schedule order for PAs that are successfully scheduled, and in program order for PAs that fail to meet constraints.

You must select an implementation to view the Operation Schedule Viewer. For the selected implementation, the window displays a list of operations for each processing array (PA).



The tabs at the top let you select one processing array to analyze from among those in the implementation.

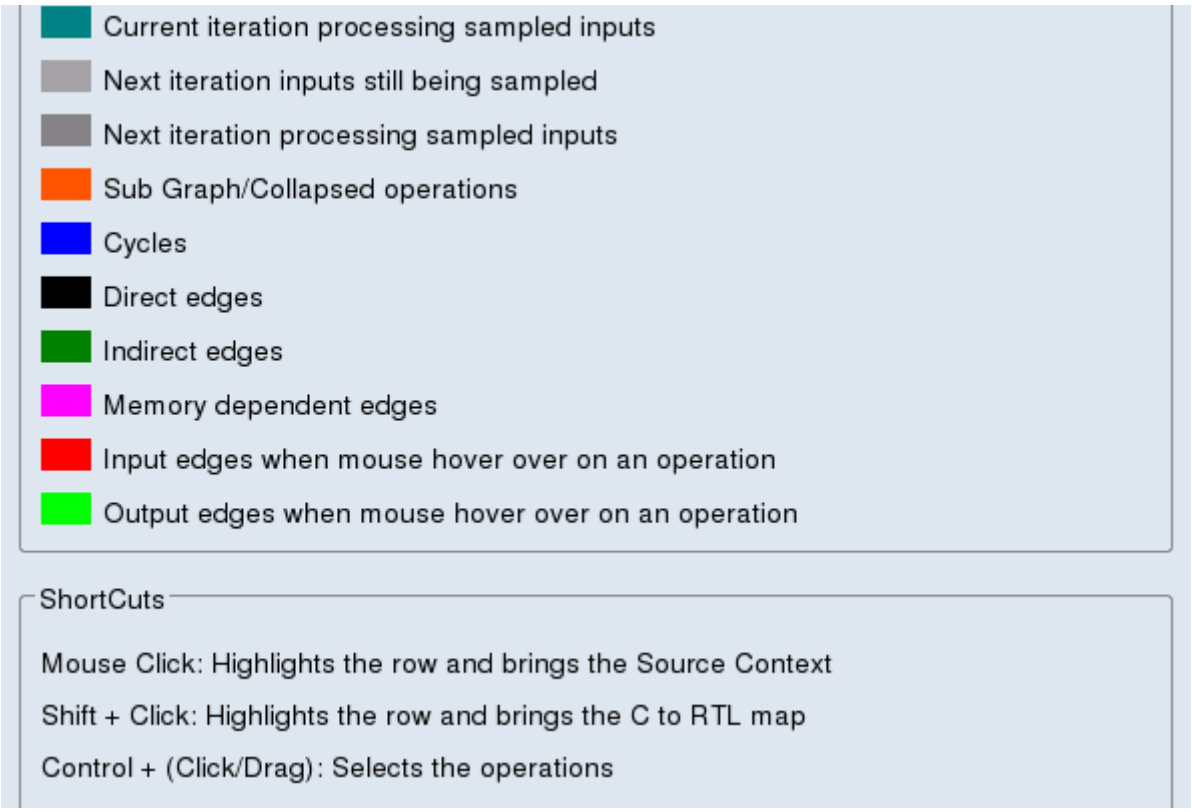
The Operations list box on the left shows a list of compiler op numbers, the compiler opcode, and the functional unit name for operations other than MOVE.

In the cycle graph on the right, a colored bar shows the estimated timing of each operation. The critical path through the operation is shown as a green colored bar. Some operations also display a cyan colored bar before the green. The beginning of the cyan section indicates the arrival of the first (noncritical) input.

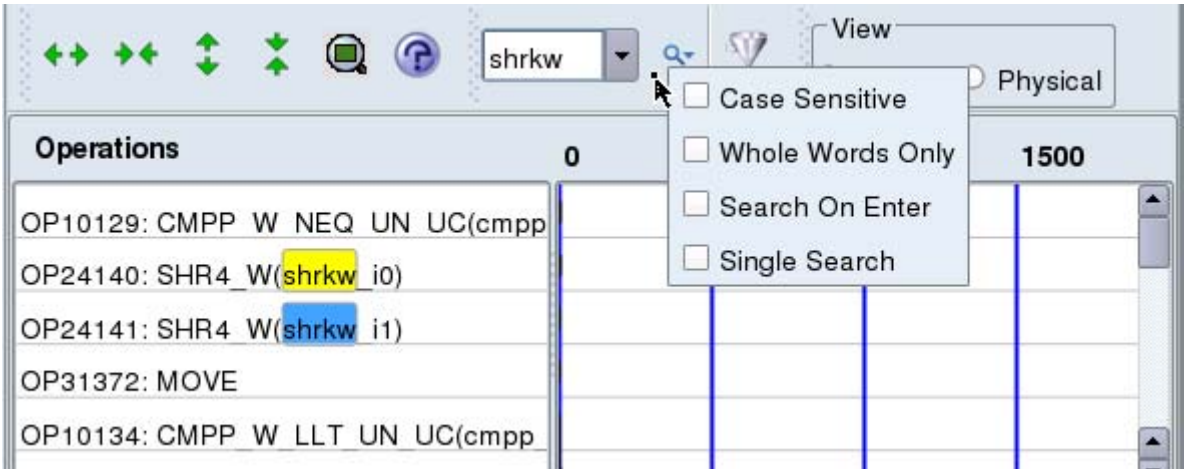
Operations that participate in feedback paths show a second, gray-colored bar. This bar shows the next execution of that operation in a subsequent loop iteration.

The green-arrow buttons let you zoom in and zoom out the cycle graph horizontally and vertically. Zooming in or out horizontally decreases or increases the number of cycles displayed in the graph. Zooming in or out vertically decreases or increases the number of operations displayed in the list. You can also drag the border between the operations list and cycle graph list to allocate the space between them.

The question mark button displays a help screen showing a color legend for the graph and a list of keyboard shortcuts:



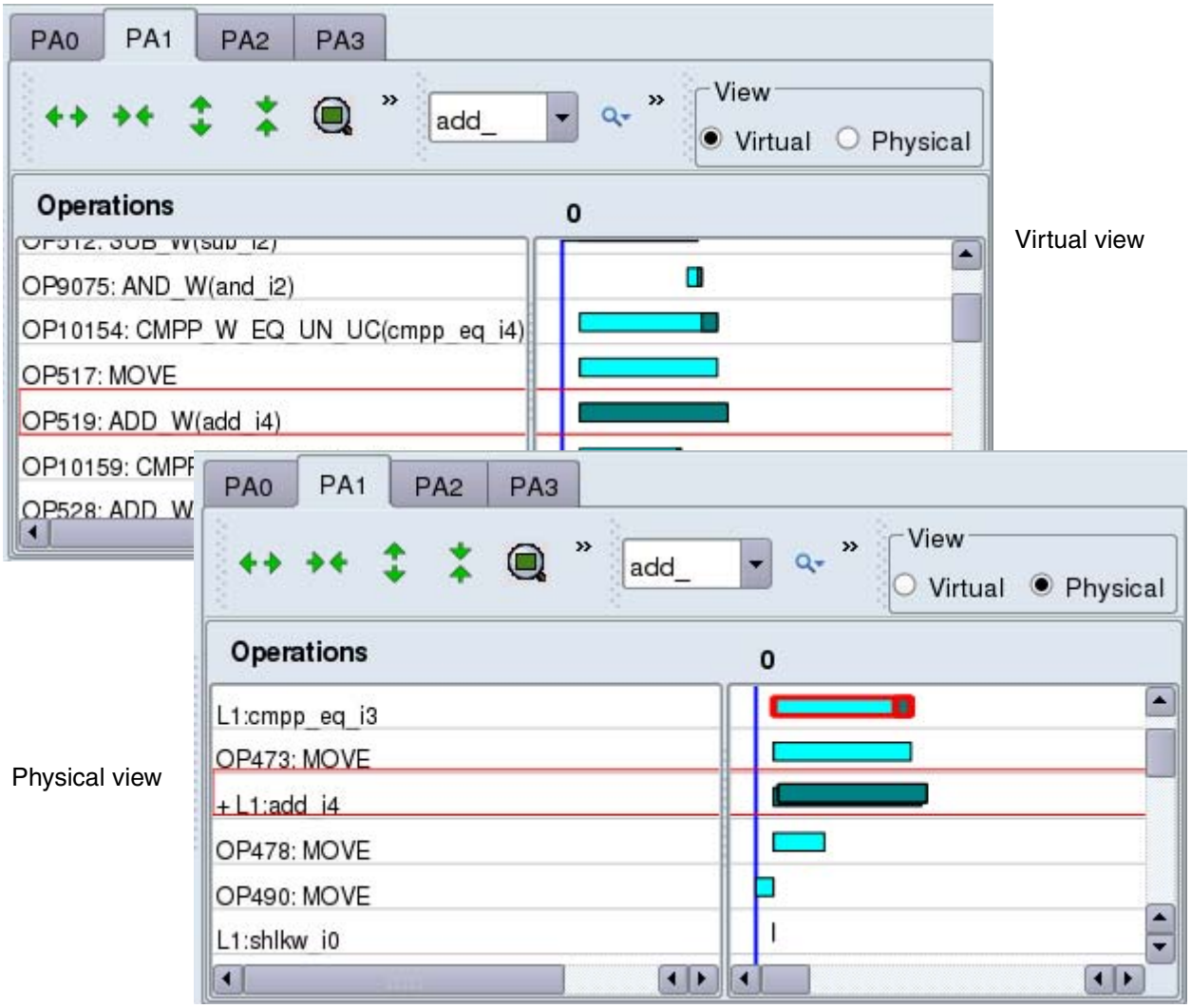
The Search text box lets you search for occurrences of a string within the Operations list. To change the search options, use the magnifying glass button to the right of the search field:



The Filter button, displayed as a filter funnel, opens a dialog box for filtering the operations displayed in the Operations list. You can restrict the operations displayed according to the type of object (Memory, Stream, or TCAB), by operation type (ADD, SUB, DIV, CMPP, and so on), by bit width, or by search string.

The View buttons select either Virtual or Physical operations in the Operations list:

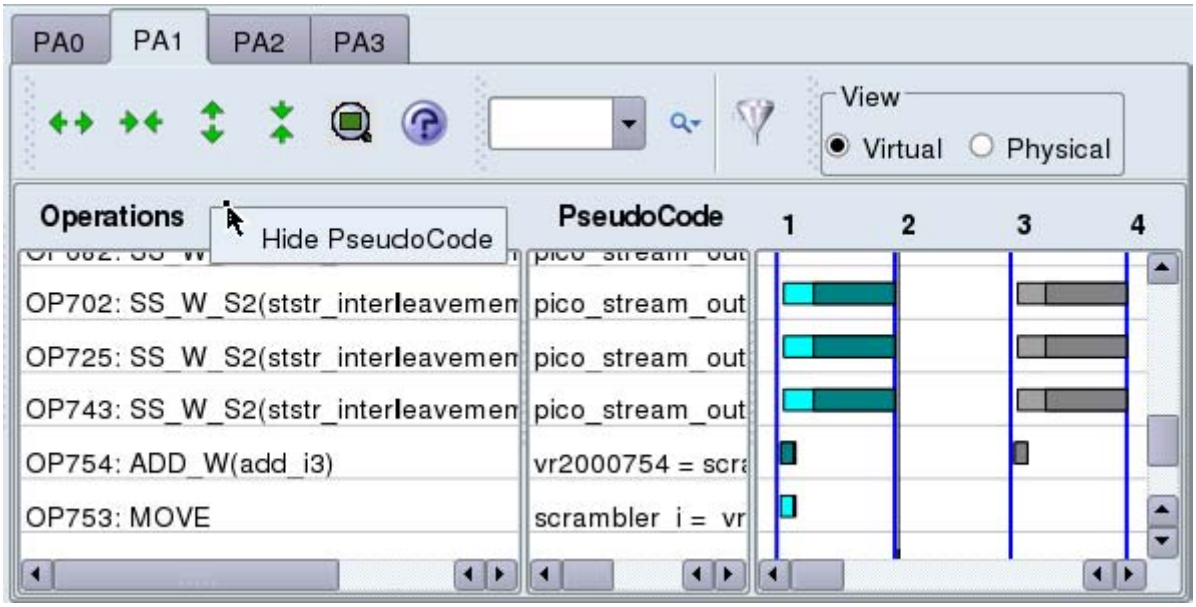
- Virtual – The program view. Each row shows an individual operation from your code.
- Physical – The hardware view. Each row shows a functional unit from the hardware. If multiple operations are mapped to one unit, the row shows the operations overlapping. Note that only two overlapped rectangles are shown, even if more than two operations are mapped to the unit; the ToolTip shows the number of operations mapped.



Virtual view

Physical view

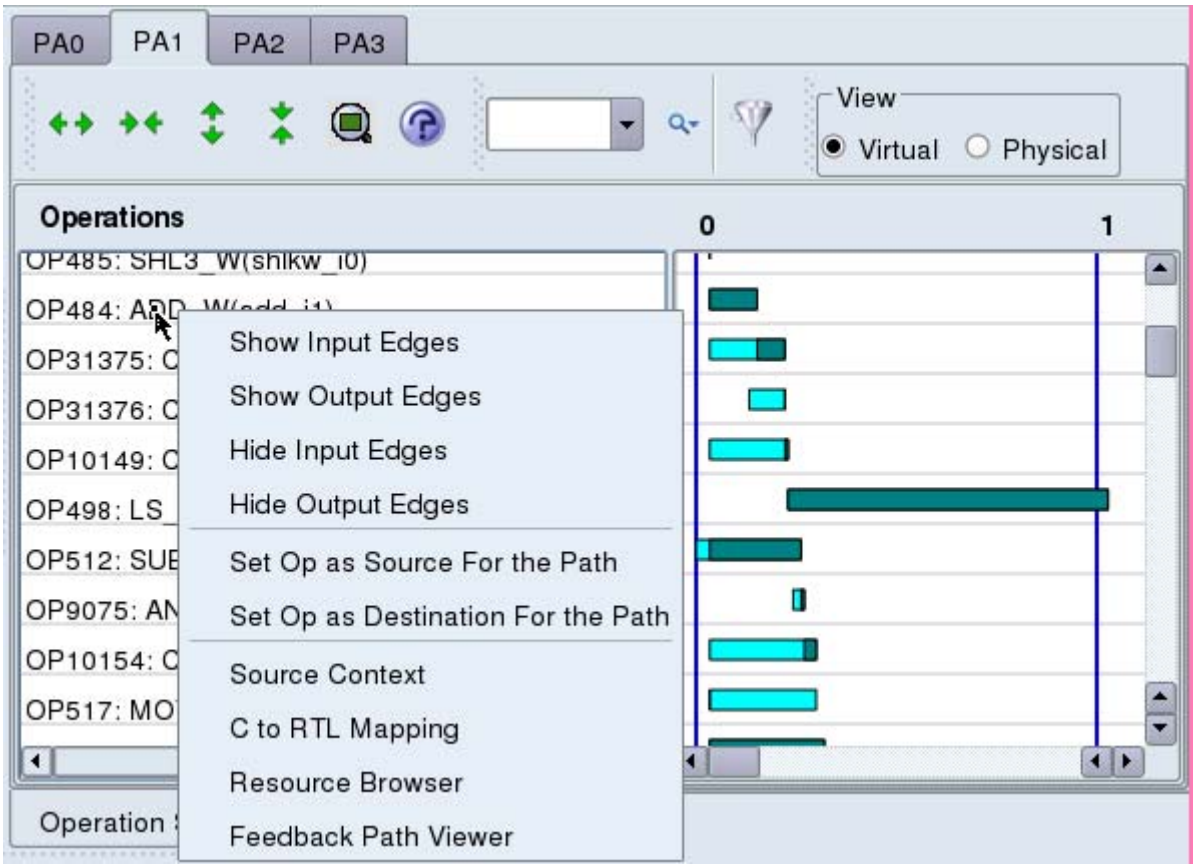
Right-click the header at the top of the Operations list or cycle graph to show or hide the PseudoCode view. Pseudo code is a detailed description of each operation showing the input and output variables. The strings are available for searching.



For the options specific to an operation, right-click an operation in the cycles area or right-click a row in the Operations or Pseudo Code area. This opens a command menu that lets you do the following:

- Show or hide input or output edges
- Set an operation as the source or destination for the path
- Open a text editor window showing the source context, the C-to-RTL mapping, or a resource browser
- Open the feedback path viewer

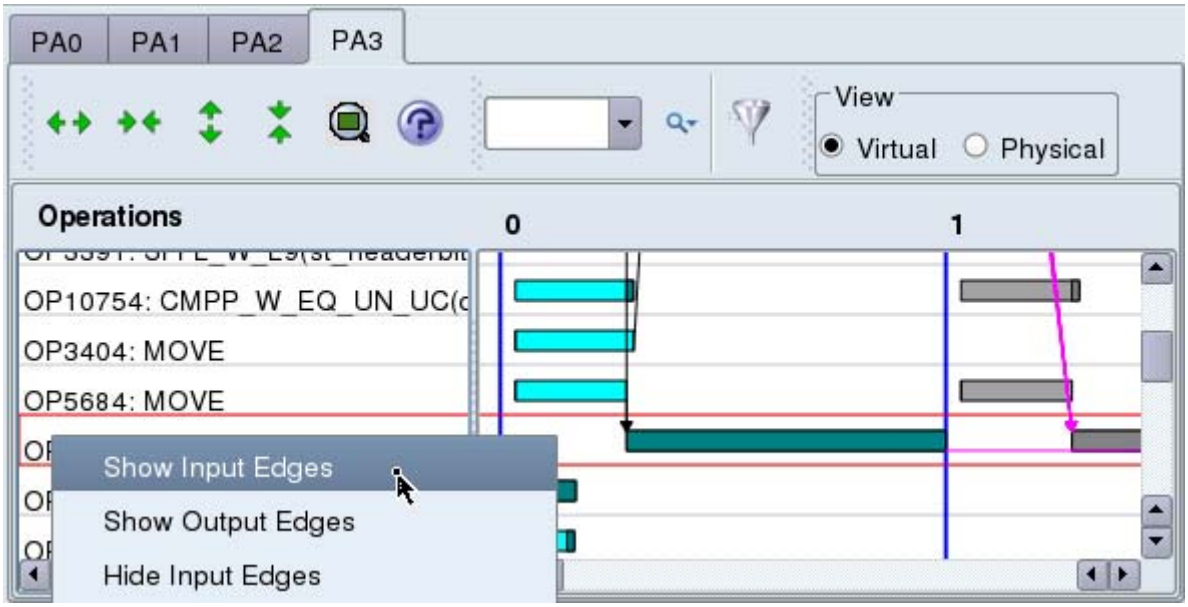
The following figure shows the menu displayed by right-clicking an operation.



Edges and Paths

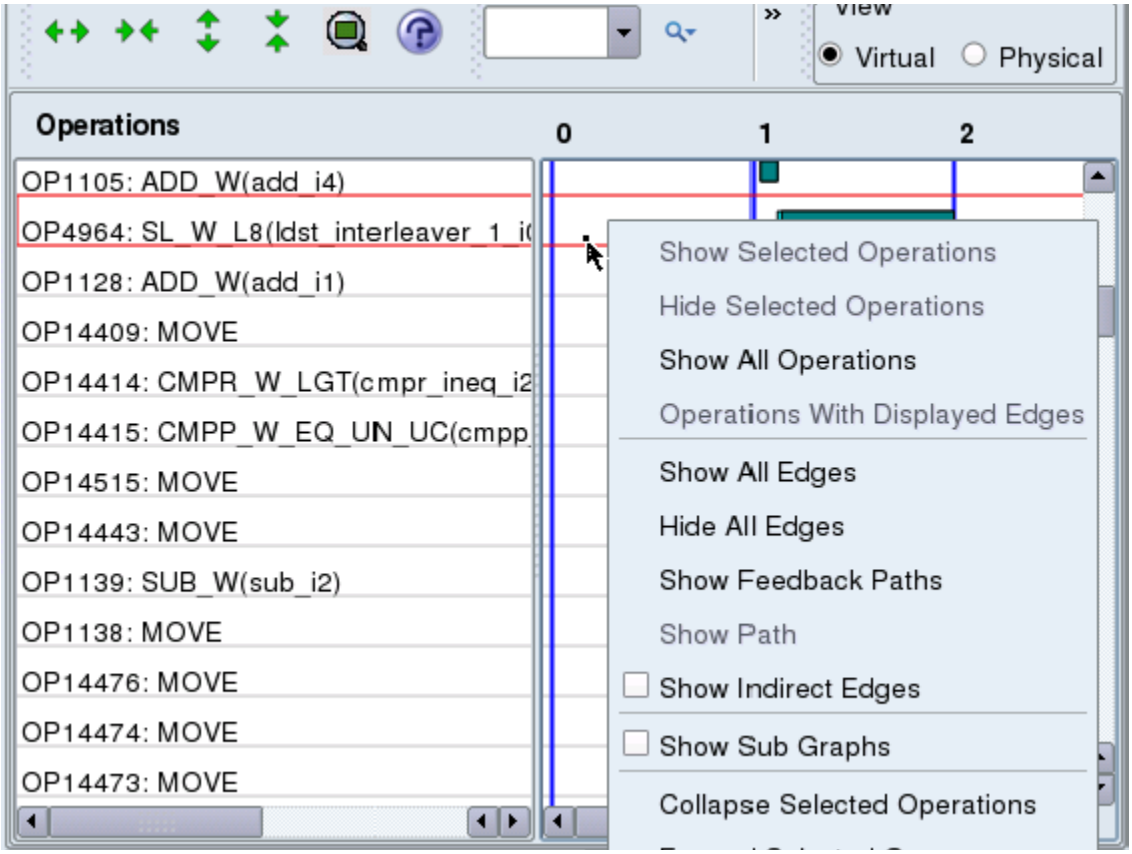
A path is a sequence of edges starting from a source operation and ending at a destination operation. You can show or hide edges, and view paths . To show or hide edges, right-click the operation, and choose the appropriate options such as Show Input Edges and Show Output Edges.

Input edges are displayed as arrows ending at the start of an operation. Output edges are displayed as arrows leaving from the end of an operation. Follow the arrows to traverse a path.



Global Operations

In the cycles view, outside of the operation text, right-click to display a context menu that belongs to all of the visible operations, as shown in the following figure.



The commands perform the actions listed in the following table.

Menu Command	Description
Show or Hide Selected Operations	When you select operations using Ctrl + (mouse click or mouse drag), selected operations can be shown or hidden.
Show All Operations	Un-hides hidden operations.
Operations With Displayed Edges	Only operations that have edges are visible.
Show or Hide All Edges	All edges for the visible operations can be shown or hidden.
Show Feedback Paths	Shows the feedback paths if any exist.

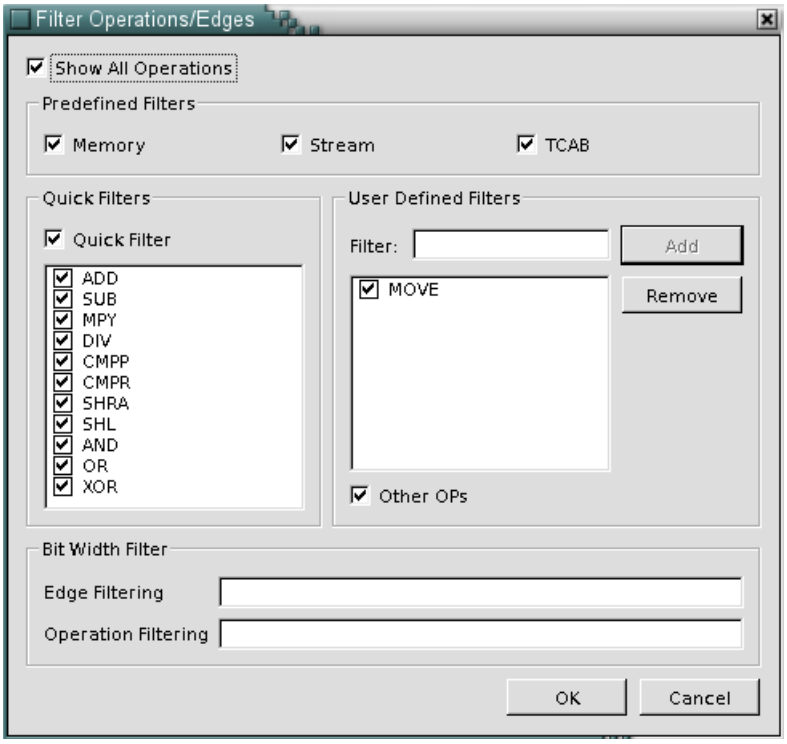
Menu Command	Description
Show Path	When one operation is selected as the source and another as the destination, this option shows the path between them. If operations are hidden between this path, it brings them into view.
Show Indirect Edges	Shows the indirect edges. If any operation is not visible, then indirect edges are shown. Uncheck this option to hide the indirect edges.
Show Sub Graphs	All operations in the subgraphs are collapsed and shown in a single row with an orange color. To expand the subgraphs, uncheck this selection.
Collapse Selected Operations	Collapses selected operations into a single row and shown in an orange color.

Operation Schedule Filter

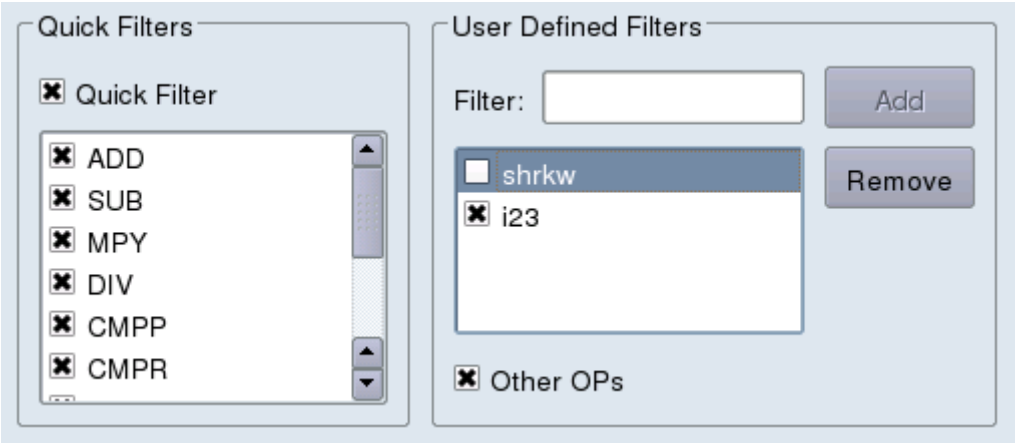
To restrict the list of operations to only those that meet the specified criteria, click the Filter Funnel button. This opens the Filter Operations/Edges dialog box.

By default, the Show All Operations option is selected. Toggle off this option to display only the operations that meet the other criteria specified in the dialog box. To selectively remove certain operations, leave the Show All Operation option selected and deselect the types of operations that you want removed from the display.

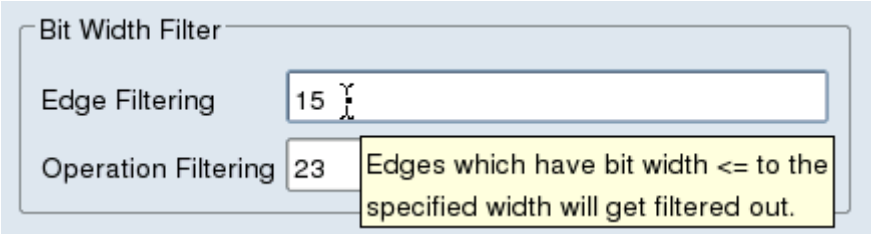
Use the User Defined Filters section of the dialog box to enable or disable the display of operations containing a specified string. Enter the string in the Filter field, then click Add to add the string as a filter. Then, you can enable or disable the display of operations containing that string.



For example, in the following figure, the strings “shkrw” and “i23” have been added as filter strings. The string “shrkw” is deselected, so the operation schedule viewer removes all operations containing that string from the display. The Other OPs option is selected, causing all other operations not listed in the User Defined Filters to be displayed.



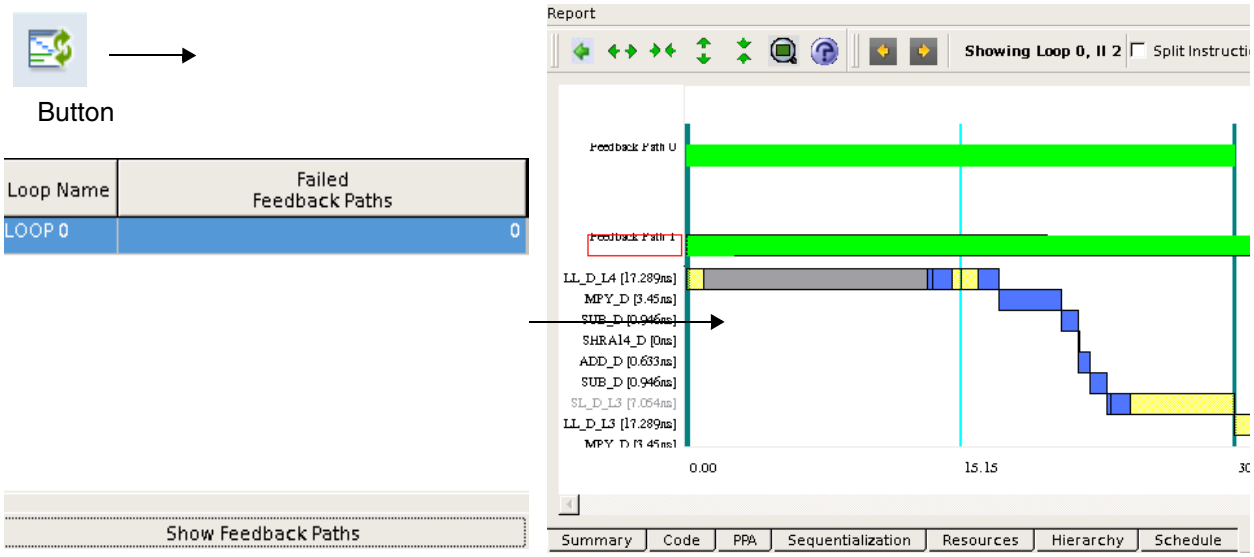
Use the Bit Width Filter section of the dialog box to enable or disable the display of edges or operations having fewer than a specified number of bits. In the following diagram, Edge Filtering is set to 15 and Operation Filtering is set to 23. This causes edges having a bit width of 15 or less to be removed from the display, and operations having a bit width of 23 or less to also be removed from the display.



Feedback Path Viewer

The feedback path viewer shows the critical feedback paths in the design on a per-loop basis. Each feedback path contains details about the operations involved in the feedback, their corresponding source positions (where available), and many other properties. You can analyze the feedback paths and take corrective action so that they are successfully scheduled with positive slack.

To open the feedback path viewer, choose Analysis > Feedback Path Viewer or click the equivalent icon. In the loop selection table, select the desired loop and click Show Feedback Paths.



Loop selection table

Feedback paths in a loop

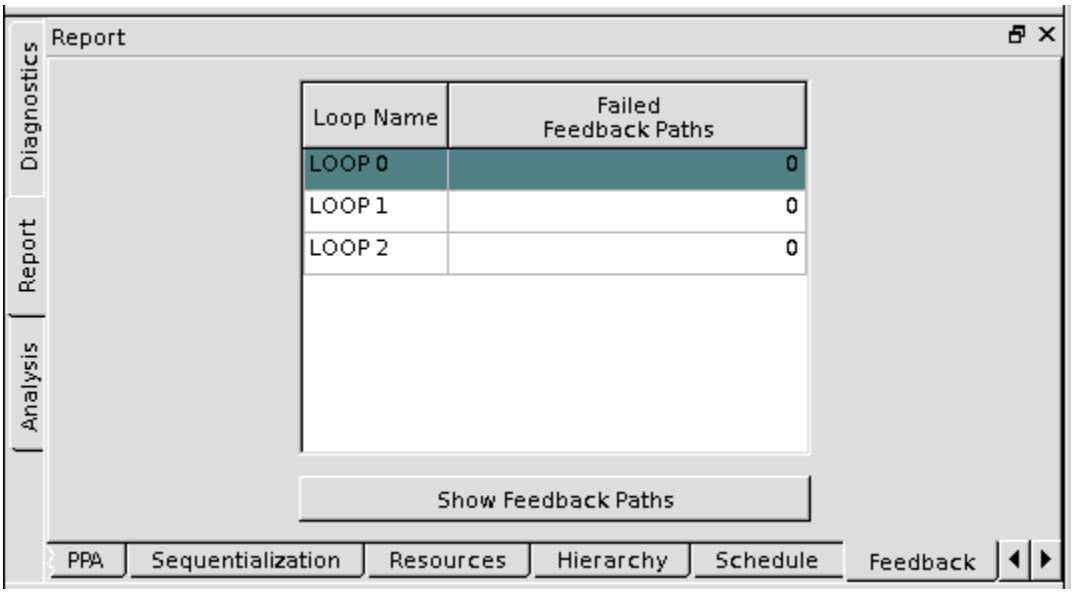
The Feedback Path Viewer displays information about the feedback paths:

- The report shows the information generated early in the synthesize step. You can look at this information to understand iteration interval (II) failures in the application.
- The report also shows information contained in the report_feedback_path.txt file, which is generated after the operations are retimed to get better QoR.

You can see the number of feedback paths that are not scheduled properly. You can select a PA in either of the tabs and click Show Feedback Paths.

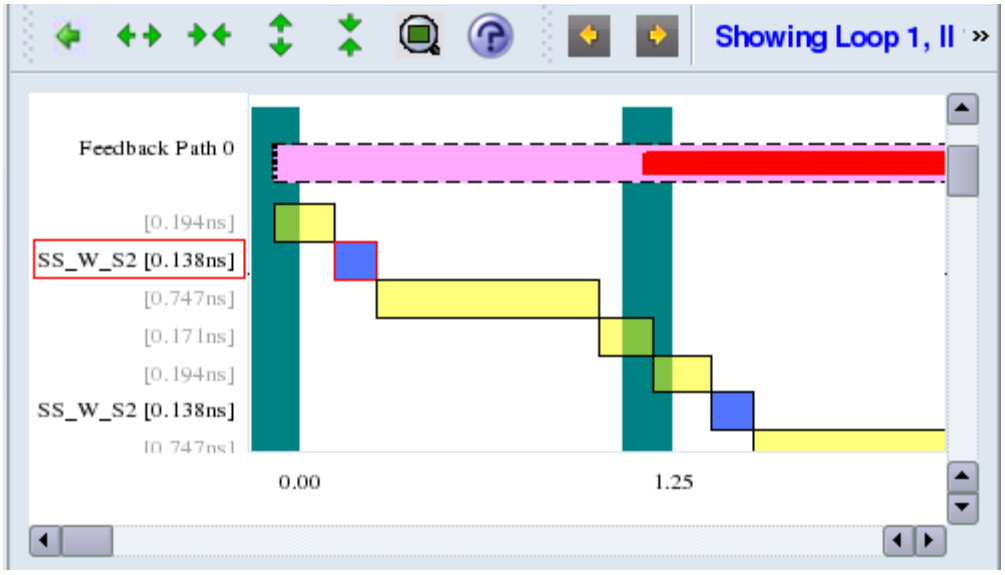
Loop Selection Table

The loop selection table window lists all the loops in the program and the number of failed feedback paths in each loop. Click the Show Feedback paths button or double-click a particular loop name to open an expanded view of the feedback paths.



Feedback Paths in a Loop

This window shows all the feedback paths in a particular loop and instructions for each feedback path. Click the green feedback bar to display or hide the instructions in the path. The bar color indicates slack: positive slack is black and negative slack is red. For a complete list of the bar color conventions, click the help [?] button above the graph.



The Feedback Path window shows the following:

Navigation Icon	Clicking the icon toggles the view to Loop Selection Table.
Zoom Toolbar	Zooms the view horizontally, vertically, or to fit all.
Help Button	Opens a dialog box showing the color conventions used to display various objects.
Navigation buttons	Navigates the instruction within the selected feedback path. The corresponding instruction is highlighted and shows the source context highlight as well.
Loop Name **Showing Loop 0, II 1**	Displays the name of the loop that is being shown.
Split Instructions option	Option to expand or collapse instructions in the graphical display.
Time on x-axis and feedback paths on y-axis	See Time on x-Axis and Feedback paths on y-Axis after this table.

List of feedback paths	Lists all the feedback paths.
Instructions in a feedback path	See Instructions in a Feedback Path after this table.

Time on x-Axis and Feedback paths on y-Axis

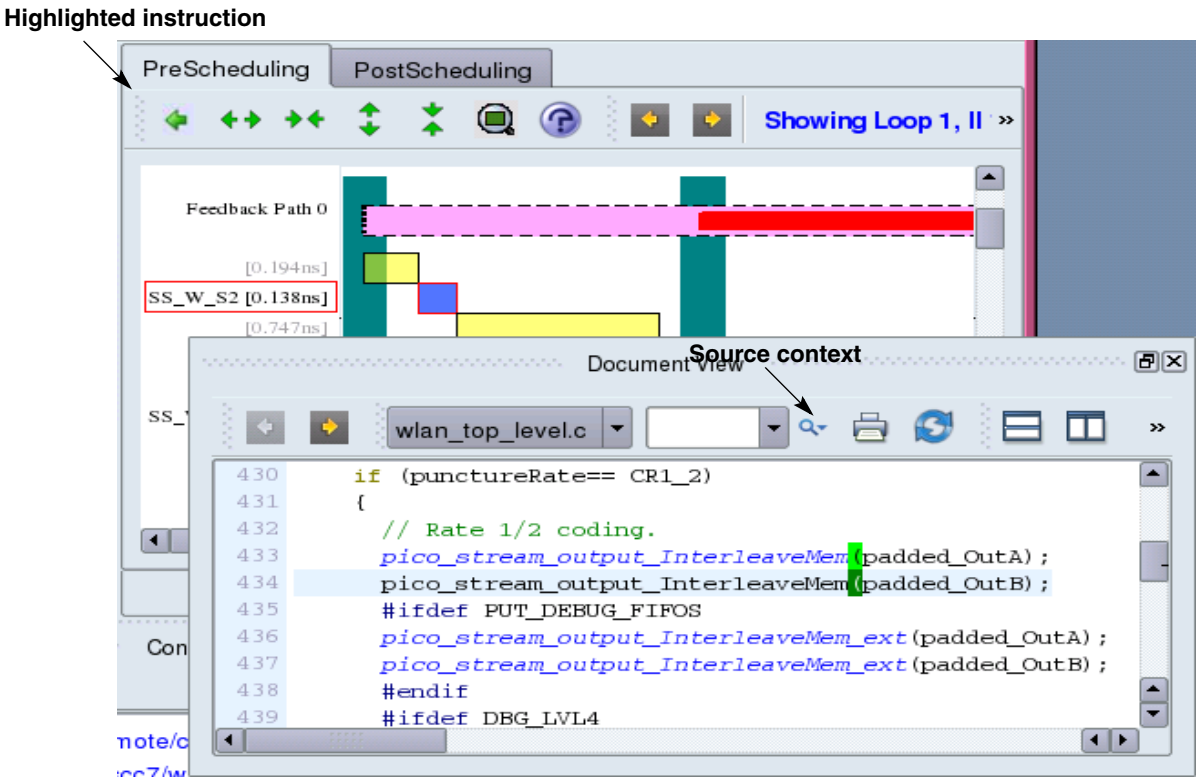
Time in nanoseconds is used as the scale for the x-axis. The initiation interval is marked with a different color to clearly distinguish it.

All feedback paths in the loop are shown in ascending order of the feedback path number, with the lowest number at the top of the y-axis. Feedback paths that pass are shown as green rectangles; failed paths are magenta rectangles with red highlighting of the excess time used.

Instructions in a Feedback Path

Clicking a feedback path expands it and shows its instructions. For each instruction, you see the opcode and time value on the left. A rectangle on the graph indicates its actual position on the timescale.

Clicking on a bar or on the bar's y-axis instruction label selects that instruction and highlights it with a red rectangle. This also displays the source context for the instruction in a document view window. The assignment statements corresponding to the instruction are highlighted in the source code. You navigate forward or backward through the instructions using the yellow-arrow navigation buttons.

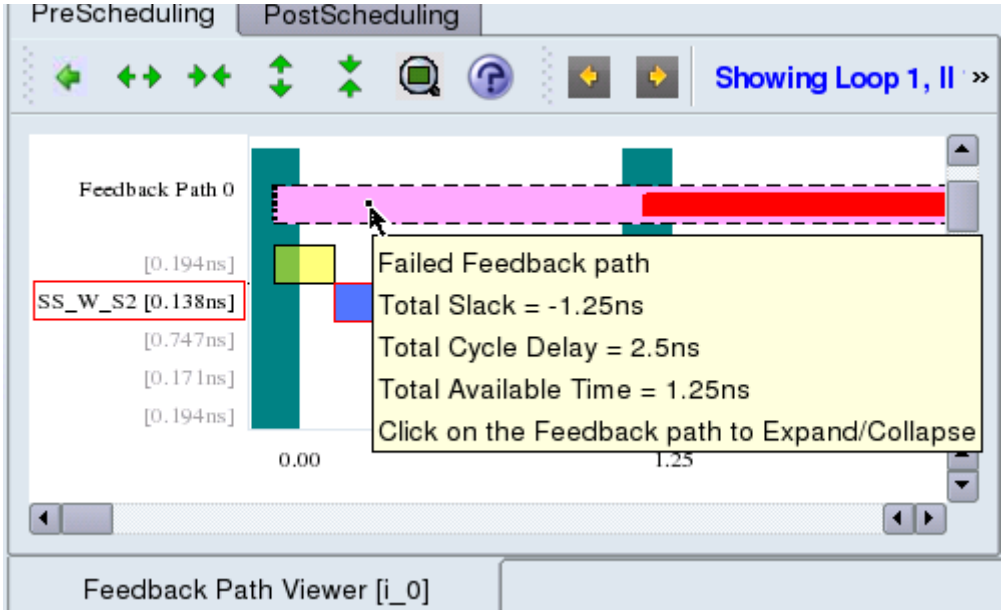


When multiple instructions point to the same source code, you can hover the cursor over the highlight in the source code to view the list of opcodes in a ToolTip window.

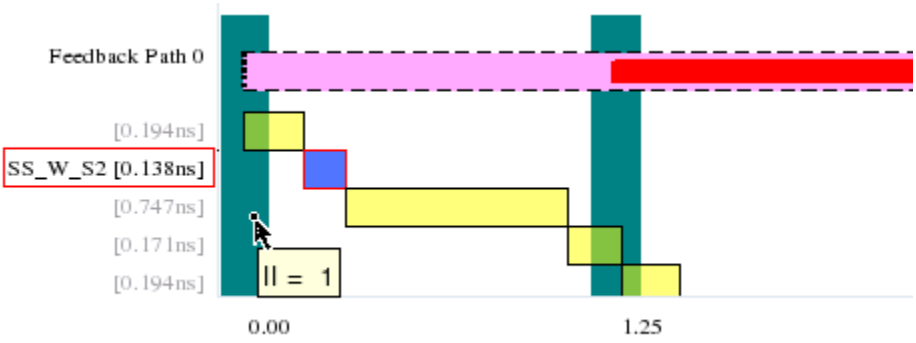
Feedback Path ToolTip

Hover over an object with the mouse to display a ToolTip. The following describe available tooltips for feedback paths:

- Feedback path rectangle and text
  Displays the feedback path scheduling status, total slack time, total cycle delay and total available time for the selected feedback path.

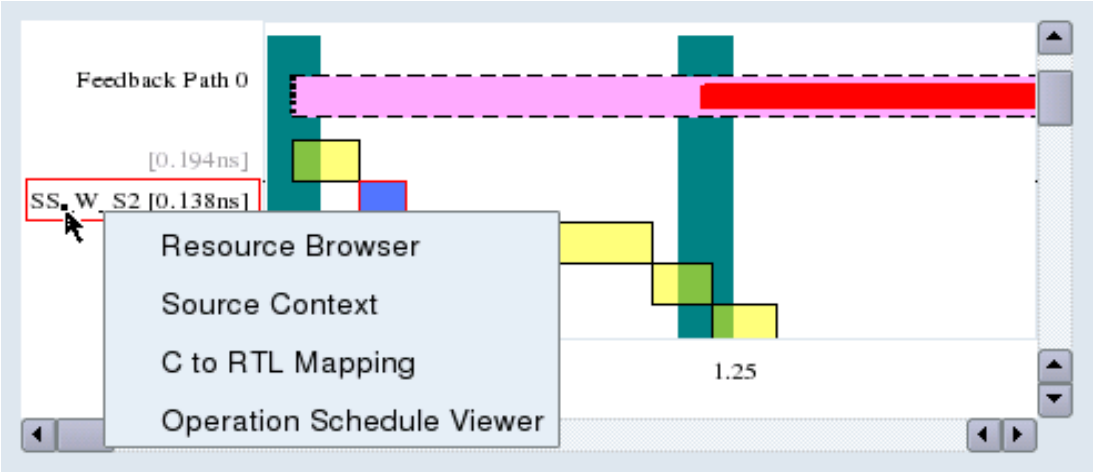


- Initiation Interval
  Displays initiation interval information for that loop:



Context-Sensitive Menu for an Instruction

Right-click an instruction to open a menu with the Resource Browser, Source Context, C to RTL Mapping, and Operation Schedule Viewer commands. Alternatively, to open the C to RTL map viewer, hold down the Shift key and click the instruction.



For more information on these analysis options, see "Resource Browser for Feedback Path View" on page 2-62 or "C to RTL Mapping" on page 2-67 or "Operation Schedule Viewer" on page 2-41.

Resource Browser

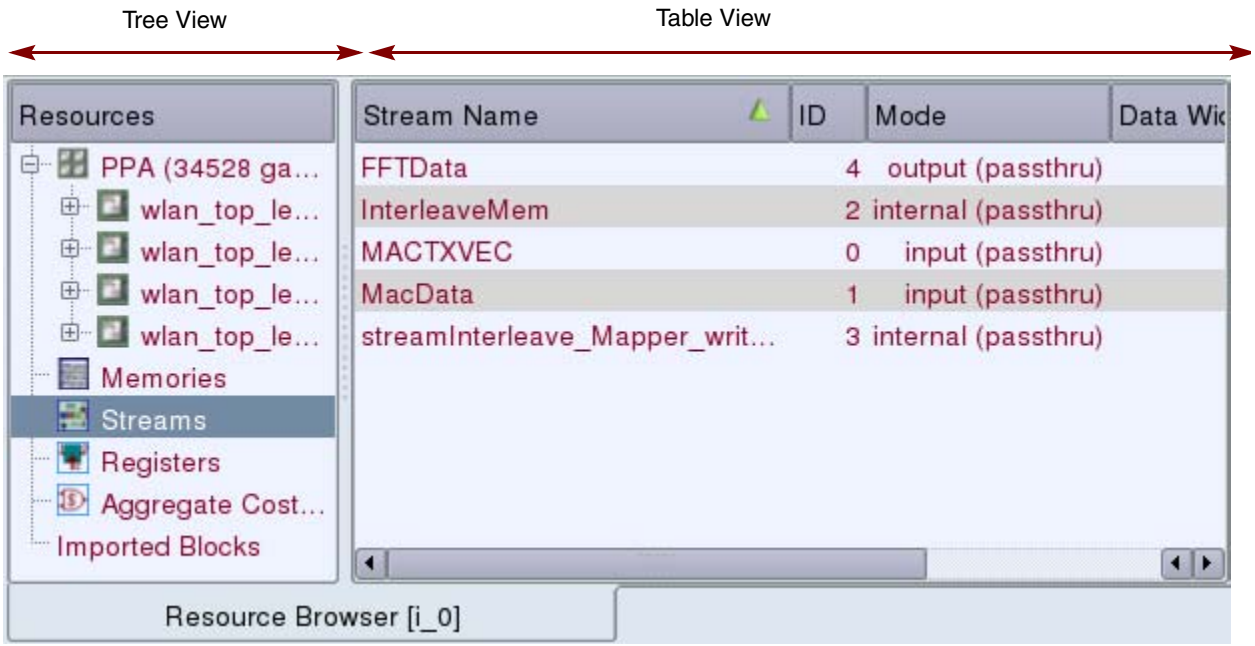
You can invoke this view by selecting Analyze > Resource Browser or by clicking the Resource Browser icon ( ). In addition, you can bring up the Resource Browser from other views to view resources for particular objects.

This view gives a detailed hierarchical list of the hardware resources needed to implement different loops. This view also shows the TCABs that are shared across PAs at the PPA level. The estimated overall cost is in terms of gates for ASIC and in terms of LUTs, FFs, DSPs and BRAMs for FPGA.

When you select Resource Browser from a context-sensitive menu, the information in this window varies, depending on the object and the view from which it was selected. See the following:

- Resource Browser Basics
- Resource Browser for Sequentialization Graph
- Resource Browser for PPA Graph
- Resource Browser for Feedback Path View

Resource Browser Basics



The window consists of a tree view that lists all the hardware resources, and a table view that lists all the properties of a selected item in the tree view.

This section describes the following:

- Hierarchy and Elements
- Navigation
- Element Properties in Tree View
- Context-Sensitive Menu

Hierarchy and Elements

This is a list of elements displayed in the Resource Browser:

- PPA > PA > Module > Instance > All Phases
- Memories > All Memories
- Streams > All Streams
- Registers < All Registers
- Aggregate Cost > Macrocell

Navigation

In the tree view, a node with +/- sign specifies that it is a leaf node and can be expanded or collapsed to display or hide its child nodes. If you select a node in the tree view, its child nodes appear in the table view. Use these techniques to navigate the elements in the browser:

- + or - signs
  Clicking the + sign in the tree view expands the node to show all its child nodes, and clicking the - sign collapses the node and hides all children.
- Double-clicking in the Tree view
  This toggles between expanded and collapsed states.
- Double-click in Table view
  Double-clicking a table row shows all the child properties, if the row has an element representation in the tree view. If the row is not represented in the tree view, double-clicking has no effect.

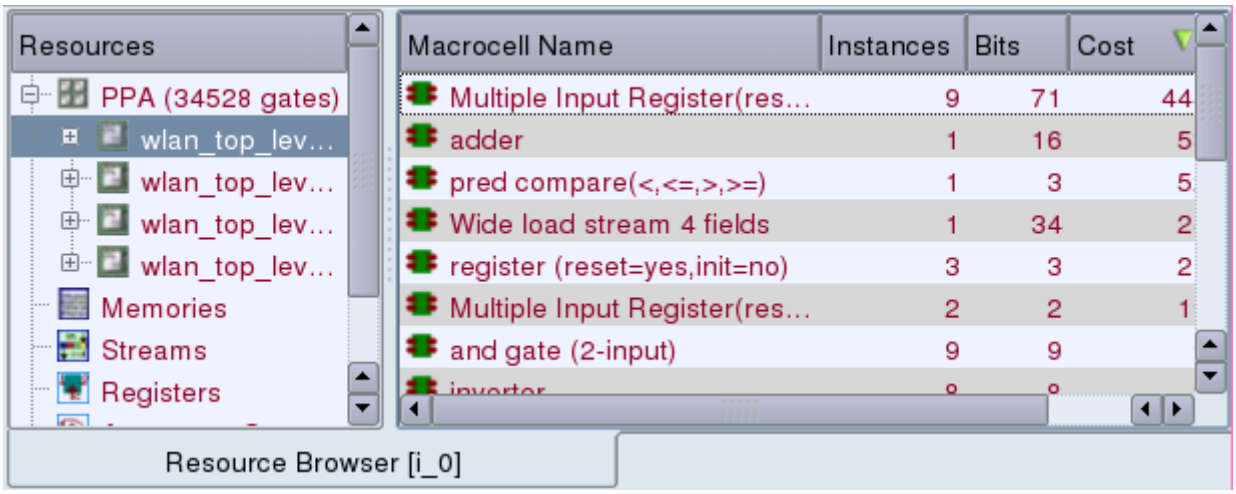
Element Properties in Tree View

This table shows the element in the tree view and their associated properties.

Element in Tree view	Properties in Table view
PPA	PA Name, Initiation Interval, Schedule Length, Iterations, Total Cycles, Total Cost
PA	Module Name, Macrocell Description, Cost, Instances, Bits
Module	Instance Name, Total Phases, Utilization in percentage, Output Bit Width
Instance	Phase, Operation Number, Code Position, Opcode, Input1, Input2, Output1, Predicate
Memories	Memory Name, ID, provider, Type, Words, Width, Buffers, RO Ports, WO Ports, RW Ports, Read Bandwidth, Write Bandwidth, RO Arb Reqs, RO Arb Ports, WO Arb Reqs, WO Arb Ports, RW Arb Reqs, RW Arb Ports,
Streams	Stream Name, ID, mode, Data Width, Depth
Registers	Register Name, Bits, Bits Cumulative
Aggregate Cost	Macrocell Name, Description, Gates, Count, Percent, Percent Cumulative

Context-Sensitive Menu

Right-click in the Table view to select the row and bring up the menu for that element. You only see this menu when an instance is selected in the Tree View.



The Source Context command opens a text editor window with C or C++ source code. The cursor is placed on corresponding source code line and column in the C or C++ source code file. Navigating the Instance Names or Total Phases using the up/down arrow keys or by clicking the items brings up the source context.

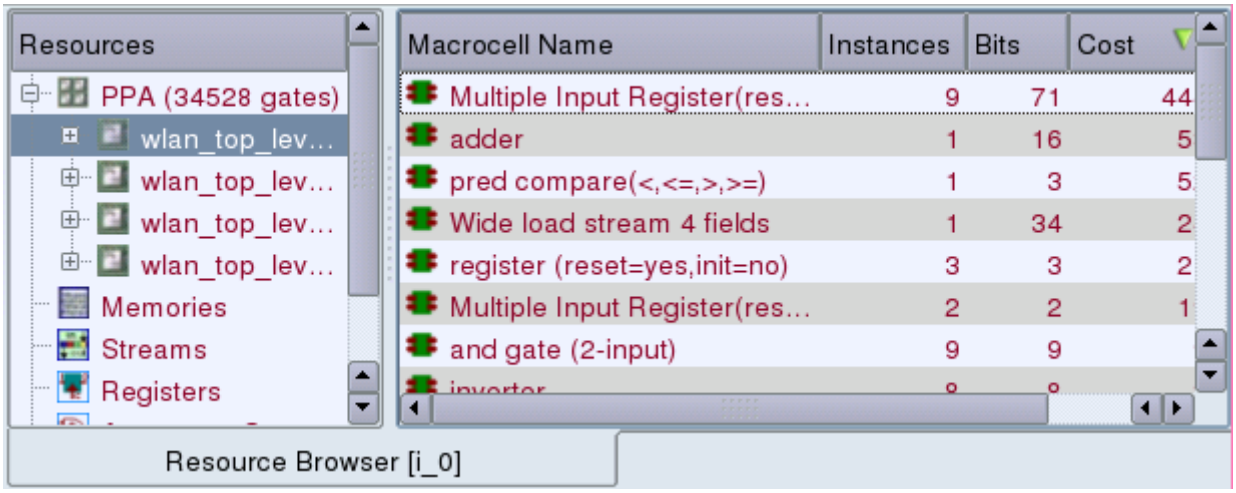
Resource Browser for Sequentialization Graph

The only objects in the Sequentialization graph are loops. Select the hardware resource in the left panel of the Resource Browser to display its properties in the right window, as shown in the following sections.

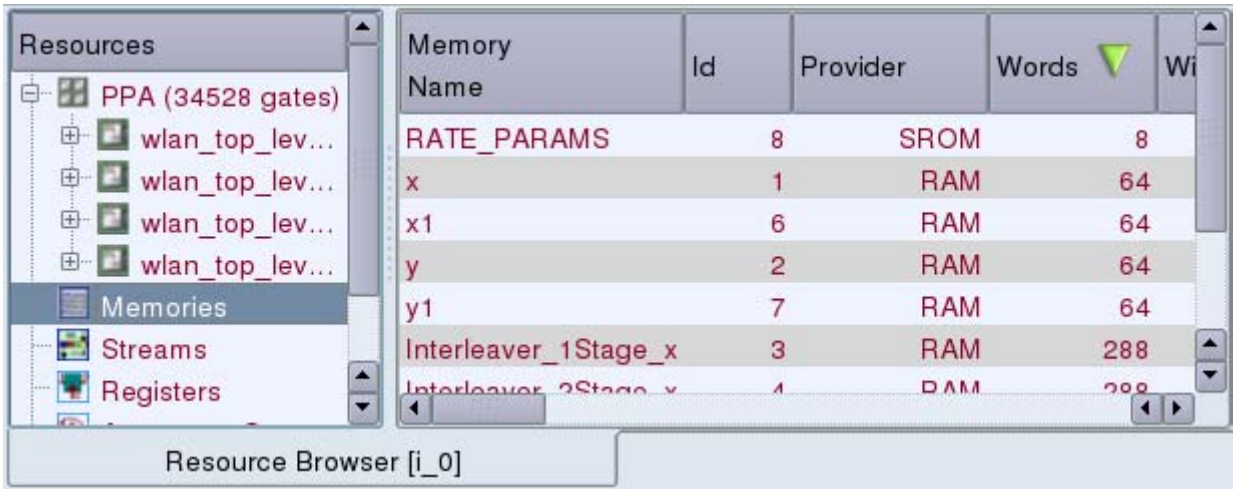
Resource Browser for PPA Graph

Selecting Resource Browser from the PPA Graph displays the Resource Browser with the selected element highlighted in the Resources column. The following figures show the Resource Browser for a Loop, Stream and Scalar.

- For a loop, the hardware resource is selected in the left pane of the Resource Browser and its contents are displayed in the right pane.



- For memories, the Memories node is selected in the left panel of the browser and contents are displayed in the right pane.



- For streams, the Streams node is selected in the left panel of the browser and contents are displayed in the right pane.

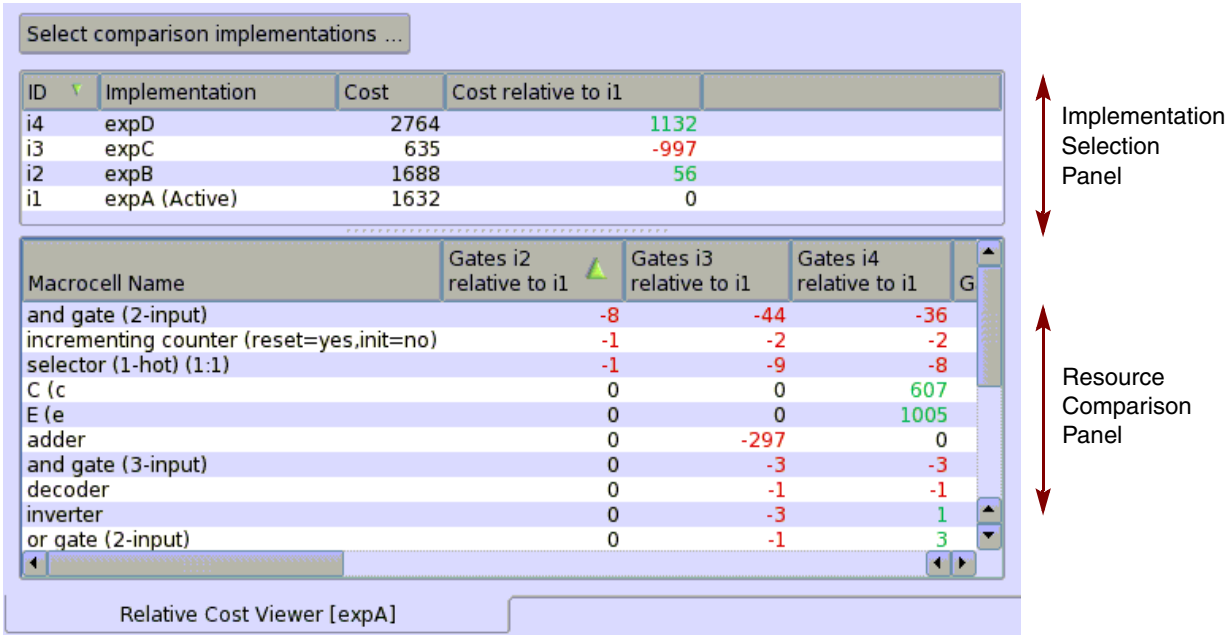
Resource Browser for Feedback Path View

When you select the Resource Browser from the Feedback Path view, it focuses on the selected instruction, as shown:

Relative Cost Viewer

To open the Relative Cost Viewer, select Analysis > Relative Cost Viewer or click the Relative Cost Viewer icon ( ). The Relative Cost viewer has two parts:

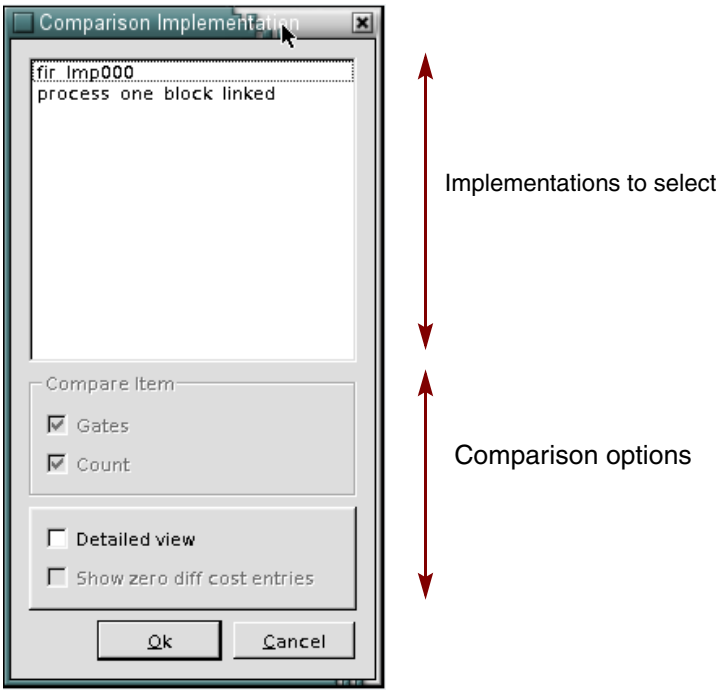
Select Comparison Implementation	Selects the implementations to be compared against the active implementation. Opens the Comparison Implementation dialog box when you click Select comparison implementations. For more information, see "Implementation Comparison" on page 2-63.
Resource Comparison Panel	Shows the differences of functional units instantiated in selected implementations and their cost. For ASIC technology libraries, the cost is represented as gate count. For FPGAs, the cost is represented as the number of LUTs, FFs, DSP and Block RAMs.



The current implementation becomes the base implementation against which all other selected implementations are compared. This implementation is marked as "(Active)" in the implementation selection panel.

Implementation Comparison

The Comparison Implementation dialog box opens when you click Select comparison implementation in the Implementation Selection panel.



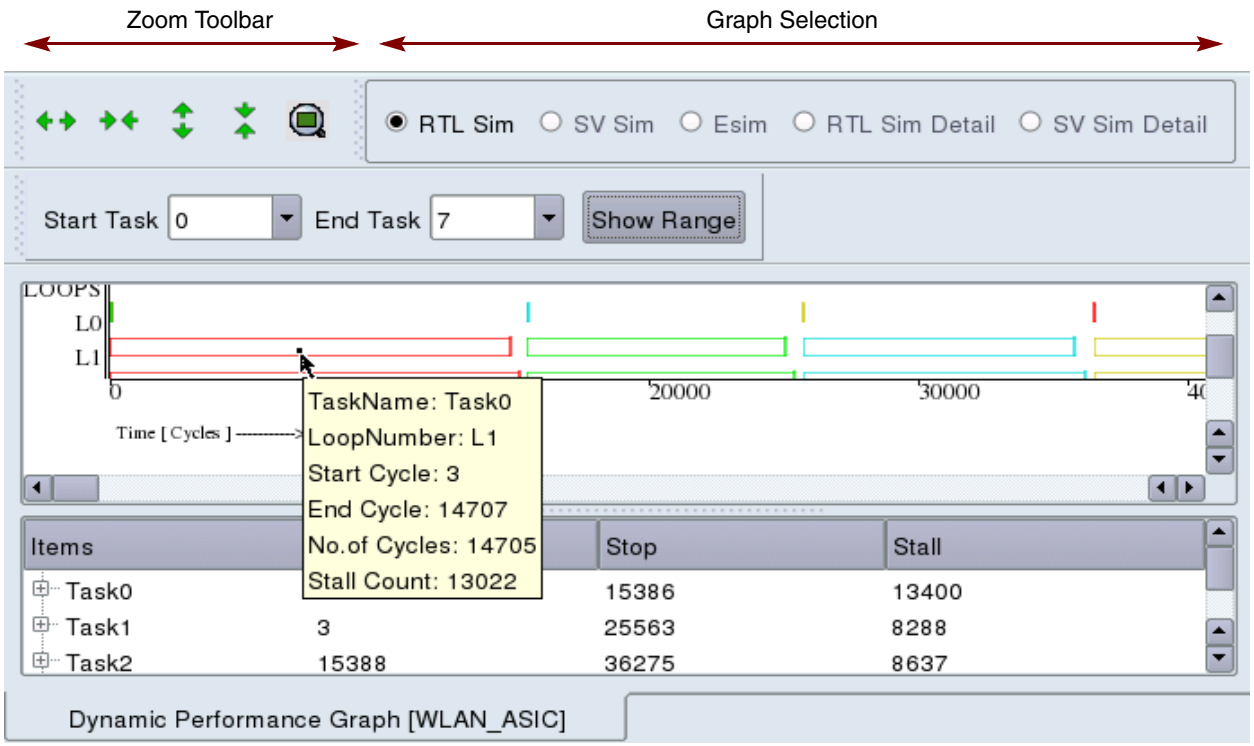
Implementation list	Lists the implementations available for comparison. The implementations share the same technology library. Select implementations to compare against the current implementation. Click implementations in succession to select multiple implementations.
Gates	Enable this to compare the number of gates in an ASIC design.
Count	Enable this to compare the number of LUTs, FFs, Block RAMs and DSPs in an FPGA design.
Detailed View	When disabled (default), all functional units are grouped together based on the functionality and the count and cost is displayed for the group of instances. Enable this option to expand the group and show individual instances instead.

Show zero diff cost entries	When enabled (default), shows zero difference entries. If two implementations have the same cost because they have the same number of instances of a particular kind, their cost difference for the components is zero. The default behavior of the tool is show the zero difference entries. Disable this option to see only the entries that have a cost difference.

Dynamic Performance Graph

To invoke this view, select Analyze > Dynamic Performance Graph or click the Dynamic Performance Graph icon ( ).

This view allows you to analyze the performance of each loop in terms of the number of clock cycles needed for execution. A sample view is shown:



This view has six sub views that correspond to the various simulations: RTL simulation, RTL simulation, event simulation, RTL simulation with detailed performance, and RTL simulation with detailed performance.

This section describes the following features of the view:

- Graph and Task Range Selection
- Task Table
- Tooltips for Loops

Graph and Task Range Selection

You must run at least one simulation before invoking this view. If no simulation has been run, an information message box is displayed. The appropriate button in the Graph Selection group box is only enabled if the corresponding simulation has been run and the XML files are available. The default order of selection is RTL offline simulation, RTL simulation, event simulation, detailed RTL simulation, and detailed RTL simulation.

When you select a simulation button, the Task Number combo box in the Task table shows all the valid tasks. Select one of them and click Show Range to refresh the window with the requested task. If you select RTL simulation, the task table displays only one combo box for Task Number.

Task Table

The task table lists the Start, Stop, and Stall cycle counts for every task in the graph. You can expand a task to show all the loops in the task and their corresponding cycle numbers, by clicking the + sign before the task name. Double-clicking the task name expands or collapses the task to show or hide all the loops present in the task.

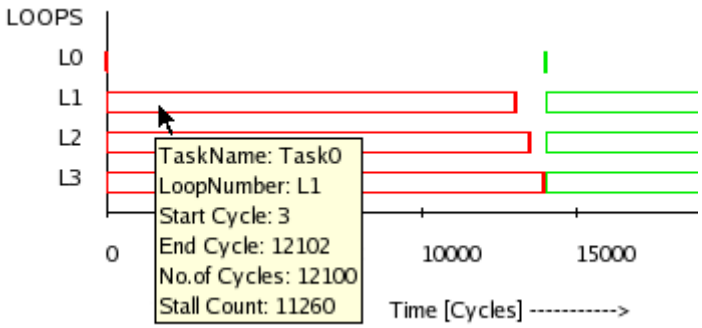
Items	Start	Stop	Stall
Task0	2	12732	18730
Task1	12733	21131	11806
Task2	21132	28600	8924
Task3	28601	33645	5546
Task4	33646	39142	6181
Task5	39143	42911	3808
Task6	42912	47544	5609
Task7	47545	51313	4338

Tooltips for Loops

Hovering over a loop item with the mouse displays a ToolTip with the following information:

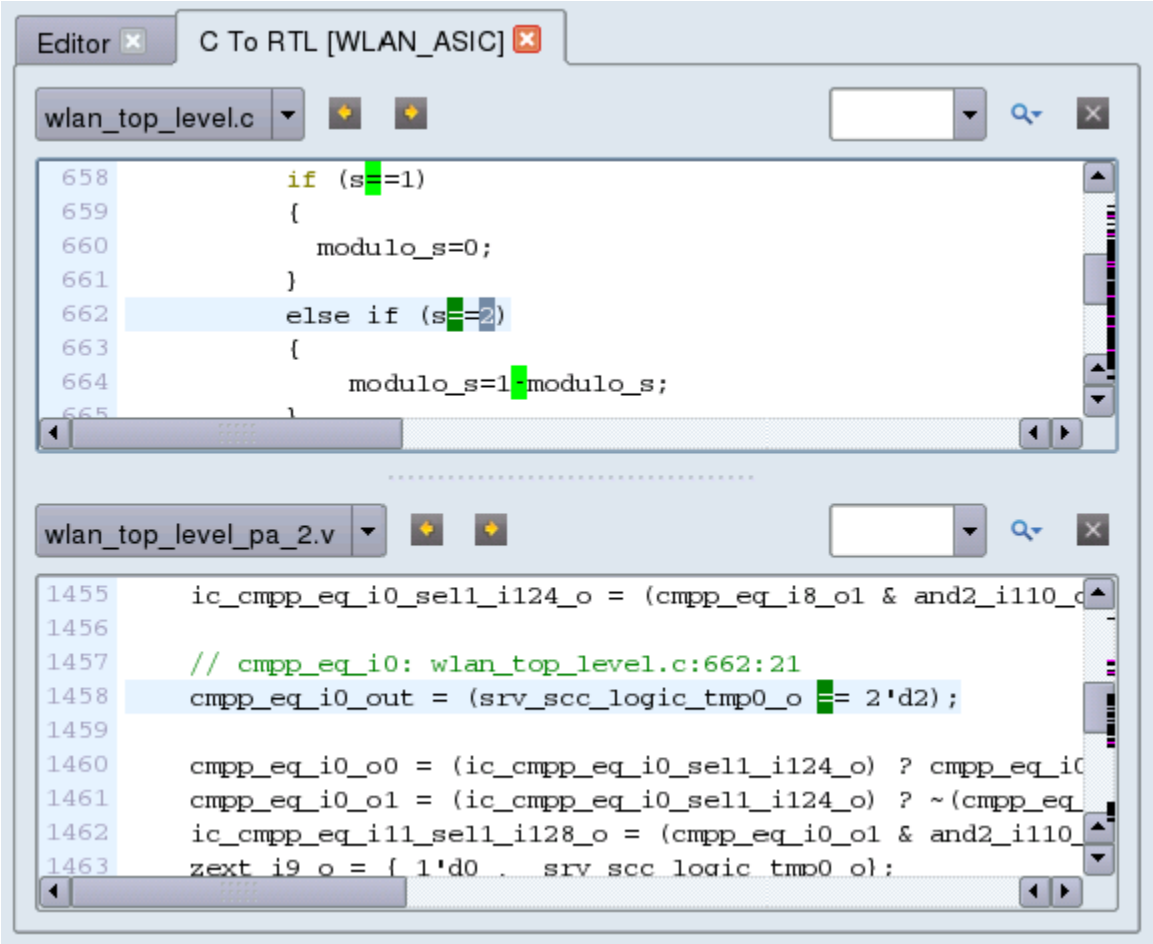
- Task Name
- Loop Number
- Start Cycle

- End Cycle
- Number of Cycles
- Stall Count



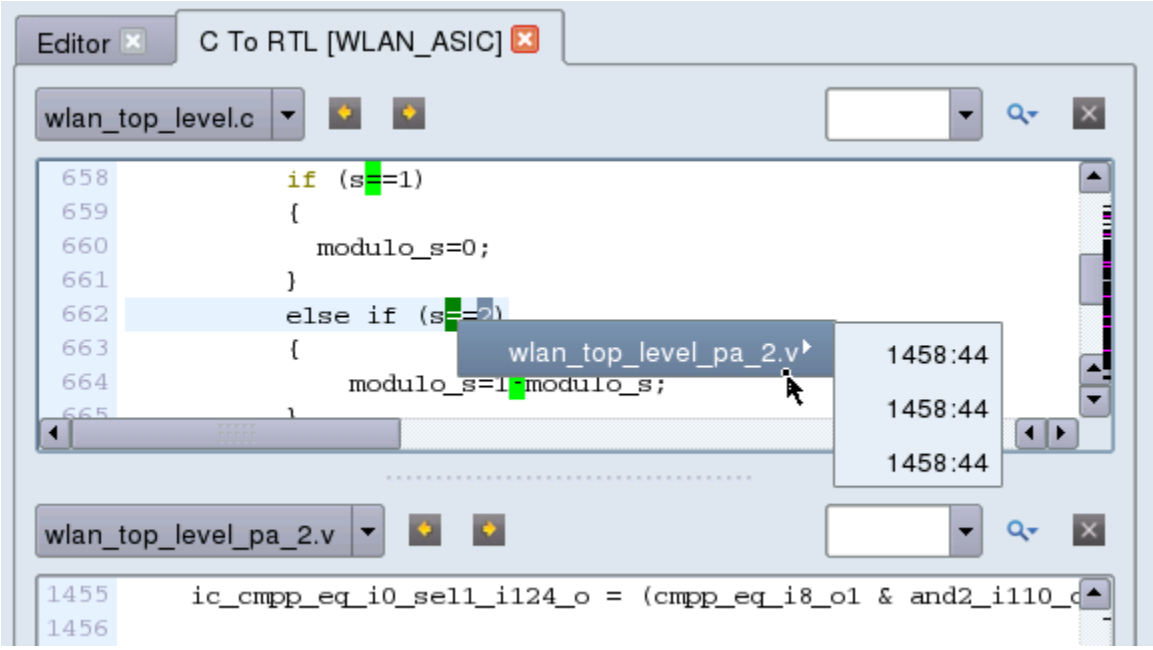
C to RTL Mapping

The C to RTL Mapping feature lets you compare the generated RTL code against the original C source code. Launch the window using Analyze > C to RTL Mapping. It displays a split window with the two files, where you can view the C to RTL mapping.



You can navigate and cross check the C code and RTL description in several ways:

- In the C file, click a green area to highlight its equivalent RTL code. If there are multiple mappings in the RTL code, then all mappings are highlighted in red and the first mapping is highlighted in dark green. If you click the same highlight again in the C file, the dark green highlight advances to the next mapping in the RTL code.
- In the RTL file, click a green area to highlight its equivalent C code. If there are multiple mappings in the C code, then all mappings are highlighted in red and the first mapping is highlighted in dark green. If you click the same highlight again in the RTL file, the dark green highlight advances to the next mapping in the C code.
- For each highlight, right-click to open the context menu, which shows you a list of files and the *row:column* within each file related to the selected highlight.



From the context menu, select the desired file and *row*:*column* to view the related mapping on the other side.

- The magenta dots in the scroll bar indicate the positions of all highlights in that file. A dense concentration of highlights is displayed in black instead of magenta.
- In either window, use the yellow-arrow navigation buttons to scroll to the previous or next cross-referenced highlight in the file.
- Search for a text string by entering the text in the search field. To change the search options, use the pull-down menu on the left of the search field. The options are Case Sensitive, Whole Words Only, Search on Enter, and Single Search.
- To change the text window arrangement between horizontal (side-by-side) tiling and vertical (one above the other) tiling, right-click in the area just above the text window.
- To navigate to other C and RTL files by name, use the drop-down box at the top.

3

Synphony C Compiler Commands

This chapter describes the Tcl commands available in the Synphony C Compiler tool. See the following for details:

- About the Command Line Interface
- Command Descriptions
- Additional SCC Options
- Synphony C Compiler EV Commands

About the Command Line Interface

In addition to using the GUI, you can enter equivalent commands at the command line. This section describes general information about these commands.

- Command-Line Entry
- Tool Startup Commands
- Commands by Type
- Alphabetical List of Commands

Command-Line Entry

The line-interface commands of the Synphony C Compiler tool share these two characteristics:

- You can get more information about the command syntax by typing the command with the `-help` option. This gives you a list of all the options for that command.
- You do not have to type in the entire command text. You can use an abbreviated version, typing in as much of the first few characters as needed to distinguish it as a unique command or command option.

Tool Startup Commands

After the Synphony C Compiler tool is installed at your site, you can launch the line-interface mode by invoking the top-level command from the command line. The top-level commands are as follows:

ASIC	`synphonycc [-help|-h] | [-version|-v] | [-report] | [-gui|-g] | [-cui|-c] | [-log` *`directory`*`] | [-noSessionLog] | [-x command] | [-f script] | [`*`script`*`]`
FPGA	`synphonycc_fpga [-help|-h] | [-version|-v] | [-report] | [-gui|-g] | [-cui|-c] | [-strict] | [-log` *`directory`*`] | [-noSessionLog] | [-xilinx|-altera] | [-x command] | [-f script] | [`*`script`*`]`

The following table contains descriptions of the command arguments:

`-help`	Prints the syntax for the command.
`-version`	Prints the version of the Synphony C Compiler tool that is installed.
`-report`	Prints out relevant information on your machine and environment. This helps to debug issues like the following: • Installation failure • Failure to get a license • Failure to run or find a Verilog simulator • Environment or path issues • Machine OS or version issues • Machine capacity or runtime issues
`-gui`	Starts the Graphical User Interface. This is the default option.
`-cui`	Starts the command line interface.
`-log` *`directory`*	Specifies the name of the directory for the Synphony C Compiler tool to log its session.
`-noSessionLog`	Disables the Synphony C Compiler session log. The -log *directory* and -noSessionLog options are mutually exclusive and cannot be used together.
`-f script`	Runs the specified script through the Synphony C Compiler tool in noninteractive non-GUI mode.
`-x command`	Runs the specified command through the Synphony C Compiler tool in noninteractive non-GUI mode.
`-xilinx`	(FPGA only.) Starts the Xilinx FPGA version of the tool.
`-altera`	(FPGA only.) Starts the Altera FPGA version of the tool.
`script`	Runs the specified script through the tool in non-interactive non-GUI mode.

Commands by Type

The command-line commands are described in "Command Descriptions" on page 3-6. The following table groups the commands into several categories.

Project Commands	Commands to create a new project, open an existing project, close a project, and delete a project • new_project • open_project • close_project • delete_project
Implementation Commands	Commands to create new implementation, select, link, lock, and export implementation and to set the parameters for the implementation • new_implementation • select_implementation • link_implementation • lock_implementation • unlock_implementation • export_implementation • delete_implementation • get_implementations • get_defaults • get_params • reset_defaults
RTL Generation Commands	Commands to analyze the source files and generate the RTL. Also contains command to restore the RTL package to its original state • analyze • build • reset_rtl_package
RTL Synthesis Command	Commands to run RTL synthesis • rtl_synthesize (ASIC) • rtl_synthesize (FPGA) • estimate_power

RTL Simulation Command	Runs RTL simulation. • verify
RTL Reporting Commands	Generates summary and detail reports for cost, performance, timing, and address maps. • report_implementation
Miscellaneous Commands	• show • view_waveform

Alphabetical List of Commands

The list of the command-line commands.

- analyze
- build
- close_project
- delete_implementation
- delete_project
- estimate_power
- export_implementation
- get_defaults
- get_implementations
- get_params
- help
- link_implementation
- lock_implementation
- new_implementation
- new_project
- open_project
- quit
- report_implementation

- reset_defaults
- reset_rtl_package
- rtl_synthesize (ASIC)
- rtl_synthesize (FPGA)
- select_implementation
- show
- unlock_implementation
- verify
- view_waveform

Command Descriptions

The commands are in alphabetical order. For information about tool invocation commands, see "Tool Startup Commands" on page 3-2.

analyze

The `analyze` and `build` commands are the two important steps in building your design. The analyze command compiles the files that you specify and generates intermediate files that are used by subsequent build and verify commands.

`analyze [`*`implementation_name`*`]`

build

Runs all the functions for the current implementation.

Syntax

`build [imp] [-from fromimp]`

imp	Name of the implementation to build.
-from fromimp	Build the implementation from `analyze` output of the `fromimp` implementation. `fromimp` must be an implementation whose target is higher in the call graph of the program and which imports the current implementation.

Description

The `analyze` command is a required step for the `build` command to be run. However, it is possible to use the result of `analyze` step from another top-level implementation to run a `build` command on the current implementation. Therefore, if you plan to build from another implementation's `analyze` step, use the following command:

```
build -from fromimp
```

If you plan to use your local `analyze` run itself, run only the `build` command.

Example

```
build imp <implementation_name>
```

---

close_project

Closes an open project

Syntax

`close_project` *[project_name]*

Example

```
close_project
```

---

delete_implementation

Deletes the specified implementation.

Syntax

`delete_implementation` *implementation* [*implementation* ...]

implementation	You must specify the name of at least one implementation to avoid deleting the current implementation, which is the default. If you do not provide a name, you get an error message.

Description

Deletes one or more implementation directories and any references to the implementation.

Example

```
delete_implementation impl_1
```

---

delete_project

Deletes the specified project.

Syntax

```
delete_project [-force] projName
```

projName	You must specify the name of the project to delete.
-force	Deletes the project even when the project contains locked implementations.

Example

```
delete_project -force project_1
```

---

estimate_power

Run power analysis on the generated RTL.

Syntax

```
estimate_power
[-file filename]
[-format fileFormat]
[-library path]
[-sv]
[-tool pwrEstimationTool]
```

-file *filename*	Specifies the alternate file name of the file containing the activity data. This file is generated by the `verify -rtl` command.
-format *fileFormat*	Specifies the file format of the activity file. Currently, only the VCD dump format is supported.
-library *path*	Specifies the path to the physical library used for design synthesis.
-sv	Specifies the RTL simulation.

-tool *pwrEstimationTool*	Specifies the power estimation tool to be used. Two tools are supported: • power_theater (Default) • spyglass_power

Description

(ASIC only.) The `estimate_power` command runs power estimation, using the activity data generated by the `verify -rtl` command. Only the VCD dump format is supported for the activity data file. For information about data creation for `estimate_power`, see "verify" on page 3-30.

export_implementation

Exports the current implementation.

Syntax

```
export_implementation
  [implementation]
  [-filename filename]
  [-logs yes | no]
  [-reports yes | no]
  [-hierarchy yes | no]
```

implementation	Specifies the name of the implementation to be exported. The default is the current implementation.
-filename *filename*	Specifies the name of the .tar.gz file in which to store the implementation data. If not specified, the file name is *implementation*.tar.gz.
-logs yes | no	Specifies whether to include the contents of the *implementation*/Log directory in the written .tar.gz data file. The default is no.
-reports	Specifies whether to include the contents of the *implementation*/Reports directory in the written .tar.gz data file. The default is no.

-hierarchy	Specifies whether to generate the settings for building the full hierarchy of the implementation, in addition to the implementation at the current level. The default is yes. If used with the -logs or -reports option, the generated .tar.gz data file includes the contents of the *implementation*/Logs or *implementation*/Reports directory in the .tar.gz data file.

Description

This command exports the current implementation for others to use. The command creates a `*.tar.gz` file containing the implementation data. If there is an existing implementation with the same name, the command overwrites the old file.

The data file written by the command contains the following information:

- The project sources and settings used by the implementation
- The implementation source, header, data, and results files
- The implementation constraints
- The files in the Logs directory (when the `-logs yes` option is used)
- The files in the Reports directory (when the `-reports yes` option is used)
- The log files, report files, and constraints of the entire hierarchy imported into the implementation (unless the `-hierarchy no` option is used)

Example

```
export_implementation -filename myblock1.tar.gz -reports yes
```

This command writes out the current implementation into the file myblock1.tar.gz. The report files in the myblock1/Reports directory are included in the written file.

get_defaults

Gets the default of the specified parameter.

Syntax

```
get_defaults -parameter
```

parameter	Name of the parameter.

get_implementations

Gets the implementation. If you do not specify the -owned or -linked option, then the UI returns the list of implementations (both owned and linked) in the current project.

Syntax

```
get_implementations [-owned] [-linked]
```

-owned	Returns a list of implementations owned by the current project.
-linked	Returns a list of implementations owned by another project.

get_params

Returns the current value of the specified parameter.

Syntax

```
get_params [impl] -parameter
```

-impl	Specifies the name of the implementation.
-parameter	Specifies the name of the parameter.

Description

Returns the current value of the parameter. The current value of the parameter is the value that you set using the `set_params` or `set_defaults` commands. If you did not set the parameter, the tool returns the default.

help

Displays all the commands or help for the specified command.

Syntax

`help [commandName]`

commandName	Specifies the command name. The tool displays help for the command. If you do not specify this option, the tool displays a list of all the supported commands.

Description

The command displays a screen that shows all of the Synphony C Compiler commands, or shows help for a specific command.

You can use the list of commands to identify a command. You can then type the command with the -help argument to view the options associated with that command.

link_implementation

Links the specified implementation to an implementation in another project

Syntax

`link_implementation projectfile from_imp [to_imp]`

projectfile	Specifies the name of the project that owns the implementation, from where the implementation is linked
from_imp	Specifies the name of the implementation to be linked
to_imp	Specifies the name of the implementation in current project

Description

The `link_implementation` command allows an implementation from another project to be viewed and used for hierarchical importing in the current project. Linked implementations can be used in hierarchical builds only if they are locked. For more details about locking an implementation, see lock_implementation.

lock_implementation

Locks the specified implementation for use in another project' s build flow or to avoid accidental changes.

Syntax

`lock_implementation [imp]`

imp	Locks the specified implementation

new_implementation

Creates a new implementation.

Syntax

`new_implementation [options] implementation`

implementation	Specifies the name of the new implementation
-from *implementation*	Specifies the name of the implementation to copy from when you create an implementation from another implementation
-params_only	Copies only the parameter settings from the source implementation into the new implementation

Description

This command creates a new implementation. You can create a new implementation from scratch or use an existing implementation as a starting point.

Example

`new_implementation impl_2 -from impl_1`

new_project

Creates a new project.

Syntax

`new_project [-w workdir] proj_name`

proj_name	Specifies the name of the project
-w	Specifies the directory where implementation files are stored

open_project

Opens an existing project.

Syntax

`open_project` *`proj_name`*

proj_name	Specifies the name of the project to be opened

Example

`open_project` *`proj_1`*

quit

Ends the current tool session.

Syntax

`quit`

Description

This command ends the current session and quits the Synphony C Compiler tool.

report_implementation

Writes out result reports after an implementation is run.

Syntax

`report_implementation [`*`implementation`*`] [options]`

The following table describes the command options in alphabetical order. It is required that you specify at least one option to the command.

implementation	Specifies the name of the implementation for which you want to generate reports. If no implementation name is provided, the command generates reports for the current implementation.

-address_map	Generates a report of the mapping of PPA controls, memories, and inputs or outputs in the memory map.
-array_info	Generates a report of the arrays in the design, and any optimizations done on them.
-c2rtl	Launches the C to RTL viewer utility, which lets you to navigate between the C code and the RTL and compare information. For instance, you can navigate from a variable declaration or a C operator to the corresponding RTL elements in the generated RTL, or navigate from the RTL back to the C source files. To get online help for the viewer utility, type Ctrl-h during the interactive viewer session.
-cost	Generates a detailed report on cost breakdown in the RTL macrocells.
-feedback_path	Generates a report of all feedback cycles in the PPA code and their total weight, which should be non-positive. If the tool discovers a feedback cycle of dependence which does not allow the initiation interval to be achieved for a loop, then this cycle is displayed.
-fu_utilization	Generates a report of all allocated functional units, and their utilization by scheduled operations.
-loop_graph	Generates a report to display the loop graph of the generated PPA as a PDF or a PostScript file.
-loop_unrolling	Generates a report with simple information about loops, like the number of unrolled loops, the number of iterations, and the number of operations created in unrolling them.
-memory_access_trace	Generates a report that compares a sequential memory access trace with a non-sequential or scheduled memory access trace to detect any memory dependence violations.
-opid_rtl_map	Generates a report of the mappings of RTL names to internal operation IDs.
-opid_source_map	Generates a report of the mappings of internal operation IDs to original source code elements.
-original_loop_to_line_number	Generates a report with the line number for each loop in the original code (before perfectization).

-partial_schedule	Reports the operations that were successfully scheduled before the schedule operation failed. This report aids debugging, when used with the associated error messages.
-perfectized_loop_info	Generates a report with the results of perfectization.
-performance [-sv] [-tasks *n* | *m* - *n*]	Prints various performance reports according to the options you specify. • -performance -sv Shows the graphical report produced after the `verify -rtl` command is run. • -performance -tasks *n* / *m -n* Lists specific tasks for which you want data displayed. Alternatively, you can define a range of tasks for which you want to generate a report. The generated files are in the imp/Reports/ directory, and their names reflect their contents: imp_performance_type.{ps|pdf} - without a task specifier, covers all tasks imp_performance_type_n.{ps|pdf} - covers only task n imp_performance_type_m_n.{ps|pdf} - covers tasks in the range m through n.
-performance_constraints	Generates a report showing the synthesis decisions made by the tool that affect performance, such as clock rate, throughput, and latency. You can use the information in the report to determine the performance characteristics of the implementation and to set the performance constraints for subsequent synthesis runs.
-interface	Shows the design interface.
-pseudo_code	Generates a report to display the pseudo code generated from the sorted unrolled schedule report. Use the pseudo code produced in this report to get an approximation of the schedule produced by the tool. However, the tool does not attempt to mirror what the real hardware looks like.
-rate_match	Generates a report on the results of rate matching including automatic loop initiation-interval determination from the overall Task_II specification.
-register	Generates a report with detailed information about registers produced in the implementation, such as source mapping, width, and type.

-register_source_map	Generates a report of the mapping between internal register names and variable names in the original code.
-rolled_schedule	Generates a report of the rolled schedule of operations in initiation-interval steps.
-sorted_unrolled_schedule	Generates a report of an unrolled schedule of operations, sorted by time.
-stream_info	Generates a report about the streams in the design.
-summary	Generates a summary report of basic cost and performance data.
-timing	Generates a report of the top critical paths, as analyzed by the SCC tool.
-unrolled_schedule	Generates a report of an unrolled schedule of operations, in original program order.
var_map_rtl	Generates a report of the mappings between the variable names in the original code and the RTL names.

Description

This command generates one or more result reports after an implementation is run. All generated reports for an implementation are saved under the Reports sub-directory within that implementation.

You must specify at least one option for this command.

Example

```
report_implementation impl1 -cost -timing -address_map -rolled_schedule
```

reset_defaults

Resets the value of the specified parameter to the tool default .

Syntax

```
reset_defaults [-parameter]
```

-parameter	Specifies the parameter that needs to be reset.

reset_rtl_package

Resets the RTL package to its original state.

Syntax

```
reset_rtl_package impl
```

impl	Resets the RTL package to its original state

Description

Resets the RTL package to its original state. If you do not specify the implementation name, the command uses the implementation set by previous `select_implementation` command.

Example

```
reset_rtl_package impl_1
```

rtl_synthesize (ASIC)

Runs RTL synthesis on an implementation.

Syntax

```
rtl_synthesize
[-tool synopsys_asic|cadence_asic]
[-level levelName]
[-scriptname filename]
[-variable name=value]
```

-tool synopsys_asic | cadence_asic	Specifies the synthesis tool. The default it synopsys_asic.
-level *levelName*	Specifies the effort level for each tool. *levelName* can be dc, dcultra, physopt, base, and pro. If -level is not specified, the default is dcultra, used with synopsys_asic.
-scriptname *fileName*	Specifies the name of an alternative top-level execution script.

-variable *name=value*	Specifies variables for the synthesis run, where *name* is one of the following: scc_design_name, scc_clk_period, scc_clk_period_in_ps, scc_half_period, scc_clk_transition, scc_clk_uncertainty,scc_in_delay, scc_out_delay, scc_default_load, scc_synthetic_library, scc_tech_lib_db, scc_tech_lib, scc_driver_cell, scc_driver_pin, scc_rtl_filelist, scc_macrocell_filelist, SCC_RTL_SYNTH_DONT_FLATTEN_BLOCKS, scc_tech_wireload_mode, scc_tech_wireload_selection_group, scc_user_cmd_before_compile, scc_user_cm, d_after_compile, or scc_tech_gate_scaler.

Description

Invokes the backend ASIC synthesis tool. You can specify the Synopsys Design Compiler tool or the Cadence® synthesis tool.

Example

```
rtl_synthesize -tool synopsys_asic -level pro -variable scc_out_delay=12

rtl_synthesize -variable scc_tech_lib_db=file1.db:file2.db -variable
scc_clk_uncertainty=0.4
```

rtl_synthesize (FPGA)

Invokes the backend FPGA synthesis tool.

Syntax

```
rtl_synthesize [-tool vendor_fpga|synplify_fpga] [-logic_synthesis_only]
```

-tool vendor_fpga | synplify_fpga	Specifies the FPGA tool to be used for RTL synthesis. You can specify vendor_fpga (the default) or synplify_fpga.
-logic_synthesis_only	Runs only logic synthesis and does not perform any design mapping or partitioning.

Example

```
rtl_synthesize -logic_synthesis_only
```

select_implementation

Selects an implementation.

Syntax

`select_implementation` *`implementation`*

implementation	Specifies the implementation.

Description

This command selects an implementation and makes it the current implementation. You can then run other implementation commands on the current implementation.

Example

`select_implementation` *`pronto`*

set_params

Set the parameters for the implementation.

Syntax

`set_params [impl] [-a] [options]`

where,

`impl` is the implementation name,

`-a` appends the values that are already present in a parameter instead of overwriting,

`options` as described in the following tables:

Description

Set the specified parameter for the implementation.

Basic Settings

Options	Description

-target name	Specifies the procedure name for the PPA and TCAB implementations. For the channel implementation, specifies the channel class name, that should be used as the target for synthesis.
-type ppa | tcab | channel	Specifies that the design should be synthesized as a PPA, TCAB or a channel. When you do not specify this parameter, the tool builds the design as a PPA, by default.
-clock_freq n	Specifies the clock frequency in MHz. The default is 100 MHz.
-design_intent import | deploy	Specifies whether the design should be built for importing or for deploying in another design. If you do not specify this option, the tool uses import by default for TCABs and channels, and deploy for the PPAs. In hierarchical designs, a design built with deploy option cannot be imported in another hierarchical design.
-techlib path	Specify the target technology library file for implementation
	Source File Settings
-source_dir directory	Specifies the location of your input files such as source files, header files, data, reference, and restult files.
-sources "filename1, filename2, ..."	Specifies the names of the source files
-headers "filename1, filename2, .."	Specifies the names of the header files
-data "filename1, filename2, ..."	Specifies the names of the data files
-results “result_file[:reference_file]..."	Specifies the names of result files that are useful for post-simulation comparison.
-appfiles all | "filename1 filename2 ..."	Specifies a subset of source files that are application files. By default, all source files are considered as application files
	Performance Settings
-ii number | "loop=loop_num1:n1 loop=loop_num2:n2"	Specifies either a loop specific II or global II. Specify “” for resetting the previous specification.

-max_schedule_length number | "loop=lopp_num1:n1 loop=loop_num2:n2"	Specifies loop specific or global max_schedule_length. Applies to all loops for which loop-specific information is not specified. Use "" for resetting this specification.
-iterations number | "loop=loop_num1:n1 loop=loop_num2:n2"	Specify loop specific or global iterations (applies to all loops for which loop specific information is not specified). Use "" for resetting this specification
-wait “label1:label2:sched1": TYPE:CONSTR_TYPE	Specify relative schedule time between the two SCC labels. Possible values for TYPE are EQ, LT, GT, LTE, GTE. EQ is default. Absolute schedule time in inferred when you omit the first label. The CONSTR_TYPE implies the constraint type and can be "hard" or "soft". Default is hard. Last two fields are optional.
-task_overlap always | infer | never	Specifies the extent of task overlap across PAs. never implies no task overlap. When you specify "infer/always" Synphony C Compiler decides the extent of task overlap that is necessary to meet the task_ii. Default is "infer".
	Hardware Settings
-rtl_clock_freq n	Specify clock frequency in MHz to use with rtl_synthesize. Default setting is "".
-host_address_map filename	Specify host memory mapping file
-clock_gating yes | no	"yes" implies that clock gating enable outputs are provided at the hardware interface. Default is "no"
-fifo_hierarchy priority	Possible values for priority are : <top:src:dest>, <top:dest:src>, <src:top:dest>, <src:dest:top>, <dest:top:src>, <dest:src:top> For example, when you specify <dest:src:top> the tool tries to place the FIFOs in the reader or writer or in the top PPA layer, in that priority order. You can specify this option only for PPA and CHANNEL implementation.
-generate_abort_logic yes | no	Specify whether or not to generate abort logic. Default is "no".
-init_data_registers yes | no	Specifies whether datapath register need to be initialized. Default is "no".

-reset_data_registers yes | no	Specifies whether datapath registers need to be reset. Default is "no".
-always_enabled yes | no	Specifies whether or not to enable the hardware. Default is "no".
-auto_start yes | no	Specifies whether to enable the ports for starting the task on hardware. Default is "no".
-reset_level high | low	Specifies active-high or active-low reset. Default is active-high for FPGA and active-low for ASIC
-reset_sensitivity sync | async	Specifies synchronous or asynchronous reset. Default is sync for FPGA and async for ASIC.
-base_address h	Specifies hexadecimal base address. Default is 0x10000.
-dont_touch yes | no	Specifies if the current block can be modified at higher levels or not. Default value is "no".
	Interface Settings
-interface_type type	Specifies the control interface type to this module. The possible types are, handshake and memory_mapped_handshake. Default is handshake.
-hostlm_access_latency uniform | minimal	Specifies if host accesses have uniform latencies or the smallest achievable in each address range. Default is "minimal".
-interface_protocol [native | ocp]	Specifies the protocol to be used for the host interface port. Default is "native".
-disable_ppa_on_host_access_to_memory	Specify whether or not to disable the PPA when a memory is accessed by the host via the direct/memory-mapped ports. Default is "no".

-host_memory_access mode	Specifies host access properties for memories. The possible values for mode are: always | never | infer, lm | direct, [dedicated], readonly | writeonly | readwrite. The first field specifies the accessibility. Default is infer. The second field specifies whether the access is memory-mapped or direct. The third field (optional) specifies if the access is dedicated. Default is shared access. Fourth field specifies if the access is readonly | writeonly | readwrite.
-host_scalar_access mode	Specifies host access properties for scalar input and outputs. The possible values for mode are lm | raw. lm : memories mapped access. raw : raw access. Default host scalar access mechanism is "raw".
-memory_forward_boundary_register always | never | infer | datapath	Specifies the presence/absence/inference/datapath of registers on ports with output to memory. Default is "never"
-memory_return_boundary_register always | never | infer | datapath	Specifies the presence/absence/inference/datapath of registers on ports with input from memory. Default is "never"
-memory_forward_path_delay delay | % clock_period	Specifies the delay of the forward path from the memory port on the processor boundary to the memory. Default is 0.
-memory_return_path_delay delay | % clock_period	Specifies the delay of the path from the memory port on the processor boundary to the memory. Default is 0.
-instream_boundary_register <always/never/infer/datapath>	Specifies if the fifo on input streams should act as an insulating boundary register without a pass-through. Default is "never"
-instream_forward_path_delay <delay in ns or % of clock period>	Specifies the delay of the path from the input stream port on the processor boundary to the input stream. Default is 0.
-instream_return_path_delay	Specifies the delay of the path to the input stream port on the processor boundary from the input stream. Default is 0.

-outstream_boundary_register <always/never/infer/datapath>	Specifies if the fifo on output streams should act as an insulating boundary register without a pass-through. Default is "never"
-outstream_forward_path_delay <delay in ns or % of clock period>	Specifies the delay of the path from the output stream port on the processor boundary to the output stream. Default is 0.
-outstream_return_path_delay <delay in ns or % of clock period>	Specifies the delay of the path to the output stream port on the processor boundary from the output stream. Specifies the delay of the path to the memory port on the processor boundary from the memory. Default is 0.
-scalar_input_delay <delay in ns>|"name1:<delay in ns> name2:<delay in ns> ...">	Specifies the external delay constraint for raw scalar inputs for tcab only . Default is 0.
-scalar_output_delay <delay in ns>|"name1:<delay in ns> name2:<delay in ns> ...">	Specifies the external delay constraint for raw scalar outputs for tcab only. Default is 0.
-system_port_name <"key1:value1 key2:value2 key3:value3 ...">	For clock gated ports <"key:loop_id/tcab_proc_name:value1 ..."> specifies names for the PPA ports.
-system_port_polarity <"key1:value1 key2:value2 key3:value1 ...">	Specifies polarity for the PPA control ports (high | low).
-system_port_production <"key1:value1 key2:value2 key3:value1 ...">	Specifies presence/absence of PPA ports(yes|no).
-top_module_name <name>	Specifies PPA or CHANNEL module name.
	Hierarchy Settings
-import "exp1 exp2 ..."	Specifies TCABs, CHANNELs or PPAs for import. To reset the specification use "".

-integrate_block "<externalize | localize | infer | tcab_name:externalize | tcab_name:localize> ..."	Specifies whether all imported blocks (IPPA or TCAB) or specific imported blocks are to be created inside the current design. externalize : imported block(s) are created outside the current design localize : imported block(s) are created inside the current design infer : Create inside or outside based on sharing opportunity Default is 'infer'.
-sharable_across_blocks "<always | never | infer >"	Specifies the sharability property of the current design (IPPA or TCAB) in container. always : Always share this in container regardless of cost. never : Never share this in container. infer : Share this if it is worth sharing with respect to cost. The default is 'infer'.
-allow_latency_violation yes | no	Specifies if the user specified architectural/resource_release latency can be violated resulting in runtime stalls. Default is yes. Applies to TCAB only.
-latency_independent_schedule yes | no	Specifies whether scheduling should be aware of architectural or resource_release latency. Default is no. Applies to TCAB only.
-tcab_input_registers <always|never|infer>	Specifies whether to register raw scalar inputs for TCAB only. Default is infer.
-tcab_output_registers <always|never|infer>	Specifies whether to register raw scalar outputs for TCAB only. Default is infer.
-architectural_pipelinability < number >	Specifies the target architectural pipelinability for TCAB only. The number specified should be greater than zero.
-input_sample_time < <infer|0>|"name1:<infer|0> name2:<infer|0> ...">	Specifies input sample time for scalar inputs for TCAB only. Default is 0. This specification applies to the importable TCABs. It does not apply for IPPAs and CHANNELs.

-output_production_time [build(default)|import] <number|"name1:num1 name2:num2 ...">	Specifies the output production time constraint. build : This specification applies to the importable design being built. For TCABs, global and scalar-wise specifications are allowed. For IPPAs, only global specification is allowed. This is not applicable to deployable designs or CHANNELs. import : This specification applies to only IPPAs that are being imported into the current design.
-resource_first_use_time < <infer|0>|"name1:<infer|0> name2:<infer|0> ...">	Specifies the resource first use time for array variable/stream for TCAB only. Default is 0. This specification applies to the importable TCABs. It does not apply for IPPAs and CHANNELs.
-resource_last_use_time [build(default)|import] <number|"name1:num1 name2:num2 ...">	Specifies the resource last use time constraint. build : This specification applies to the importable design being built. For TCABs, global and scalar-wise specifications are allowed. For IPPAs, only global specification is allowed. This is not applicable to deployable designs or CHANNELs. import : This specification applies only to IPPAs that are being imported into the current design.
	Simulation Settings
-compiler_args *string*	Specify C/C++ compile-time options. These settings override previous -ccompiler_args specification
-linker_args *string*	Specify C/C++ linker options
-exec_args *string*	Specify runtime arguments for C and SystemC simulations
-exec_dir *directory*	Specify directory where Synphony C Compiler runs simulations
-ccompiler_args *string*	Specify C language compile-time options
-simulator *name*	Specify vcs or ncv. Default is vcs
-verify_rtl_tasks m | m-n | m- | all	Specifies the tasks to be executed during RTL verification. The task numbering starts from 0. m : Executes task m m-n: Executes a range of tasks from m to n, including m and n. m- : Executes task m to the last task, inclusive of both. all : Executes all tasks

-vcompiler_args *string*	Specify Verilog compile-time options for the specified simulator
-vexec_args *string*	Specify Verilog simulation options
-host_mem_access_for_verify_rtl rtl | testbench	Specify this option to speedup RTL simulation during host accesses memories with a large number of entries. To generate appropriate testbench, specify this option at build time. This option has no effect on the hardware. rtl : During RTL simulation, host access to memories is via host path in the RTL. The memory read and write operations are performed on a per-cycle basis. This can be slow for memories with a large number of entries. Default is rtl. testbench : During RTL simulation, host access to memories is via testbench. The memory read and write operations are performed in a single cycle via testbench. This speeds up the RTL simulation;but does not test the hostpath.
-systemc_source yes | no	Specifies if the application uses System C. When specify this option without any value, the tool uses "yes". When you do not specify this option, the default is "no".
-memory_size "array_arg_name1:size array_arg_name2:size ..."	Specify the size of the array arguments for the synthesizable function.
Miscellaneous Settings	
-reporting verbose | normal | quiet	Specifies the type of the report
-no_transcript	Toggle the transcript mode

Example

```
set_params -target process_one_block
```

show

Displays all parameters for the options requested. If no options are specified, the entire environment is displayed.

Syntax

```
show
[-defaults]
[-params]
[-rtl_packages]
[-implementations]
[-techlibs]
[-capture]
[-project]
[-work]
[-implementation]
```

-defaults	Shows all the default parameters.
-params	Shows all the implementation parameters.
-rtl_package	Shows all created RTL packages.
-implementations	Shows all the implementations that are part of the project.
-techlibs	Shows the list of installed technology libraries.
-capture	Use this option in conjunction with any other option to capture the output of the show command into a string. If this option is specified, the show command does not display the output on stdout, instead it returns a string containing the text.
-project	Shows the project filename with absolute path.
-work	Shows the name of the work directory, -relative to the project directory if it is a subdirectory -absolute path if it is not a subdirectory of the project directory
-implementation	Shows the name of the currently selected implementation.

unlock_implementation

Unlocks the specified implementation. You can run commands that modify the implementation.

Syntax

```
unlock_implementation impl1
```

verify

Performs RTL generation and simulation commands.

Syntax

```
verify  [imp] -golden | -lint | -rtl | -clean
[-compile_only] [-run_only] [-nostdout] [-gcov] [-clear_gcov_history]
-effort_level low | normal | high
-simulator vcs
-dump_vcd -dump_fsdb -dump_vpd
-detailed_perf_report -debug_on -coverage_on -interactive_on
-perturb_on -perturb_file filename
-perturb_random_seed integer
-task_overlap always | never | infer
-alt_ppa filename
-norefcompare
-tool name
-format file_format
-dotasks <R>
```

Description

The `verify` command runs simulation on the RTL generated by the `build` command. Therefore, you must run the `build` command before you run the `verify` command.

Example

```
verify -simulator vcs -coverage_on \
  -perturb_on -dump_vpd -interactive_on \
  -alt_ppa synth/run/gateppa.vg \
  -vcompiler_args "-y libfile_path"
```

view_waveform

If RTL simulation is run with waveform dump options, then the `view_waveform` command can be used to open the waveform of RTL and RTL monitor signals. The `view_waveform` command supports multiple viewers. RTL simulation must be run with the `-dump_vcd` option for DVE viewer, `-dump_fsdb` option for Verdi viewer, and `-dump_trn` option for Simvision viewer.

Syntax

```
view_waveform [impl][-dve(default)/-verdi/-simvision]
```

Switching Activity Interchange Format (SAIF) File

During `verify -rtl` compilation, you can generate a switching activity interchange format (SAIF) file using the following options:

```
-vcompiler_args "+define+SCC_DUMP_SAIF" +vcs+saif_libcell;
```

This creates a SAIF file dump_sv.saif, in the rtl_package/simulation/uvm directory. For example,

```
verify -rtl -vcompiler_args "+define+SCC_DUMP_SAIF" +vcs+saif_libcell
```

The SAIF file generation is supported only for the VCS simulator.

Additional SCC Options

This section describes the additional options available for the schedule and synthesize commands.

- Schedule Auxiliary Option
- Synthesize Auxiliary Options

Schedule Auxiliary Option

The following FPGA-only command selects the multiplier pipeline stages.

```
setvar -append schedule_auxopts "-Fmultiplier_type= type"
```

In this syntax, type is one of the following:

- single_cycle – Instantiates single cycle multiplier.
- two_cycle – Instantiates two cycle multiplier.
- three_cycle – Instantiates three cycle multiplier.
- four_cycle – Instantiates four cycle multiplier.
- infer_type – Default. Infers the multiplier type automatically.

Synthesize Auxiliary Options

See the following for information about additional SCC options you can specify for synthesize with the synthesize_auxopts option:

- Multiplexing Non-Mutex Memory Ports

- Arbitration Priority
- TCAB Sharing Controls
- Threshold for Cost Based TCAB Sharing
- Interactions Between Sharing and Timing
- Data Width Threshold for Predicated Multiplier Sharing
- Minimum width for sharing functional units
- II-based Opportunistic Sharing
- Sharing Versus Increased Schedule Length
- Characterization Data Scaling
- Control for Interconnect Delays
- Interconnect Delay Estimation
- Latency Selection
- Default Case Assignment to 0

Multiplexing Non-Mutex Memory Ports

This flag forces the multiplexing of memory ports to the specified array even if the tool cannot prove the ports to be mutually exclusive.

```
setvar -append synthesize_auxopts
  "-Fmem_port_muxing__assume_mutex_accesses =
  arrayName0:read|write|readwrite[arrayName0;read|write|readwrite...]
```

When you use this flag, it is your responsibility to make sure that the accesses are mutually exclusive. The read, write, and readwrite qualifiers specify if the flag is to be applied to the read, write, or read/write types of memory accesses. The specification is applied to both accesses within a PA and across PAs. *ArrayName* is the C source name. If there are multiple matches at different scopes, the flag is applied to all arrays. There is no linting.

To use this flag, set the -Finternal_mem_read_muxing_allowed flag to yes to allow muxing of the internal-fast and internal-ROM read ports. If the array has separate read and write ports, the `-Fmem_port_muxing__assume_mutex_accesses` flag applies to read and write ports separately. In this case, the read qualifier can be used to multiplex down to only one read port. Similarly, the write qualifier can be used to multiplex down to only one write port.

The `-Fmem_port_muxing__assume_mutex_accesses` flag does not handle cases where PA0 has two non-mutex accesses and PA1 has one access, and you want the Synphony C Compiler tool to assume that PA0 and PA1 are mutex with regard to this memory access.

Arbitration Priority

The Synphony C Compiler tool lets you specify arbitration priorities to synthesize, using the following syntax:

```
setvar -append synthesize_auxopts "-Farbitration_priority =
  loopnum>loopnum>[loopnum ...],
  arrayName:loopnum>loopnum>[loopnum ...]"
```

In the following example, the default priority for arrays is L2>L1>L0, and the priority is L1>L0>L2 for array foo. Loops not mentioned have lower priority than the defined loops, and have default priorities with respect to one another. Note that the loop numbers are post-perfectized numbers (if L1 comes before L2 in perfectized code, then L1 has higher priority than L2).

```
setvar -append synthesize_auxopts
  "-Farbitration_priority=2>1>0,foo:1>0>2"
```

For an incomplete global priority specification, fill the unspecified values to the end in SCC default order. Take this 5-loop example:

```
setvar -append synthesize_auxopts "-Farbitration_priority=3>0"
```

The tool interprets this as follows:

```
setvar -append synthesize_auxopts
  "-Farbitration_priority=3>0>1>2>4"
```

For an incomplete per-memory priority specification, fill the unspecified values to the end as per global order. If global order is unspecified, fill in the Synphony C Compiler default order. In the following 5-loop example:

```
setvar -append synthesize_auxopts
  "-Farbitration_priority=3>2>1>0>4,foo:1>2"
```

is considered as:

```
setvar -append synthesize_auxopts
  "-Farbitration_priority=3>2>1>0>4,foo:1>2>3>0>4"
```

TCAB Sharing Controls

The shareability property set at build time using `set_params -sharable_across_blocks` can be overridden at import time using the following flag.

```
setvar -append
  synthesize_auxopts
  "-Ftcab_sharing_across_blocks ={always|infer|never} |
  TCAB_proc:{always|infer|never}"
```

The values for this flag are explained .

`always` – share regardless of TCAB area

`infer` – share only if worth sharing and TCAB is larger than a threshold

`never` – do not share

The following flag enables sharing of TCAB in cases where the Synphony C Compiler is unable to prove mutually exclusive accesses. Using this option requires sharing to be enabled in the command-line interface.

```
setvar -append
  synthesize_auxopts
  "-Ftcab_sharing_across_blocks_assume_mutex_calls =
  tcabProcName1:tcabProcName2[:tcabProcNameN ...]"
```

The Synphony C Compiler tool does not share TCAB with input-to-output combinational paths that have the potential for creating a combinational loop. This option specifies the list of TCABs that can be shared even if they have combinational paths in them and does not create combinational loops.

```
setvar -append synthesize_auxopts
 "-Ftcab_sharing_across_blocks_mux_factor=muxInputs"
```

In this syntax, muxInputs can be any nonnegative integer.

- If muxInputs is 0 or 1, the mux delay is not to be taken into account
- If muxInputs is equal to or greater than 2, the delay of a corresponding mux (2 implies a 2-by-1 mux) is taken into account on all scalar inputs of that TCAB

This option is used to account for delay factors due to a mux present for sharing. The default value for *maxInputs* is 2. This option requires sharing to be enabled in the command-line interface.

Threshold for Cost Based TCAB Sharing

The following flag shares a TCAB only when the number of Synphony C compiler estimated gates is greater than the *numberGates* value.

```
setvar -append synthesize_auxopts
  "-Ftcab_sharing_across_blocks_cost_threshold = numberGates"
```

Interactions Between Sharing and Timing

The Synphony C Compiler tool attempts to share resources in several ways. Expensive resources (such as multiplies, divides, load or store, stream, and TCABs) are aggressively shared when their calls are mutually exclusive or when the initiation interval is sufficient to share the resources. *Aggressive* in this case means that the Synphony C Compiler tool moves the operations around from the schedule slots that they would occupy without any resource constraints. Moderately inexpensive resources (such as adders, subtractors, variable shift registers, and some comparators) are shared only opportunistically, meaning that they are scheduled without regard to resource constraints and are not moved in the schedule to facilitate sharing.

```
setvar -append synthesize_auxopts "-Ftreat_as_very_cheap_op=list"
setvar -append synthesize_auxopts "-Ftreat_as_moderately_cheap_op=list"
setvar -append synthesize_auxopts "-Ftreat_as_expensive_op=list"
setvar -append synthesize_auxopts "-Ftreat_as_very_expensive_op=list"
```

Each flag takes a colon separated list.

These are valid inputs to the list:

- TCAB names
- mpy
- scc_asm_MPY*

When scheduling becomes difficult due to feedback paths, a modulo scheduling attempt fails and Synphony C Compiler tool *buys* more units of some type to break some resource constraints and to provide more flexibility in scheduling. This process proceeds through the various modulo scheduling attempts. Even more constraints are broken in the last attempt when more than one attempt is allowed. The criteria for determining the added units are not very sophisticated, so in complex designs you may see that the incorrect units are being added. Possible results are that the schedule still fails, or you end up with more units of some type than expected or intended.

Sharing and timing interact very tightly when the shared operations are in a feedback path. Sharing a unit creates false paths that are timed by Synphony C Compiler tool. Currently, sharing is done with timing, but only to the extent that the timing paths are checked and verified to pass. In other words, the Synphony C Compiler tool does not elect to share a different set of operations on a hardware resource based on timing, unless one of the options fails timing. The net result is that certain sharing decisions can impact timing negatively which, in turn, impact the achievable frequency for the design.

A common signature of such designs is that the scheduling fails at relatively low frequencies, but the feedback paths shown appear to support much higher frequencies. The feedback paths of the shared hardware do not reflect either resource conflicts or false timing paths.

Data Width Threshold for Predicated Multiplier Sharing

The following flag specifies the minimum width of the multipliers to be considered for sharing when they are mutually exclusive. The default width is 1.

```
setvar -append synthesize_auxopts
  "-Fdata_width_for_mutex_multiplier_sharing=width"
```

Minimum width for sharing functional units

The following flag specifies the minimum width of the operation to be considered for sharing functional units with mutually exclusive operations.

```
setvar -append synthesize_auxopts
  "-Fdata_width_for_mutex_resource_sharing=width"
```

The default length is 16 for ASIC and 65 for FPGA. This setting effectively turns off predicated opportunistic sharing for moderately inexpensive operations during modulo scheduling. Using this option can have a beneficial impact on timing and often on cost.

II-based Opportunistic Sharing

The following flag specifies the minimum width of the II-based operation to be considered for sharing.

```
setvar -append synthesize_auxopts
  "-Fdata_width_for_II_resource_sharing=width"
```

The default value is 16 for ASIC and 65 for FPGA. This setting effectively turns off II-based opportunistic sharing for moderately inexpensive operations during modulo scheduling. Using this option can have a beneficial impact on timing and often on cost.

Sharing Versus Increased Schedule Length

The default decisions for sharing the functional units are different in TCABs and PPAs. TCAB defaults are performance-centric. Any sharing decision that could potentially increase the schedule length is off by default for TCABs. The decision can be modified using the following options.

```
setvar -append synthesize_auxopts "-Fpipelinability_focus=value"
```

value is one of the following:

`performance` – does not create sharing that could potentially elongate schedules

`area` – tries to share functional units as much as possible

`infer` – maps to 'performance' in TCAB and to 'area' in PPA (default value).

The following flag controls the amount of sharing at the expense of increasing the schedule length.

```
setvar -append synthesize_auxopts
  "-Fmax_sched_length_increase_with_sharing=number"
```

In this option, number specifies the increase in schedule length in cycles. Consider the following example:

```
for (j=0; j < 1; j++) {// dummy loop
    a = b * c;
    for (i=0; i < 10; i++) {
        a = a * d;
        x[i] = a + i;
    }
}
```

This example shows sharing of a multiplier unit. It may be required to have the dummy loop created to force this choice of perfectization. The multiplier would be scheduled at different time slots.

Characterization Data Scaling

The following flags are used to scale characterization data:

```
setvar -append synthesize_auxopts
 "-Farea_scale_factor_macrocellName=value[:typical_data|all_data]"
setvar -append synthesize_auxopts
 "-Fdelay_scale_factor_macrocellName=value[:typical_data|all_data]"
```

The all_data qualifier scales the entire characterization data while the typical_data qualifier changes the initial typical implementation that Synphony C Compiler tool picks for that macrocell. Default is typical_data.

The macrocell name in these flags is the RTL-packaged name instead of the original name in the macrocells directory. The area_scale_factor flag currently affects only the estimated area output. However, if it is qualified with all_data, then the characterization data seen by the Synphony C Compiler tool is scaled and can make different typical-implementation and width-clustering decisions.

Control for Interconnect Delays

The following flag is available only with FPGA and allows you to control the interconnect delay estimate that the tool uses for its timing estimation.

```
setvar -append synthesize_auxopts
 "-Fdelay_scale_factor_interconnect=value"
```

The minimum value for this option is 1.0 and the default value is 1.5. The higher the value for this option the greater the estimated interconnect delay if the timing constraints of the design are not violated.

Interconnect Delay Estimation

The following flag is available only with FPGA and allows you to enable (yes) or disable interconnect estimation.

```
setvar -append synthesize_auxopts
 "-Fenable_interconnect_estimation=yes|no"
```

The default value is yes. If interconnect estimation is switched off (-Fenable_interconnect_estimation=no) for any reason, the interconnect factor should be set to 1.2 using the option "-Fdelay_scale_factor_interconnect=1.2".

Latency Selection

The following flag is available only with FPGA and is used to enable or disable latency selection for multipliers.

```
setvar -append synthesize_auxopts "-Fenable_latency_selection =yes|no"
```

The default value is yes. When this flag is enabled and a single-cycle multiplier does not fit within a cycle, it picks a two cycle multiplier, and if that does not fit within two cycles, it picks a three cycle multiplier.

Default Case Assignment to 0

If the SCC tool knows that the default branch of a Verilog case statement is unreachable, it sets the output value to be x for the default. This action enables better logic optimization. However, for partial case statements, equivalence checking tools such as LEC do not understand this value and cannot prove equivalence. You can assign a default value of 0 with this command:

```
setvar -append synthesize_auxopts "-Freplace_x_with_0=yes|no"
```

Setting the -Freplace_x_with_0 flag to yes ensures that every partial case statement assigns a default value of 0 to the output instead of x, during RTL creation and macrocell packaging. The default for the command is no.

Note that this flag has an area or timing impact on the final RTL as the macrocells are characterized using x; so the timing numbers used by the SCC tool differ from the numbers seen in the final hardware.

Synphony C Compiler EV Commands

With the SCC tool, simulations run sequentially on an implementation so as not to overwrite results from another simulation. The Synphony C Compiler EV tool organizes the simulations so that they do not overwrite one another when run in parallel, and lets you run a large number of simulations. It is a simulation-intensive tool that includes the verification capabilities in the Synphony C Compiler tool. It lets you run the simulation steps to verify your design.

See the following for more information:

- Synphony C Compiler EV Launch Command
- Synphony C Compiler EV Supported Commands

For information about using the EV tool to run regressions, see “Running Synphony C Compiler EV Regressions” in the *Synphony C Compiler User Guide*.

Synphony C Compiler EV Launch Command

Start the SCC EV tool using the following command:

```
$ synphonycc_ev
```

When invoked, information similar to the following is displayed:

```
Synphony C Compiler EV E-2010.09
Copyright (c) 2010 Synopsys, Inc. All rights reserved.
Build Time 08/23/10 21:42:55
```

WARNING: The Synphony C Compiler tool does not support the opening of multiple sessions simultaneously for an implementation. If an implementation is open and is being modified in different sessions simultaneously, each session can potentially overwrite information essential to another session. You are advised to open an implementation in only one session at a time. This applies to all sessions of the Synphony C Compiler tool.

Synphony C Compiler EV Supported Commands

The Synphony C Compiler EV tool uses a subset of the Synphony C Compiler commands, specifically the simulation commands. This subset of commands and their supported options are listed .

verify -golden [-*options*] -run_only	See verify.
verify -rtl | -status [-*options*] -run_only	See verify.
new_implementation *newImplementation* -from *sourceImplementation*	See new_implementation.
select_implementation delete_implementation report_implementation	Same as the SCC tool. See select_implementation, delete_implementation, and report_implementation for details about the command options.

4

Projects and Implementations

This chapter describes project, implementations, directory structure in the Synphony C Compiler tool and is organized as follows:

- Projects and Implementations
- Directory Structure
- Project and Implementation Commands

Projects and Implementations

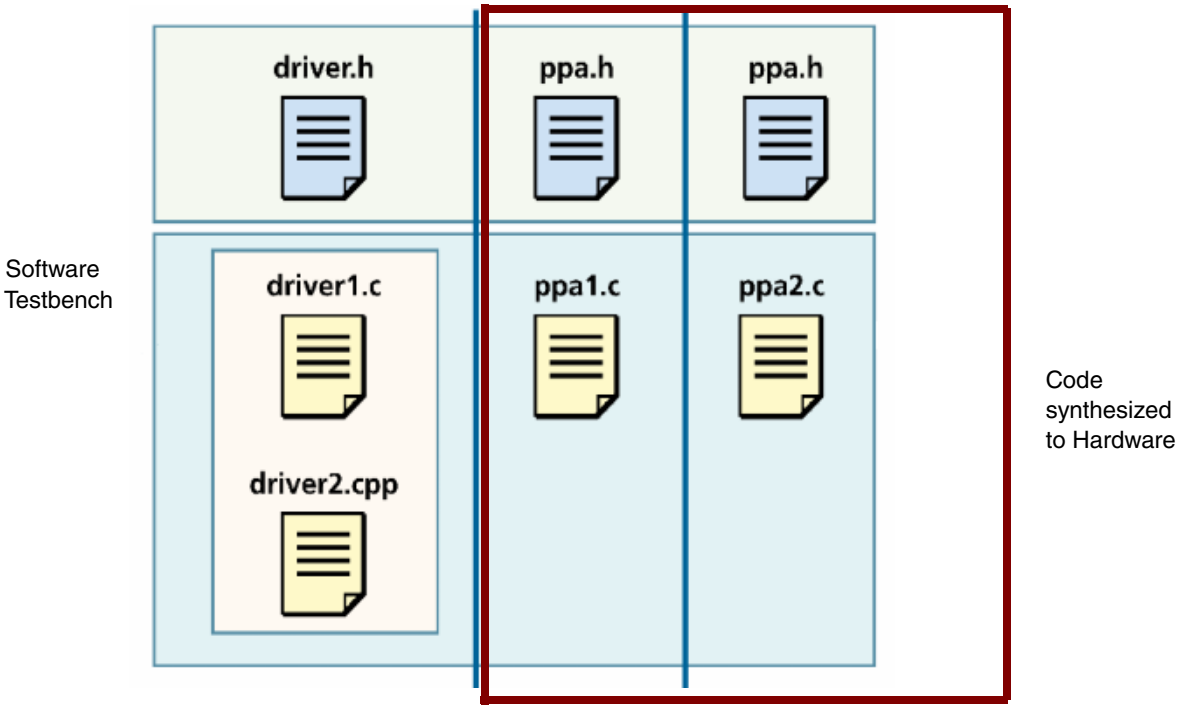
The design model for the Synphony C Compiler tool is based on the idea of projects and implementations. For further details, see the following descriptions:

- Projects
- Implementations

Projects

A project consists of a set of C or C++ source files, header files, input data files, and reference result files. The source files are compiled together and then run. The project might include many source files that are translated into hardware. The code in the source files can be classified into two parts:

- Synthesizable code
  This is the C or C++ algorithm to be converted to hardware and can be in one or multiple files. In hardware-software codesign, this can be the processor code. In hardware-software codesign, this can be the portion of the application requiring hardware acceleration.
- Driver code
  This is the testbench code used to exercise the algorithm to check for functional correctness. In hardware-software codesign, this can be the processor code. For processor-based systems, the tool creates a driver for the processor to communicate with the generated hardware.



Implementations

An implementation is a C procedure with specific parameters that determine area and performance. One project can have many implementations with different parameters. You can run an implementation with different parameters, multiple times.

All project files are cached in the implementation directory. The implementation includes all the information used or produced in the tool process: input files (sources, headers, input data and expected results), output files (generated RTL, customized back-end scripts, unit-test vectors, testbenches), user feedback files (logs, reports, graphs) and any intermediate or temporary files.

Sets of related implementations can be collected into groups. You can generate detail and summary reports for all the implementations within a group. And also, you can package the RTL from your *best* implementations and save for standalone simulation and synthesis.

Directory Structure

When you create projects, the Synphony C Compiler tool creates directories and sub-directories based on the commands you set. The project directory is the where the tool runs, and stores parameters and implementations.

- Project Directory

  By default, the project directory is set to the directory from where the Synphony C Compiler tool is launched. You can define another project directory when you set the project parameters. Alternatively, you can set the project directory with the File > Open Project command, or with the `set_params -projdir` command.

  The project directory contains all files related to the project. You do not need to have the input files in the same directory where the project is run. You can specify the directory from which the input files are to be picked up for the current project.

  The Synphony C Compiler tool uses the project directory to pass file names to executables, copy files, and generate scripts. Altering the directory and file structures of a project might cause compilation commands and scripts to fail.

  In addition, implementations are set up under the project directory.

- Source Directory

  The `src` source directory is the directory from where the input files for the project are taken. The Synphony C Compiler tool looks for the implementation files in the source directory. These implementation files consist of all the files associated with a given implementation, including source files, header files, data files, and result files.

  By default, the source directory is set to the project directory. You can specify another source directory with the `set_params -source_dir` command. If you are using the GUI, the directory is inferred from the location where files are selected when you set up the project with the Project Setup command.

Structure of a Project Directory

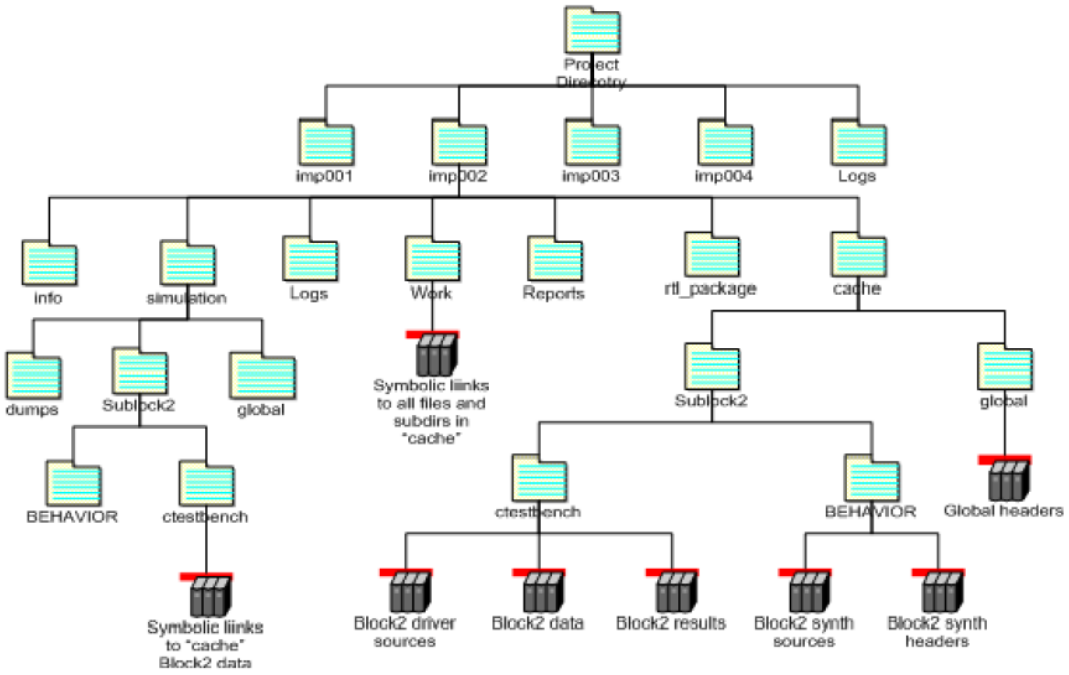
This is the typical structure of a project directory:

```
Projdir
     src
     include
     test
          input
          output
     src
          test
     include
          test
```

Structure of an Implementation Directory

A Synphony C Compiler implementation collects all the information used or produced in the process of generating an artifact in one place: input files (sources, headers, input data, expected results), output files (generated RTL, customized back-end scripts, unit-test vectors, test benches), your feed-back files (logs, reports, graphs), and intermediate and temporary files used in the process.

The following figure shows the implementation directory setup:



The directory holding all these has the following structure:

<implementation>/

- Reports/ – the most-recent version of all the reports generated so far.
- info/ – additional feedback files, useful in performance analysis or application debugging.
- rtl_package/ – the complete set of files required to run simulations and to take the generated artifact through back-end synthesis.
- simulation/ – results of the simulation. Each file is marked with the phase in which it was produced.
- simulation/scc_dumps/ – test vectors produced during simulation.

- Logs/ – logs of all the commands executed, ordered by command name, execution date and time.
- Work/ – tool-internal intermediate and temporary files. Most of these files are in machine-readable format.
- cache/ – contains source, header, data, and result files or symbolic links for the data and result files.

The files in the Reports directory have the same information as produced by running the `report_implementation` command with the specific option. For example, the file report_summary.txt contains the same information as produced by running the `report_implementation -summary` command. The tool automatically refreshes these files to contain the most recent information available. The reports produced in each build and simulation phase are listed:

- `synthesize`: address_map, cost, fu_utilization, opid_rtl_map, partial_schedule, feedback_path, rolled_schedule, sorted_unrolled_schedule, summary, interface, timing, unrolled_schedule, var_map_rtl.
- `verify -rtl`: Simulation performance

The `report_implementation -performance` command produces the performance report generated by the `verify -rtl` command as a graphical report (in PDF and PostScript formats).

The info directory contains files, which provide feedback in addition to the existing information at the terminal or in the reports. These files contain intermediate versions of the transformed application code or ancillary information detailing the artifact's integration in the system. These files are automatically refreshed by the tool to contain the most recent information available.

The synthesize command generates the following reports.

- procName.rtlmap: A compact list of all the ports at the boundary of the RTL module of the generated artifact
- procName.rtlmap: A compact list of all the ports at the boundary of the RTL module of the generated artifact

Project and Implementation Commands

This section describes the commands used in the design process to work with projects and implementations:

- Project Setup Commands

- Example of Design Command Sequence

Project Setup Commands

The following table lists the Tcl commands to use to build a project. To use the GUI, select the corresponding commands in the GUI.

set_defaults -projdir *dirPath*	Set up project directory
set_defaults -sources "*files*"	Define source files
set_defaults -headers "*header_files*"	Define header files
set_defaults -data "*data_files*"	Define data files
set_defaults -results "*files*"	Define result files
new_implementation	Create implementation
set_params -appfiles	Define implementation parameters

The sources and headers must be under the directory specified by `set_defaults -source_dir dirName`. You can specify absolute paths or relative paths. If you specify relative paths, the sources, headers, data and result files must be under the project directory. You can cache the data and results files for the implementation using the `set_params -cache_data_files yes` and `set_params -cache_result_files yes` commands.

Example of Design Command Sequence

The tool provides many other command options, but the following example shows a typical combination of commands that you can use:

1. Set the project parameters (`set_params`). This establishes your project directory, compiler, and source, header, input data, and result files.
2. Create an implementation (`new_implementation`), which allows you to set up the unique name for the implementation to run and whether the required run files are to be copied from a pre-existing implementation or to be derived from your input C application only.
3. Set up the implementations parameters (`set_params`). This command identifies the application file for the implementation and the procedure within it. In addition, specify the

Task_II, clock frequency, maximum task frames, and other hardware configuration parameters for the implementation.

4. Run compliance checking for the implementation (lint) to ensure that your application can be compiled, the golden run, and passes several linting check points, such as syntactic, semantic, bit width range, etc.

   You can also run static and dynamic linting individually. Run dynamic linting using the `verify -lint` command to encompass all the check points.

5. Run a single implementation or multiple implementations:

   - For a single implementation that has passed compliance checking, you can run your implementation through the build phase (build). Then run one or several C simulation verifications of your design.
   - Run several implementations to compare and evaluate the results of various performance and area parameter settings.

6. You can now generate RTL and run verification . Alternatively, use the verify command.

   After you have an RTL package that meets your design requirements, you can hand it off to the hardware engineer to continue the Verilog verification, synthesis and integration processes.

Command Logging

All commands issued at the Command-Line interface prompt are logged. There are three separate log files created in the project directory under <projdir>/Logs.

1. command.log. This lists all Tcl commands with time stamps showing when they were issued. This may be useful for scripting.
2. scc.log. This lists all user interface commands with time stamps showing the underlying Synphony C Compiler commands that were issued to the Synphony C Compiler core.
3. transcript.log. This is created if the -transcript option is selected when invoking the `set_params` command. Selecting that option results in a transcript.log file, which is a transcript of the current session. If the transcript.log was previously created, this file is appended to it. (The option `-no_transcript` switches off transcription.)

In addition, the output of each command issued in the context of an implementation is also captured and logged in the implementation directory, <implementation>/Logs, along with a timestamp. Command outputs are captured in these log files which can be sometimes useful for analysis of designs or debugging issues.

5

Programming Model Basics

This chapter describes the Synphony C Compiler tool generated hardware elements. For more information, see the following:

- Overview of the Programming Model
- Synthesizable Code and Testbench
- Task Processing in PPAs
- Loop Nests and Processing Arrays
- Compiler Transformations
- Performance Metric
- Feedback Paths
- Data Structures and Hardware Mapping
- Communicating with the External Environment
- Communication Between PA Modules
- Hardware Sharing
- Parallelism

- Task Overlaps
- Implications of Coding Styles and Syntax
- Good Hardware Techniques for the Compiler

Overview of the Programming Model

The Synphony C Compiler tool translates your un-timed C or C++ algorithm into timing-aware parallel hardware. The tool explores the architectural space, automatically generates optimal design alternatives, and produces efficient, synthesizable RTL from the best area and performance implementation among those alternatives.

The hardware generated by the Synphony C Compiler tool can be built as either of the following structures. Unless specifically stated otherwise, all features described here apply to both pipeline of processor arrays (PPAs) and tightly coupled accelerator blocks (TCABs).

- A top-level design (PPA).
  A PPA is a pipeline of processor arrays, and refers to the top-level hardware block that the tool generates from a single C application procedure.
- A building block (TCAB).
  TCABs are low-level building blocks that can be imported into higher-level designs. You can also build a TCAB as a standalone module.

The process of converting the software algorithm into RTL involves the following steps:

1. Taking an algorithmic C procedure as input and running initial semantic and syntactic checks of the input.
2. Exploring the design space and identifying optimal design alternatives, based on user-defined performance and area targets.
3. Running through several synthesis and verification paths and comparing output against the golden standard (verify -golden).
4. Producing synthesizable RTL, which successfully meets the area and performance targets.
5. Generating an RTL artifact that is ready for processing by industry-standard tools.

Synthesizable Code and Testbench

You must specify the top-level entry point for synthesis. The tool synthesizes all the logic in this function, as well as all the functions it contains, as hardware. This top-level function is called the *target procedure* in this chapter. You can specify it in one of these ways:

- Specify the function with the `set_params -target` *`function`* command.
- Specify the function name alias with a `$SCC` label. For more information, see "$SCC Labeling" on page 7-3.

The target procedure must be defined in one of the appfiles. PPA or synthesizable code refers to a portion of the application that is a candidate for hardware acceleration. Driver code refers to the rest of the application which can be used for testing and verification purposes. For example, the driver can be the application code minus the PPA code, or it could be a set of files designed specifically for verifying the PPA code. The Synphony C Compiler tool also allows you to mix the functionality of testbench and synthesizable hardware in the application source files.

The testbench supports all of C/C++ syntax support. However, synthesizable portion of code supports a subset of C or C++ syntax. For more information on the syntax supported in the synthesizable subset of code, see the Chapter 6, “C/C++ Synthesizable Syntax.

Task Processing in PPAs

A *task* is one complete execution of the PPA procedure call. When there are multiple invocations of the PPA procedure, C/C++ executes each call sequentially. However, the PPA hardware can be built to execute the tasks sequentially or with different levels of task overlap.

The PPA’s control interface is either a raw handshake interface or a memory mapped handshake interface. For the raw handshake interface, the timing of the control handshake signals on the PPA interface allow the external hardware to exercise the best task overlap on the PPA. If the PPA is memory-mapped on the system bus, then a memory-mapped handshake interface provides the same information. However, the C driver running on the processor should be rewritten to use the task overlap capability. For details, see “Deployment of Memory-Mapped PPA” on page 14-26.

In Synphony C Compiler, all the simulations except RTL simulation execute tasks sequentially. The `verify -rtl` command executes tasks in an overlapped manner wherever possible, showing the real throughput.

In terms of programming, a task is the amount of computation that is encapsulated by one complete invocation of the target procedure. A task is completed only when all the activity related to that invocation in all loop nests is complete. In the following example, the testbench segment calls the synthesizable function block( ) several times, each call to the synthesized function block( ) is a task.

```
int main()
{
      …
      for (i=0;i<3;i++)
            {
             block();  // Synthesized as hardware.
            }
      …
}
```

```
void block(char scale) {

int i, j;
char x[8][8], y[8][8];
short dc = 0;
for(i = 0; i < 8; i++)
     for(j = 0; j < 8; j++)
          x[i][j] = x.get();
for(i = 0; i < 8; i++)
     for(j = 0; j < 8; j++)
          y[i][j] = y.get();
for(i = 0; i < 8; i++)
     for(j = 0; j < 8; j++)
          dc += y[j][i] * x[j][i];
for(i = 0; i < 64; i++) {
     short data = _input_z() - (dc >> 6);
     scc_stream_output_inter(data);
}
for(i = 0; i < 64; i++) {
     short data = inter.get() * scale;
          scc_stream_output_s(data);
     }
}
```

In strict sequential semantics, a new task can be started only when a previous task is finished. This is due to the stack-based procedure call and return mechanism that ensures the caller effectively waiting for the called procedure to finish before it can start the next invocation of the same procedure. This is a classic example of a low throughput sequential design.

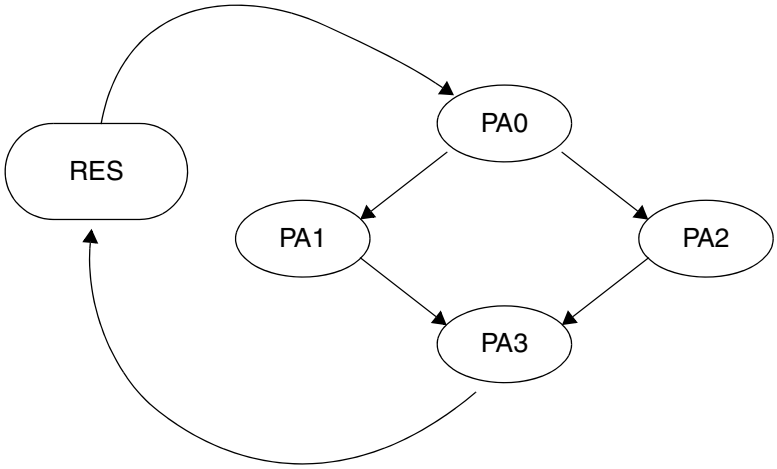
Hardware modules execute in a parallel fashion. The Synphony C Compiler tool analyzes the application and is capable of extracting several levels of parallelism in your application which enhance the performance of the generated hardware. The following sections describe this in more detail.

Shared Resource Handling

A PPA contains one or more processing arrays (PAs). PAs communicate via flow-controlled data FIFOs or a shared resource such as a scalar or memory stream port. Any shared resource (scalar, memory, or stream port) must be protected from conflicting accesses by the PAs. For this reason, a dependency graph is built for each resource. The dependency graph contains a resource node and PAs. An edge direction shows the flow of resource token (permission). The source of the edge is a resource releaser and the destination is a resource acquirer.

Figure 5-1 shows four PAs that access a shared resource in a certain way (PA0: Write, PA1, PA2: Read, PA3: Write).

Figure 5-1 Example of Resource Sharing Between PAs



In this example, the resource token is initially present in the resource pool (RES). PA0 acquires the resource and does a write access. The resource token is then passed to PA1 and PA2, which do read access. Note that both PA1 and PA2 can hold the token simultaneously since they do only read access, which is non-conflicting. After both PA1 and PA2 complete read accesses, PA3 can start the write access. Finally, the resource is released back to the resource pool. For the next task, the token flow repeats.

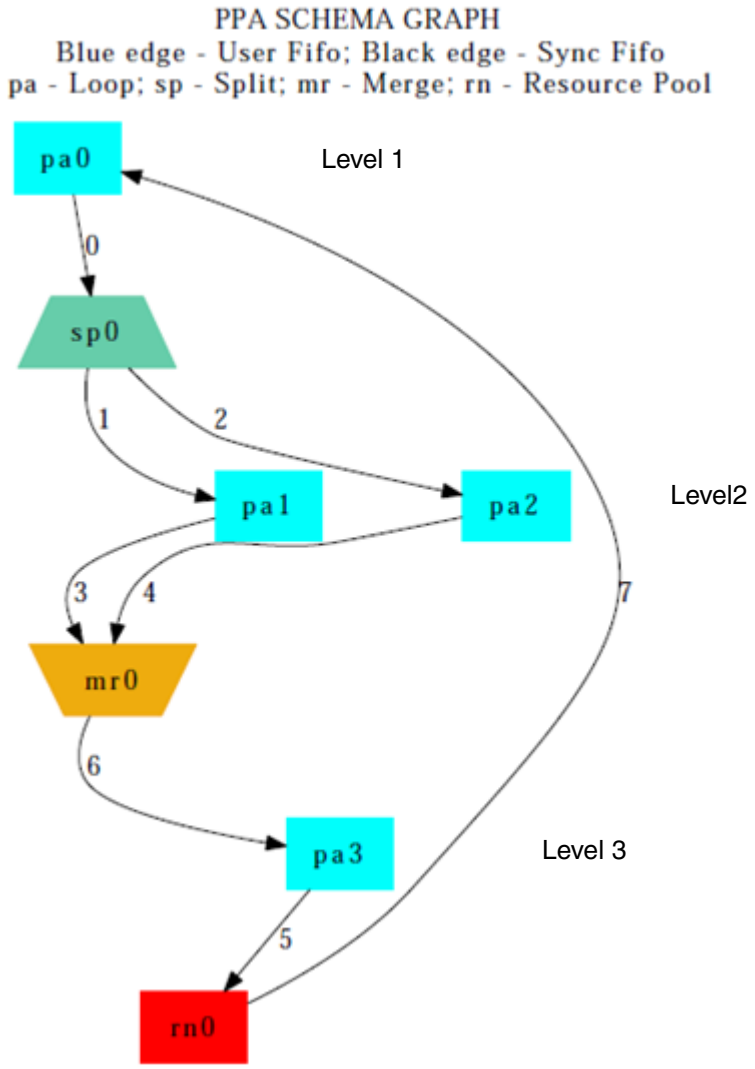
The resource pool contains resource tokens. There can be multiple tokens for a multi-buffered resource. If the resource is a scalar variable written by a PA and read by another PA, then the token is also accompanied by data.

For PPAs with external control for task start and done, a special resource is created. The external hardware (host) can be viewed as the resource pool manager of this resource. It provides the task start token to all PAs along with any scalar inputs for the task. The PAs return the token back to the external hardware (host) along with any scalar outputs of the task.

PPA Hardware

When a PPA has task overlap, the PAs could be executing different tasks. Figure 5-2 shows the hardware used to implement the resource graph of Figure 5-1 and the ordering of the PAs based on their dependencies.

Figure 5-2 PPA Hardware Resource Graph



In this figure, Level(PA0) = 1, Level(PA1) = 2, Level(PA2) = 2, Level(PA3) = 3. When there is task overlap, PAs at lower-numbered levels can be executing current tasks while PAs at higher-numbered levels are still executing older tasks. For example, PA0 can execute Task 3 while PA3 is still executing Task 1.

Due to this overlap, there could be multiple tokens and data released before consumption. A requirement for task overlap is to provide enough buffers. In this example, the resource pool must have three resources (buffers) available to achieve full task overlap.

Whenever a token is released to multiple consumers, a split hardware block is created that acknowledges the releaser only when both the acquirers are ready to accept the token. Similarly, whenever a token is acquired from multiple releasers, a merge hardware block is

created that acknowledges the acquirer only when the releasers are ready to release the token.

The token-passing hardware is created using a 2-way handshake between resource releaser and acquirer. Since multiple tokens might be released before acquisition in a task-overlapped PPA, FIFOs of appropriate depth are placed on the handshakes.

Task Overlap Scenarios

Task overlap in a PPA can happen in different ways:

- No task overlap: Only one task is active in the entire PPA. After completing a task, PAs are idle until the task is complete on the PPA.
- Inter-PA task overlap: Multiple tasks are active on the PPA, but only one task is active on a PA. In this scenario, a PA starts a new task only after completing the current task.
- Inter-PA and Intra-PA task overlap: Multiple tasks are active on the PPA and even on a PA. In this scenario, when a task on a PA enters its draining phase, it immediately starts a new task in a pipelined way, without losing any cycles.

Handshake Event Types

Resource release and acquisition can happen in two different ways:

- `task_boundary`: The PA acquires resources at the beginning of the task and releases resources only at the end of the task. This impacts performance but reduces cost of resource handshake by merging handshakes.
- `resource_exact`: The PA acquires resources only when it requires them (as late as possible) and releases when it is done with its use (as soon as possible).

Continuous Processing

A PPA can be specified to be built as a continuous processing design by using the `set_params -continuous_processing always` command. This specification automatically turns on inter-PA and intra-PA task overlaps and selects `resource_exact` type resource release and acquisition. Such settings allow maximum performance (task II) to be achieved at the expense of area and timing.

For continuous processing, Synphony C Compiler tries to eliminate any dead cycles between tasks. If dead cycles appear between tasks, you can modify the application or add more buffers to achieve higher performance. The `Reports/report_summary.txt` file reports the pipelining loss for each PA and the associated reason.

By default, Synphony C Compiler builds a PPA with no continuous processing. In this configuration, there is no task overlap in the PPA, and resource release and acquisition occur at the task boundary. Handshakes are insulated for better timing. Basically, such a PPA has better area and timing at the expense of performance (task II).

RTL simulation using `verify -rtl` command reports the achieved task II. If there is a deadlock due to insufficient FIFO sizes, then you can increase the corresponding FIFOs in the `rtl_package/rtl/`*`ppa_name`*`_define.v` file.

You can run `verify -rtl -vcompiler_args "+define+PEAK_FIFO_USE +define+ALL_FIFOS"` to give an upper bound on the FIFO sizes. The upper bound on the FIFOs can be seen in `peak_fifo_lengths.v` in the `rtl_package` folder of your implementation.

Low-Level Control Using synthesize_auxopts

You can use the `synthesize_auxopts` command for advanced, low-level control of task execution. The command options used for these controls are described in the following subsections.

Intra-PA Task Pipelining

You can enable or disable task pipelining within a PA. You have global control as well as control of individual PAs. Use the following command:

```
setvar -a synthesize_auxopts
 "-Fpipelined_pa=always|never|infer,loop_num:always|never|infer"
```

- `always` – Allow intra-PA task pipelining
- `never` – Disallow intra-PA task pipelining
- `infer` – Allow intra-PA task pipelining if continuous processing is selected; otherwise disallow it. This is the default behavior.

Handshake Event

The `resource_handshake_events` variable can to used to specify `task_boundary` or `resource_exact` handshake. It allows global control as well as control for individual PAs. Use the following command:

```
setvar -a synthesize_auxopts
 "-Fresource_handshake_events=task_boundary|resource_exact|infer,
 loop_num:task_boundary|resource_exact|infer"
```

- `task_boundary` – Acquire at task start and release at task end.
- `resource_exact` – Acquire at first use and release at last use.
- `infer` – `resource_exact` handshaking is selected if continuous processing is selected; otherwise, `task_boundary` is selected. This is the default behavior.

Handshake Insulation

The `registered_output_handshake` option can to used to specify whether a handshake has to be registered at the releaser for better timing.

```
setvar -a synthesize_auxopts
 "-Fregistered_output_handshake=always|never|infer"
```

- `always` – Registered handshake.
- `never` – Unregistered handshake.
- `infer` – Unregistered handshake is used if continuous processing is selected; otherwise, registered handshake is used. This is the default behavior.

Input Data Register Optimization

Scalar data acquisition by a PA contains a control register and a data register. A dedicated control staging register is always needed. However, a PA core datapath register can be reused as the data staging register in some cases, which saves area at the expense of performance.

```
setvar -a synthesize_auxopts "-Facquire_data_register=always|never|infer"
```

- `always` – Acquire handshake has its own data register.
- `never` – Acquire handshake shares the PA core datapath register.
- `infer` – Acquire handshake has its own data register if the PA has intra-task pipelining; otherwise, it uses a PA core datapath register. This is the default behavior.

Resource access window specification

For continuous processing designs, Synphony C Compiler tries to achieve a schedule where a resource is accessed for less than or equal to (II * *num_buffers*) cycles. For other designs, there is no across-task constraint for the resource access window. Using the following syntax, you can specify an upper bound on the resource access window in an iteration of a PA.

```
setvar -a synthesize_auxopts
 "-Fper_iter_singlebuf_res_access_window_ub=panum:n, panum:n, ..."
setvar -a synthesize_auxopts
 "-Fper_iter_multibuf_res_access_window_ub=panum:n, panum:n, ..."
```

Delaying PA starts

For continuous processing designs, there can be situations where the RTL simulation deadlocks and due to non-FIFO size reasons. In such cases, you should re-synthesize the design after specifying a delay for some PA task starts to avoid deadlocks.

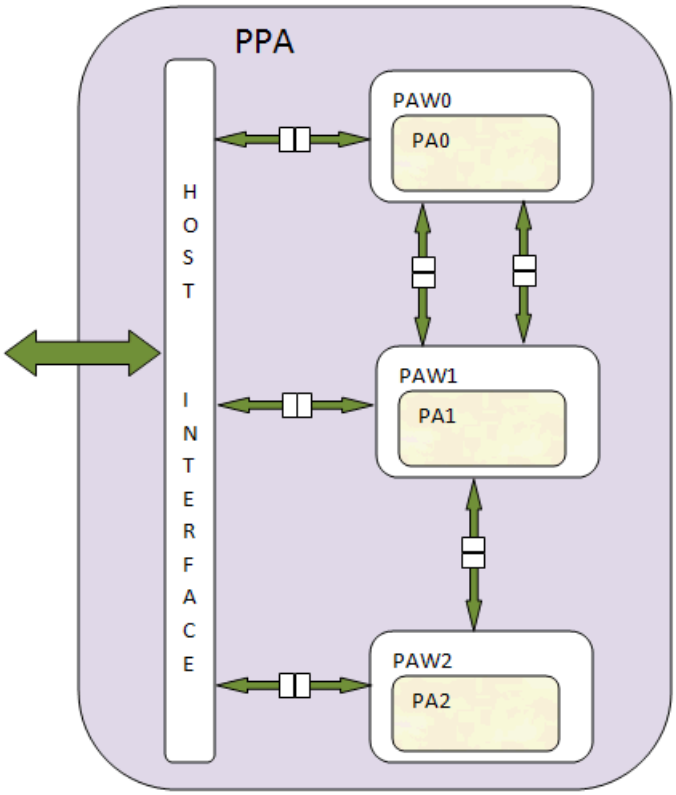
```
setvar -a synthesize_auxopts
 "-Fper_iter_task_ii_lb=panum:n, panum:n, ..."
```

Loop Nests and Processing Arrays

To get the most out of the Synphony C Compiler designs, it is important to have a high-level understanding of the hardware architecture that the tool produces. This section briefly outlines the template of the generated hardware and the Chapter 16, “Hardware Architecture,” covers this in more detail.

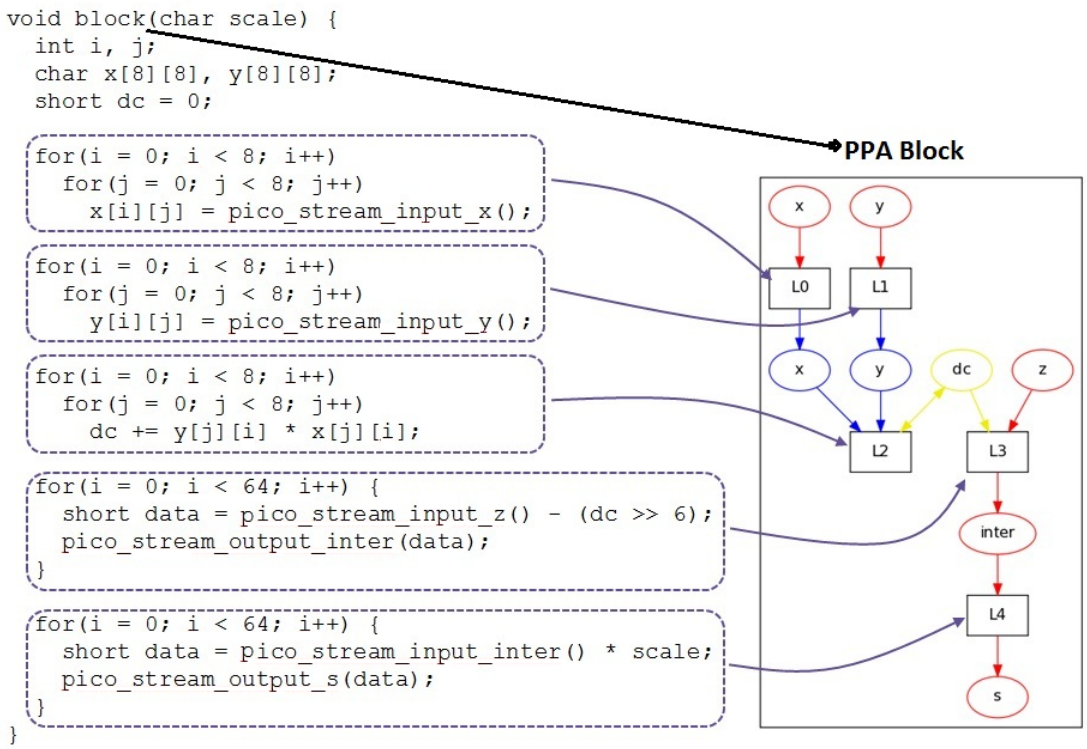
The Synphony C Compiler tool maps the PPA procedure (and all procedures inlined into it) to a Pipeline of Processing Arrays, or PPA. Individual blocks of code within that procedure will be mapped to Processing Arrays or PAs. A PA is a hardware block that performs some iterative function, though some PAs may execute a single iteration.

Each PA is optimized to meet the performance needs of the iterative function being mapped to it. For a simple block, such as a single instruction `x = a + b`, a PA may be just a few hundred gates (example, an adder and some registers). For a more complex block, such as an H.264 deblocking filter, a PA may be thousands of gates. All the PAs are instantiated in the top-level PPA module with the relevant timing controller and additional logic for host interfaces. A conceptual diagram of the PPA block is as shown. For more information, see Chapter 16, “Hardware Architecture.”



The PPA architecture has a single clock domain. However, to enable high performance designs, each individual PA operates in its own stall domain, allowing a PA to continue executing even if another PA is stalled waiting for data.

Consider the code shown on the left side in the following diagram. If you synthesize block( ) function as the top-level design, the Synphony C Compiler tool recognizes that your code consists of five loop nests and infers the data passing mechanism between them. In the generated hardware, each loop nest is synthesized as a separate module called a Processing Array (often referred to as PA) and a top-level design (PPA) instantiates the different PA modules after inserting additional wrappers around them. For more information about the PA and its wrappers, see Chapter 16, “Hardware Architecture.”



In this chapter, the term loop refers the loop nest in the software code and the term PA is used to denote the hardware module created for the loop. The loop nests and PAs are numbered starting with a 0 (zero). In this case the five PA units (PA0, PA1, PA2, PA3 and PA4) correspond to the loop nests (L0, L1, L2, L3 and L4). The right hand side of the diagram shows a PPA graph for the piece of code shown on the left side. Since there is a one-to-one correspondence between the loop nests and PAs, these terms are used interchangeably in a few places.

Compiler Transformations

The previous section provided details of the Synphony C Compiler PPA architecture. This section provides a programming-centric view of how the compiler of the tool maps to the PPA architecture.

The overall flow of the compiler of the tool is as follows:

`analyze` --> `build`

All procedure calls in the PPA procedure are inlined by default, and all loops marked for unrolling with #pragma unroll are unrolled. You can view ppafilename_inline.c, ppafilename_unroll.c, and ppafilename_preprocess.c to see the Synphony C Compiler transformations after inlining, unrolling. You can also view a loop unrolling summary report using the Tcl command `report_implementation -loop_unrolling`.

You have an option to build a hierarchy of designs using the Synphony C Compiler tool. For more information, see the Chapter 16, “Hardware Architecture. RTL for that module is imported from the TCAB or IPPA implementation and directly instantiated in relevant places in the current design.

Loop unrolling has the effect of repeating the code in the loop a number of times equal to the range of the loop index. For example,

```
for(k=0;k<3;k++) { L1; }
```

when unrolled will repeat L1 four times:

```
k=0; { L1; }
k=1; { L1; }
k=2; { L1; }
k=3; { L1; }
```

Procedure inlining has the effect of copying the code in the procedure into the call location. Assume you have a procedure foo:

```
void foo(int x) { P1; }
```

There is a call to foo:

```
foo(x);
```

The resulting code after inlining will look like this:

```
{ P1; }
```

After these transformations have been done, the Synphony C Compiler tool accepts the code in the form described. Note that this discussion deals only with the structure of the code into loop nests and straight-line code segment, and furthermore, all code is subject to Synphony C Compiler syntax restrictions.

1. At the scope of the PPA procedure, any sequence of straight-line code segments and loop nests is acceptable.
2. if( ) statements may be placed around loops at any nesting depth including the top level.
3. Inside each loop, there can be any number of loops (both nested and those which execute sequentially), which can optionally be preceded and followed by straight-line code segments. A loop may be present in both then and the else part of an if( ) statement.
4. Rule #2 can be applied recursively. In other words, a loop nest may have an arbitrary nesting depth

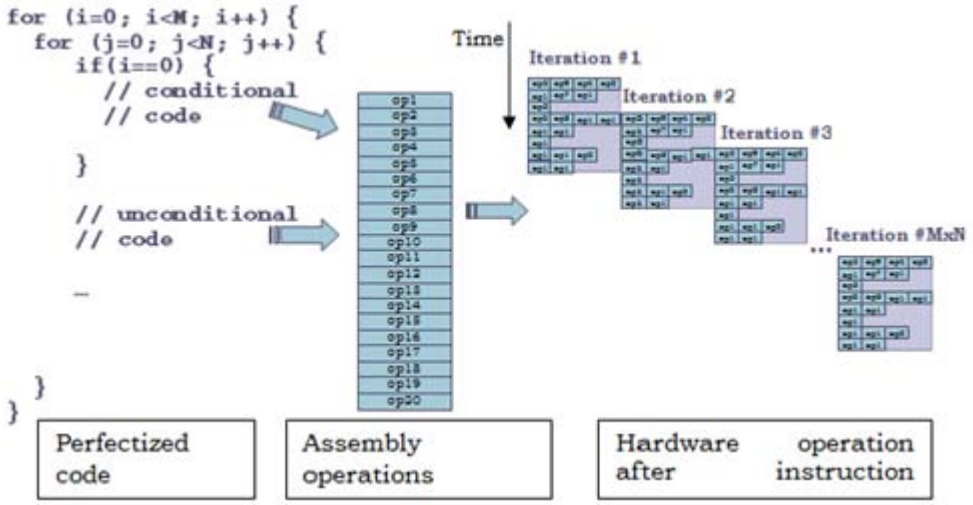
You can specify the desired performance of the design through clock frequency and a parameter called Task_II. For more information on Task_II, see "Performance Metric" on page 5-26.

The first phase in the schedule step is perfectization, which converts the code in the PPA procedure into a set of perfect loop nests. For more information, see "Perfectization" on page 5-16.

The next phase in the schedule step is rate matching. Rate matching determines the relative rates or IIs of all loops in the design after perfectization. Rate matching will set rates to achieve Task_II (if specified), while accounting for stall cycles caused by streaming. In addition, rate matching sets the number of buffers required to meet Task_II for each array in the design. For more information, see "Banking and Multi-Buffering of Memories" on page 8-15. You can view a rate matching summary report using the `report_implementation -rate_match` command. For more information, see "Rate Matching" on page 5-25.

The first phase of the synthesize step is hardware allocation. The Synphony C Compiler tool does an analysis of the performance targets and instruction mix to determine the amount of functional units to allocate in each PA. The next phase of synthesize is instruction scheduling, which schedules instructions onto the PAs functional units.

The tool hardware uses a predicated execution model where each operation has a predicate or condition that determines if the operation is active in each cycle. The natural mapping of code to this hardware is shown in the following figure.



Instruction scheduling is timing driven. The tool maintains critical paths and determines the best schedule based on characterized data from the target technology library.

After instruction scheduling is complete, the synthesize step generates RTL for each PA. Special register structures are instantiated that reduce the area required. A timing controller is designed to control the execution of all PAs. Finally, the top-level interfaces are created and RTL is produced for the whole PPA.

Perfectization

Perfectization step converts the code in the PPA procedure into a set of perfect loop nests. A loopnest is a set of nested FOR statements, such as i-j-k in:

```
for(i=0; i < 10; i++) {
y[i] = 0;
for (j=0; j < 20; j++)
for (k=0; k < 30; k++)
y[i] += x[j] - a[k];
}
```

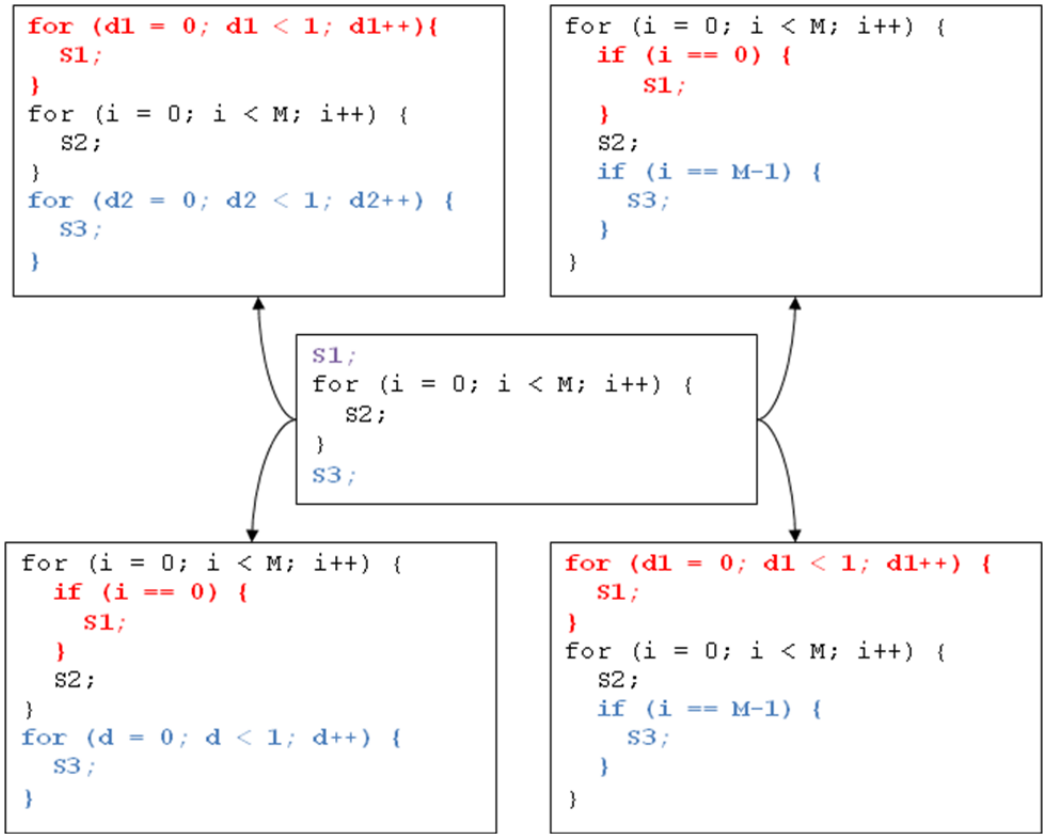
A perfect loop-nest is one in which all the code is inside the innermost loop (that is, there is no code between any consecutive FOR statements within the nest). The previous i-j-k nest is not perfect since there is code between loop i and loop j.

The Synphony C Compiler tool performs perfectization to prepare the code for the PPA artifact architecture, which is optimized to execute perfectly nested loops. Each loop nest executes on a PA in the PPA architecture. This approach enables very high performance designs since each PA has its own stall domain, and all loop nests after perfectization can run in parallel if the data dependences allow for it.

You may have code outside any for loop as shown where S1, S2 and S3 are blocks of code.

```
void ppa() {
     …
     S1;
     for (i = 0; i < M; i++) {
          S2;
     }
     S3;
     …
}
```

The tool explores several mapping options and chooses one among them. For example, for the code shown, the tool has options to make either statement blocks S1 and S3 as individual loops or merge either or both of them into the existing loop nest containing code block S2. This is shown in the following diagram.



Perfectization Choices

Perfectization attempts to minimize the hardware cost while meeting Task_II. However, this can be a complex task and the tool may not always correctly predict the best choice. In addition, small changes in Task_II might change the perfectization choices.

The Synphony C Compiler tool begins perfectization by identifying possible mappings of code to PAs that are supported by the Synphony C Compiler PPA hardware template. For each piece of out-of-loop code and between-loop code, the various sinking or wrapping options are generated.

For each of the options, rate matching is run to choose loop IIs. Any options that do not meet your performance target (Task_II) are eliminated. At this point, no feedback analysis is being done so it is not known if it is possible to meet the chosen IIs.

A simple cost analysis is done for each solution. The set of solutions is sorted by cost.

The solutions are processed in order of lowest expected cost through iteration scheduling (with inter-loop-iteration data feedback analysis). The first solution that passes is chosen.

This solution is known to meet the Task_II target as it has already passed in the rate matching, and now each loop nest (PA) has a valid iteration schedule. Note that this solution must still pass through the detailed instruction scheduling in the synthesize phase.

When optimizing the design you should understand the choices made by perfectization, and be prepared to override them. Perfectization choices may only be overridden by modifying the code (perfectizing by hand).

There are two sources of information about perfectization choices:

- The `report_implementation` command supports the following options:

  `-original_loop_to_line_number`: List the line number for each loop in the original (before perfectization) code

  Example:

```
Loop nest #0: h264_encoder.c:1083: Created by perfectization
Loop nest #1: h264_encoder.c:1091: Created by perfectization
Loop nest #2: h264_encoder.c:439: Was loop 0 before perfectization
Loop nest #3: h264_encoder.c:1103: Was loop 1 before perfectization
Loop nest #4: h264_encoder.c:1196: Was loop 2 before perfectization
Loop nest #5: h264_encoder.c:1345: Was loop 3 before perfectization
Loop nest #6: h264_encoder.c:1447: Was loop 4 before perfectization
Loop nest #7: h264_encoder.c:1635: Was loop 5 before perfectization

Loop nest #8: h264_encoder.c:1720: Was loop 6 before perfectization
Loop nest #9: h264_encoder.c:1816: Was loop 7 before perfectization
Loop nest #10: h264_encoder.c:1917: Created by perfectization
Loop nest #11: h264_encoder.c:862: Was loop 8 before perfectization
Loop nest #12: h264_encoder.c:961: Was loop 9 before perfectization
```

  `-perfectized_loop_info`: Presents information about the results of perfectization

  Example:

```
Loop nest #0: h264_encoder.c:439: Outermost index variable: b
Loop nest #1: h264_encoder.c:1103: Outermost index variable: block0
Loop nest #2: h264_encoder.c:1196: Outermost index variable: block_pair1
Loop nest #3: h264_encoder.c:1345: Outermost index variable: block_pair4
Loop nest #4: h264_encoder.c:1447: Outermost index variable: block2
Loop nest #5: h264_encoder.c:1635: Outermost index variable: block_pair3
Loop nest #6: h264_encoder.c:1720: Outermost index variable: block6
Loop nest #7: h264_encoder.c:1816: Outermost index variable: block_pair7
Loop nest #8: h264_encoder.c:862: Outermost index variable: b
Loop nest #9: h264_encoder.c:961: Outermost index variable: b
```

- A modified C file, applicationName_perfected.c, is produced in the implementation directory. This file shows the perfectized code that was chosen by schedule.

Hand Perfectization: Methods and Quality

In thinking about perfectization with respect to the Quality of Results, the critical factors are:

- In each PA, what does the code imply about the computation done on any given iteration? This will determine the resources required.
- What is the assignment of code to PAs (see ppafilename_perfected.c)? Were any operations that could have been shared assigned to different PAs? If so there will be an impact on the quality of results.

Sometimes you may want to override the perfectization decisions by doing perfectization manually to obtain better quality of results. There are four standard methods for converting code with a non-perfect loop-nest into code with one or more perfect loop-nests:

- Code sinking
- Loop unrolling
- Loop distribution
- Extending the iteration space

This section describes these techniques and their respective implications on cost. We start with the following non-perfect loop-nest:

```
for (i=0; i < 1000; i ++) {
     w[i] = a * b[i] - g * d[i];
     x[i] = c * d[i] - g * b[i];
     for (j = 0; j < 4; j++) {
          z = x[i] + j;
          e[j] = input[j][z] * w[i];
     }
     y[i] = f * z;
}
```

The performance target for this code is Task_II = 4000.

Code Sinking

Code sinking is a transformation that moves out-of-loop code into the inner-most loop under a condition that maintains the semantics of the original code.

For this example, two statements are moved to the beginning of the inner loop under the condition j==0, and one statement is moved to the end of the inner loop under condition j==3.

```
for (i=0; i < 1000; i ++) {
     for (j = 0; j < 4; j++) {
          if (j==0) {
               w[i] = a * b[i] - g * d[i];
               x[i] = c * d[i] - g * b[i];
          }
     z = x[i] + j;
     e[j] = input[j][z] * w[i];
     if (j==3) {
```

```
            y[i] = f * z;
          }
     }
}
```

Although code sinking is often the easiest way to perfectize a loop-nest, it can lead to the most expensive hardware. After code sinking, the performance requirement of Task_II=4000 translates into one cycle per iteration.

The tool synthesizes hardware for all code in the inner-most loop, even if it rarely executes. In this example, there will be at least five multipliers instantiated to handle the iterations in which j==0. If the tool is unable to time-multiplex the f * z multiplier, then a total of six multipliers will be synthesized. All but one of these multipliers will be under-utilized. As a result, code sinking may lead to many under-utilized resources.

Code sinking is discouraged if the out-of-loop operations require expensive hardware. However, for simple operations such as shift, mask, add, subtract and comparison, code sinking will almost always be the most cost effective method of perfectizing the code.

Code sinking is unsafe and may change the semantics of your code if the containing loop nest does not execute at all (since the sinked-in code executes in the first iteration).

Code sinking may also have implications to performance through shared data. Consider this example:

```
c0 = control >> 3 & 0xff;
c1 = coeff*c0;
for (i=0; i < M; i ++) {
     x[i] = f1(y[i],c1);
}
for (j=0; j < N; j ++) {
     z[j] = f2(y[j],c1);
}
```

Assume that the tool chooses to perfectize the code by creating a separate PA for the first two lines of code. In this PPA, the two loop nests may run entirely in parallel, after a small number of cycles in which c0 and c1 are computed. Now suppose that we sink the out-of-loop code into the first loop. The result is as follows:

```
for (i=0; i < M; i ++) {
     if(i==0) {
          c0 = control >> 3 & 0xff;
          c1 = coeff*c0;
     }
     x[i] = f1(y[i],c1);
}
for (j=0; j < N; j ++) {
     z[j] = f2(y[j],c1);
}
```

Now the two loop nests are sequentialized, since the first loop defines c1 and the second loop uses it. Sequential execution of the two loops may make it extremely hard to get the required performance.

When your out-of-loop code consists of any expensive operations, such as multipliers, dividers, loads, or stores, or when the total number of out-of-loop operations is large, it is preferable to use other techniques such as loop unrolling, loop distribution, and extending the iteration space.

Loop Unrolling

If the inner-most loop is small enough, loop unrolling is often a better alternative to code sinking. In this example, the inner-most loop has a trip-count of four and can easily be unrolled as shown on the next page.

After loop unrolling, the performance requirement of Task_II =4000 translates into four cycles per iteration. As a result, the tool time-multiplexes the nine multiply operations onto three multipliers. This is a tremendous savings over the five or six multiplier result achieved with code sinking.

```
for (i=0; i < 1000; i ++) {
     w[i] = a * b[i] - g * d[i];
     x[i] = c * d[i] - g * b[i];
     z = x[i] + 0;
     e[0] = input[0][z] * w[i];
     z = x[i] + 1;
     e[1] = input[1][z] * w[i];
     z = x[i] + 2;
     e[2] = input[2][z] * w[i];
     z = x[i] + 3;
     e[3] = input[3][z] * w[i];
     y[i] = f * z;
}
```

Loop unrolling is not always possible. The performance requirements and the trip counts must be such that after unrolling the resulting initiation interval is small (II between one and eight is ideal).

Loop Distribution

Loop distribution forms a new perfect loop-nest containing a targeted set of statements - an out-of-loop code block in the case of perfectization. Our example has two out-of-loop code blocks, yielding two new loop-nests after distribution.

```
// Loop 0
for (i=0; i < 1000; i ++) {
     w[i] = a * b[i] - g * d[i];
     x[i] = c * d[i] - g * b[i];
     scc_stream_output_x(x[i]);
}
// Loop 1
```

```
for (i=0; i < 1000; i ++) {
      for (j = 0; j < 4; j++) {
          if (j==0) cur_x = x.get();
          z = cur_x + j;
          e[j] = input[j][z] * w[i];
          if (j==3) scc_stream_output_z(z);
     }
}
// Loop 2
for (i=0; i < 1000; i ++) {
     y[i] = f * z.get();
}
```

To enable parallelism, all communication between loop nests is implemented using streams. As a result, we created two FIFOs: one to pass the x array values from loop 0 to loop 1, and one to pass the z scalar values from loop 1 to loop 3. The stream calls in loop 1 are guarded by the same conditions seen in the code sinking example (j==0 and j==3).

After loop distribution, the performance requirement of Task_II =4000 translates into four cycles per iteration for loops 0 and 2, and one cycle per iteration for loop 1. Different performance targets for each loop allow the tool to obtain much better resource sharing than code sinking.

The Synphony C Compiler tool synthesizes one multiplier for loop 0 (100% utilized), one multiplier for loop 1 (100% utilized), and one multiplier for loop 2 (25% utilized). This yields a total of three multipliers - the same result obtained from loop unrolling.

However, loop distribution has additional streaming and control overhead not present in loop unrolling. In addition, the tool does not share resources across loops. Loop unrolling keeps a single loop and enables more sharing.

Note:
Loop distribution is not always possible because of data dependencies.

Iteration Space Extension

A common occurrence in some algorithms doing frequency/time or frequency/spatial domain transformations (example: FFT, DCT) is to handle DC and AC components differently. Sometimes this is expressed as a first step for DC and then a loop for AC:

```
encode_DC();
     for(i=1;i<64;i++) {
     encode_AC(i);
}
```

Suppose that encode_DC( ) and encode_AC( ) share a memory — DC value is written to element 0 and AC values to elements 1-63. In addition, they may have common operations such as multiplications, divisions, etc. The Synphony C Compiler tool considers the following two perfectization choices for the code:

- Wrapping the out-of-loop code into a PA:

```
for(i=0;i<1;i++) {
     encode_DC();
     }
for(i=1;i<64;i++) {
     encode_AC(i);
}
```

- Sinking the out-of-loop code into the loop:

```
for(i=1;i<64;i++) {
     f(i==1) encode_DC();
     encode_AC(i);
}
```

In this case, neither of these is the best choice. Assume that encode_DC( ) and encode_AC( ) each make one read and one write access to a shared memory m[ ], and that each performs one multiplication. Furthermore, assume that performance target is Task_II of 64-one cycle per data element.

If we wrap the out-of-loop code, we will have two PAs. Each one requires a load unit, a store unit, and a multiplier. We need two ports to m[ ] but they can be arbitrated or multiplexed between the two PAs.

If we sink the out-of-loop code, we will have a single PA. However, on its first iteration it makes four memory accesses for encode_DC( ) and encode_AC( ). It also performs two multiplications on the first iteration. At Task_II = 64, we will have II=1 and require four ports to m[ ]. In this case, the best thing to do is modify the code by hand to get the most efficient operation:

```
for(i=0;i<64;i++) {
     if(i==0) encode_DC();
     else encode_AC(i);
}
```

This coding will have one load, one store and one multiply on each iteration. These will be shared between the encode_DC( ) and encode_AC( ) functions because these calls are now mutually exclusive.

Conditional Loop Statements

The Synphony C Compiler tool supports loops under if( ) statements. It converts this syntax into a perfect loop nest form through compiler transformations. A different transformation is applied based on whether such a condition is outside the entire loop nest or inside a loop. The following examples show the transformations.

Example 1: Conditional execution of loops at the outermost level

```
if (cond) {
     for (i=0; i<M; i++) {
     L1;
```

```
     }
} else {
     for (j=0; j<N; j++) {
          L2;
      }
}
```

It is transformed into the following code:

```
tmp = cond;
for (i1=0; i1<M; i1++) {
     if (!tmp) break;
     L1;
}
for (i2=0; i2<N; i2++) {
     if (tmp) break;
     L2;
}
```

Each loop nest can be executed at a different II and loop numbering will be in the order written in your program. Depending on the performance target for the PPA and the cost estimation, perfectization can choose to either evaluate the condition evaluation statement into a separate loop nest allowing the loop nests to execute in parallel or sink in to the first loop nest causing them to execute sequentially.

Example 2: Conditional execution of loops at inner levels

```
for (outer=0; outer<size; outer++) {
     if (cond) {
          for (i=0; i<M; i++) {
               L1;
          }
     } else {
           for (j=0; j<N; j++) {
                L2;
            }
     }
}
```

The inner loops will be jammed into a single loop nest with the loop body of each loop nest executing conditionally.

```
for (outer=0; outer<size; outer++) {
     tmp = cond;
     ub_mn = cond ? M : N;
     for (ij=0; ij<ub_mn; ij++) {
          if (tmp) {
               L1;
          } else {
               L2;
          }
     }
}
```

The loop numbering and the reports will refer to the jammed loop nest. The (min, target, max) iterations specified through num_iterations pragma will be merged as (min(i,j), max(i,j), max(i,j)). The hardware between the inner loops can be shared in the generated PA corresponding to this loop nest. Note that the bitsize of the new loop index will be large enough to contain all the merged loop indexes.

The Synphony C Compiler tool will not create any new loops to enable loop jamming. For example,

```
if (cond) {
     S1;
} else {
     for(i = 0; i < M; ++i) L1;
}
```

will be transformed as

```
tmp = cond;
if (tmp) S1;
ub = (cond ? 0 : M);
for(i = 0; i < ub; i++) if (!cond) L1;
```

In this case, cond and!cond will not be determined as mutually exclusive and therefore S1 and L1 cannot share hardware. The user can force jamming in this case by enclosing S1 in a single trip count loop.

Rate Matching

Rate matching determines the relative rates - or IIs - of all loops in the design after perfectization. Rate matching will set rates to achieve Task_II (if specified), while accounting for stall cycles caused due to streaming data. In addition, rate matching sets the number of buffers required to meet Task_II for each array in the design. For more information, see “Banking and Multi-Buffering of Memories” on page 8-15. You can view a rate matching summary report using the `report_implementation -rate_match` command.

After the IIs for the loops are chosen by the rate matcher, the schedule step performs iteration scheduling. This step attempts to find an efficient execution ordering of iterations in each loop nest that satisfies estimated timing constraints of feedback paths in the code. If no iteration schedule can be found that satisfies all feedback paths, the feedback is printed and a new perfectization choice (if any) is considered. For more information, see “Handling Feedback Paths” in the *Synphony C Compiler User Guide.*

When only the Task II is specified, the tool attempts to compute II values of those loops for which you have not specified an II value. If Task_II and IIs are specified, the tool checks if those values are consistent (using the IIs specified would in fact deliver the required task II).

If Task_II is specified, the compiler computes II for each loop so that the rate of production and consumption of stream data is matched. This leads to small, bounded FIFOs. This also ensures that there is only one transaction per cycle for each stream port. The overall Task_II is achieved with as less area as possible. Overall Task_II may be computed, or verified.

Rate Matching with Variable Loop Bounds

If loop bounds are fixed at compile time then the tool determines the exact number of iterations. If loop bounds are variable, the user can assist with Task_II targeting using the following pragma:

#pragma num_iterations (<min>,<target>,<max>)

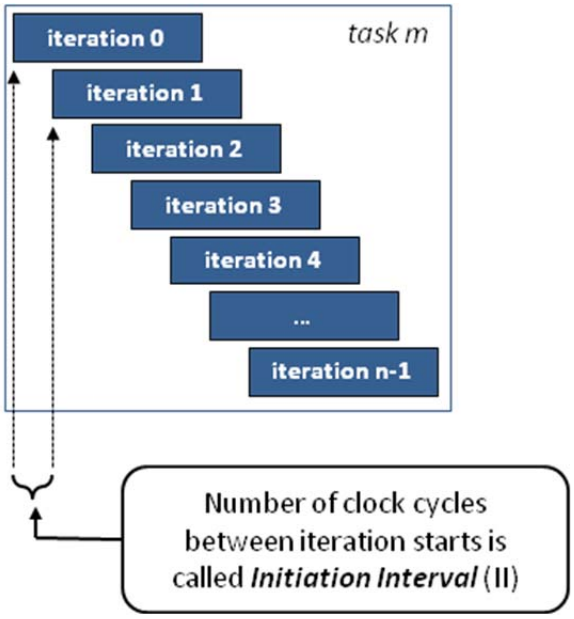
Place this statement immediately before the loop. You may specify any or all of the three values. If specified, the target value is used by the compiler to meet Task_II.

Performance Metric

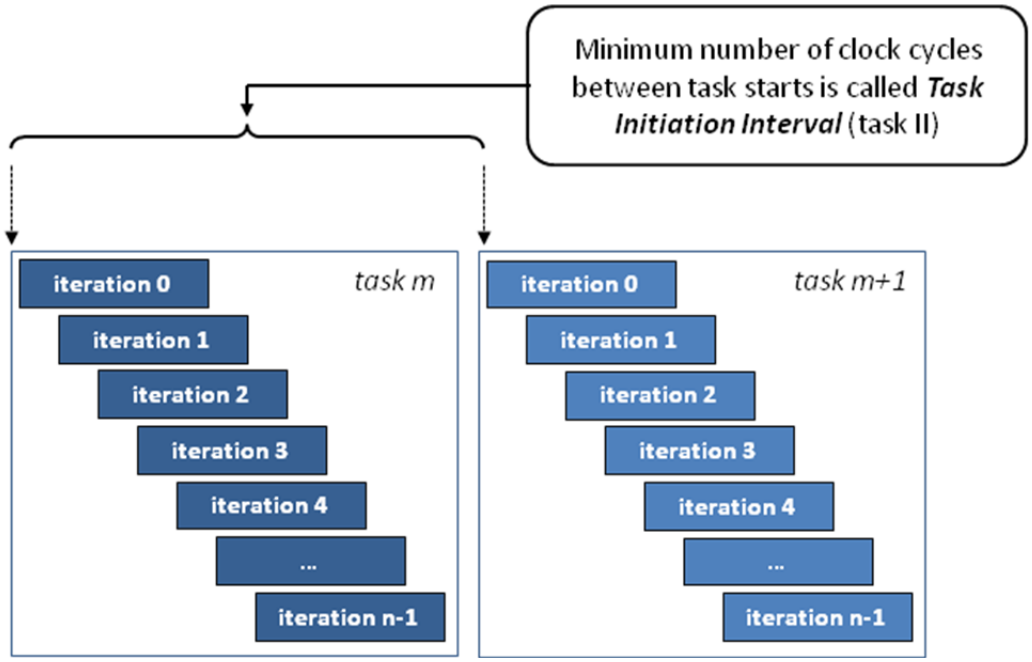
II (Initiation Interval) is the number of cycles between starting new iterations of a loop. Consider the following code segment.

```
void f(char a[32], char b[32], char c[32],
     char qa[32], char qb[32], char z)
{
      unsigned char k;
     for(k = 0; k < 32; k++) {
          long t = (a[k]*qa[k] + b[k]*qb[k]) >> 2;
          t = t * z;
          if (t > 127) t = 127;
          else if (t < -128) t = -128;
               c[k] = (char) t;
          }
}
```

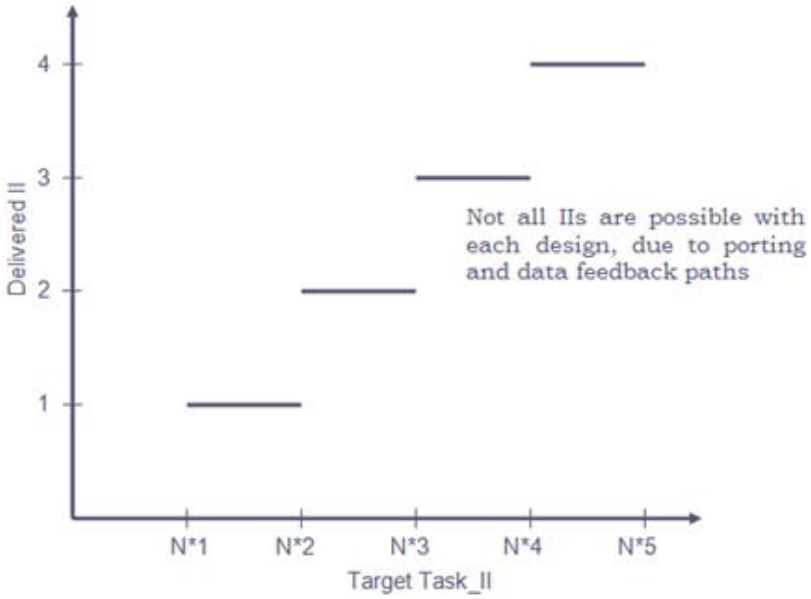
Here, II is the number of cycles that elapse between successive iterations of the k loop. You can specify II values of some loops, all the loops or none of them. If IIs are specified, the tool checks the design for memory and stream bandwidths and other resource usage requirements.



Task_II (Task Iteration Interval) is a goal for the shortest time between invoking successive calls to the function mapped to hardware (number of cycles between task m and task m+1 in the following figure). This is a target specification which the tool tries to achieve. The delivered task II may be different due to several reasons. It is a static calculation based on loop iterations and throughput, not a simulation-based number.



Note that there is a step-wise relationship between Task_II and II (the relationship becomes more complex for PPAs with multiple PAs): This is shown in the following figure.



Effect of II on Generated Hardware

It is very important to understand how the Synphony C Compiler tool creates hardware and the meaning of iteration interval (II) in that hardware. Consider the following code:

```
for(i=0;i<N;i++) {
     a[i] = a[i]*coeff1 + b[i]*coeff2;
}
```

Assume we target a Task_II of N. This design will have a single PA corresponding to the loop nest. It will have to run at II=1 to meet our Task_II target. If we look at the operations implied by the code, we have to read and write an element of a[ ] on each iteration. We also have to perform two multiplications in each loop iteration. Finally, we need one addition and another load for b[ ].

To do all this at II=1, we require the following functional units:

- Two load units (one for a[ ], one for b[ ]);
- One store unit (for a[ ]);
- Two multipliers;
- One adder.

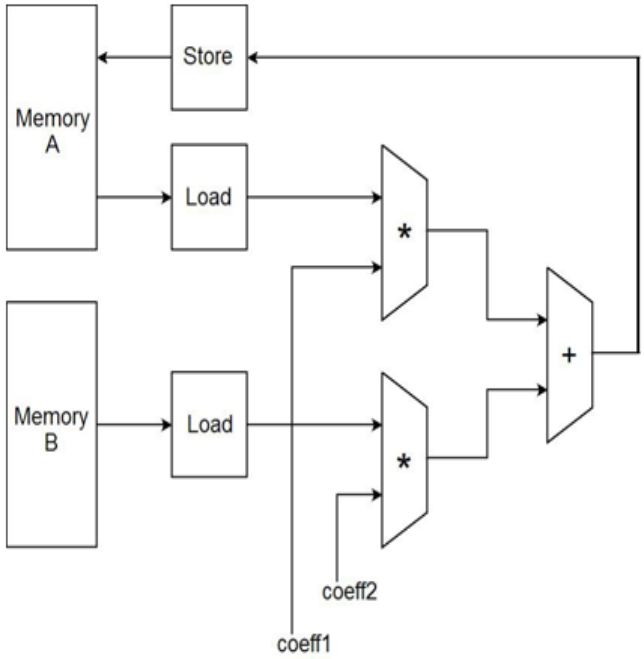
Now, assume we relax Task_II to 2*N. This will allow the design to run at II=2. This means that for any given loop iteration, we are able to use a set of resources twice - on two phases of II - to perform the computation. This would require a different set of units:

- One load unit (for b[ ]);
- One load/store unit (for a[ ]);
- One multiplier;
- One adder.

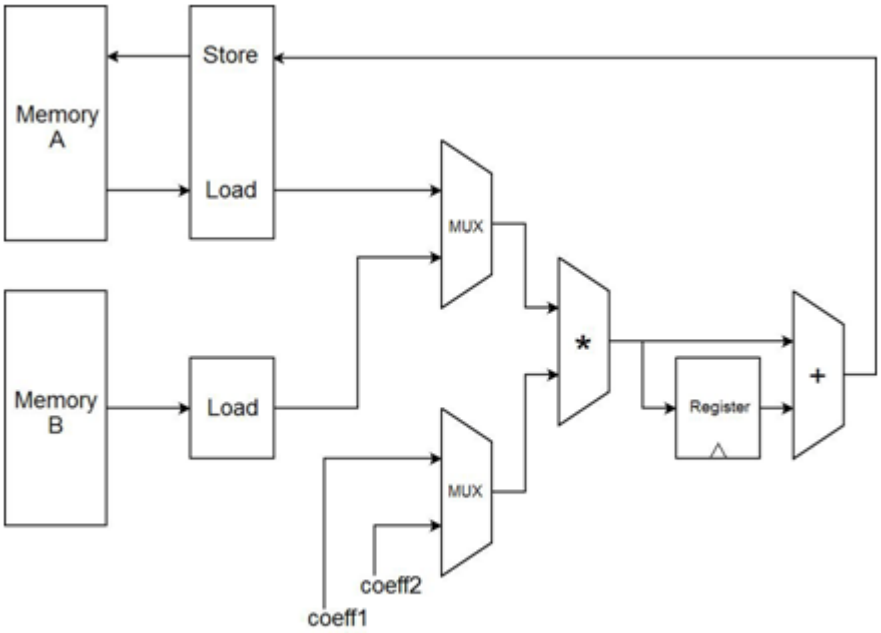
An additional benefit is that we require only one memory port for a[ ].

The load/store unit is a compound functional unit that can execute either a load or store on a single memory port. It is useful in the II=2 design because it can execute both operations on alternate cycles. Since the II=1 design must execute both load and store for a[ ] every cycle (implying two memory ports), it cannot benefit from this shared unit.

Schematics of the II=1 and II=2 implementations are shown in the following figures.



Implementation with II=1



implementation II = 2

The physical implementation of the resource sharing involves multiplexing the inputs and fanout on outputs of functional units. For instance, the single multiplier in the II=2 design has a multiplexer at the input to choose between coeff1 and coeff2 for one input, and between a[i] and b[i] for the other. Its output feeds the input of the adder, as well as a register to store an intermediate result (this register's output feeds the adder's other input).

The II=1 datapath is much more straightforward and resembles the dataflow graph of the operations in the code. This difference in construction between II=1 designs and higher II designs is critical to quality of results. Synthesis tools can perform many more optimizations on functional units that are directly connected than those that are connected through multiplexers and other logic.

As II goes higher, the style of the design stays the same, but more multiplexing is added as functional units perform different tasks on the additional phases of II. These higher II designs will have increased register and interconnect cost, but lower functional unit cost.

Factors Affecting II

Understanding the factors which impact II is useful to interpret the automatic II selection process in the rate matching phase. Broadly, the following rules apply.

- Bandwidth of streams: There can be only one call to a stream per cycle. So, if you have more than one call to a stream within an iteration, the achievable II of the loop is higher.
- Constraints due to imported TCABs: The II achievable for a PA depends on the pipelinability of the TCABs used within the PA. If the PA utilizes the epilog stage pipelining capability of a TCAB, then the target II of the PA should be compatible with the pipelinability of the TCAB. This is because the resources get shared between different time slots within a TCAB pipeline stage. So it would be incorrect for the PA to issue a new task to the TCAB on a non-stage boundary when there is a previous task still executing on the TCAB.

  For instance, consider a non looping TCAB that has a pipelinability of 2 and the schedule length is 4. In this case II=3 is not feasible for a PA that uses the TCAB. This is because a new task would be issued to the TCAB at time 3 and since the TCAB resources are shared between times 3 and 4, the new task and the previously issued task in flight would conflict for resources. However, if the schedule length of the TCAB were 3, then at II=3, the PPA is not going to use the epilog stage pipelinability of the TCAB and therefore it would be a feasible II for the PA.

Specified Performance Versus Delivered Performance

The compiler uses Task_II at compile time to design hardware which meets the required performance. The delivered performance may be different for two reasons:

- Dynamic effects like stalls because of non-availability of data
- Different tasks may take different time to execute

You can use performance charts to find the delivered Task_II. In case multiple tasks are run, the average, min and max Task_II values are reported.

Common Performance Bottlenecks

Performance bottlenecks can be due to high level code organization or due to operation scheduling. The following guidelines can be used to deal with the different categories of the performance issues.

- Block-level performance issues

  For meeting throughput requirements, you have to first identify the desired parallelism at the block or loop nest level. Then identify any perfectization choices that contribute to the existing performance bottleneck and re-code with perfectization techniques described in "Hand Perfectization: Methods and Quality" on page 5-18. This may require you to unroll a few loops which in turn cause an increase in the number of copies of hardware.

  Answers to the following questions might guide you to the right parallelism strategy.

What is the right task size? Is task overlap required?

Where should streams versus memory connections be used?

Resolution of memory related bottlenecks may require adopting a proper memory architecture including changing the type of memory implementations. For more information on meeting block level performance issues, see the *Synphony C Compiler User Guide*.

- Operation-level performance issues

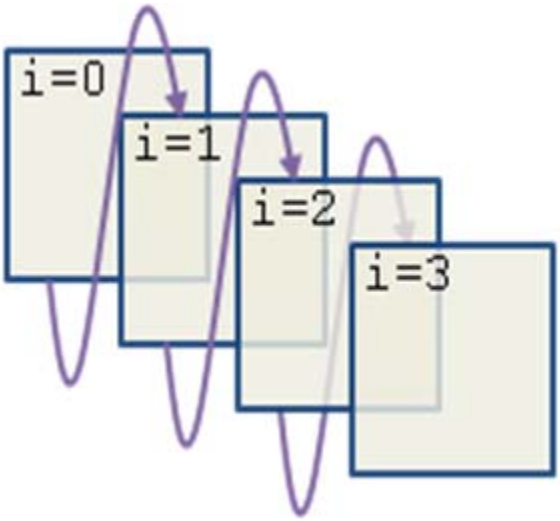
  Inability to meet II or frequency in your design can be addressed by identifying the right critical feedback paths in your design and resolve the items through any possible re-coding or usage of additional constraints or relaxing existing constraints. It is important in several cases to optimize your design for the common case and encapsulate the infrequent and long critical path into a different hierarchy.

Feedback Paths

Applications written for the Synphony C Compiler tool often include data feedback paths where the value computed in the current iteration of a loop-nest depends upon the values computed in previous iterations. Feedback paths can span across a single iteration or across several iterations. The number of iterations between a value's computation and its last use is called the feedback depth.

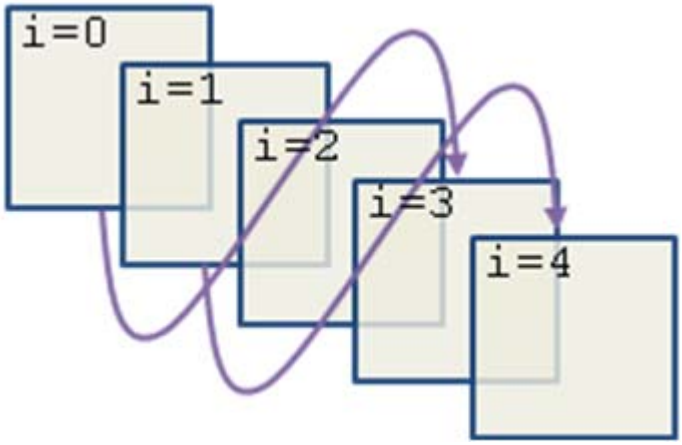
Example1: (depth=1)

```
for(i = 0; i < N; i++) {
          if(c)
               c = c.get();
     }
```



Example 2: (Depth = 3)

```
for(i = 0; i < N; i++) {
     X[i+3] = f(X[i]);
}
```



The greater the feedback depth the more is the parallelism exposed because many more operations can be executed in parallel before a dependent value is needed.

The Synphony C Compiler tool easily expresses and analyzes feedback paths through affine arrays. For example, the Synphony C Compiler tool determines that x[i] = x[i-8] * z (where i is a loop index) has a feedback depth of 8. C also supports implicit feedback paths for scalars, such as x++, which is a feedback depth of one (a dependence on the previous iteration). However, a feedback depth that is shallow is much harder to parallelize.

In an affine array, all the indexes are integer loop combinations of the loop indexes. This makes it easy for the tool to parallelize the code. For non-affine arrays, the tool tries to make the code parallel, but if it cannot, it sequentializes the code.

Data Structures and Hardware Mapping

Scalar Variables

Globally declared, input and output scalar variables map to a register or in-put/output wires. Such registers can be loaded or examined by the host using an API if the protocol for making such an access is not violated. The register or wire may be sized according to the specified or inferred datawidth. An external hardware controller can either set the data or read the data from the registers or wires.

Local scalar variables may map to wires or registers depending on operation schedule. No hardware state is retained across tasks calls for these scalar variables.

Arrays

Arrays in your code can be mapped as any of the following.

- External memories
- Synthesized memories
- Registers/Wires
- Block Rams (in case the target device is an FPGA)

For more information on memory support, see “Memory Support” on page 8-6.

Array Partitioning

The Synphony C Compiler tool maps array variables to the system's memory hierarchy. The tool automatically creates dedicated memory ports or uses arbitration among multiple clients to provide shared access to a single array, depending upon the bandwidth needed.

To achieve more cost-effective implementations, the user may manually partition the arrays in the application into multiple sub-arrays such that the number of accesses per array is reduced. This enables the tool to synthesize a banked memory structure at a lower cost due to reduced porting requirements.

Example: (before partitioning)

```
char in[4][1024], out[1024];
void app( )  {
     int i;
     for (i = 0; i < 1024; i++)
     out[i] = (in[0][i] + in[1][i] + in[2][i] + in[3][i]) / 4;
}
```

There are four references to array in per iteration of the loop-nest. If the performance target is one cycle per iteration (Task_II = 1024), Synphony C Compiler will map array in to a 4KB four-ported memory (an external SRAM in this case). Such a memory may be prohibitively expensive (or even infeasible). Lowering the performance target to four cycles per iteration (Task_II = 4096) would reduce this to a 4KB single-ported memory. However, a simple code rewrite allows both high performance and single porting.

Example: (after partitioning)

```
char in0[1024], in1[1024], in2[1024], in3[1024], out[1024];
void app( )  {
     int i;
     for (i = 0; i < 1024; i++)
     out[i] = (in0[i] + in1[i] + in2[i] + in3[i]) / 4;
}
```

Now if the performance target is one cycle per iteration, the tool maps the four in variables to four 1KB single-ported memories - a substantially cheaper memory design than one 4KB four-ported memory.

The tool performs this banking optimization automatically under certain conditions. For more information, see "Banking and Multi-Buffering of Memories" on page 8-15.

Memory Porting

For arrays mapping to SRAM or ROM, the number of memory references can also affect the cost and performance of the design. The tool attempts to reduce memory references through various methods.

The following code (where none of the array references to b[ ] may be eliminated), would require four ports at II=1, two ports at II=2, and one port at II=4:

```
for(i=0;i<N;i++) {
     x1 = b[i][0];
     x2 = b[i][1];
     …
     b[i][2] = f(x1);
     b[i][3] = f(x2);
}
```

Common sub-expression elimination (CSE) and other optimizations may reduce the number of memory references. CSE will remove multiple
references with the same array indexes. The effective number of memory
references per iteration may be reduced if certain references can be proven by the compiler to be mutually exclusive. While coding, you should understand the meaning of the array references in the context of these optimizations.

Managing memory porting is critical to good Quality of Results. To get the right porting, you need to write efficient code as well as understand the strengths and limitations of the compiler.

For additional information on minimizing porting, see the following:

- "Memory Arbitration" on page 8-14
- "packed Pragma" on page 7-53
- "Banking and Multi-Buffering of Memories" on page 8-15
- "Array Partitioning" on page 5-35
- "Reducing Porting" on page 5-38.

However, there are limitations to this CSE. The matching of index expressions to detect common array references is purely syntactic (i.e. they must be identical). For example,

```
j = 2 * i + 1;
  k = 2 * i + 1;
  result = a[j] + a[k];
```

will result in two array references instead of one.

Array CSE will not hoist a reference outside a condition. This is to avoid adding references along a path where there were none to start with. In few cases, you may have to introduce additional references along such paths to trigger CSE. For example,

```
if (cond_1)
       a = x[i];
  if (cond_2)
       b += x[i];
```

can be transformed into

```
tmp = x[i];
  if(cond_1) a = tmp;
  if(cond_2) b += tmp;
```

to make the x[i] unconditionally read statically.

In the presence of break, continue or return statements, it might be impossible to guide the compiler in this manner since it introduces the guard conditions to implement the semantics of the early exits.

```
if (cond) a = x[i]; // S1
    if (error) return -1; // S2
    b += x[i]; // S3
```

If intermediate return statement is infrequently executed, it might actually be a good idea to reuse x[i]. However, there is no way to write this so that the compiler detects this (both S1 and S3 will be guarded and the therefore the read of array x at S1 is no longer unconditional).

You may need to manually perform CSE in this case (assuming the read x[i] is safe to do).

```
    int x_i = x[i];
    if (cond) a = x_i;
    if (error) return -1;
    b += x_i;
```

In certain cases, the compiler cannot prove that certain access are same and it would end up creating multiple ports to meet the performance criteria. For example consider the snippet of FFT example:

```
for (i = 0; i < m; i++){
       n1 = n2;
       n2 = n2 + n2;
       for (j = 0; j < n1; j++){
            c = cos_fix[j];
```

```
            s = sin_fix[j];
            for (k = j; k < n; k = k + n2){
                 t1 = (c * x[k + n1] - s * y[k + n1]) >> 15;
                 t2 = (s * x[k + n1] + c * y[k + n1]) >> 15;
                 x[k + n1] = x[k] - t1;
                 y[k + n1] = y[k] - t2;
                 x[k] = x[k] + t1;
                 y[k] = y[k] + t2;
            }
       }
  }
```

In the piece of code, the compiler cannot prove that n1 is not zero. Therefore, the compiler assumes that there is an intervening write in between two reads. However, the user can prove that n1 is never zero and the write happen to a different location. The user can avoid creating multiple ports in this case by using temporary variables (xtmp, ytmp) as shown:

```
for (i = 0; i < m; i++){
       n1 = n2;
       n2 = n2 + n2;
       for (j = 0; j < n1; j++){
            c = cos_fix[j];
            s = sin_fix[j];
            for (k = j; k < n; k = k + n2){
                 int xtmp = x[k];
                 int ytmp = y[k];
                 t1 = (c * x[k + n1] - s * y[k + n1]) >> 15;
                 t2 = (s * x[k + n1] + c * y[k + n1]) >> 15;
                 x[k + n1] = x[k] - t1;
                 y[k + n1] = y[k] - t2;
                 x[k] = xtmp + t1;
                 y[k] = ytmp + t2;
            }
       }
  }
```

Reducing Porting

The user may be able to reduce memory porting by restructuring their code. Consider the following statements:

```
if (mode == 0)
     m = X[a][b];
else if (mode == 1)
     m = X[c][d+e];
else
     m = X[d][e];
```

The Synphony C Compiler tool recognizes that all three references to X are mutually exclusive and will attempt to share a single LOAD unit. However in some rare cases, the tool will be unable to perform complete sharing due to interactions with the array references and other operations in the code.

To guarantee that the tool minimizes the number of LOAD units (and, in turn, the porting requirements), the code can be rewritten as:

```
if (mode == 0) {
     arg1 = a; arg2 = b;
}
else if (mode == 1) {
     arg1 = c; arg2 = d+e;
}
else {
     arg1 = d; arg2 = e;
}
m = X[arg1][arg2];
```

Communicating with the External Environment

The Synphony C Compiler tool provides easy to use interfaces for external environment to control the hardware, query its status and exchange data. For more information on interface control, see the Chapter 14, “System Integration and Verification. The compiler supports data communication with the external environment through the following C/C++ variable types:

- Arrays (file or procedure scope)
- Arrays (file or procedure scope, and internal_fast)
- Scalars (file scope)
- Streams (via special Synphony C Compiler constructs)

In hardware, these variable types become:

- Local memory interfaces (compatible with 3rd party SRAMs)
- Local memories (Synphony C Compiler generated register-based, synthesizable RAMs)
- ROMs (External or Synphony C Compiler synthesized)
- Memory mapped to the hostLM interface
- FIFOs, for more information, see “Stream Interfaces” on page 14-9.

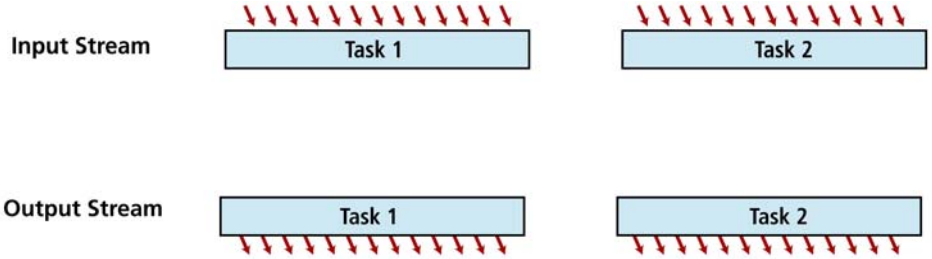
Often it is advantageous to pack many small data elements into a single wider element (up to 32-bits) for arrays or streams. For example, you can pack four bytes into a single int. For more information on packing bytes, see “packed Pragma” on page 7-53.

The following are additional requirements imposed on the way user accesses and modifies the entries mentioned. It is important to not violate this protocol from the external environment.

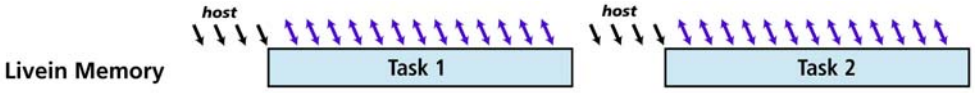
- Scalars can be updated from the host either before a task starts or in between tasks. The PPA may sample the scalar data during the execution of the task.

- Scalar outputs may be updated by the PPA during the execution of a task and host access to such scalars is valid only at the end of a task.

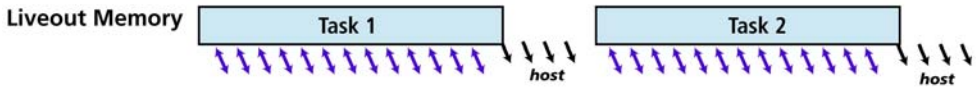
- Input and output streams are active when the task is in progress



- Memory (array) inputs into the PPA from the host need to be complete before a task starts. PPA may update the memory during the execution of the task.



- Memory (array) outputs may be modified by the PPA during the execution of a task. Any host access to such arrays should be done only at the end of a task.



Note:
Memories and scalars may be both input/output simultaneously.

Memory Mapping

The control and data interfaces provided by Synphony C Compiler can be optionally a memory mapped interface. The following table describes the options to get a memory mapped interface for any component.

Components	Options
Control	set_params -interface_type memory_mapped_handshake
Scalars	set_params -host_scalar_access #pragma host_scalar_access
Arrays	set_params -host_memory_access #pragma host_memory_access
Streams	Cannot be memory-mapped

The host memory map can be specified using the following project parameter setting.

```
set_params -host_address_map …
```

For more information, see the Chapter 14, “System Integration and Verification.

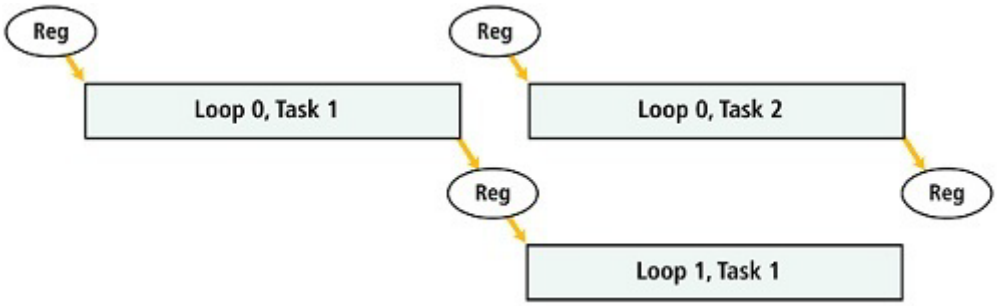
Loop Dependence Rules Across Tasks

It is useful to understand any loop dependence due to the protocol requirements for read and write accesses between PPA and the host mentioned. Consider a two loop design where L0 executes ahead of L1 in original C code.

The following examples explain *noncontinuous processing* scenarios. In continuous processing designs, loops are not sequentialized. Instead, they run in parallel but synchronize at dependence points.

Sequentialization Due to Scalar Usage (read/write)

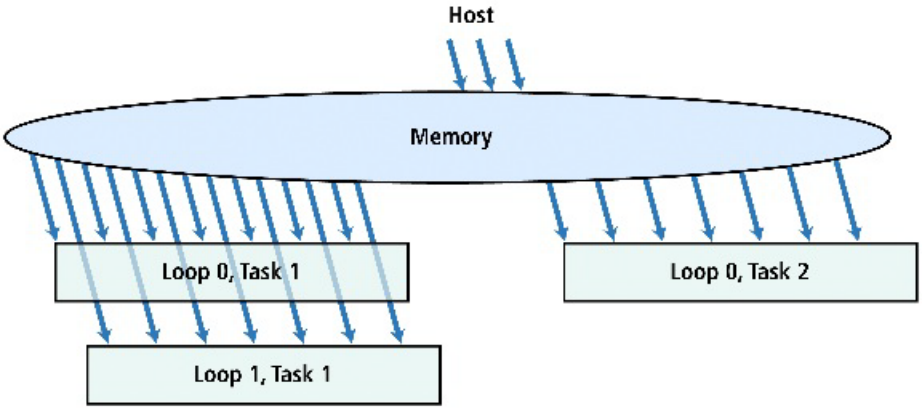
Consider the case where Loop 0 reads a scalar variable from host and updates it, and Loop 1 reads the updated value of the scalar variable.



L0 can start a new task as soon as L0 has completed the current task and L1 has started the current task.

Sequentialization Due to Memory Reads

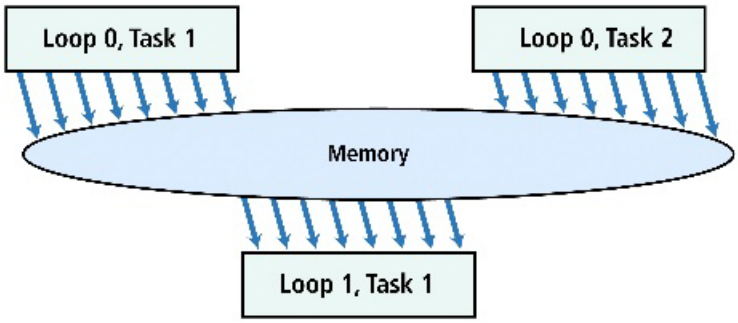
Consider the case where loop 0 and loop 1 read an input memory.



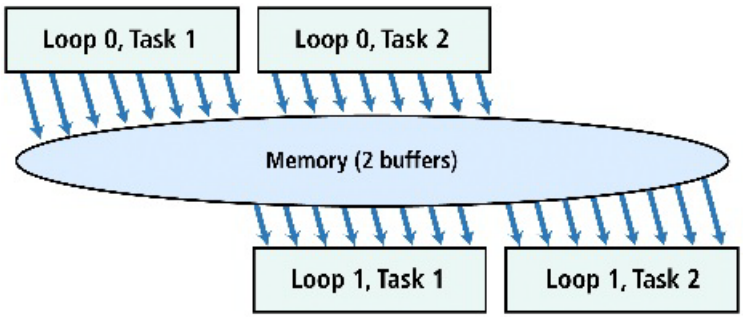
L0 can start a new task as soon as L1 has completed the current task and the host has initialized the memory for the next task.

Sequentialization Due to Memory Read and Writes

Consider the case where Loop 0 updates a memory and Loop 1 reads the memory.



If the memory has a single buffer, Loop 0 can start again for the next task when L1 has completed the current task as shown. If the memory is multi buffered, then Loop 0 can start again for the next task as soon it has completed the current task. When the Loop 0 updates a buffer, Loop 1 reads from the other buffer and vice versa.



Communication Between PA Modules

PAs can communicate through scalar variables or arrays, or through flow-controlled interfaces such as FIFOs.

- Using scalar variables or arrays to communicate between loops forces the PA modules to work sequentially for noncontinuous processing designs. PAs in continuous processing designs work in parallel but synchronize at the dependence points.

- A FIFO between loops enables the corresponding PAs to work simultaneously: one PA pushes data into the FIFO when data is available and the other PA reads the FIFO when data is required. The FIFO must be sized appropriately to ensure good performance in the hardware. Note that if two loops use a FIFO for a similar operation (both read or write to the FIFO), the loops work sequentially, not simultaneously. For more information on FIFOs, see "Stream Support" on page 8-24.

The Synphony C Compiler software automatically designs synchronization hardware for the PPA to guarantee that the PAs execute in a manner that satisfies all data dependences.

When a procedure is implemented as a hardware pipeline, multiple invocations may be pipelined through it without waiting for the previous invocation to finish. This is because the hardware resources of one loop nest may become free as soon as that loop nest completes usage and can be reused for the next task even if subsequent loop nests are busy with the previous task. For more information, see "Task Overlaps" on page 5-57.

Each scalar shared between PAs is registered by each PA that writes it and each PA that reads it. This is required for correct operation with stall domains with task overlap. However, if there is no task overlap and the reading PA is consuming the scalar as read-only, then the tool only creates a register for the writing PA, and the reading PA reads directly from that register.

Sometimes you can express variables either as scalars or as small arrays, or even arrays of a single element. These choices have their pros and cons:

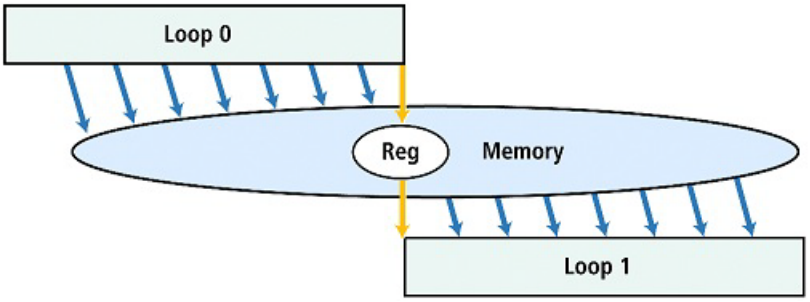
- Shared scalars automatically provide double-buffering to allow task overlap, but this means that storage is duplicated. By contrast, an array (that becomes internal_fast) enable the smallest possible storage: a single register. On the other hand, it makes task overlap impossible, unless the array is multi-buffered.
- The compiler has more flexibility in the handling of feedback paths and data dependences when variables are expressed as scalars. You might achieve higher performance targets when variables are expressed as scalars.

To summarize, using shared scalars generally provides the easiest path to coding and meeting performance, but you might consider using small arrays to fine-tune the cost of the design.

Loop Sequentialization Rules Within a Task

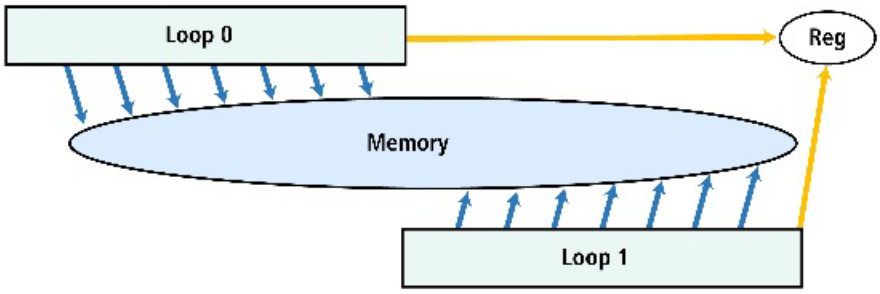
The following examples explain *noncontinuous processing* scenarios. In continuous processing designs, loops are not sequentialized. Instead, they run in parallel but synchronize at dependence points.

- Loop 0 writes memory or scalar x and loop 1 reads x.



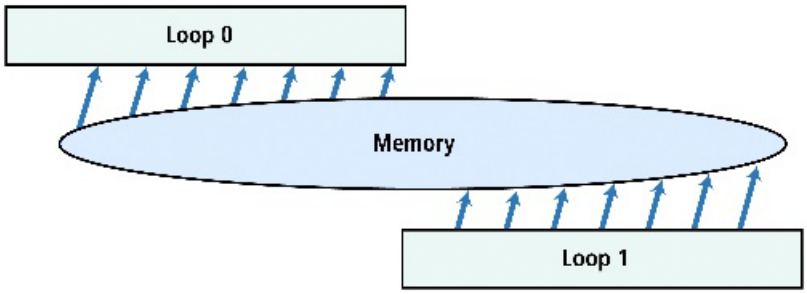
In such cases, loop 1 start must follow after loop 0 finishes.

- Loops 0 and 1 write memory or scalar x.



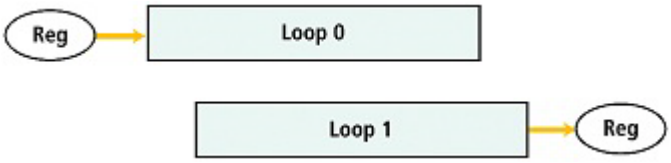
In such cases, loop 1 start must follow after loop 0 finishes.

- Loop 0 reads memory x and loop 1 writes x.



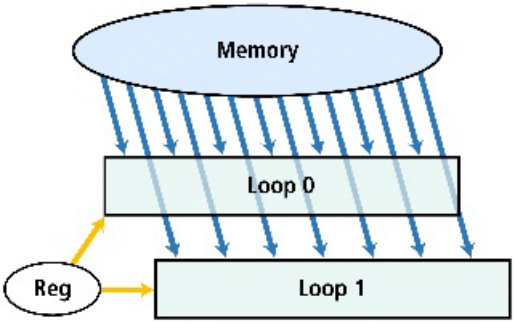
In such cases, loop 1 start must follow after L0 finishes.

- Loop 0 reads scalar x and loop 1 writes x.



In such cases, both loops can run in parallel.

- Loops 0 and 1 read a memory or scalar x

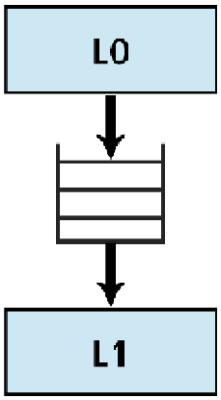


In such cases, both the loops can run in parallel.

Note:
Check the loop graph to understand if you have inadvertently added any sequentialization.

Streaming Data

Streaming data channels enable the connected loop nests (or PAs in hardware) to execute in parallel. Hardware implements a 2-way handshake using a 3-signal interface (dat, request, and ready) to transfer data. Optionally, the FIFOs can be specified to have additional storage.



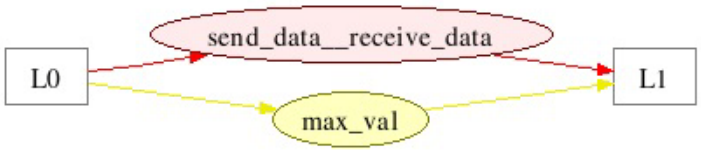
This facilitates time-independent synchronization and automatic flow control in the design. The consumer of the FIFO would stall if data is not available in the FIFO to service a request from the consumer. Similarly, if a FIFO is already full, any modules which attempt to write into the FIFO would stall till the FIFO has at least one empty slot. For additional information, see "Stream Support" on page 8-24.

You can specify any function in your application as either an input or stream. When such a procedure is executed in software, a request signal is asserted (either for read or write operation). If the corresponding ready is true, the transaction takes place and PA proceeds. If the ready is false, the PA stalls and continues asserting request until ready goes true. For more information, see "System Interface" on page 14-4.

In hardware, querying the status of FIFO is explicitly handled by the designer. Streaming interfaces in the tool implement the functionality with pre-verified blocks to ensure correctness. You need not have any additional code to achieve this functionality in your software model.

Synchronization and Flow Control

Without streams, the PPA would always execute in a pre-determined time. Streams introduce variability due to dynamic synchronization and flow control. Streams wait for input data to be available and output buffer to be free. Producer and consumer block independently while still retaining deterministic computation. Streams may cause deadlocks due to bounded buffers. Deadlocks can always be removed by increasing FIFO sizes. To get optimal FIFO sizes, you should analyze your design to eliminate any unintended sequentialization in the code. Consider an application where two loops exchange data using a scalar and stream as shown in the following figure.



In this case, the FIFO element needs to have sufficient buffer to accommodate all values produced by the loop L0 before the max_val scalar variable is consumed by loop L1.

You may use the following guidelines for proper usage of streams and memories to communicate between the loop nests.

- Use streaming data when loop nests must run in parallel on the same task
- Use memory when loop nests can be sequentialized
- Use memory when loop nests must run in parallel across tasks (with multi-buffering)
- Use streaming when read/write access pattern is sequential
- Use memory when read/write access pattern is complex or random

Hardware Sharing

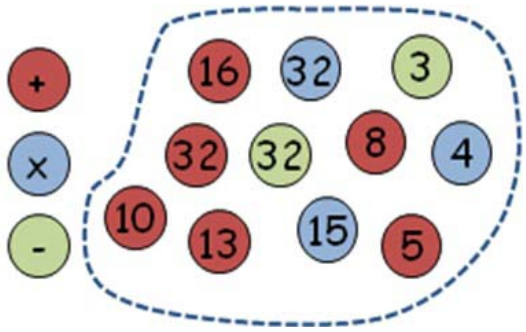
The Synphony C Compiler tool provides several types of hardware sharing. It is important to understand where the compiler can automatically share hardware, and where it cannot. In some cases, you will need to write code in such a way that sharing is explicit, but in other cases, it is best to let the compiler perform the sharing.

When the Synphony C Compiler tool synthesizes hardware from C, sharing is performed at the operator level, (for example add, subtract, multiply, load/store, TCAB calls, and scc_asm_* calls) but not at the level of code blocks or inlined procedures. Operations are split into groups

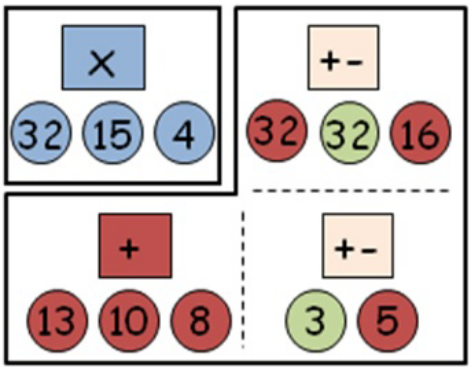
- Expensive: The Synphony C Compiler tool works very hard to share multipliers, dividers, TCABs and load or store units, modifying the instruction schedules to enable optimal mapping of these operations to a minimum number of functional units.
- Moderately cheap: Units such as add, subtract, inequality compares, and variable shift are treated differently. These units are only shared opportunistically when the scheduling is convenient, as modifying the operation schedule requires the addition of registers.

- Cheap: For these less expensive units, sharing can become counter productive due to the added multiplexing and register cost.

Expensive and moderate operations are further clustered by width and shared to obtain better QoR. Consider the following case where an application has several multipliers, adders and subtract units. In the figure, all similar operations have the same color and the numbers represent the widths of the operations.



The compiler groups the different operations based on their widths and tries to share the operations with close bitwidths. In the example of groups of operations, the 32- and 16-bit adders are shared on an add-sub unit (the same unit is capable of doing both addition and subtract units). Similarly, the smallest adder and subtractors are shared on the same add-sub unit. The remaining adders are shared on the same add instance. This is shown in the following diagram.



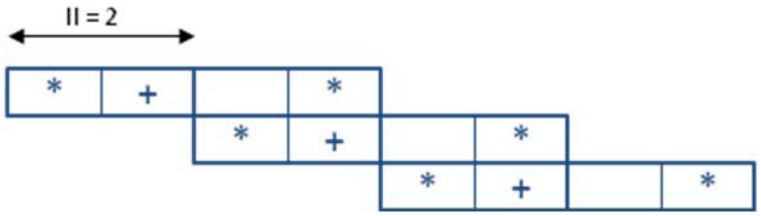
width clustering

There are two main opportunities for sharing of operators as explained in the following sections.

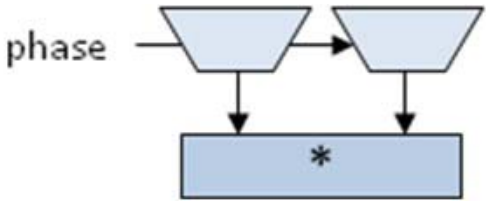
Sharing Across Cycles in an II>1 Design

If II is 2 or greater, there are multiple phases of execution in the design. A single hardware resource such as a multiplier or a TCAB may be scheduled to perform a separate operation from the source on each phase.

Consider the schedule of operations for an II=2 design as shown. Each iteration uses two multipliers. Since II=2, the tool was able schedule the operations so that only one multiplier is active on any given cycle.



The multiplier is shared across the different callers after suitably handling the multiplexing of inputs and demultiplexing of the outputs as shown.



Synphony C Compiler offers some control over this sharing through the `synthesize_auxopts` variable:

```
setvar synthesize_auxopts "-Fdata_width_for_II_resource_sharing=bits"
```

This flag controls the width threshold above which the tool attempts to share moderately cheap units such as, add, subtract, compare, and variable shift. The default is 16 bits.

Sharing of Mutually-Exclusive Operators

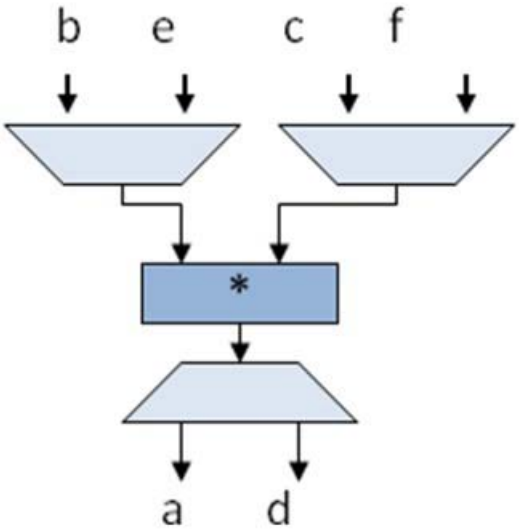
There may be mutually-exclusive uses of expensive operators such as multiply, divide, load, store, TCAB, etc. The compiler will identify the mutually exclusive operations and can map them to a single functional unit.

Note that sharing is not done for all mutually exclusive operations. The compiler will only try to share units if there is a cost benefit, that is, if the cost of sharing (in registers and multiplexing) is less than the cost of the functional unit that would perform the operation.

Consider the following code segment. Here the multiplier usage is in the if-and-else blocks, which are clearly mutually exclusive at all times.

```
if()
     a = b * c;
else
     d = e * f;
```

The multiplier is shared across the different uses after suitably handling the multiplexing of inputs and de-multiplexing of the outputs as shown.



Synphony C Compiler offers some control over this sharing through the `synthesize_auxopts` variable:

```
setvar synthesize_auxopts "-Fpredicated_resource_sharing=no"
```

will disable this type of sharing. Note that this is a global flag applying to all loops and all operations, so generally you will not want to turn it off.

```
setvar synthesize_auxopts "-Fdata_width_for_PMS_resource_sharing=bits"
```

This flag controls the width threshold above which Synphony C Compiler will attempt to share moderately cheap units (e.g. add, subtract, compare, variable shift). The default is 16-bits.

Hardware Sharing Coding Strategy

The following strategy can be applied to get better sharing in your design.

- Exploit mutual-exclusion: Use structured if/else or switch/case coding to expose the mutual exclusion to the compiler explicitly. You should be aware of possible pipelining in the application and also dependency constraints which can inhibit sharing.
- Increase II when performance allows.
- Use techniques like sequential loop jamming and parallel loop jamming to move shared hardware into the same loop. This increases opportunities to share hardware.
- Use hierarchical synthesis to share larger blocks

Coding Styles to Exploit Mutual Exclusion

Following are the few coding styles that exploit mutual exclusion.

Example 1:	Example 2:	Example 3:
if(x==0) {	if(x<3) {	if(x==y) {
A;	A;	A;
}	}	}
[...]	[...]	[...]
if(x==1) {	if(x>5) {	if(x!=y) {
B;	B;	B;
}	}	}

Example 4:	Example 5:
if(cond A) {	switch(x) {
A;	case 0: A; break;
}	case 1: B; break;
else if(cond B) {	}
B;	
}	

Coding Styles to Inhibit Sharing due to Mutual Exclusion

Absence of break statements: The following example does not have a break statement in case 0. This would imply that when x is equal to 0, both A and B may execute and so A and B are not mutually exclusive.

```
switch(x) {
     case 0: A;
     case 1: B;
}
```

The tool cannot analyze arbitrary conditions, so in some cases where condA and condB are mutually exclusive, SCC will not be able to prove it

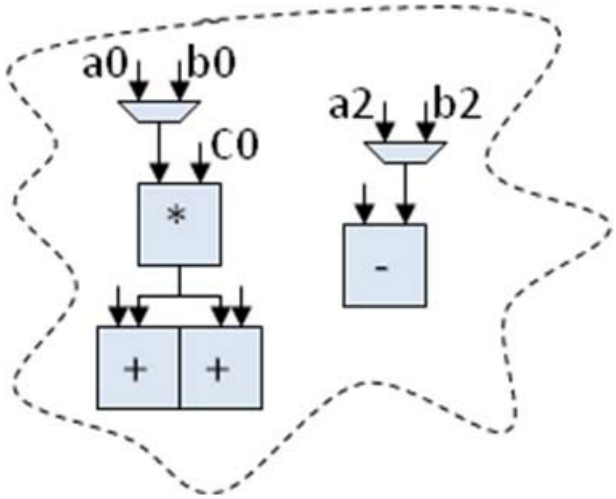
```
if (condA) {
     A;
}
[…]
if (condB) {
     B;
}
```

Sharing Larger Blocks

The compiler shares at the operator level. Datapaths made up of cheaper operations (+, -, <<, etc.) may not be entirely shared. This section describes how larger blocks can be shared in this tool. Consider the following example where the function calls to f( ) are not automatically fully shared when f( ) is inlined into g( ). Even the individual operators in f( ) get shared only opportunistically.

```
#define C0   13585
#define C1  -27456
int f(int i0, int i1, int i2) {
     return (((i0 * C0 + (1<<7)) >> 8) +
                     ((i1 * C1 + (1<<7)) >> 8) - i2 + (1<<7)) >> 8;
}
```

```
void g(unsigned char type) {
     int out;
     […]
     if (type == 0)
          out = f(a0, a1, a2);
     else
          out = f(b0, b1, b2);
          […]
}
```



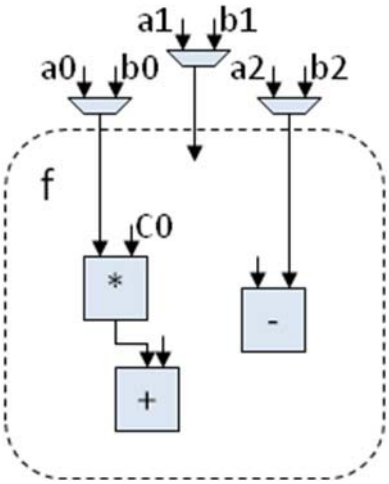
To achieve maximal sharing, you can re-code this design to move the function call out of if-else statement. Then you could use hierarchical synthesis to enhance the sharing. Create a TCAB for f( ), then import this TCAB when synthesizing the PPA. f( ) will be fully shared between the two call sites.

The modified code is as follows.

```
#define C0   13585
#define C1  -27456
int f(int i0, int i1, int i2) {
     return (((i0 * C0 + (1<<7)) >> 8) +
                    ((i1 * C1 + (1<<7)) >> 8) - i2 + (1<<7)) >> 8;
}
void g(unsigned char type) {
     int out;
     int x0, x1, x2;
     […]
     if (type == 0) {
          x0 = a0; x1 = a1; x2 = a2;
     } else {
          x0 = b0; x1 = b1; x2 = b2;
```

```
        }
        out = f(x0, x1, x2);
        […]
}
```

The hardware generated in this case would be as shown in the following figure where there is only a single instance of the TCAB for f().



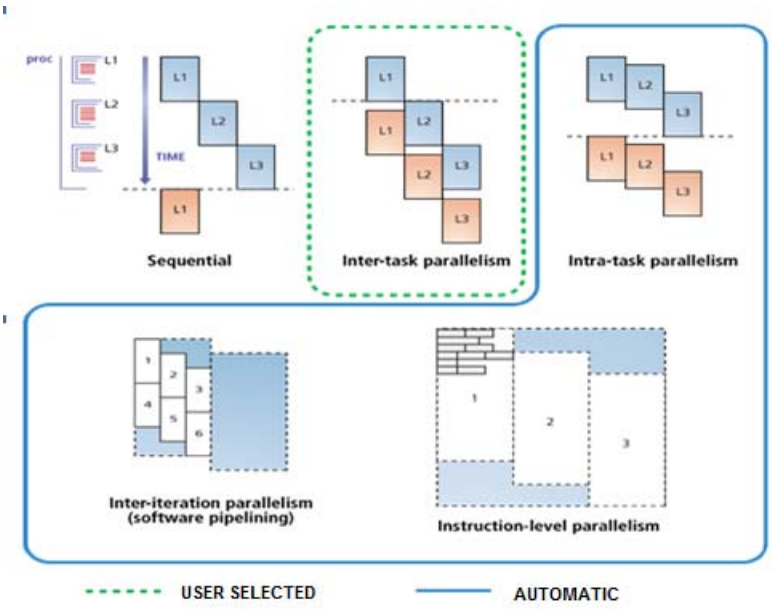
Parallelism

Software algorithms execute statements in the program order. Synphony C Compiler tool extracts parallelization opportunities at several levels in your code when building hardware.

Consider the following diagram where an application has 3 loop nests. The software model would execute as follows.

- All statements execute sequentially
- In case of loops, all operations in the current iteration need to complete before new iterations are started
- All loops execute sequentially within a task.
- A new task starts only on completion of all the loops within a task.

This is shown in the Sequential part of the figure.



The Synphony C Compiler tool analyzes the data and control dependences in the application and where possible, it allows the following optimizations which improve the performance of the application.

- An operation is scheduled as soon as all the data and control dependences are met. This is called instruction-level-parallelism.
- In each PA will schedule operations so that multiple operations execute in parallel with pipelined loop iterations. This is called software pipelining.
- Within a task, multiple PAs may execute in parallel. This is called Intra-task parallelism.
- The tool also provides a higher level of parallel execution called Task Overlap (or Inter-Task Parallelism) where successive procedure calls may be overlapped. The PPA template includes configurable
  functionality to store multiple sets of input/output data to enable a host to control it while running overlapped tasks (procedure calls).

The figure shows the various levels of parallel operation in the PPA. For more information on controlling tasks, see “Task Overlaps” on page 5-57.

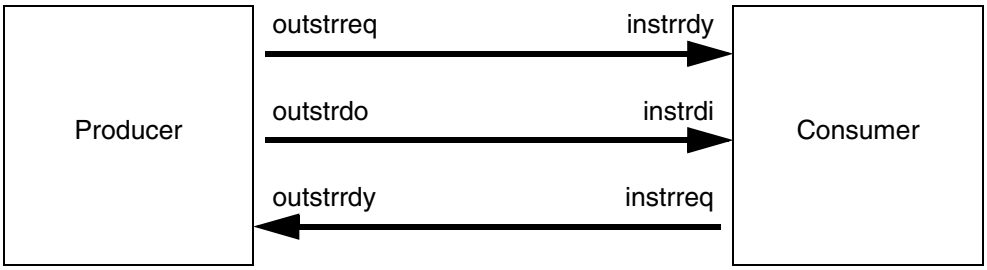
Task Overlaps

All C simulations follow the sequential execution model of the original C code. Simulations run using `verify` command follow the parallel execution model inherent in the generated hardware. The Verilog testbench and the SystemVerilog-UVM testbench are programmed to exercise the maximum task overlap supported by the generated hardware, and automatically verify the hardware outputs and hardware state against the values seen in C simulation.

Resource Allocation for Task Overlap

The pipelined hardware stage of a PPA is only one of the resources used in executing the loop nest computation that is mapped to it. In addition, to support multiple outstanding task invocations, storage resources need to be allocated and assigned to multiple data copies. A PPA may reference input configuration registers (inputs) and may produce values to be stored in output configuration registers (outputs). The stage may read or write to memory arrays that are shared with other stages. Finally, the stage may read or write to streams that communicate with other stages. Whether a PPA design can be task-pipelined to the full depth of the available stages also depends on its data usage pattern and the number of data storage resources allocated.

The stream-based communication is synchronized via a handshake at each data transaction so the producer and consumer stages may, in fact, be running in parallel on the same task. This exploits parallelism within a single task without any extra resources. However, access to shared input/output configuration registers and memory arrays is synchronized on a task-basis. Therefore, pipelining multiple tasks through the PPA requires multiple copies of the configuration registers and memory arrays to be present, otherwise two stages accessing the same set of registers or memory cannot be task-pipelined. The following figure shows the streaming of data interface via handshaking protocol.



If an array is shared across multiple stages of the PPA, then you may specify multiple memory banks to be allocated for that array using `#pragma multi_buffer` in the C source so that multiple tasks in flight may be supported.

Maximum Achievable Task Overlap

There are several factors that determine the maximum achievable task overlap in an application. These factors are listed for easy reference.

- The number of loopnests:
  Each loop nest creates a computation stage in the hardware. A stage is not allowed to execute more than one task at a time. Therefore, the maximum achievable task overlap is guaranteed not to exceed the number of stages. The maximum task overlap occurs when each stage is executing a different task.
- The data dependences between loopnests:
  The exploitable parallelism across the hardware stages depends on the data dependences within the application. On one hand, if the execution extent (II*#iterations) of each loop nest is comparable and there is only streaming communication between them, all stages will be started and will execute in parallel on the same task. Therefore, maximum achievable inter-task overlap will be small since the time between the first stage becoming free while other stages are still executing that task is small. On the other hand, if each stage is sequentially dependent on the previous stage, and a new task is started as soon as the first stage and all its required resources are free, there can be as many tasks in the pipeline as the number of hardware stages. However, the actual number of tasks in flight would still depend on the actual data dependences between the loops and the resources used by them.
- Use of raw input and output data ports:
  You can directly access inputs and outputs using raw data ports by declaring such variables with #pragma host_scalar_access. However, this may limit the amount of task overlap that is achievable in the PPA hardware depending on how often the raw input and output ports need to be changed or sampled. If the raw ports need to be changed for each task, then no task overlap would be achievable.

Invariant Properties Under Task Overlap

There are some properties that remain unchanged when you go from no task overlap execution model to the task overlapped execution model. These invariant properties are discussed.

Sequential Semantics with Task Overlap

Other than the output and the retiming discussed in "Task Overlaps" on page 5-57, the sequential semantics of the application is maintained even with task overlap. For instance, data is read from an input stream (is written to an output stream) in the same sequence as if no task overlap was present. Likewise, the reading and writing from/to arrays and scalars also maintains sequential semantics with respect to each data object. A pipeline stage accessing a memory array will start execution only after a preceding stage accessing the same array (on which it is sequentially dependent) has completed execution. This is good because you need not worry about stream and memory synchronization between loopnests due to task overlap. However, this may limit the amount of task overlap that can be exploited by the tool if the streams, scalars and array resources are shared across multiple loopnests. This is because a new task cannot be started in a stage until all the resources used by that stage are also free.

Some tips to maximize task overlap in the face of data dependences and re-source sharing are discussed.

- Do not read from or write to the same stream in multiple loopnests (stream port sharing). This will cause the stream port resource to remain busy until all the loopnests that read from it or write to it have completed. Note that this restriction does NOT apply to point-to-point FIFO streams that are written from one loop and are read in another loop.
- Sharing input scalar variables for read-only access across loopnests does not sequentialize the loopnests. However, if a scalar variable is written in one loop nest and read or written in another, the tool automatically inserts sufficient buffering such that the scalar resource is released for reuse as soon as the other stage accessing the scalar variable has started.
- Sharing memory arrays for read-only access across loopnests also does not sequentialize the loopnests. However, if a memory array is written in one loop nest and read or written in another, those loopnests are sequentialized so that the latter loop nest finishes its accesses pertaining to the current task before the next task overwrites the array. For locally declared memory arrays, higher throughput across tasks may be achieved by specifying #pragma multi_buffer identifying the number of buffers desired. The tool then automatically generates hardware to manage the set of buffers in a round-robin manner. A free buffer is available immediately for use by the next task. A used buffer is released for reuse when all the stages accessing it have completed.

Persistent State Across Tasks

As per C semantics, globally declared data arrays retain their values across tasks. This is helpful in situations when arrays are used to accumulate state across tasks or computation in one task depends upon results from previous tasks. These arrays may be initialized before a task run and read upon its completion, accumulating the state across all task invocations within a run.

Note that configuration registers are generated by declaring scalar variables in the global scope. These registers do not retain their value across tasks automatically. If present, they have to be explicitly initialized before each task and read out at the end of it. The tool automatically generates a software driver stub that keeps the state of globally scoped C variables consistent with the sequential programming model. However, as described in "Task Overlaps" on page 5-57, the effects of output retiming during task overlap also applies to these C variables and you are responsible for re-circulating the output values from one task to the input of another task with the appropriate retiming. Later sections show how to achieve this in the driver code.

Guidelines for persistent state across tasks:

- Do not use globally scoped scalar variables in your code that translate to input or output scalar configuration registers
- Declare small data arrays in the global scope that are accessed with constant indexes. The array can be marked with #pragma internal_fast if you want them to be as fast as registers. For certain small sized data arrays, this is also the default. These array variables would be accessible within the hardware just like scalar variables and would preserve state across tasks.

Hardware-Based Task Overlap Scenarios

The following sections describe writing a hardware controller and auto-start or restart.

Writing a Hardware Controller

If you want to integrate the PPA generated by Synphony C Compiler into your system directly without relying on a host processor that runs the PPA software stub and the API library, then conceptually, all the task and resource management done by the PPA software stub and the API library must be done by some hardware controller that controls the PPA. All the conceptual aspects of obtaining task overlap under various application conditions also remain the same. The only difference from the software controller is that under certain situations a much simpler hardware controller may be used that directly controls the hardware state of the PPA as exposed by its low-level signal interface. Describing how to write a hardware controller under these scenarios is beyond the scope of this document, however, the low-level signal interface and the associated integration issues are discussed in the Chapter 14, "System Integration and Verification.

Auto-Start or Restart

Under a limited set of conditions, a PPA may be synthesized to operate autonomously without a hardware or a software controller. This is accomplished when the PPA is synthesized with auto-start/restart capability using command:

```
synthesize -auto_start
```

The application constraints are essentially that no scalar inputs or outputs should exist. This eliminates the need for a programmatic way to read and write it for each task. However, global arrays, streams and raw data ports are allowed. The following sequence of events (executed either programmatically or using the low-level signal interface) enables a task overlapped execution.

1. Upon reset the PPA hardware is disabled.
2. The raw data input ports and the global memories may be initialized.
3. The PPA hardware is enabled, either programmatically using the SCC_enable_PPA API command or by setting the bits PSW[4:5]=0x3.
4. The PPA hardware automatically starts executing the first task as soon as it is enabled (auto-start).
5. The raw data inputs are guaranteed to have been sampled when the init_done event is signaled (PSW[2]=1).
6. As soon as the first task clears the first stage of the PPA, a new task is started (auto-restart). This is signaled by the busy bit (PSW[0]) going to '0' for 1 cycle and then again going back to '1'. Meanwhile, the previous task may be advancing to the subsequent stages resulting in task overlap.
7. The cycle of starting new tasks continues ad-infinitum unless the PPA is aborted using the API command SCC_abort_PPA or by setting the bit PSW[6]=1. This step is optional.
8. If the PPA is aborted then all currently executing tasks are flushed and the PPA halts in a disabled state. At this point, a new run may be initiated starting from step 2.

A task run with maximal task overlap is executed automatically between the events of enabling the PPA and aborting it. This may be used as a simple mechanism to obtain task overlap with minimal control. Note that the current value of the raw data input ports is sampled by each new task as soon as it starts. Therefore, it is recommended to keep the raw data ports stable during the entire task run and change them only when the PPA has been disabled by an abort.

Implications of Coding Styles and Syntax

Coding for high-level synthesis requires developing a different intuition about how various coding styles translate to hardware. Sometimes the best coding can run counter to common software optimizations. While some of this may be tool-dependent, there are some good practices that should help to generate efficient hardware in all cases.

Variable Scoping

Variables can have different meanings when declared at different scopes, and sometimes this can affect QoR.

Declaring a variable at file scope means that the variable is visible externally. It will be assigned a location in the memory map (either as input, output, or both). If the variable is an array, it will be implemented as a physical memory, either user_supplied or internal_fast.

Note:
File scope array variables will neither be scalarized nor multi-buffered.

Declaring a variable at procedure scope means that it is capable of holding state across iterations, but it is not externally visible. It is not a requirement that these variables must be associated with physical registers, but they often are. Synphony C Compiler is free to make optimizations like scalarization on procedure-scope arrays.

Declaring a variable at the innermost loop iteration scope means that it should hold its value only during that iteration. A variable declared at this scope requires registers only if the compiler uses it in different schedule times. Because of this potential area savings, it is recommended that you declare variables inside the iteration scope if you do not intend for them to carry state across loop iterations.

Compiler behavior is undefined if iteration-scope variables are read before being assigned in an iteration. Although this is a potential application coding error, generally gcc behaves as if the variable were declared outside of the loop. The Synphony C Compiler linting assists in identifying if you have such application errors.

Registers and Wires

In RTL it is usually explicit which signals are intended to have registers and which should be wires or combinational. While coding in C this is not so clear, because the compiler will make these decisions. However, there are some rules of thumb that can help to understand what the compiler will do.

For this discussion, we are ignoring the effects of pipelining within a loop iteration. Pipelining may introduce registers for timing purposes but these are different from registers used to

hold the state of variables across loop iterations. Here we will use loop carried to mean those registers that are required to hold variable state across loop iterations.

If a variable is assigned on each path through a loop iteration, then it is not required to carry its value across iterations and no state registers should be produced.

In these examples, x may be defined at procedure scope or inside the innermost loop scope. The guidelines apply in both cases.

Example:

```
for(i=0;i<N;i++) {
     // no loop carried register needed for x
     if(cond)       x = a;
     else       x = b;
     z = f(x);
}
```

A similar example is when a variable is not assigned on every path through the loop nest, but is assigned on every path that leads to a use:

```
for(i=0;i<N;i++) {
     // no loop carried register needed for x
     if(cond) {
          x = f(a);
          z = f(x);
     }
}
```

If a variable is not always assigned prior to use in a loop iteration, then it must carry state across iterations, and loop carried registers will be created by the compiler:

```
for(i=0;i<N;i++) {
     // loop carried register is needed for x
     if(cond) {
          x = a;
     }
          z = f(x);
}
```

If a variable is not always assigned prior to use in a loop iteration but is defined in innermost-loop scope, then no loop carried register will be needed for x:

```
for(i=0; i<N; i++) {
     // no loop carried register is needed for x
     int x;
     if(cond1) {
          x = A[i];
     }
     if (cond2) {
          B[i] = f(x);
     }
}
```

This coding is valid if either of the following holds:

- It never happens that cond2 is true and cond1 is false on the same iteration, OR
- If it is possible that cond2 is true and cond1 is false on some iteration, then the value of x on that iteration does not affect the expected program output.

In this example, the tool reports a warning if cond2 is true and cond1 is false in the same iteration. In addition, it will randomize the value of x on such an iteration and will report an error if this randomization causes a change in the expected program output.

Input Code and the Generated Hardware

To the first order, more code in the application means more hardware, so it is a good idea to minimize the amount of code where possible.

Loop unrolling implies that all the operations in the unrolled loop should be done within an iteration of the higher level loop. If the higher level loop iterations are performed at II=1, then all those operations must be performed each cycle in the hardware. For more information on unrolling, see "unroll Pragma" on page 7-70.

Function inlining is similar - it means to do all the operations in the function within the loop iteration that calls the function.

This is also an area where intuition from software development may lead to poor QoR. In software, it is generally a good idea to minimize the number of operations performed. Sometimes this means writing more code and doing more tests. It might mean having multiple procedures for similar functions, each optimized for a particular task. For synthesizing hardware, the optimal solutions may be reversed - it is better to have less code and some operations wasted.

A common example is an Inverse Discrete Cosine Transform, where a good software implementation may test for all coefficients in a row or column being zero:

```
if((in[0] | in[1] | in[2] | in[3] |
          in[4] | in[5] | in[6] | in[7]) == 0) {
          out[0] = out[1] = out[2] = out[3] =
               out[4] = out[5] = out[6] = out[7] = dc_value;
}
else {
     idct_step(out,in);
}
```

In this case, idct_step would produce the same output if the test was removed, but operations are saved in software by taking the shortcut. The hardware pipeline must be constructed to execute idct_step and take the output from either branch of the if/else, so there is no savings in time to take the shortcut. In addition, the implementation with the

shortcut would be costlier, since it involves additional comparisons and assignments that do not affect the functionality.

The best coding for hardware is simply:

```
idct_step(out,in);
```

Variable Trip Count Loops and Early Exits

Synphony C Compiler can do a lot of analysis and optimization of code inside a constant bound for loop that has an unambiguous loop index variable and statically known trip count. It is important to understand the implications of coding in ways that make these optimizations possible.

Examples of constant bound for loops:

- ```
  #define N 10
      for (i=0; i<N; i++)
  ```
- ```
  for (j=19; j>=-1; j-=2)
  ```

Note:

To be considered a constant bound, the loop index variable `i` or `j` must be a basic integer data type such as `char`, `short`, `int`, or `uint`. The tool cannot determine the trip counts of the loop when the index variable is a struct field, a class member, or a variable of any other data type such as a parameterized data type or a fixed-point type otherwise supported by the tool. Such loops are treated as variable trip count loops.

Examples of variable trip count loops:

- ```
  extern int n;
            for (i=0; i<n; i++)
  ```
- ```
  for (i=j=0; i<n && j<k; i++; j++)
  ```
- ```
  while (i<N) { fn(i); i++; }
  ```
- ```
  for (i=0;i<100;i++) {
            if (cond){ break;} // usage of break
  }
  ```
- ```
  for (i=0;i<100;i++) {
            i=i-1; // Modification of loop variable
  }
  ```
- ```
  struct ctr_t {
  int idx;
  int idy;
  } sd;
  for (sd.idx=0; sd.idx<10; sd.idx++)
  ```

- `for (s_unit<4> k=0; k<10; k++)`

The Synphony C Compiler tool generates two types of PAs (Processing Arrays)

- Fixed trip count PA, that runs a fixed number of loop iterations
- Infinite running PA

In both cases, it is possible to programmatically terminate the loop execution through a break statement and drain the pipeline. However, the break mechanism results in additional hardware. In case of fixed trip count PA, an extra counter is maintained to ensure loop termination after fixed number of loop iterations.

A loop nest with all constant bound for loops and no break statements or a break from only the outermost loop in the loop nest, results in a fixed trip count PA. In all other cases, the loop nest will result in infinite running PA with the break mechanism. When the algorithm can be expressed in terms of constant bound loops, it is almost always beneficial to do so.

Perfectization Choices

Perfectization has a limited number of options to choose. In some cases, the best options are not available to the compiler and to optimize QoR you must force the correct decisions.

Perfectization has two basic choices available to it:

- Code sinking into an existing loop nest/PA.
- Wrapping out-of-loop code into its own loop nest/PA.

Code sinking will provide good results if the operations involved are cheap in hardware, or if they are of a different type than those in the loop nest in question. It may provide poor result if it increases the number of expensive functional units required by a PA.

Wrapping out-of-loop code will tend to provide good result if the out-of-loop code wants to run at a different rate than the in-loop code. Perhaps a loop needs to run at II=1, but we need to perform five out-of-loop multiplications prior to entering the loop. Sinking the code would be a poor choice as it would imply five multiplications on the first loop iteration (at II=1, this means five additional multipliers!). Wrapping the out-of-loop code would enable it to run at II=5 or higher (TASK_II permitting) and use a single multiplier.

Wrapping out-of-loop code may provide sub-optimal results if the operations are cheap and could just as well be integrated into the loop nest. It may also be wasteful in some situations where units such as load/store could otherwise be shared with operations in the loop nest (e.g., when the out-of-loop code and the in-loop code access the same memory and the ports are not fully utilized).

In some situations, neither of these automated options produces the best results, and hand modifications can be done to reduce area. An example is shown.

Note:
The following is identical to the example in "Iteration Space Extension" on page 5-22.

A common occurrence in some algorithms doing frequency/time or frequency/spatial domain transformations (FFT and DCT are two examples) is to handle DC and AC components differently. Sometimes this is expressed as a first step for DC and then a loop for AC:

```
encode_DC( );
for(i=1;i<64;i++) {
     encode_AC(i);
}
```

Suppose that encode_DC( ) and encode_AC( ) share a memory (DC value is written to element 0 and AC values to elements 1- 63). In addition, they may have common operations such as multiplications, divisions, etc. Synphony C Compiler considers two perfectization choices for the code:

Wrapping the out-of-loop code into a PA:

```
for(i=0;i<1;i++) {
     encode_DC( );
}
for(i=1;i<64;i++) {
     encode_AC(i);
}
```

Alternatively, sinking the out-of-loop code into the loop:

```
for(i=1;i<64;i++) {
     if(i==1) encode_DC( );
     encode_AC(i);
}
```

In this case, neither of these is the best choice. Assume that encode_DC( ) and encode_AC( ) each make one read and one write access to a shared memory m[ ], and each perform one multiplication. Further, assume that we want Task_II to be about 64 for this code – one cycle per component.

If we wrap the out-of-loop code, we will have two PAs. Each one requires a load unit, a store unit, and a multiplier. We can have two ports to m[ ] but they will be arbitrated or multiplexed between the two PAs.

If we sink the out-of-loop code, we will have a single PA. However, on its first iteration it makes four memory accesses: Those for encode_DC( ) as well as those for encode_AC( ). It also does two multiplications on this iteration. If we run at Task_II = 64, we will have II=1 and require four ports to m[ ].

In this case, the best thing to do is modify the code by hand to imply the efficient operation we want:

```
for(i=0;i<64;i++) {
     if(i==0) encode_DC( );
     else encode_AC(i);
}
```

This coding will have one load, one store and one multiply in each iteration. These will be shared between the encode_DC( ) and encode_AC( ) functions because these calls are now mutually exclusive.

Trade-off Between Cost and Performance

Synphony C Compiler offers a variable trip count solution that optimizes on cycles. The solution requires additional hardware to implement the early termination of the loop. Variable trip count loops in inner loop dimensions also result in reduced analyzability and some loss of optimizations that are possible in constant bound loops. These implications are discussed in more details subsequently.

Given the trade-off between cost and performance, user should decide whether to go for a variable trip count loop or implement it with a constant bound loop. If it is not a major requirement to complete the task within the minimum number of loop iterations, a variable loop solution can be implemented using constant bound loops that are designed for the maximum problem space. For example,

```
for (i=0; i<n; i++) {
     fn(i);
}
```

Suppose the loop executes between 100 and 120 iterations and performance requirements allow us to execute 120 iterations even for a task that requires 100 iterations, writing it like this would result in less costly hardware

```
for (i=0; i<120; i++) {
     if (i<n) {
     fn(i);
     }
}
```

Affine Expressions of Loop Indexes

The Synphony C Compiler tool computes the value of loop indexes and the value of affine expressions of loop indexes used in conditions in an efficient way that reduces the width of adders required. These optimizations are done for the constant bound loops. These

optimizations are also done for the outermost variable trip count loop if the loop index can be unambiguously identified by the tool and the minimum and maximum values of the loop trip count are made available to compiler using `#pragma num_iterations`. For example,

```
#pragma num_iterations (200, 220, 240)
for (i=0; i<n; i++)
     for (j=0; j<300; j++) {
               if (2*i+j < 10) fn(i, j);
     }
```

Note:

Linear expression of the form Ax + C, where A and C are integer constants and x is a variable (loop index). For example, (2*i + 1), (i + 3*j - 8) etc. Such expressions are commonly used in indexing arrays and for control conditions inside loops.

Here the pragma specifies that the for loop i executes between 200 and 240 iterations with performance targeted at 220 iterations. For more information on pragma syntax, see Chapter 7, “Pragmas and Constraints.”

In the presence of variable trip count inner loops or inner loop breaks, this optimization is not possible. This optimization is also not possible when the upper bound on the number of iterations in the loop nest is not known. In such cases, the loop indexes are explicitly counted in the resulting hardware. For example,

```
for (i=0; i<20; i++)
#pragma num_iterations(10, 12, 15)
for (j=0; j<m; j++)
```

The optimization is not possible even for index i since the inner loop is a variable trip count loop.

```
for (i=0; i<n; i++)
for (j=0; j<10; j++)
```

In a loop nest, all loops starting at the innermost variable trip count loop to the outermost loop result in explicitly counted loops. So in an application involving a mix of constant bound for loops and variable trip count loops, the constant bound loops should be written as the inner loops as far as possible and variable trip count loops restricted to the outer loop dimensions.

Specifying Bounds on Loop Iterations

If the minimum number of iterations of a variable trip count “for” loop is not known, the compiler has to conservatively assume that the loop may not execute at all. Expressions involving the loop index of such a loop might not be analyzable by the compiler, so certain optimizations, such as the one described in the previous section, are not possible in that situation. In addition, certain compiler transformations such as loop perfectization depend

upon the fact that the loop executes at least once. The loop is transformed to execute at least one iteration and an extra break statement is inserted by the compiler to ensure semantic correctness of the program, which might result in additional hardware.

It is advisable to always specify the minimum iteration count of variable trip count loop if known, or at least 1 if it is known that the loop is always entered. For example,

```
#pragma num_iterations(1,,)
  for (i=0; i<n; i++)
```

Certain compiler optimizations, such as loop unrolling, require the maximum number of iterations to be known at compile time. The tool exits with an error message otherwise. Synphony C Compiler also performs width optimization of the loop index variable of a variable trip count “for” loop and dependent operations based on the maximum number of iterations specified through the pragma. This information also helps the tool make certain decisions in the analysis that might affect cost and performance. Therefore, it is advisable to provide this information to the compiler whenever possible. For example,

```
#pragma num_iterations(,,20)
  for (i=0; i<n; i++)
```

Loop Flattening

In Synphony C Compiler, certain loop optimizations require constant trip counts at all levels, except possibly the outermost loop. More general loop nests are converted into this form through loop transformations. The most important transformation for doing this is the loop flattening. All loops starting from the innermost non-constant trip count loop to the outermost loop are converted into a single loop (with the appropriate iteration space). This loop then simulates the index manipulation and other loop-related activities for all the loops that have been combined.

An important class of optimizations that is suppressed due to this flattening is 'feedback optimizations'. The most common of these feedback optimizations is the one which efficiently computes the next index of a loop using some information about the current value (and some previous values) of the loop index (remember that Synphony C Compiler may reorder your loop iterations if it is permitted to do so).

When a loop nest is flattened, this optimization does not happen and therefore the code, which updates the loop indexes (of all the flattened loops) may become the bottleneck.

As a general guideline, you should therefore attempt to keep the number of levels of the flattened loop nest as small as possible. You can do this by
ensuring that the inner loops have constant trip counts, and push the variable trip count loops to the outer levels.

Good Hardware Techniques for the Compiler

Some designers will begin coding without a sense of what a good hardware implementation is for the application. In many cases, however, the designer begins with a very good understanding of some properties of a good implementation. These might be good algorithms to use for the hardware, ways of reducing the number of functional units or memory ports, ways of reducing bitwidths or many other techniques.

Note:

It is absolutely critical to expose these good tricks to the compiler to obtain implementations that are competitive with a hand design.

Avoiding Natural Termination of Loop in the Presence of Breaks

In a constant bound loop, the compiler has to assume that the natural termination of the loop after fixed number of iterations is always possible and generate extra hardware to ensure the termination happens. In a loop nest with all constant bound for loops and only an outermost loop break, a fixed trip count PA is generated to ensure this. In case of variable trip count inner loops and loops with inner break statements, and extra break condition is generated for the counted exit of the loop.

When the break statement is intended to be the only possible termination of a loop, it is advisable to make it explicit to the compiler through coding to avoid generation of this additional hardware. For example,

```
for (i=0; i<200; i++) {
      x[i] = fn(i);
      if (i == n-1) break;
}
```

If (n <= 200) is always true, the same loop may be written as

```
#pragma num_iterations(1,200,200)
for (i=0; i<n; i++) {
     x[i] = fn(i);
}
```

Exposing Loop Exit Conditions to the Compiler for Timing

Compiler can understand the loop exit conditions expressed through bound expressions in 'for' statement and use this information in generating efficient translation for timing. Loop exits through break statements are considered data dependent and are not analyzed by the compiler.

Only the loop nests of the following form are fully analyzable by the compiler

```
for (i=s; i<n; i+=k)
```

where, s, n, and k are either compiler time constants or variables. Other forms of 'for' loops that are allowed in 'C' are often less analyzable and may lead to difficulty in meeting timing at high frequencies. For example,

- `for (i=j=0; i<n && j<k; i++; j++)`
- `for ( ; i<n; i++)`
- `for (i=0; i!=0; i++)`

It should be noted that in a loop nest, index computation of an outer loop depends on the exit conditions of the inner loops. Since the loop iterations are pipelined, the feedback cycle for index computations of the outer loops involves the computation of exit conditions for the inner loops. However, if the bound conditions are inputs to the loop nest and are analyzable by the compiler, then the conditions can be computed speculatively and such loop nests can achieve higher frequencies. It is advisable to compute the exit conditions outside the loop nests and make them inputs.

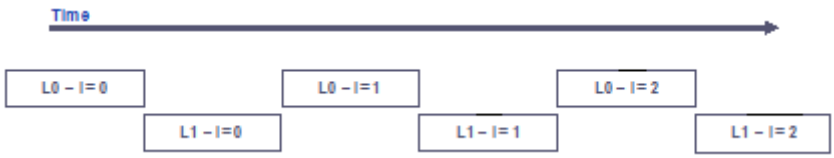
Outer Loops for Getting the Desired Results

In the presence of outer loops there are many possible desired results for the hardware. Consider the following code snippet:

```
for(i=0;i<M;i++) { // outer loop
     // Start of L0
     for(j=0;j<N0;j++) {
               Loop0_body; // body of Loop0
      } // End of L0
     // Start of L1
     for(j=0;j<N1;j++) {
               Loop1_body; // body of Loop1
     } // Endof L1
}
```

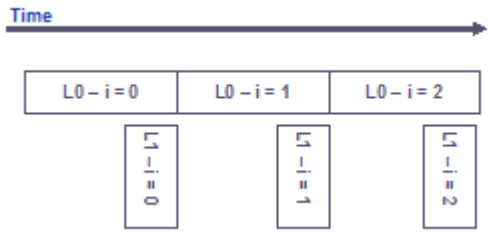
There are many cases for the desired hardware. We will consider these cases in the order of increasing complexity.

Case 1: Sequentialized Inner Loops



The performance chart clearly shows that L0 and L1 execute in sequence. If this performance is desired, then we can re-code the inner loops as procedures and make TCAB out of these procedures. Alternatively, Synphony C Compiler would automatically sequentialize the inner loops inside the outer loop.

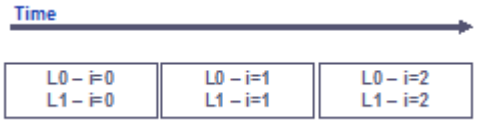
Case 2: Unrolling



In many cases, additional inner loops represent functionality that should not be iterative in hardware. Examples could be shifting of shift registers or summation across filter taps in a high performance FIR filter. In these cases, the correct solution is just to unroll one or more inner loops. We can modify the code as shown:

```
for(i=0;i<M;i++) {
     for(j=0;j<N0;j++) {
          Loop0_body;
     }
     #pragma unroll
     for(j=0;j<N1;j++) {
          Loop1_body;
     }
}
```

Case 3: Loop Merging



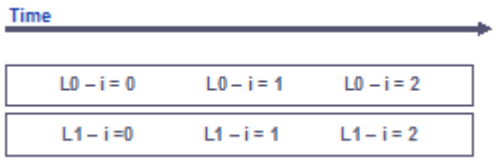
A common special case is where N0 = N1 and the inner loops can be merged as shown:

```
for(i=0;i<M;i++) {
     for(j=0;j<N0;j++) {
          Loop0_body;
```

```
            Loop1_body;
      }
}
```

This requires that iteration n of L1 is not dependent on any iteration of L0 greater than n. This is a common case, for example in a pre-processing step like level shifting prior to signal processing.

Case 4: Streaming Pipeline (Loop Distribution)



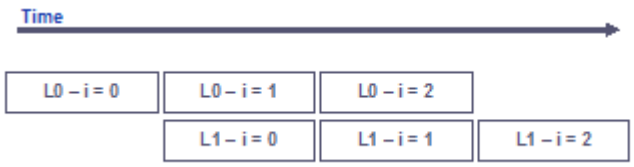
Parallelism is automatically enabled if L0 and L1 communicate via streaming data, or in certain special cases with memory communication. In case of communication using memory the user has to dual-buffer the memory that is used for communication. That is, the user has to use two copies of the memory for different tasks.

It must be noted that this transformation is not possible for all code (eg. if there is feedback from L1 to L0)

```
for(i=0;i<M;i++) {
      for(j=0;j<N0;j++) {
            Loop0_body;
      }
}
for(i=0;i<M;i++) {
      for(j=0;j<N1;j++) {
            Loop1_body;
      }
}
```

This transformation is often referred to as Loop Distribution since the outer loop is distributed across all inner loops.

Case 5: Outer Loop in Driver



This outer loop needs to be in hardware, but Synphony C Compiler Express can design the complex internal control if the code is written this way. An external hardware controller can be added to launch the tasks.

```
main() {
     for(i=0;i<M;i++) ppa();
}
ppa() {
     for(j=0;j<N0;j++) {
          L0;
     }
     for(j=0;j<N1;j++) {
          L1;
     }
}
```

This option can be chosen for any design that requires complex control (multi-buffering of memories, timing control between loops) to be designed automatically.

Loop Merging

Because each loop is in its own stall domain and Synphony C Compiler does not arbitrate for resources across PAs (except memories), segments of code that map to different PAs yield no sharing of functional units such as multipliers, dividers, load/store units, shifters, etc. A simple way of thinking of this is that there is no sharing of computation resources between loop nests (TCABs are an exception and can be used for critical sharing across loops that execute sequentially).

In many cases, there are several ways to code various functions. For example, an image pipeline might have code like this:

```
for(line=0;line<NLINES;line++) {
     for(col=0;col<NCOLS;col++) {
          rgb_gamma_correction();
     }
}
```

```
for(line=0;line<NLINES;line++) {
     for(col=0;col<NCOLS;col++) {
               rgb_color_correction();
     }
}
for(line=0;line<NLINES;line++) {
     for(col=0;col<NCOLS;col++) {
          rgb_to_yuv_conversion();
     }
}
```

Assume that the function calls operate on a single pixel, with streaming data flowing between the loop nests (in other words, we have the choice of doing all the gamma correction on the full picture, then all the color correction, then all the RGB to YUV conversion, or we can do all three operations on each pixel before proceeding to the next pixel). In this case, it is straightforward to merge these loops since their loop structure is the same:

```
for(line=0;line<NLINES;line++) {
     for(col=0;col<NCOLS;col++) {
          rgb_gamma_correction();
          rgb_color_correction();
          rgb_to_yuv_conversion();
     }
}
```

Making this transformation has several benefits:

- It eliminates overhead associated with multiple PAs and streams be-tween them
- It reduces the overall latency of processing for each data element, and this may further reduce the number of pipeline registers required.
- It opens up the possibility to share resources across these functions
- From a pure software perspective there may be benefits to this coding as well: We eliminate intermediate stores for the data arrays, reducing the number of memory references

In many cases, it is not so straightforward to merge loops together. Consider this case:

```
for(i=0;i<N;i++) {
     // initial processing on a data set
}
for(i=0;i<M;i++) {
     // mainstream processing
}
```

This type of processing might have many opportunities for sharing of resources between the initial processing and the mainstream processing, but we need to do all the initial processing

before proceeding to the mainstream. In this case, we can make the following transformation:

```
for(i=0;i<N+M;i++) {
     if(i<N) {
     // initial processing on a data set
     }
     else {
     // mainstream processing
     }
}
```

Now all the code is mapped to the same PA and Synphony C Compiler knows that the initial and mainstream portions are mutually exclusive, so the ex-pensive resources may be shared.

The decision to merge or not merge loops is often not clear. Some cases where it may be best NOT to merge loops:

- Only one of the loops uses any given expensive operation (multiply, divide, load, or store)
- There is little or no potential sharing after merging o Both loops operate at II=1, and their operations are not mutually exclusive
- The loops operate at II>1 but there are no empty slots available for additional operations - for instance each loop operates at II=2, but each one already performs two multiplications per iteration
- The coding complexity of merging the loops is high, for instance it re-quires modifying the loops structures, and the savings are not large enough to justify the merging
- The loops should run at different IIs for efficient operation

Reducing II

From the previous discussion of the multiplexing involved in II>1 designs, it should be clear that if a function can be implemented with the same Task_II at different IIs, then, the lower the II, the more efficient the design. This often comes at the expense of coding complexity.

Sometimes designs are very difficult to code to achieve lower II with the same Task_II. Consider an 8x8 DCT decomposed into a row transform and a column transform, each further decomposed into 8x1 transforms. For example, we look only at the row transformation:

```
for(i=0;i<8;i++) {
     dct_8x1_transform();
}
```

The dct_8x1_transform( ) function is an optimized set of operations on 8 inputs, producing 8 outputs. You will often run this design at II=8, consuming 8 inputs and producing 8 outputs per iteration, for a net throughput of one input and one output per cycle.

In other cases, the code may be more amenable to decomposing to a finer granularity, thus enabling a lower II design with the same Task_II. An example is a radix-4 FFT. The inner loop of the FFT has four similar operations that imply a certain data path and set of functional units (only the innermost loop is shown):

```
for(i=0;i<N;i++) {
     // read 4 complex numbers
     radix_4_step0(…);
     radix_4_step1(…);
     radix_4_step2(…);
     radix_4_step3(…);
     // write 4 complex numbers
}
```

Assuming we pack both components of a complex number into a single memory location, there are 8 memory references per iteration. A common performance target might be II=4 (with two ports) or II=8 (with one port). A high performance implementation might separate the input and output memories and run at II=4 or II=2. Let us assume that we are targeting II=4 with one two-ported memory.

The compiler will inline all the operations from the four radix_4_step functions. This will cause the loop nest to have many additions, subtractions and multiplications. Targeting II=4 (or a Task_II that yields II=4 for this loop nest) enables Synphony C Compiler to share functional units for these operations (4 operations per functional unit across the 4 phases of II).The radix_4_step* functions are very similar and it is possible to write a single function:

```
radix_4_step(int step, …)
```

that replaces all four of the originals. If we can re-code with four times the number of loop iterations, then II=1 will have to be targeted to meet the same performance (Task_II). Even though the performance of the two codings is the same, the single function version (with four times as many iterations and II=1) will typically have much lower area than the four function version (with ¼ as many iterations and II=4). Synphony C Compiler is about to generate more efficient hardware with the single function coding.

One issue that arises in the single function coding is that we need all the inputs before we can start any of the computations:

```
int invalues[2][4];
for(i=0;i<N+1;i++) {
     if(i>0) { // capture the next values to process
          invalues[1][0] = invalues[0][0];
          invalues[1][1] = invalues[0][1];
          invalues[1][2] = invalues[0][2];
          invalues[1][3] = invalues[0][3];
     }
     for(j=0;j<4;j++) {
```

```
            if(i<N) {
                 // read 1 complex number (index is function of i,j)
                 invalues[0][j] = …;
            }
            if(i>0) {
                 radix_4_step(j,invalues[1]);
            }
}
```

We have created a pipeline where several values are read and stored locally for a computation that starts after all of them are read (for the FFT, this requires four complex numbers). This technique is quite common when reducing II for efficiency.

Concatenating Memories

Often, concatenation of memories can be used to reduce cost. This is equivalent to packing multiple data structures into a local memory. It is useful for cases such as Huffman coding or decoding and quantization where there may be multiple tables kept in memory. These tables are logically separate but are often accessed one at a time. For example:

```
if(luma) {
     if(DC) q = luma_DC[index];
     else q = luma_AC[index];
}
else { // chroma
     if(DC) q = chroma_DC[index];
     else q = chroma_AC[index];
}
```

In this coding, the four arrays luma_DC, luma_AC, chroma_DC and chroma_AC are mapped to four separate memories. Since they are all accessed in mutually exclusive fashion, we could pack them all into one memory without increasing the porting:

```
if(luma) {
     if(DC) q = tables[LUMA_DC_OFFSET + index];
     else q = tables[LUMA_AC_OFFSET + index];
}
else { // chroma
     if(DC) q = tables[CHROMA_DC_OFFSET + index];
     else q = tables[CHROMA_AC_OFFSET + index];
}
```

Shifters

Synphony C Compiler instantiates two types of shifters: variable shift amount and constant shift amount. Variable-amount shifters can be expensive as they must implement a data path for every possible shift amount.

Wherever a constant width shift is possible, you should code so that it is evident to the compiler. Synphony C Compiler will not make optimizations from variable shifters to constant shifters except where it can propagate a constant to the shift amount.

Example: Assume a 3-bit unsigned input variable shift_amount. Perhaps you know that this variable can only take on two values: 6 or 3. The following code is used:

```
x >>= shift_amount;
```

To optimize the design, this should be recoded as:

```
if(shift_amount==3) x >>= 3;
else x >>= 6;
```

A more complex case is the extraction of bytes from a 32-bit word:

```
for(i=0;i<4;i++) {
…
byte = (word >> (8*i)) & 0xff;
… }
```

If this loop is unrolled, it will automatically generate shift-by-constant functional units, since 8*i will always be constant after unrolling. However, if the loop remains iterative, then you can make the following change:

```
for(i=0;i<4;i++) {
…
if(i==0) byte = word & 0xff;
else if(i==1) byte = (word >> 8) & 0xff;
else if(i==2) byte = (word >> 16) & 0xff;
else byte = (word >> 24) & 0xff;
…
}
```

Now the code exposes the fact that there are only four possible values for i and they are all constant (known at compile time). The result will be four shifts by constant and four extractions. Both of these operations are essentially free after the RTL is synthesized to gates, as they are pure wiring operations. There is some cost for the comparisons and multiplexing of the four values, but this is equivalent with the implementation of a sparse shifter.

Scalarization Versus Internal Fast Memory

Scalarization is different from internal_fast (synthesized memory). With scalarization, individual elements of the array may have very different treatment including bitsizing, scheduling, and even removal via optimization. An internal_fast memory would still be treated as a single array in the hardware and compilation.

A brief list of the advantages and disadvantages of using scalarization and internal_fast is listed. You can specify #pragma scalarize to force a choice of scalarization. See the following text for details.

Scalarization Advantages Versus internal_fast:

- Allows complete separation of analysis for components, e.g. x[0] and x[1] (scalarized to x_0 and x_1 become separate variables and have no common dependences. By contrast, internal_fast memory will allow raw access to x[0] and x[1], but these memory references must still be analyzed along with other accesses to this memory.
- If the variable is accessed only inside one loop nest, the physical implementation is not required to use any storage – it may result in a pure wiring operation. internal_fast memory always physically implements the array in registers.
- If scalarized arrays are shared across loops, they automatically support task overlap for those loops. internal_fast memories do not support this task overlap unless they are multi-buffered.

Advantages of Internal_fast Over Scalarization:

- If scalarized arrays are shared across loops, the storage will be duplicated with an output register per writer loop and an input register per reader loop. internal_fast requires only the original array to be implemented in registers.
- Internal_fast allows both raw (constant index) and variable indexed addressing and implements both efficiently. Scalarization allows only raw addressing.

Some examples and use cases of scalarization follow.

In some cases, an array is intended to convey a wiring or combinational function rather than state. In these cases, it is absolutely critical to scalarize, as other options will create the state. For example:

```
{
   unsigned char byte[4];
   #pragma unroll k
   for(k=0;k<4;k++)
      byte[k] = (word >> (8*k)) & 0xff;
   out = byte[i%4];
}
```

In this case, the code describes an extraction and multiplexing function rather than any state. A real memory (internal_fast, due to the size) will be created. To optimize this case, you can specify the scalarization pragma:

```
#pragma scalarize byte
```

6

C/C++ Synthesizable Syntax

An application generally consists of several functions. Only some of them contribute to hardware synthesis; the others form the testbench that verifies its functionality. This chapter describes the supported C and C++ syntax in the functions that are used for hardware synthesis:

- Synthesizable Code and Testbench Code
- Type Syntax
- Procedure Syntax
- Statement and Expression Syntax
- Testbench Code
- ANSI Support

Synthesizable Code and Testbench Code

Synthesizable code refers to the portion of the application that is implemented in hardware. The remaining code consists of testbench files and data sets that exercise the application to test and verify it. The testbench can be the application code minus the PPA code, or it could be a set of files designed specifically for verifying the PPA code.

The Synphony C Compiler tool can synthesize hierarchical designs by building lower-level designs and importing them into higher-level designs. A TCAB is a hardware block that is meant to be integrated into another block during hierarchical synthesis. The Synphony C Compiler tool uses an import mechanism to automatically integrate the design. See *Building Hierarchical Designs* in the *User Guide* for information about using TCABs.

You can specify the function to be built as hardware either PPA or TCAB using the following implementation parameter setting.

```
set_params -target function_name
```

For more information, see "Synthesizable Code and Testbench" on page 5-3.

The Synphony C Compiler tool synthesizes all the logic inside the specified function as hardware automatically. All the remaining code becomes part of the testbench.

The Synphony C Compiler tool uses the PPA code and the testbench code to automatically construct and verify the hardware respectively. For successful synthesis of applications with the Synphony C Compiler tool, you need to modify the PPA code to comply with certain requirements. This set of constraints defines the synthesizable subset of C or C++ for the Synphony C Compiler tool.

Other requirements are prescriptive in nature. For example, certain control structures like constant trip count loops have very efficient mapping onto hardware by the Synphony C Compiler tool. Though the Synphony C Compiler tool supports all kinds of C loops such as for-loops, do-while, while, and break and the continue statements, use of arbitrary structures might not result in efficient hardware.

There is another set of constraints, which cannot be statically checked by the compiler. Nevertheless, they are important for the correct behavior of the synthesized hardware. An example of such a constraint is "array indexing should be within the bounds of the array". Static checking cannot always guarantee that this condition holds. The Synphony C Compiler tool relies on your test coverage during simulation to impose these conditions. The effect of violating these constraints in the synthesized hardware is undefined. The C simulation supports gcov option for profiling and test coverage. Synopsys recommends you to rewrite your testbench or use additional data sets to increase the coverage.

The following sections describe the syntax supported by the Synphony C Compiler tool in the synthesizable portion of your application.

Type Syntax

This section describes the following aspects of C/C++ syntax support in the Synphony C Compiler tool:

- File Syntax
- Base Type
- Modifiers
- Enum and Typedef
- Parameterized Integer Data Types
- Arrays
- Pointers
- Static Variables and State Encapsulation
- Class and Structures
- Constructors and Destructors
- Unsupported C++ Syntax
- Procedure Parameters and Return Values
- Initializations in Declarations

File Syntax

The Synphony C Compiler tool has some restrictions on the file syntax to be used in the application files:

- The Synphony C Compiler tool reserves the following prefixes for its own use; `scc`, `SCC`, `pico`, `PICO`. Do not define names that use any of these strings as a prefix.
- The Synphony C Compiler tool allows you to use C preprocessor directives in the application files as well as the testbench files.

Base Type

The Synphony C Compiler tool supports the following scalar (base) data types, both signed and unsigned:

char	8 bits
short	16 bits
int	32 bits
long	32 bits
long long	64 bits

The widths are host-dependent; the ones in the table are valid on a 32-bit machine. The maximum bitwidth supported on integral types is 64 bits. To ensure that a variable always has a particular bitsize, use the bitsize pragma. For more information about this pragma, see "bitsize Pragma" on page 7-20.

The Synphony C Compiler tool also supports the following data types:

- s_int<n> (n bits, 1 <= n <= 64)
- s_uint<n> (n bits, 1 <= n <= 64)

The Synphony C Compiler tool also supports the following fixed-point data types:

- s_ufixed <int wl=32, int iwl=32, scummed Q=S_TRN, zoomed, O=S_WRAP, int n_bits=0>
- s_fixed <int wl=32, int iwl=32, s_q_mode Q=S_TRN, s_o_mode, O=S_WRAP, int n_bits=0>

For more information on requirements to use SystemC data types, see "Fixed-Point Library Support" on page 9-2.

The tool does not support the following C types:

- float
- double

Modifiers

The Synphony C Compiler tool supports the following modifiers:

extern	For globally scoped variables. Affects how variables are mapped to hardware storage structures. For more information, see "Memory Support" on page 8-6.
const	Affects how variables are mapped to hardware storage structures. For more information, see "Memory Support" on page 8-6. An array variable with const modifier would result in a ROM. For more information, see "ROMs" on page 8-10.
signed	
unsigned	
static	Implements a single copy of the hardware in the created hardware. See "Static Variables and State Encapsulation" on page 6-11 for more details.

The tool does not currently support these modifiers:

- register
- volatile

Enum and Typedef

The tool supports enums of supported data types. The enum constants must be in the [-(2^31-1), 2^31-1] range. The GCC compiler accepts the entire integer range, so the tool automatically converts values beyond 2^31-1 to unsigned values.

Use typedef to define synonyms for the supported types. For example:

```
typedef int WORD;
typedef unsigned int UWORD, BOOL;
typedef struct {
          int r[100];
          int g1[100];
          int g2[100];
          int b[100];
} pixel_array;
```

You can specify typedef for bit size; for more information, see "Non-Standard Variable Widths" on page 8-2.

Parameterized Integer Data Types

The Synphony C Compiler tool provides native support for width-parameterized integer data types. Unlike bitsize pragmas on C data types, the width in a parameterized data type is part of the program semantics, and overflow is well-defined and does not trigger errors or warnings. This can be an advantage if you want such overflow semantics or if you are confident about the widths and do not require checking for overflow. The maximum width for these data types is 64.

Native Data Types s_int and s_uint

To use the Synphony C Compiler native integer data types `s_int` and `s_uint`, you must include the `scc_sint.h` file in the application files, using the preprocessing directive syntax:

```
#include "scc_sint.h"
```

You can use these data types in the synthesizable source code as well as in the testbench code. Here is an example of `s_int` data type usage:

```
s_int<4> a;
s_int<7> b, c;
a = 35;
b = a + c;
```

You can convert native Synphony C Compiler integer data types to other data types and vice versa. To convert from another data type to a native integer data type, directly assign the variable to it. For example,

```
unsigned int a;
s_int<32> b;
b = a;
a = b.to_uint();
```

You can use an `s_int` or `s_unit` variable as the loop index of a for loop, but the tool cannot analyze this syntax. Such loops are treated as variable trip count loops and loop bounds must be provided explicitly by using the `num_iterations` pragma. For details, see “Specifying Bounds on Loop Iterations” on page 5-69.

Note that, the execution semantics of `s_int` is of a `long long` type, and `s_uint` is really that of an unsigned `long long` type. For example,

```
s_int<32> x, y;
long long z;
z = x * y;
```

This example has `long long` semantics, which is a different behavior from the following code snippet:

```
int x, y;
long long z;
z = x * y;
```

In the first example, the multiply operation has `long long` multiply semantics. In the second example, the multiply operation has integer semantics, which means that the result of the multiply is truncated to the width of an integer before being assigned to the variable on the left side.

When performing bitwise initialization of `s_int` and `s_uint` type variables, lint simulation might incorrectly report uninitialized warning and show a mismatch in the simulation outputs. For example, the following code might lead to mismatches in the outputs of lint simulation:

```
s_uint <16> var;

#pragma unroll
for (int i=0; i<16 ; i++)
  if (i<NUM)
     var[i]=1;
  else
     var[i]=0;
```

To avoid this limitation, initialize the `s_int` and `s_uint` type variable to zero before you begin to perform bitwise initialization by modifying the code as shown:

```
s_uint <16> var=0; //initialize to zero before bitwise initialization
#pragma unroll
for (int i=0; i<16 ; i++)
  if (i<NUM)
     var[i]=1;
  else
     var[i]=0;
```

Supported Operators for SCC Integer Data Types

The SCC native integer data types support the following operators:

```
Constructors
      s_int();
      s_int(int_type v);
      s_int(const s_int<W>& a);
      s_int(unsigned long a);
      s_int(long a);
      s_int(unsigned int a);
      s_int(int a);
      s_int(uint64 a);
Assignment Operators
      s_int<W>& operator= (int_type v);
      s_int<W>& operator= (const s_int<W>& a);
```

```
      s_int<W>& operator= (unsigned long a);
      s_int<W>& operator= (long a);
      s_int<W>& operator= (unsigned int a);
      s_int<W>& operator= (int a);
      s_int<W>& operator= (uint64 a);
Bit Selection and Part Selection Operators
      s_int_bitref operator[ ] (int i);
      s_int_bitref_r operator[ ] (int i) const;
      s_int_bitref bit(int i);
      s_int_bitref_r bit(int i) const;
      s_int_subref operator() (int left, int right);
      s_int_subref_r operator() (int left, int right)const;
      s_int_subref range(int left, int right);
      s_int_subref_r range(int left, int right) const;
Prefix/Postfix Increment and Decrement Operators
      s_int_base& operator++ (); // prefix
      const s_int_base operator++ (int); // postfix
      s_int_base& operator-- (); // prefix
      const s_int_base operator-- (int); // postfix
Implicit/Explicit Conversions
      operator int_type() const;
      int to_int() const;
      unsigned int to_uint() const;
      long to_long() const;
      unsigned long to_ulong() const;
      int64 to_int64() const;
      uint64 to_uint64() const;
Arithmetic, Bitwise, and Comparison Operators
      +, -, *, /, %; &, |, ^, <<, >>;
      +=, -=, *=, /=, %=; &=, |=, ^=, <<=, >>=;
      ==, !=, <, <=, >, >=;
Reduce Methods
      bool and_reduce() const;
      bool nand_reduce() const;
      bool or_reduce() const;
      bool nor_reduce() const;
      bool xor_reduce() const;
      bool xnor_reduce() const;
```

The tool does not fully support the comma (`,`) concatenation operator. You can use the `concat` macro on supported parameterized data types and bit and part selection operations, as shown in the following example:

```
concat( L, R )
```

However, you cannot use `concat` on `bool` types.

SystemC Data Types sc_int and sc_uint

You can convert native SCC integer data types to SystemC data types and vice versa in testbench code, but not in synthesizable code. To convert from native SCC integer data

types to the SystemC `sc_int` and `sc_uint` data types, you can use the `to_sc_int` and `to_sc_uint()` methods. For example,

```
sc_int<32> a;
s_int<32> b;
b = a;
a = b.to_sc_int();
```

You can synthesize code with `sc_int` and `sc_uint` data types without manually replacing each `sc_int`/`sc_uint` with `s_int`/`s_uint`. To do so, include `scc_sint.h` in the application file and define the compiler flag `CONVERT_SCINT_TO_SINT`:

```
set_params -cppcompiler_args "-DCONVERT_SCINT_TO_SINT"
```

This automatically converts `sc_int`/`sc_uint` to `s_int`/`s_uint` throughout the application code. Linking with a SystemC library is not required unless you want to use true SystemC data types in the testbench code. Note that the synthesizable support of `sc_int`/`sc_uint` under this flag is limited to that of `s_int`/`s_uint`, as described in the previous sections.

If you are using true SystemC `sc_int`/`sc_uint` data types in the testbench code and synthesizing `sc_int`/`sc_uint` data types in the application code (via defining `CONVERT_SCINT_TO_SINT`), then converting between data types in the testbench needs a namespace qualifier of either `scc` or `sc_dt`. For example,

```
sc_int<32> a; // Testbench variable; default namespace is SystemC sc_dt
scc::sc_int<32> b; // Synthesizable variable in SCC namespace scc
b = a;
a = b.to_sc_int();
```

Note that the `scc_sint.h` file redefines some of the functionality provided by the `systemc.h` file, so it must be included after the `#include` line for `systemc.h`.

Arrays

The Synphony C Compiler tool supports single and multidimensional arrays of the supported base data types, arrays of structures and class objects.

If you index a multidimensional array beyond its declared bounds, the Synphony C Compiler tool produces an undefined result. This behavior differs from standard ANSI-C semantics. Consider the example of a 2-D array declared as shown:

```
int X[10][10];
```

In ANSI-C, a dynamic reference to X[2][10] is the same as a reference to X[3][0]. However, the Synphony C Compiler tool treats a dynamic reference to location X[2][10] as undefined. Lint simulation signals an error if it encounters out-of-bound situations, but the efficacy of this check depends on the coverage provided by the data sets. For more information, see “Out-of-Bounds Array Access Checks” on page 12-6.

Pointers

The Synphony C Compiler tool lets you use pointers in the synthesizable code if the pointers can be statically resolved. This means that the location or variable the pointer is referring to must be known at compile time. You cannot use global pointers; you can only specify them in the scope of the procedure.

- The Synphony C Compiler tool supports pointers if the pointer can be statically resolved as pointing to a variable or function in the synthesizable code.

```
int a[100];

int *q = &a[i];
```

- The Synphony C Compiler tool supports pointer arithmetic only if the pointers involved in the arithmetic elements point to the same scalar array.

```
int a[100], b[100];
int c[10];
char C1;
int *p = a;
ptr ++;
ptr += i;
ptr1 - C1; // where “C1” is int/char
int *q = &a[5];
int *r = &b[5];
*(p-q)  // legal
*(p-r)  // not legal as p and r point to different arrays
if (p!=q) { ... } // legal as p and r point to same array
if (q==r) { ... } // not legal as p and r point to different arrays
```

  When using user-defined C++ classes with operator overloading, a common coding style might use:

```
SomeClass& SomeClass::operator=(const SomeClass &that) {
 if (this != &that) {   // check for self-assignment
    ... // Deallocate, allocate new space, copy values...
  } // end if (this != &that)
  return *this;
} // end operator==()
```

  In this example, the variables that the pointers refer to are not known at compile time, causing an error. To correct this error, you need to remove the self-assignment check and verify that there is no self-assignment in the target code.

  Illegal pointer comparisons are flagged as errors during the linting simulation by the tool.

- The tool permits comparison (<,<=,>,>=,==,!=) of pointers of the same object. If you compare operands that point to different objects, the behavior is undefined.

```
int *p=&a[i];
int *q=&a[j];
```

```
int *r=&b[i];
X=(p<q); //Legal
Y=(p<r); //Undefined behavior since p and r point to different
               //arrays.
```

- You can compare null pointers, and run equality (== and !=) comparisons against such pointers.

```
int*p=NULL;
if (cond) p = &a[i];
     if (p != NULL) {
     // code
     }
```

- You cannot include pointers as members of structs and classes in the synthesizable code.
- The tool only allows inlined functions to return pointers. If a function takes a pointer as an argument, that function is synthesizable only if the pointer is statically resolvable.
- Class member functions can also take pointers as arguments and return pointers if the pointer usage satisfies the conditions described in this section.
- The tool does not support pointers to pointers, or arrays of pointers.
- You cannot convert pointer types:

```
     int *temp;
     char *temp2;
     temp = (int*)temp2; //This is not allowed.
```

Static Variables and State Encapsulation

The static modifier is commonly used in C or C++ syntax to restrict the scope of a variable and extend its lifetime. You can apply this modifier within the scope of a procedure or a file.

You can use static variables to preserve the state across multiple calls to the TCAB or PPA. This state is encapsulated at the PPA or TCAB layer in which it is defined.

State encapsulation enables modular design. When internal fast memories and scalars are declared as static in a TCAB, the persistent state is abstracted away from higher levels of hierarchy. There will not be any ports at the TCAB level that expose the state to the instantiating entity. This makes the RTL cleaner and makes the design more modular.

If the static variable is defined at the file scope, it is visible inside that file; if it is defined in procedure scope, it is visible inside that procedure. The Synphony C Compiler tool encapsulates the static variable at the scope of the TCAB or PPA layer where it was defined. The tool does not provide any external access beyond that layer except for user-supplied memories. For user-supplied memories, the ports are exposed from the PPA layer all the way through the module that uses the static memories. The generated RTL has the static

state encapsulated within the module where the variable was defined, but you can control this with the `localize_access` pragma, which is described in "localize_access Pragma" on page 7-33.

The Synphony C Compiler tool maintains a single copy of a static variable. If a static scalar is encapsulated in a PPA and used by multiple PAs, it is shared. A static scalar encapsulated in a TCAB also has a single copy. TCABs containing encapsulated states cannot have multiple instances because of the single copy requirement.

If a static scalar is not encapsulated in the current design, its value is copied from the original single copy at the beginning of the task and copied back at the end of the task. If memory is not encapsulated in the design, the tool has ports on the design boundary to access the memory.

File scope static variables can have the same name in different files. To add constraints to these variables, use this syntax when referring to the variable:

```
variable @ filename
```

For example, you can specify `input_sample_time` for the MacData variable inside the wlan_top_level.c file by using the following statement:

```
set_params -input_sample_time
     MacData@wlan_top_level.c:infer
```

Class and Structures

The Synphony C Compiler tool supports C++ encapsulation of data, functions, constructors and destructors inside C++ classes and struct type elements. For more details on constructor and destructor support by the Synphony C Compiler tool, see "Constructors and Destructors" on page 6-13.

When you instantiate objects of a class, Synphony C Compiler inlines all the code at the place of use. Access control specifiers like private, protected or public are also allowed in synthesizable code. Data members can be any of the types mentioned so far in this chapter except pointers. The restrictions on usage of pointers are same as listed in the section "Pointers" on page 6-10.

The Synphony C Compiler syntax supports friend specification on both functions and classes, usage of templates, namespaces and overloading. The Synphony C Compiler tool supports single inheritance. Overloaded functions can be built as TCABs or PPAs, as described in "Overloaded Functions" on page 7-5.

The Synphony C Compiler tool supports specification of nested classes, references, function returning references (for inline functions only), functions with default arguments and lvalue usage.

The Synphony C Compiler syntax does not support the following:

- union construct
- SystemC data type as fields of C structures
- Bitfield specifications

Bitsizes of the fields of C structures can be specified using `#pragma bitsize`. For additional information, see "Non-Standard Variable Widths" on page 8-2.

Constructors and Destructors

The Synphony C Compiler tool supports constructors and destructors. It calls the constructor and destructor routines as defined by the C++ language, and creates hardware accordingly. This section describes how the tool handles loops, unused values, and global objects when they are used with constructor and destructor code.

Loops in Constructors and Deconstructors

Constructors and destructors cannot contain loops. Either unroll the loops in the constructor and destructor, or create a new function with the same functionality, with explicit calls to that function for the objects as applicable.

The following example shows how to rewrite code to avoid using loops:

Original Code with Loop Inside Constructor

```
class X {
public:
     int a[20];
     X() {
         for (int i=0;i<20;i++)  a[i]=0; //Loop inside constructor not supported
     }
};

void ppa() {
     X p[10];
     }
}
```

Rewritten Code with Unrolled Loop

```
class X {
     public:
     int a[20];
     X () {
          #pragma unroll
          for (int i=0;i<20;i++) a[i]=0;
     }
};

void ppa() {
     X p[10];
}
```

Unused Values

The constructor and destructor functions can have side effects, and the Synphony C Compiler tool honors these effects inside the synthesized hardware.

The next example shows a constructor that contains a stream call for which the return value is not used:

```
extern int valIn.get();
class X {
     public:
     int A;
     X () {
          A=valIn.get();
     }
};
int ppa() {
     X a,b,c;
     return (a.A+b.A);
}
```

The variable c is not used in the PPA, but a call to the stream from the constructor of c is still part of the synthesized code. Copies of constructors for all the class objects are inlined at the point of declaration, and the hardware that the tool creates for this example issues three stream calls for a, b, and c.

Global Objects

Constructors and destructors of global objects are considered testbench code, and are not synthesized into hardware. Standard testbench coding practices should be applied for global constructors and destructors.

In the following example, the global array g_var1 can be accessed by the host, and all its members are part of the global state. The constructor of the class X modifies a member (p) of the array. This constructor invocation is part of the testbench code and the hardware is not synthesized.

```
class X {
     public:
          int p;
          X() {          // Constructor
          p=0;
          }
};
X g_var1[10];
void ppa( )  {
     int a;
     a = g_var1[0].p + g_var1[4].p;
     g_var1[5].p=a;
}
```

Unsupported C++ Syntax

The tool does not support the following C++ syntax:

- Virtual functions
- Multiple inheritance
- Exceptions
- RTTI (typeid, dynamic_cast, and so on)
- Dynamic memory allocation
- Pointer members of a class/struct
- export keyword
- Pointer to data members in a class (.* and ->* syntax)
- Dynamic initialization
- User-defined names that begin with any of these prefixes: `scc`, `SCC`, `pico`, `PICO`
- The float and double data types
- The register and volatile modifiers
- Pointers as members of structs and classes in the synthesizable code
- Pointers to pointers, or arrays of pointers
- Pointer arithmetic where the pointers in the arithmetic elements point to different scalar arrays
- Union constructs, SystemC data type as fields of C structures and bit field specifications for classes and structures

- Loops in constructors and deconstructors

Procedure Syntax

The application files can include multiple procedures, but you must designate one of them as the top-level procedure, or PPA procedure. It is recommended that you use lowercase for the PPA procedure and application file names, because some Verilog simulators restrict the use of upper-case names. The Synphony C Compiler tool generates Verilog signal names similar to the variable names in your application whenever possible.

Besides the PPA procedure, you can have TCAB procedures for functions that are built as TCABs to be imported into higher-level designs. For more information, see Chapter 10, "Hierarchical Synthesis." The TCAB procedures are called from the PPA procedure or from other procedures in the PPA procedure call tree. Non-TCAB procedures that are called from the PPA procedure are inlined.

The following subsections describe the syntax requirements for procedures and restrictions in the synthesizable code:

- Procedure Parameters and Return Values
- Statement and Expression Syntax

Procedure Parameters and Return Values

All procedures, including the PPA procedure, can have parameters and return values. If the procedure is called from the PPA procedure, and is not built as a TCAB, it must conform to the inlining requirements outlined in the section "Statement and Expression Syntax" on page 6-17.

Scalars and Array Parameters

The scalar parameters and return values of a PPA procedure become PPA inputs and outputs respectively. During integration, connect memory ports to the appropriate memories.

- Array parameters passed to the PPA or TCAB procedure result in external memory ports on the PPA interface, if they have not been scalarized.
- Array parameters whose accesses within the procedure are only through constant indexes are converted into scalar parameters with appropriate inputs and outputs.
- An internal_fast array cannot be passed as a PPA parameter, because arrays passed as parameters are considered part of the system outside the PPA. For this reason, the restriction does not apply to TCABs. Create access to the actual array or memory

outside the PPA or TCAB. You can pass arrays with non-zero base addresses for inlined functions, but not for functions synthesized as TCABs.

In the following example, a slice of the mem array is passed to the sum function. The Synphony C Compiler tool supports the inlining of the sum( ) function, but would error out if sum( ) were built as a TCAB.

```
int sum(int *x) {…}
void ppa() {
     int mem[4][10];
     sum_row = sum(mem[i]);
     …
     }
```

Reference Parameters

Reference parameters are supported in C++. The following example shows one such procedure being called, using the address-of operator (&):

```
int g, h;
fn (int *x, int& y){
     *x = 10;
     y = 20;
     k = g;
     }
fn(&g, h);
```

The Synphony C Compiler tool implements this logic as described here:

- It creates a copy of the variables inside the function (copy-in), and performs all operations using that copy.
- At the end of the function, it copies any modifications to these variables back to the original arguments (copy-out). In true reference semantics, k gets the value of 10, whereas in copy-in or copy-out semantics k gets the previous value of g.

  The Synphony C Compiler tool issue an error message during linting if there is any possibility of aliasing between parameters and globals within the PPA code. You must ensure that no such aliasing occurs across the testbench-accelerator interface.

Statement and Expression Syntax

The Synphony C Compiler tool supports almost all ANSI-C syntax within a procedure body, with the following restrictions:

- No goto or exit statements.
- No pointers, except where they can be statically resolved, or for address of operators and pointer dereferencing.

If you pass a scalar address as a function argument, you can use a pointer dereferencing operation (*<variableName>) in the function body. For a discussion of some other requirements, see "Statement and Expression Syntax" on page 6-17.

The Synphony C Compiler tool supports most ANSI-C statements and expressions.

The Synphony C Compiler tool support calls to procedures defined in application files. There are some requirements for these calls:

- All called procedures must be defined in the application files.
- You cannot use recursive or mutually-recursive procedure calls.
- After inlining and unrolling, the code must not violate any restrictions outlined in this chapter.

The Synphony C Compiler tool inlines all procedure calls in the PPA procedure except the imported TCAB procedures and library calls, like the ones listed:

- Loop stream communication calls ("Stream Support" on page 8-24)
- Custom opcode calls such as scc_asm_MIN_W( ) ("Custom Opcodes" on page 8-44)
- Multi-level loop break calls scc_break_loop( ) and scc_end_loop( ) ("Multi-Level Break Statements" on page 8-43)

Define all called procedures in the app files, so that they can be analyzed and inlined by the Synphony C Compiler tool. If a function definition appears after its use, use the appropriate function prototypes to declare the function before it is used. You can call inlinable functions from the testbench, but they are not inlined in the testbench.

The following example shows simple inlining with scalar arguments:

Original Code	After Function Inlining

`int f(int x) {`
`  int j;`
`  for(j=0;j<5;j++) x++;`
`  return x;`
`}`
`int x;`
`void ppa(void) {`
`  int y = 3+x;`
`  x = a+b+c + f(y*y);`
`}`
	`int x;`
	`void ppa() {`
	`  int y;`
	`  int scc_tmp_ret_1;`
	`  y = 3+x;`
	`  {`
	`  int x, j;`
	`  x = y*y;`
	`  for (j=0;j<5;j++) x++;`
	`  scc_tmp_ret_1 = x;`
	`  }`
	`  x = (a+b+c) + scc_tmp_ret_1;`
	`}`

You can also declare arrays as formal parameters and pass them as arguments. When you declare multi-dimensional arrays as formal parameters, specify the size of all but the first dimension as a constant so that addresses are generated correctly. For example:

```
int det(int a[2][2]) {
     int c;
     c = a[0][0]*a[1][1] - a[0][1]*a[1][0];
     return c;
}
void copy(int a[][2], int *b, int n){
     int i;
     #pragma unroll
     for(i=0; i<n; i++) {
          a[0][i] = a[1][i];
          a[1][i] = b[i];
     }
}
void scc_stream_output_detx(int);
void myppa() {
     int j,d;
     int x[2][2], p1[2];
     x[1][0] = x[1][1] = 0;
     for(j=0; j<100; j++) {
          p1[0] = j;
          p1[1] = j+1;
          copy(x, p1, 2);
          d = det(x);
          scc_stream_output_detx(d);
     }
}
```

This example also shows the unroll pragma being used to unroll the inner loop automatically after the function copy is inlined. The trip count of the loop becomes a constant only after the inlining. For more information, see “unroll Pragma” on page 7-70.

Initializations in Declarations

The Synphony C Compiler tool supports initialization when you declare a variable. If you use the `reset_data_registers` implementation parameters, the tool automatically initializes all data registers to the required initialized values. Initialization of static variables is done automatically on entry to the function which use static variables, and on every call to the function when using other variables.

Initialization of Variables in Global or File Scope

Unlike the C language, the Synphony C Compiler tool does not support automatic initialization to zero for global and static variables. This prevents initialized hardware from

being generated unless explicitly indicated. To indication the initialization of variables to zero, explicitly specify this syntax:

```
int a[20] = {0};
char b = 0;
static int c[10] = {0};
```

The following describe how the tool handles initialization for specific cases:

- Initialization of memories or scalars synthesized inside the tool

  If you use the `reset_data_registers` implementation parameter, the Synphony C Compiler tool initializes the scalars or arrays that are synthesized in the tool to zero. It uses resettable data registers to initialize to zero.

  For global and file static variables, the tool treats non-zero initialization values as testbench code and does not synthesize them. It is up to the host to do non-zero initialization.

- Initialization of unsynthesized memories or scalars

  For user-supplied memories, where the memories were not synthesized by the Synphony C Compiler tool, the initialization code is treated as testbench code and is not synthesized. Any initialization must be done by the host.

- Internal block RAM (FPGA)

  In FPGA designs, the tool treats initialization syntax for internal block RAMs in file scope as testbench code and does not synthesize it. Any initialization must be done by the host.

  If the scope of a global memory is changed to procedure scope using the localize_access pragma, the tool handles initialization as if it was defined at that scope.

Variables in Local or Procedure Scope

The Synphony C Compiler tool synthesizes the initialization code when the function is called. The following describe some of the details:

- Initialization of memories or scalars synthesized inside the tool

  Zero initialization:
  The Synphony C Compiler tool synthesizes the hardware that initializes these variables to zero. For static variables defined locally, the tool creates resettable registers to initialize the variables to zero. It only does this if you explicitly specify the initialization, using the following syntax:

  ```
  static int a[20] = {0};
  static char b = 0;
  ```

  Non-zero initialization:
  If you have specified non-zero initialization values in the code, the tool synthesizes a ROM with the specified values. It then copies the contents of the ROM to the memories

to be initialized. For static variables, the tool does this the first time the function is called, For non-static variables, it does this every time the function is called.

For array variables of C++ data types like `s_int<N>` and `scc_fixed`, partial initialization syntax does not initialize the uninitialized array elements to zero. Use complete initialization as shown to initialize all elements in such an array.

```
s_int<5> x[4] = { 1, 0 } ; // Initializes only x[0] to 1 and x[1] to 0
                           // Other elements are uninitialized

s_int<5> y[4] = { 0,0,0,0 } ; // Initializes all elements of y to 0
```

- User-supplied memories and block RAM (FPGA)

  For these memories in FPGA designs, the tool uses a loop to initialize to zero or non-zero. To initialize to a non-zero value, the tool creates a ROM that holds the values indicated by you in the code, and then initializes the memories by copying from the contents of the ROM in a loop. For static variables, the tool does this the first time the function is called, For non-static variables, it does this every time the function is called.

  If a user-supplied array in local scope is initialized, it might result in multi-ported memory, depending on the II and Task_II constraints you set by you. The initialization would have a separate loop nest, and therefore, a separate PA. If you do not want to generate this loop nest, write explicit initialization code.

- Constant variables

  The tool supports initialization if the variable is declared as a ROM. Declare an array as a ROM by specifying the const storage modifier or by using #pragma rom. For more information on memory, see “Memory Support” on page 8-6.

Testbench Code

The Synphony C Compiler tool supports most C++ syntax for use in the testbench code. Restrictions vary, depending on whether the testbench code in contained within the list of application files.

- Testbench code outside the application files

  There are no syntax restrictions on testbench code that is outside the application files. All C++ syntax is supported. However, to generate a testbench that can properly plug into the simulation flows, there are some important guidelines to follow, as specified in the next section.

- Testbench code inside the application files

  If the testbench code is included in the application files, there are some restrictions:

  - Virtual classes and exceptions are not supported.

- Standard libraries that require virtual classes and/or exceptions (such as iostream.h) cannot be included in application files.

Testbench Code Examples

The following examples illustrate how to specify testbench code:

- Testbench Example for Scalar Input Initialization
- Testbench Example for Reading an Output

Testbench Example for Scalar Input Initialization

Consider the array given, declared in your PPA code. The memory for this array will be part of the hardware synthesized for the PPA.

```
int filter0_local[64][16];
```

You can initialize an element of the array in the testbench code, using standard C syntax:

```
filter0_local[10][3] = 1000;
```

Similarly, the array can be initialized by copying another array declared in the testbench code. Suppose you have the following array declared in your testbench code and never used in your PPA code:

```
int filter0[64][16];
```

Then you can copy all of filter0 into filter0_local by writing the following in your testbench code:

```
for (i = 0; i < 64; i++)
for (j = 0; j < 16; j++)
           filter0_local[i][j]= filter0[i][j];
```

Byte arrays are represented in the address map as packed arrays to match the programming model but cannot be accessed as a packed 4-byte array to a word. This is because in hardware the byte arrays are synthesized as a byte-wide memory bank, which cannot do 4-byte transactions at the same time. So the array can only be accessed 1 byte at a time with a byte address.

Testbench Example for Reading an Output

The outputs behave similar to the inputs. To copy all of filter0_local into filter0 write the following in the testbench code:

```
for (i = 0; i < 64; i++)
          for (j = 0; j < 16; j++)
               filter0[i][j] = filter0_local[i][j];
```

There are some variations, depending on whether you use SystemC, C, or C++:

- SystemC Testbench

  The SystemC arrays must be initialized through the SystemC API functions.

  ```
  sc_int<11> filter0[64][16];
  for (i = 0; i < 64; i++)
  for (j = 0; j < 16; j++)
  SCC_writeMemInput_sc_int(filter0,16*i+j, filter0_local[i][j]);
  ```

  For proper linkage with SystemVerilog testbench wrapper provided by the Synphony C Compiler tool, the `main` function and all the external scc_stream_input_ and scc_stream_output_ functions must conform to the C calling convention with the following signature:

C Testbench	C++ Testbench
`extern int main(int argc, char**argv);`	`extern “C” int main int argc, char **argv);`

```
extern int xin.get();
extern "C" int main(int argc, char **argv);
extern "C" int xin.get();
```

If you get a linker failure that either main (in c simulations) or scc_main (in SV simulations) is undefined, the following could be the causes:

- Testbench is a C++ file but C linkage declaration for “main” was missing.
- Proper prototype signature for main was not used.
- scc.h was not included in the testbench file. That inclusion allows your main to be renamed to scc_main in SV simulations since SystemC inserts its own main.

ANSI Support

The Synphony C Compiler tool accepts applications that are ANSI C compliant. However, there are important exceptions where the tool does not comply with ANSI C. This section identifies and explains those exceptions.

Unsupported ANSI C Elements

This table lists the ANSIC C elements that are not supported by the SCC tool.

ANSI C Element	Description
Control Flow Constructs	
goto, "label:"	No control constructs that involve direct jumps.
Expressions	
Type casting to unsupported types	No type casting to unsupported types (see the Floating point types entry in this table). For example, (double)<*expression*>
Variable Types	
Pointers	No pointer declaration or usage/dereference (no int *var, *var, var->, or &var). Allowed only as formal parameters to procedures, when they refer to array names or as an actual argument to a procedure, which takes a reference parameter.
Union	Not supported.
Floating point types	Only char, short, long, and int types, and the unsigned and signed qualifiers types are allowed. No float, double, or long double.
typedef to unsupported types	Aliasing of types is only permitted for SCC supported types.
Initialization	
String constants, use of “ "	No string constants are allowed for initializing or assigning array variables.
Function Parameters	
Passing slice of array	Passing slice of array is allowed for inlined functions, but not for functions synthesized as TCABs.

ANSI C Elements Supported with Specific Semantics

The following table lists ANSI C elements that are supported by the Synphony C Compiler tool, but whose semantics in the tool may not be as general as those used for general-purpose C compilers.

ANSI C Syntax	Description
Expressions/Operators	
sizeof	The tool uses this to reflect the physical word size of the underlying host machine. For example, the sizeof an int is 4 bytes. If this operator is used in a C application, you might want it for different purposes. For example, to reflect the size of the words in your target synthesized system).
Storage-class Specifiers & Type Qualifiers	
auto, register	These qualifiers are ignored by the tool
volatile	Not supported.
printf	Can be used up until the preprocess -partial stage, to catch any application issues.

Implementation-Defined ANSI C Elements

There are some items in the ANSI C specification that are “implementation-defined”. The following table lists the implementation done in the Synphony C Compiler tool.

Operator / Data Type	ANSI C	SCC
Right shift operator (>>)	The result of E1 >> E2 is E1 right-shifted by E2 bit positions. If E1 has an unsigned type or if E1 has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of E1 / 2E2. If E1 has a signed type and a negative value, the resulting value is implementation-defined.	If E1 has a signed type and a negative value, the resulting value is the arithmetic shift of E1 by E2 bits.
char	char = 8 bits	char = 8 bits

Operator / Data Type	ANSI C	SCC
short	short = 16 bits	short = 16 bits
int	int = 32 bits	int = 32 bits
long	long = 32 bits	long = 32 bits
long long	long long = 64 bits	long long = 64 bits

7

Pragmas and Constraints

This chapter describes the different methods to specify user constraints for a design. This chapter describes the following:

- User-Specified Constraints
- Tcl Constraint Commands
- Overview of Pragmas
- Pragma Descriptions

User-Specified Constraints

You can specify constraints that the Synphony C Compiler tool uses when it generates the hardware. The following describe the two ways in which you can specify constraints:

- Tool Interface Constraints
- Code Pragmas

Pragmas defined in the C code override corresponding constraints set in the tool interface. Use the tool interface to specify global constraints that affect all the variables in the code. For example, the Synphony C Compiler tool allows host access through global arrays. You can disable host access globally from within the tool, or use pragmas to disable host access to global arrays on an individual basis. Not all pragmas have tool interface equivalents.

Tool Interface Constraints

Use the tool interface to set global constraints. You can set these constraints either from the command line or through the GUI.

- GUI constraints
  Set the constraint values in the various dialog boxes.
- Command-line interface
  Type the relevant commands in the console. You can also put the Tcl commands in a Tcl file, and then execute the set of commands by typing synphonycc <fileName>.tcl in the Linux shell. See "Tcl Constraint Commands" on page 7-6 for a list of the commands you can use to set constraints. For detailed information, refer to Chapter 3, "Synphony C Compiler Commands.

Code Pragmas

Add pragmas in the synthesizeable code to specify constraints. Use pragmas to override global settings on individual objects. Not all pragmas have equivalents within the tool.

For more information about different pragmas, see "Overview of Pragmas" on page 7-12, and the individual pragmas documented in the rest of the chapter.

$SCC Labeling

To create a label for an expression, statement, or function declaration, use a comment in the following form:

```
/*$SCC label label_name*/
```

The label applies to the operator following the comment. For example,

```
x = a + b /*$SCC label e1*/ *c - d;
```

or

```
x = a + /*$SCC label e1*/ b*c - d;
```

In either of these examples, the label `e1` applies to the operation `b*c` that immediately follows the comment. Scalar operands are ignored during label association and the given label is associated with the next operator.

To label a region of code, use the following syntax:

```
/*$SCC label_begin label_name*/ code_region /*$SCC label_end label_name*/
```

For example,

```
x = a + /*$SCC label_begin e2*/ b*c - d /*$SCC label_end e2*/;
```

The `e2` label applies to both the operations (`b*c`) and (`(b*c) - d`) that are enclosed by start/end label comments.

The region can span multiple lines, as in this example:

```
/*$SCC label_begin e3*/
for (i=0; i < 10; i++) { CODE }
while(1) { CODE }
y[10] = …;
/*$SCC label_end e3*/
```

The `e3` label applies to all statements and expressions enclosed between the start/end label comments.

$SCC Labeling Examples

The following examples further demonstrate the usage of `$SCC` labels.

```
x = a /*$SCC label e4*/+ b*c /*$SCC label e4*/- d;
```

The `e4` label applies to both the operation + and –, but not to *.

```
x = /*$SCC label my_load*/b[i][j] * c + d;
x = b/*$SCC label my_load*/[i][j] * c + d;
```

The `my_load` label applies to the array reference if it is not scalarized, or it is ignored otherwise. To label the `*` operation, the comment must be placed between `b[i][j]` and `*`.

```
/*$SCC label my_store*/x[i] = b;
```

The `my_store` label applies to array store operation if not scalarized, or it is ignored otherwise. To label the assignment operator, the label must be placed between `x[i]` and `= `.

```
x = /*$SCC label my_load*/b.chroma[1] * c + d;
```

The `my_load` label applies to the array reference if it is not scalarized, or it is ignored otherwise. The field selection operator "`.`" cannot be labeled.

```
X = a + /*$SCC label e5*/foo(b*c, y[i]);
X = a + foo/*$SCC label e5*/(b*c, y[i]);
```

The `e5` label applies to the call `foo`, but not to any other expressions.

```
X = a + /*$SCC label_begin e6*/foo(b*c, y[i]) /*$SCC label_end e2*/;
```

The `e6` label applies to the call `foo` as well as the expressions `b*c` and `y[i]`.

```
void foo(int n) {
   #pragma unroll
   for (i=0; i < n; i++) {
      x = a + /*$SCC label e7*/b*c - d;
   }
}
void filter() {
...
   /*$SCC label e8*/foo(10);
...
   /*$SCC label e9*/foo(7);
}
```

Label `e7` applies all 17 multiply operations resulting from inline and unroll.

Label `e8=>e7` applies to the 10 multiply operations resulting from inlining `foo(10)` and unrolling the `i` loop.

Label `e9=>e7[4]` applies to fourth instance of multiply resulting from unrolling loop `i` through call `foo(7)`.

An $SCC label can also be used to build class member functions as hardware by using the label just above the member function declaration or definition. For example,

```
/*$SCC label foo_name*/
void classA ::foo_name( int inArr[10]);
```

For template classes or functions, the label must be applied on a declaration with the correct template parameters set. For example,

```
template void classA<IN_WIDTH,DEF_VAL> {
   void foo () {
       ...
   }
};

/*$SCC label foo_name*/
template void classA<5,0>::foo_name(
     int inArr[10]);
```

In this example, the class A is instantiated with the correct parameters before a label is applied on the function to be built as hardware.

Overloaded Functions

When functions are overloaded, you might want to identify a particular function from the set of overloaded functions to specify a constraint on it. You can do this by setting a unique name with a `$SCC` label. After applying a name to this function, you can use Tcl commands and pragmas with the new name.

For example, you might want to build a particular function from among the set of overloaded functions as a TCAB. You can name the function using a `$SCC` label, and then use this name in the `set_params` command to build the TCAB for the function.

The following example shows an incorrectly specified pragma:

```
// INCORRECT CODING:  Ambiguous "put" in pragmas causes errors
void put(s_fixed<13,4>);
#pragma output_stream put
#pragma fifo_length put 10
void put(int);
#pragma output_stream put
#pragma fifo_length put 5
```

You can correct this condition by naming the stream and then later referring to it by name in constraint pragmas:

```
// CORRECT CODING: $SCC label refers to overloaded versions of "put"
// $SCC label mystream
void put(s_fixed<13,4>);
// These pragmas can be placed anywhere after the $SCC label mystream
#pragma output_stream mystream
#pragma fifo_length mystream 10
```

```
// $SCC label mystreamA
void put(int);
// These pragmas can be placed anywhere after $SCC label mystreamA
#pragma output_stream mystreamA
#pragma fifo_length mystreamA 5
```

Wait Constraints on Unrolled Copies of a TCAB

Consider the following example of a TCAB named `TCAB_T` being imported in a top-level design `PPA()` inside an unrolled loop:

```
void TCAB_T() {
   ...
}

void PPA() {

    #pragma unroll
    for (i=0;i<2;j++){
         ...
         /*$SCC label inst*/ TCAB_T();
         ...
    }
}
```

A `$SCC` label `inst` has been applied on the TCAB call. If you need to schedule the copies of the TCAB (created by unrolling) apart by five cycles, then you need to apply the following implementation parameter in the PPA implementation:

```
set_params -wait "inst\[0\]:inst\[1\]:5:EQ"
```

Note the additional backslash characters used in this syntax, which are required for applying any Tcl-based constraints on the label.

Tcl Constraint Commands

The following table lists the various options of the `set_params` command that you can use to specify user constraints..

PERFORMANCE SETTINGS
`set_params -n`
`set_params -task_ii n`
`set_params -task_overlap always | never | infer`
`set_params -continuous_processing always | never`

PERFORMANCE SETTINGS

```
set_params -wait "lb1[:lb2]:n[:condition][:hard|soft] ..."

set_params -miti n
```

HARDWARE SETTINGS

```
set_params -type "ppa | tcab |channel" -target "implementation_name"

set_params -clock_gating yes | no

set_params -techlib path

set_params -simulator name

set_params -generate_abort_logic yes | no

set_params -init_data_registers yes | no

set_params -reset_data_registers yes | no

set_params -always_enabled yes | no

set_params -edge_clock posedge | negedge

set_params -reset_level high | low

set_params -reset_sensitivity sync | async

set_params -base_address hex

set_params -memory_size "array_arg_name1:size_arg_name1
array_arg_name2:size_arg_name2 ..."

set_params -false_path_handling yes | no

set_params -macrocell_prefix string
```

INTERFACE SETTINGS

```
set_params -interface_type handshake, memory_mapped_handshake

set_params -hostlm_access_latency uniform|minimal
```

INTERFACE SETTINGS *(Continued)*

```
set_params -interface_protocol native | ocp

set_params -disable_ppa_on_host_access_to_memory yes | no

set_params -memory_access_latency num

set_params -host_memory_access  always | infer | never, lm | direct, [dedicated],
readonly | writeonly | readwrite

set_params -host_scalar_access lm | raw

set_params -memory_forward_boundary_register  always | never | infer | datapath

set_params -memory_return_boundary_register  always | never | infer | datapath

set_params -ram_synthesis_threshold words:bits

set_params -internal_blockram_memory_read_write_ports separate | combined |
no_arbitration | separate,no_arbitration | combined,no_arbitration

set_params -user_supplied_memory_read_write_ports separate | combined |
no_arbitration | separate,no_arbitration | combined,no_arbitration

set_params -user_supplied_fpga_memory_read_write_ports separate | combined |
no_arbitration | separate,no_arbitration | combined,no_arbitration

set_params -memory_forward_path_delay
 (delay in ns or  % of clock period)

set_params -memory_return_path_delay
 (delay in ns or  % of clock period)

set_params -top_module_name module_name

set_params -system_port_name "key:signal_name ..."

set_params -system_port_polarity "key:high | low ..."

set_params -system_port_production "key:yes | no ..."

set_params -stream_signal_polarity "key:high | low ..."

set_params -instream_boundary_register  always | infer | never | datapath

set_params -instream_forward_path_delay
 (delay in ns or  % of clock period)
```

INTERFACE SETTINGS *(Continued)*

```
set_params -instream_return_path_delay
 (delay in ns or  % of clock period)

set_params -outstream_boundary_register   always | infer | never | datapath

set_params -outstream_forward_path_delay
 (delay in ns or  % of clock period)

set_params -outstream_return_path_delay
 (delay in ns or  % of clock period)
```

HIERARCHY SETTINGS

```
set_params -force_independent_stalldomain_tcab yes | no

set_params -allow_latency_violation yes | no

set_params -latency_independent_schedule yes | no

set_params always_enabled yes|no
conditional_start yes|no conditional_output yes|no

set_params -tcab_input_registers always | never | infer

set_params -tcab_output_registers always | never| infer

set_params -architectural_pipelinability number

set_params -input_sample_time infer|0 | "name1:infer|0  name2:infer|0 ..."

set_params -output_production_time [build(default) import] number|"name1:num1
 name2:num2 ..."

set_params -scalar_input_delay "nsDelay|"a:nsDelay b:nsDelay ...

set_params -scalar_output_delay <delay in ns>|"name1:<delay in ns> name2:<delay in
ns> ...">

set_params -resource_first_use_time [build(default) import] <  infer|0 |
"name1:infer|0 name2:infer|0 ...">

set_params -input_stall_path_insulation always|never|infer

set_params -output_stall_path_insulation always|never|infer
```

Cycle-Level Scheduling

The cycle-level scheduling feature lets you control scheduling of operations for specific protocol modeling at the interface. You can label the expression or expression block using `$SCC` labels and use implementation parameters to control the absolute scheduling of expressions and relative scheduling of expression blocks. This kind of control is necessary to model specific bus protocols at the interface.

You can implement the following types of scheduling:

- Relative scheduling – Two operations are scheduled to start *n* cycles apart.
- Absolute scheduling – Operations are scheduled to start *n* cycles after the start of the schedule.

To schedule specific operations, you first need to identify the operations by inserting `$SCC` comment labels where the operations are defined in your C/C++ code. Then you use the `set_params -wait` command to specify the absolute or relative scheduling of those operations.

Scheduling the Operations

To specify a scheduling constraint, use the following command syntax:

```
set_params -wait "lb1[:lb2]:n[:condition][:hard|soft] ..."
```

where *lb1* and *lb2* are `$SCC` labels applied on an expression or expression block and *n* is the number of cycles of difference between the two implemented blocks. Multiple scheduling constraints can be specified between the quotation marks.

You can optionally specify the scheduling constraint as `hard` or `soft`; the default is `hard`. A hard constraint triggers an error if the constraint cannot be met, whereas a soft constraint allows scheduling to proceed even when the constraint cannot be met. You can also specify a number-of-cycles comparison condition, as shown in the examples that follow.

For absolute scheduling, specify a single operation label followed by the number of cycles after the start of the schedule for the operation to begin. For example,

```
set_params -wait "opA:2"
```

For relative scheduling, show the first operation label, followed by the second operation label, followed by the number of cycles difference between the operation start times. For example,

```
set_params -wait "opA:opB:2"
```

For co-scheduling, use the relative scheduling syntax with 0 as the number of cycles difference. For example,

```
set_params -wait "opA:opB:0"
```

The number of cycles difference can be specified as a logical condition such as “greater than or equal to 2 cycles later,” as in the following example:

```
set_params -wait "opA:opB:2:GTE"
```

The following qualifying condition labels are supported:

- `EQ`: equal to (default condition)
- `GT`: greater than
- `GTE`: greater than or equal to
- `LT`: less than
- `LTE`: less than or equal to

The following example demonstrates various forms of cycle scheduling. In this example, IP is getting address and data from the bus at every cycle and IP needs to respond to bus at every cycle by sending response data.

```
    void get_req(uint1 &req_type, unsigned long long &haddr) {
        /* $SCC label req_lbl1*/s_req.nb_get(req_buf);
        /* $SCC label resp_lbl1*/s_resp.nb_put(resp_buf);
        haddr    = req_buf.haddr;
        req_type = req_buf.hwrite;
    }
    void get_data(unsigned long long &hwdata) {
        /* $SCC label req_lbl2*/s_req.nb_get(req_buf);
        /* $SCC label resp_lbl2*/s_resp.nb_put(resp_buf);
        hwdata = req_buf.hwdata;
    }
    void send_err() {
        resp_buf.hreadyout = 0;
        resp_buf.hresp = OKAY;
        /* $SCC label er1*/s_resp.nb_put(resp_buf);
        resp_buf.hresp = ERROR;
        /* $SCC label er2*/s_resp.nb_put(resp_buf);
        resp_buf.hreadyout = 1;
        resp_buf.hrdata = 0xBAADF00D;
        /* $SCC label resp_lbl3*/
        /* $SCC label er3*/s_resp.nb_put(resp_buf);
        /* $SCC label req_lbl3*/s_req.nb_get(req_buf);
    }

ppa () {
    /* $SCC label_begin st_a*/
    get_req(rw_req,addr);
```

```
    /* $SCC label_end st_a*/
    if((addr==0x00A)) {
            send_err();
    } else {
            /* $SCC label_begin st_wd*/
            get_data(data);
            /* $SCC label_end st_wd*/
    }
}
```

Here are the scheduling requirements and the corresponding commands for enforcing those requirements:

- In `get_req` and `get_data`, reading from stream `s_req` and writing to `s_resp` must be done in the same cycle:

  ```
  set_params -wait "req_lbl1:resp_lbl1:0"
  set_params -wait "req_lbl2:resp_lbl2:0"
  ```

- `get_data` must be scheduled after one or more cycles after `get_req` execution:

  ```
  set_params -wait "st_a:st_wd:1:GTE"
  ```

- In `send_err`, three successive occurrences of writing to `s_resp` are to be scheduled in three consecutive cycles:

  ```
  set_params -wait "er1:er2:1:soft er2:er3:1:soft"
  ```

By default, the compiler enforces relative scheduling of operations based on possible constraints and fails with an error if the scheduling constraints cannot be satisfied. However, using the `soft` keyword in the constraint makes it a soft constraint; if the constraint cannot be met, the compiler continues and issues a warning that the constraint was not met. You can use the keyword `hard` to explicitly specify a hard constraint (the default behavior).

When you use a co-scheduling constraint between streams, both streams should have similar `boundry_register` specifications. Any differences in these specifications can introduce extra cycles of difference between the streams. When you specify cycles using wait constraints, consider the extra wait cycles. For example, if the streams have a difference of *x* cycles between them due to the boundary register specification, and *n* is the expected cycle difference, use $n + x$ cycles in the wait constraints.

Overview of Pragmas

Pragmas are constraints that you define in the source code. You set them on individual objects, and they override any equivalent global constraints set with the set_params command from within the tool. Some pragmas have no equivalent within the tool and can only be set in the source code.

See the following for additional details:

- Pragma Specification
- Pragmas by Function
- Pragma Descriptions

The rest of the chapter describes the pragmas, which are grouped together by function.

Pragma Specification

You can attach pragmas to various objects, as described:

- Attach pragmas to C or C++ variables, array names, or stream names. This is true for all pragmas unless specifically stated otherwise. One exception is the packed pragma.

  Note that the convention is only a list of names separated by '.' and not an arbitrary C expression.

- You can also specify most pragmas directly on class and struct member variables and on typedefs, without an object. There are some exceptions, like `localize_access` and `packed`; the exceptions are noted in the pragma descriptions. You cannot specify pragmas on array or reference parameters of inline functions and pointers.
- You can also specify pragmas inside a method, using this.<member>.
- Positional pragmas apply to the statement following the pragma, such as unroll and num_iterations.

The following describe some aspects of specifying pragmas:

- If a pragma is applied to an array that gets scalarized, the pragma is automatically propagated to the scalarized elements. In the following example, the tool applies the bitsize pragma to all the elements of array a after scalarization.

```
int a[4];   //Array that gets scalarized because of the
            //usage pattern
#pragma bitsize a 10
```

- Apply the pragma on the array name only, not on the scalarized components. The following pragma is not specified correctly:

```
#pragma bitsize a[0] 10
```

- If more than one pragma applies because multiple pragmas were specified at different hierarchical levels, the instance-specific pragma overrides the more generic specification. Similarly, an instance-specific pragma overrides a more global specification with the set_params command.

- You can use macro names instead of literal constants in pragmas. However, such variables with macro names must expand to literal constants during macro preprocessing. For example, you can specify the bit width of a variable as follows.

```
# define bitsize_x 10
      int x;
#pragma bitsize x bitsize_x
```

- Some pragmas contain an optional parameter proc_list, which is a subset of function names in the application. The Synphony C Compiler tool applies the pragmas when any of the functions in the proc_list is built as a TCAB. The proc_list may contain both actual function names and names assigned with `$SCC` labels. If you do not specify a proc_list for a pragma that can have a proc_list, the tool only applies the pragma when the function that contains the pragma is built as a TCAB. If the function is inlined in another top-level function that is being synthesized, the pragma is ignored.

  In the following example, the function D( ) is called from A and B. The function D( ) contains a pragma specification that includes a proc_list that is function A( ). If D is built as a TCAB, the tool applies the pragma inside the function D( ). If D is inlined into A( ) and B( ) and these functions are built as TCABs, the pragma specification is applied only when A is built as a TCAB and not B. This is because the proc_list of the pragma inside D contains only A. If no proc_list is specified and D( ) is inlined into other TCABs, the pragma is ignored.

```
A() {
      D(); // Call to D
}
B() {
      D(); // Call to D
}
D() {
      #pragma pragmaName argument "A"
}
```

Pragmas by Function

The following table provides a quick reference to the pragmas supported by the Synphony C Compiler tool, grouped together by function:

Host Access

The tool includes various pragmas that define host access:

stable Pragma	Specifies that the data be consumed directly from the input port, without using a register
hold_register Pragma	Creates a register to hold constant input.

host_memory_access Pragma	Defines host access for memories.
host_scalar_access Pragma	Defines host access for scalar inputs and outputs.
not_input, not_output Pragmas	Specifies that global variables are not inputs or outputs.

Array

The following table lists the pragmas you can specify for arrays. Additional pragmas, that apply to both arrays and streams, are described in "Arrays and Streams" on page 7-17.

dont_care_code Pragma	Specifies ROM values to be treated as don't cares.
force_constant_addressing Pragma	Converts non-constant address to constant addresses.
internal_blockram Pragma	Maps an array to block RAM (FPGA only).
internal_fast Pragma	Maps an array to register-based memory.
localize_access Pragma	Controls access to static and global variables.
memory_structure Pragma	Specifies desired level of banking of local arrays, global arrays, and array parameters.
memory_access_latency Pragma	Specifies the intrinsic latency of the memory to which the array is mapped.
multi_buffer Pragma	Specifies the number of buffers to create for multibuffering.
no_inter_loop_memory_analysis Pragma	Allows loop nests that share a memory to operate in parallel.
no_inter_task_memory_analysis Pragma	Allows tasks that share a memory to operate in parallel.
no_dependence Pragma	Constrains the allowed distance between memory accesses.
no_inter_tcab_subtask_memory_analysis Pragma	Allows pipelining of subtasks on the TCAB.

preserve_array Pragma	Prevents automatic scalarization and banking of an array.
read_write_ports Pragma	Specifies memory ports for user-supplied memories.
rom Pragma	Specifies that the array be implemented as ROM.
scalarize Pragma	Dissolves an array and maps it to wires and registers.
user_supplied, user_supplied_fpga Pragmas	Implements the specified array outside the PPA in ASIC and FPGA designs.

Stream

The following table lists the pragmas you can specify for streams. Additional pragmas, that apply to both arrays and streams, are described in "Arrays and Streams" on page 7-17.

bus_name Pragma	Specifies a bus name for related streams (FPGA only).
bus_standard Pragma	Specifies the bus standard (FPGA only).
bus_type Pragma	Specifies the bus type (FPGA only).
data_name Pragma	Specify bus signals (FPGA only). See data_name, ready_name, request_name Pragmas.
fifo_length Pragma	Defines the number of FIFO entries
no_inter_loop_stream_analysis Pragma	Allows loop nests that share a stream to operate in parallel.
no_inter_task_stream_analysis Pragma	Allow tasks that share a stream to operate in parallel.
ready_name Pragma	Specify bus signals (FPGA only). See data_name, ready_name, request_name Pragmas.
request_name Pragma	Specify bus signals (FPGA only). See data_name, ready_name, request_name Pragmas.
signal_polarity Pragma	Specifies the the signal polarity of a stream control port to either active-high or active-low.

signal_production Pragma	Specifies whether to produce particular stream interface control signal in hardware.

Arrays and Streams

There are some pragmas that can be used with both arrays and streams. These common pragmas are listed here:

boundary_register Pragma	Specifies the creation of boundary registers for streams and memories. See "boundary_register and Other Boundary Register Pragmas" on page 7-20.
forward_boundary_register Pragma	Specifies the creation of boundary registers for streams and memories. See "boundary_register and Other Boundary Register Pragmas" on page 7-20.
forward_path_external_delay Pragma	Specifies external delays. See "forward_path_external_delay and Other External Delay Pragmas" on page 7-29
return_boundary_register Pragma	Specifies the creation of boundary registers for streams and memories. See "boundary_register and Other Boundary Register Pragmas" on page 7-20.
return_path_external_delay Pragma	Specifies external delays. See "forward_path_external_delay and Other External Delay Pragmas" on page 7-29

Arrays, Streams, and Scalars

The packed pragma can be used with arrays, streams, and scalars.

packed Pragma	For packing and unpacking arrays, packing scalar data at the PPA interface, and unpacking stream data at the PPA interface.

Miscellaneous

This section lists the miscellaneous pragmas:

bitsize Pragma	Sets the bitwidth of a variable.
flush Pragma	Specifies that the pipeline be flushed
ii Pragma	Defines initiation interval.
max_schedule_length Pragma	Specifies the maximum schedule length.
num_iterations Pragma	Specifies the number of times a loop executes.
signal_name_prefix, signal_name_identifier, and signal_name_suffix Pragmas	Specifies the names of stream, scalar, and memory ports on a PPA interface.
unroll Pragma	Unrolls loops.

Pragma Descriptions

The pragmas are described in alphabetical order. Some pragmas are grouped and described together.

- bitsize Pragma
- boundary_register Pragma (boundary_register and Other Boundary Register Pragmas.)
- bus_name Pragma
- bus_standard Pragma
- bus_type Pragma
- stable Pragma
- data_name Pragma (data_name, ready_name, request_name Pragmas)
- dont_care_code Pragma
- fifo_length Pragma
- flush Pragma
- force_constant_addressing Pragma
- forward_boundary_register Pragma (boundary_register and Other Boundary Register Pragmas)

- forward_path_external_delay Pragma (forward_path_external_delay and Other External Delay Pragmas)
- hold_register Pragma
- host_memory_access Pragma
- host_scalar_access Pragma
- ii Pragma
- internal_blockram Pragma
- internal_fast Pragma
- localize_access Pragma
- max_schedule_length Pragma
- memory_structure Pragma
- memory_access_latency Pragma
- multi_buffer Pragma
- no_dependence Pragma
- no_inter_loop_memory_analysis Pragma
- no_inter_loop_stream_analysis Pragma
- no_inter_task_memory_analysis Pragma
- no_inter_task_stream_analysis Pragma
- no_inter_tcab_subtask_memory_analysis Pragma
- not_input, not_output Pragmas
- num_iterations Pragma
- packed Pragma
- preserve_array Pragma
- read_write_ports Pragma
- ready_name Pragma (data_name, ready_name, request_name Pragmas)
- request_name Pragma (data_name, ready_name, request_name Pragmas)
- return_boundary_register Pragma (boundary_register and Other Boundary Register Pragmas)

- return_path_external_delay Pragma (forward_path_external_delay and Other External Delay Pragmas)
- rom Pragma
- signal_name_prefix, signal_name_identifier, and signal_name_suffix Pragmas
- signal_polarity Pragma
- signal_production Pragma
- scalarize Pragma
- unroll Pragma
- user_supplied, user_supplied_fpga Pragmas

bitsize Pragma

Miscellaneous

This pragma sets the bitwidth of a variable in the source code. This is the syntax for using this pragma:

```
#pragma bitsize variableName width
```

The variable can be an array name, stream procedure name, scalar, typedef, or field of a struct or class. The tool dynamically checks the bitsizes for possible overflows during lint simulation. It is recommended that you set explicit bit sizes for interfaces (memories, streams, input scalars, output scalars), state variables (variables that maintain state across iterations), and the inputs and outputs of any DSP API. The Synphony C Compiler tool uses this specification to optimize bitwidths of registers and functional units such as multipliers, adders, and so on.

boundary_register and Other Boundary Register Pragmas

Arrays and Streams

Use these pragmas to specify boundary registers to provide timing insulation at the PPA boundary. By default, the tool does not insert any boundary registers for either streams or memories. You can specify boundary registers for external streams, user-supplied memories, and block rams in FPGA designs.

This is the syntax:

```
#pragma boundary_register streamProcName always|infer|never|datapath
#pragma forward_boundary_register arrayName always|infer|never|datapath
#pragma return_boundary_register arrayName always|infer|never|datapath
```

The options are described in the following table:

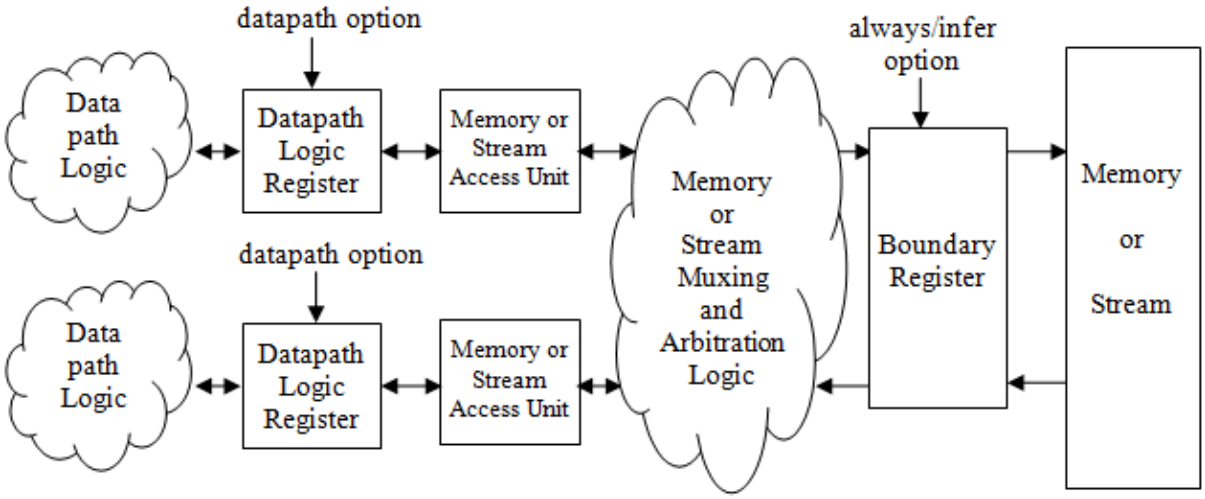
	Datapath Logic Registers	Boundary Registers	Comments
never	No	No	
always	No	Yes	
datapath	Yes	No	Only datapath logic is registered. Memory or stream muxing and arbitration logic is not registered.
infer	No	Based on external delays	

See the following for additional information:

- "Boundary Register Specification for Streams" on page 7-22
- "Boundary Register Specification for Memories" on page 7-23

You can insert boundary registers with an implementation parameter or a pragma. Pragmas override the Synphony C Compiler defaults and any implementation-wide settings specified with the `set_params` command. You can specify this pragma on a typedef, or members of a class or struct.

Source mapping is not available for boundary registers introduced in the control path. The hardware that gets generated for memory or stream operations is shown in the following figure.



Boundary Register Specification for Streams

External streams can have a FIFO of certain length at the RTL boundary. The boundary FIFO might have passthru paths, or provide insulation between the RTL internals and external hardware. The Synphony C Compiler tool provides controls for both insulation and length. A FIFO created for insulation has a default length of 2 and cannot have zero length. A noninsulating FIFO can have length >=0.

For external streams, the insulation property can be controlled using the following values in the implementation parameter or pragma for the boundary register:

- `always`: creates an insulating FIFO at the boundary of the RTL.
- `never`: creates a passthru FIFO at the boundary of the RTL.
- `infer`: lets the tool decide about the insulation based on timing analysis.

Both external and internal streams can be specified to have boundary registers of type datapath. This creates a boundary register in the datapath of PAs, providing good but not perfect insulation. For internal streams, the datapath specification applies to both ends; there is no control to apply it only to one end (for example, the output side).

The parameters and pragmas for controlling stream boundary registers are:

```
set_params -instream_boundary_register
     always|infer|never|datapath
Applies to external input streams.
```

```
set_params -outstream_boundary_register
     always|infer|never|datapath

Applies to external output streams.

#pragma boundary_register streamProcName always|infer|never|datapath
Applies to both external and internal streams. streamProcName must be the
entire stream name.

#pragma fifo_length streamProcName fifo_length
Applies to both external and internal streams. streamProcName must be the
entire stream name.
```

Boundary FIFOs of nonzero size have some impact on simulation or interaction with external hardware. Input boundary FIFOs prefetch the data and therefore could make more requests than the actual RTL. The external stream hardware must tolerate *n* additional requests where *n* is the length of the boundary FIFO. On the output side, RTL signals completion after it finishes writing to the output boundary FIFO (if any). External hardware must drain any residual data in the output boundary FIFO, even after the task completion.

Boundary Register Specification for Memories

You can specify boundary registers for forward and return paths on memories using these implementation parameters:

```
set_params -memory_forward_boundary_register
   always|infer|never|datapath
set_params -memory_return_boundary_register
   always|infer|never|datapath
```

To override this global specification, use the following pragmas on a per memory basis:

```
#pragma forward_boundary_register arrayName
   always|infer|never|datapath
#pragma return_boundary_register arrayName
   always|infer|never|datapath
```

A forward or return boundary register created out of always or infer settings completely insulates the forward or return paths in the PPA from the external driver. A boundary register created out of a datapath setting creates the boundary register on the data path as shown in the previous figure. The forward path signals include memory request, data to memory, memory address and memory mode. The return path signals include data from memory and the memory acknowledgement.

If you need perfect insulation, use the always option> Note however that this increases the latency of the memory accesses, which in turn increases the register cost because of additional replay buffers. The datapath option gives good but not perfect insulation. However, this reduces the replay buffer cost.

bus_name Pragma

Stream

In FPGA designs, this pragma specifies the bus name that is used for communication. Some bus standards are best modeled as a combination of multiple streams, and giving all the related streams the same bus name allows a back end tool like Xilinx EDK to connect them. This is the pragma syntax:

```
#pragma bus_name streamName busName
```

You cannot specify this pragma on members of a class or struct. If you do not specify a name for the bus, the default name is the name of the stream without the `scc_stream_input` or `scc_stream_output` prefix.

bus_standard Pragma

Stream

In FPGA designs, you can specify the bus standard that the stream uses for communication with this pragma:

```
#pragma bus_standard streamName busStandardName | none
```

You cannot specify this pragma on members of a class or struct.

The following example infers that scc_stream_output_A is available for integration using a bus standard called STANDARDX:

```
#pragma bus_standard scc_stream_output_A STANDARDX
```

You can specify none as a bus standard. If you do this, the stream is not b bound to any bus protocol and the signals of the stream appear as separate ports and are visible as independent entities to a third-party integration tool. You can manually hook up the port connections for this particular stream as required. This is especially useful in cases where some behavior of an interface can be exploited to reduce the amount of hardware and the number of interconnects. For example, if data is always available, the ready signal can be tied to its active level. This cannot be done when if the stream is associated with a bus standard (where all the ports are required to be connected to a particular signal of the other protocol).

Default Behavior

If you do not specify a bus standard, and you use a struct stream, the name of the standard is the name of the struct. You must name the struct. While using a typedef, it is required to name the struct as well.

In the following code, the struct name XIL_LL_STREAM is used as the bus standard name for any struct stream of this type:

```
typedef struct XIL_LL_STREAM {
unsigned int LL_Data;
unsigned int LL_SOF_n;
unsigned int LL_EOF_n;
} XYZ;
```

If you do not specify a bus standard, and have a base C datatype stream, the name of the standard will be SCC_STREAM<datawidth>, where datawidth is the width of the data port of the stream (1...64).

bus_type Pragma

Stream

In FPGA designs, specify the bus type used for integration with the following pragma.

```
#pragma bus_type streamName busTypeName
```

You cannot specify this pragma on members of a class or struct. If you do not specify a bus type, then by default, the bus_type is TARGET for input streams and INITIATOR for output streams.

This is an example of using the pragma:

```
#pragma bus_type scc_stream_output_sample SLAVE
```

stable Pragma

Host Access

By default, the Synphony C Compiler tool registers every input into the PPA or TCAB. The following command avoids generating input register in a TCAB:

```
set_params -tcab_input_registers never
```

However, this only implies that the first consumer of the input should consume directly from the input port rather than a register. The syntax to directly use the values at ports for all variables in TCAB is as follows:

```
#pragma stable <variable name> [<proc_list>]
```

An input marked as `stable` to a TCAB procedure should have the same value for pipelined calls to the TCAB. This is possible when container of the TCAB either is passing a constant value or is passing an input of the container. Further, the TCAB should not modify a `stable` input.

Effectively, the `stable` pragma tells the Synphony C Compiler tool that it need not create new registers and instead can resample the input ports at anytime during the schedule. This can save substantial area in some cases, for example:

```
int x,y,global;
int  tcab_fn(int a,int b)
{
      #pragma stable a
      #pragma stable b
      int temp = pico_asm_MPY2C_W(a,b);
      temp = pico_asm_MPY2C_W(a,temp);
      return temp;
}
void ppa(){
{
   int j;
      for (j = 0; j < M; j++) {
         global += tcab_fn(x,y);
   }
}
```

In this example, no input registers are created for inputs a and b within the TCAB. The values passed to these input ports in the PPA, namely, x and y, are constant across all pipelined calls to tcab_fn.

You can specify this pragma on a typedef, or members of a class or struct.

data_name, ready_name, request_name Pragmas

Stream

In FPGA designs, these pragmas specify the bus signals of the protocol to which data, request and ready signals of the stream interface should bind. These pragmas are applicable only when the implementation is synthesized as a PPA.

```
#pragma data_name streamName busDataName
#pragma request_name streamName busRequestName
#pragma ready_name streamName busReadyName
```

You cannot specify these pragmas on members of a class or struct. If you do not specify these pragmas, the default values are DATA, SEND and ACK respectively for output streams and DATA, ACK, and SEND respectively for input streams.

The information specified in the pragmas does not directly affect either the PPA or the wrapper created for the integration, but conveys information about the names of the data, request and ready signals used in the stream. When the PPA is integrated into EDK environment, this information is placed in the implementation/rtl_package/pcores/PPA_NAME_version/data/PPA_NAME_v2_1_0.mpd file.

dont_care_code Pragma

Array

This pragma specifies that some values in the ROM can be treated as don't cares. This is the syntax for the pragma:

```
#pragma dont_care_code value
```

Example

In the following example, locations F[0][0] and F[1][1] are treated as don't care values:

```
#pragma dont_care_code 99

const unsigned char F[10][10] = {{99, 20,}, {0, 99, 21},};
```

fifo_length Pragma

Stream

For each internal and external stream in the code, the Synphony C Compiler tool implements a FIFO in the RTL. You can specify the number of entries in your FIFOs using the fifo_length pragma on the stream procedure name. The scc_stream default FIFO length is 0 if the fifo_length pragma is not specified.

This is the syntax for using this pragma:

```
#pragma fifo_length stream_procedure_name fifo_length
```

You can specify this pragma on objects of the scc_stream class. Use <object>:input for read ports and <object>:output for write ports.

Examples

The following show how to specify FIFO length using scc_stream class and FIFO:

```
scc_stream<int> y;
#pragma fifo_length y 10
FIFO(x, int);

#pragma fifo_length x.get() 10
```

flush Pragma

Miscellaneous

To achieve high performance, the Synphony C Compiler tool builds a pipeline of operations. At any given cycle, there could be many loop iterations operating. Each pipeline is in its own stall domain and any stall condition causes the whole pipeline to stall. Constraints on the external environment often make it necessary to flush the pipeline of old iterations before issuing an operation that might cause a stall.

This pragma lets you flush the pipeline:

```
#pragma flush
```

For more information on pipeline flushing, see "Pipeline Flushing" on page 8-40. An alternative approach to pipeline flushing that might be better for continuous processing designs is to use a non-blocking stream interface, as described in "Stream Support" on page 8-24.

force_constant_addressing Pragma

Array

This pragma converts a reference with a non-constant address into a reference with a constant address. Use it to force constant addressing on an array. This is the syntax:

```
#pragma force_constant_addressing arrayName [all | none]
```

Constant addressing is implemented in one of three modes (infer, forced constant addressing, no forced addressing), depending on the pragma specification:

Specification	Implementation
No pragma specified	Infer mode. The tool uses heuristics to infer whether to convert to a constant reference.
Pragma specified with none	The tool leaves the addressing as is.
Pragma specified without all or none Pragma specified with all	The default is all. The tool forces constant addressing on the specified array.

Here are some examples of the syntax:

```
int X[8][64][4];
#pragma force_constant_addressing X
```

```
int Y[8][64][4];
#pragma force_constant_addressing Y none
```

forward_path_external_delay and Other External Delay Pragmas

Arrays and Streams

By default, the Synphony C Compiler tool assumes an external delay of 0% on forward and return paths for memory and stream ports. You can specify external delay on a per-array or per-stream basis with the following pragmas on arrays or streams. Specify the delay in nanoseconds.

```
#pragma forward_path_external_delay array | stream delay
#pragma return_path_external_delay array | stream delay
```

You can specify this pragma on a typedef, or members of a class or struct.

This pragma specification overrides the Synphony C Compiler default and any implementation-wide settings. You can set the implementation-wide delays with the following commands.

Memories	set_params -memory_forward_path_delay set_params -memory_return_path_delay
Input streams	set_params -instream_forward_path_delay set_params -instream_return_path_delay
Output streams	set_params -outstream_forward_path_delay set_params -outstream_return_path_delay

hold_register Pragma

Host Access

You can use the `hold_register` pragma to hold data stable on a per-task scalar input or output, a `stable` scalar input, an output stream, or an array mapped to a user_supplied memory. You can apply it to a typedef or members of a class or struct.

This is the pragma syntax:

- Input scalar

  `#pragma hold_register` *`scalar`* `single_buffer`

  For information about the effect of hold registers on scalar inputs, see "Scalar Interface" on page 14-8.

- Output scalar

  ```
  #pragma hold_register scalar single_buffer
  ```

  For information about the effect of hold registers on scalar outputs, see "Scalar Interface" on page 14-8.

- Output stream

  ```
  #pragma hold_register stream single_buffer
  ```

  The output data remains stable after a successful handshake and until the next one, and is persistent across tasks.

- Array

  ```
  #pragma hold_register array single_buffer
  ```

  This pragma applies only to external SRAMs, and not to internal_fast and BRAM memories. By default, the output data, address, and mode (read/write bit) are not guaranteed to be stable after the store transaction. However, when you use this pragma, after a memory store transaction on a port, the output data remains stable until the next store transaction on the same port. The output address and output mode remain stable until the next load or store transaction is issued on the port. If the memory has forward boundary registers, then they hold the data stable. Otherwise, the tool inserts separate hold registers to ensure data stability.

host_memory_access Pragma

Host Access

This pragma controls the nature of host access on a per-memory basis. The syntax of this pragma is shown:

```
#pragma host_memory_access arrayName none | lm | direct,

[dedicated],[readonly | writeonly | readwrite]
```

none	Specifies that the host has no access to the memory.
lm	Specifies memory-mapped access. This can only be readwrite. You cannot specify memory mapped (hostlm) access for struct memories.
direct	Specifies that the host has direct access to the memory using raw ports, with one raw port per memory.

dedicated	Specifies that the host has dedicated access to the memory, regardless of whether the access type is memory-mapped or through direct ports.
readonly | writeonly | readwrite	Specifies the kind of access through direct ports. Does not apply for memory mapped access.

You can specify this pragma on a typedef, or members of a class or struct.

You can also use `set_params -host_memory_access` command to specify global host access settings.

host_scalar_access Pragma

Host Access

This pragma specifies the type of host access for a scalar input or output variable. If a variable is both input and output, the pragma applies to both write and read access:

```
#pragma host_scalar_access <variable> <lm | raw>
```

lm	Host can access the input or output variable using memory_mapped (hostLM) port.
raw	Host can access the input or output variable using raw ports.

You can specify this pragma on a typedef, or members of a class or struct.

Example

```
int x;
#pragma host_scalar_access x raw
```

To globally map all scalar inputs and outputs to memory or raw ports, use the following command within the tool:

```
set_params -host_scalar_access <lm | raw>
```

ii Pragma

Miscellaneous

This pragma sets the II value of a loop nest in the source code.

```
#pragma ii value
```

The following are the rules for applying this pragma:

- The pragma must appear before a loop.
- The pragma can be attached to outermost loop and the loops nested within the outer loop. In this case, the inner loop II must match the outer loop II.

For a loop nest, this pragma specification overrides the default II specified by the `set_loop_params -ii` Tcl command.

Examples

The following example shows three loops, each with different II values.

```
void ppa( ) {
      int i, j, k;
      for(i=0; i < 100; i++) { . . . }
         #pragma ii 4
      for(i=0; i < 100; i++) { . . . }
         #pragma ii 1
      for (j=0; j < 400; j++) { . . .}
         #pragma ii 2
      for(k=0; k < 200; k++) { . . . }
}
```

The next example shows II specified on an inner loop. As there is no outer loop II, the loop is set to II=1.

```
void ppa( ) {
      int i, j;
      for(i=0; i < 4; i++) {
         #pragma ii 1
      for (j=0; j < 100; j++) { . . .}
      }
}
```

---

internal_blockram Pragma

Array

This pragma is only for FPGA designs. It maps an array to block RAM. When applied to streams, the stream storage maps to block RAM. The syntax of this pragma is as shown:

```
#pragma internal_blockram array | stream
```

You can specify this pragma on a typedef, or members of a class or struct.

The tool ignores this pragma if the array is scalarized. You can prevent the array from being scalarized with #pragma scalarize *arrayName* no, which is described in "scalarize Pragma" on page 7-68.

internal_fast Pragma

Array

The internal_fast pragma causes an array to be mapped to a register-based memory. Unlike accesses to user_supplied memory or block RAM, a read or write to a constant address does not require a memory port and directly accesses the corresponding register. When applied to streams in FPGA designs, the stream storage maps to SRLs.

The syntax of this pragma is shown:

```
#pragma internal_fast array | stream
```

You can specify this pragma on a typedef, or members of a class or struct.

The tool ignores this pragma if the array is scalarized. You can prevent the array from being scalarized with #pragma scalarize *arrayName* no, which is described in "scalarize Pragma" on page 7-68.

localize_access Pragma

Array

The `localize_access` pragma controls the accessibility of static and global variables. It tells the Synphony C Compiler tool that the variable is never accessed (even indirectly) outside the call tree of a given procedure. This allows the Synphony C Compiler tool to reduce or eliminate hardware that would otherwise be required to access the variable.

The syntax of the pragma is shown:

```
#pragma localize_access x yes [function] | no
```

`no`	Provides access to all functions, including driver code.
`yes`	Restricts access to the given procedure.
`function`	Only applies for `yes`, and specifies that only the function and functions in its call tree can access the state. *`function`* must be a TCAB or a PPA; it cannot be an inlined function.

To localize the access of an array to a specific procedure, use the following syntax:

```
#pragma localize_access array yes procedure_name
```

To make access of an array global, use the following syntax:

```
#pragma localize_access array no
```

Note:

- Automatic inference of banking and scalarization based on constant indexes will not happen on global or class members variables even if they are localized using `localize_access` pragma.
- Automatic multibuffering (using `multi_buffer` pragma) on global variables or class member variables is not supported even if they are localized using `localize_access` pragma.

In C/C++, static variables declared in procedure scope cannot be accessed outside the procedure. However, the procedure containing the static variable has few restrictions on callers. When synthesizing hardware that contains a static variable, the Synphony C Compiler tool must decide whether or not the static variable can be accessed indirectly from outside the synthesized block by a call to the procedure containing the static variable from outside the synthesized procedure.

For example, if a procedure containing a static is synthesized and also called from driver code, then the synthesized hardware needs host access to the static state.

In the following code, the scope of static variable x is restricted within the function foo. The function foo is inlined inside tcab_A as well as the PPA. However, inlining is not possible unless the static state is shared across the TCAB and the PPA.

```
foo()
{
     static int x;
     …
}
tcab_A()
{
     foo()
     …
}
ppa()
{
     …
tcab_A();
     foo();
     …
}
```

In this case, you can use the `localize_access` pragma to localize the scope of the variable x to the PPA scope. That is, the static variable is visible to the scope of the indicated function:

```
#pragma localize_access x yes ppa
```

Using the localize_access Pragma

Some common situations where you can use the `localize_access` pragma are:

- Localize global variables that should not be exposed to the host to the PPA or TCAB where they are used.
- Make static variables in a TCAB procedure scope visible to the PPA with `localize_access no`, or `localize_access yes` [PPA procedureName]. This offers the following advantages:
  - The PPA will now contain a copy of the static variable, so it can optimize the pipelinability of the TCAB and reduce II.
  - Making all static variables of a TCAB visible to the container allows the Synphony C Compiler tool to create multiple instances of the TCAB to improve performance. Otherwise, it can only create a single copy of the TCAB, because it cannot duplicate the state of the static variable.
- Use this pragma to share a function with a static variable that is inlined in both the PPA and the TCAB between the TCAB and PPA.
- A class member variable can only be localized to a class member function that is synthesized as a PPA or a TCAB. Constructors of such a class are treated as part of the testbench code. Before using the member variables, you must explicitly initialize them inside the function where the class member is localized, to generate correct hardware.
- In the application, instantiate only one object of a class that has `localize_access` specified on the data members.

---

max_schedule_length Pragma

Miscellaneous

This pragma specifies the maximum number of cycles in which the operations of a loop nest can be scheduled. For example, if this parameter is set to 4, the tool schedules the operations of a PA in 4 or fewer cycles.

```
#pragma max_schedule_length n
```

This pragma must be specified before a loop nest that becomes a PA. For example,

```
void ppa( ) {
      int i, j;
      #pragma max_schedule_length 4
      for(i=0; i < 100; i++) { . . . }
```

```
        for (j=0; j < 400; j++) { . . .}
}
```

In other words, the pragma cannot be applied inside an inner loop of an outer loop that becomes a PA.

The tool tries to schedule the operations within the specified schedule length. If this constraint cannot be met, even after relaxing any other soft constraints, the tool issues an error message during synthesize phase, relaxes the constraint, and generates RTL.

memory_structure Pragma

Array

This pragma specifies the desired level of banking of local arrays, global arrays, and array parameters.

```
#pragma memory_structure arrayName bank[, bank[, bank]...]
```

The number of instances of the keyword `bank` specifies the number of levels of banking to be used for banking the specified array, starting with the outermost index. For example,

```
int x[2][2][4];
#pragma memory_structure x bank, bank
```

This pragma invokes first-level and second-level banking, producing four banks: `x_0_0[4]`, `x_0_1[4]`, `x_1_0[4]`, and `x_1_1[4]`.

Note:

Using the `memory_structure` pragma can sometimes create situations in which different banks can be passed to different TCABs, which is not allowed.

memory_access_latency Pragma

Array

This pragma specifies the intrinsic latency of the memory to which the array is mapped. Latency has to be greater than or equal to1 and the memory is assumed to be pipelinable. Any specification of boundary registers increases the latency over and above this specification.

```
#pragma memory_access_latency arrayName latency_value
```

You can specify this pragma on a typedef, or members of a class or struct.

The pragma specification overrides the global memory_access_latency implementation parameter.

multi_buffer Pragma

Array

The Synphony C Compiler tool automatically performs multi-buffering, but you can use the following pragma to constrain it. If you specify the pragma and multi-buffering is not possible, the pragma is ignored.

```
#pragma multi_buffer arrayName numBuffer
```

You can specify this pragma on a typedef, or members of a class or struct.

Multi-buffering enables greater task overlap in certain applications. For more information on multi-buffering, see "Banking and Multi-Buffering of Memories" on page 8-15. The Synphony C Compiler tool automatically generates the multiple buffers and the corresponding addressing logic that selects the correct buffer in each task. Multi-buffering incurs additional cost, so the Synphony C Compiler tries to minimize buffers for the given performance constraint.

Do not use this pragma for direct duplication of ROMs. The tool does direct duplication automatically, based on the number of simultaneous accesses to the ROM. If you specify the number of buffers for ROMs to be greater than one, the Synphony C Compiler tool resets the number of buffers to one with a user warning.

no_dependence Pragma

Array

Synphony C Compiler analyzes the array addresses and conditions under which the references are present to determine the pipelined iteration separation of aliasing references. This separation distance information provides flexibility to the compiler during pipelining to schedule operations out of order without violating the original program dependences.

However, the compiler analysis may be conservative under certain situations. You can provide additional guidance to the compiler by using the following pragma:

```
#pragma no_dependence array
  [source-access:[source-region]]
  [sink-access:[sink-region]]
  [distance]
```

where *source-access* and *sink-access* are each the string `read` or `write`, *source-region* and *sink-region* are each a memory region, and *distance* is an integer.

The following examples demonstrate usage of the pragma.

- In its simplest form, the pragma specifies only an array name:

  ```
  #pragma no_dependence X
  ```

In this example, none of the reads and writes of array X alias with each other, so the tool has full flexibility in scheduling read and write events.

- The pragma can optionally specify the distance interval as an integer:

  ```
  #pragma no_dependence X 3
  ```

  In this example, any read or write of array X does not alias with the next three reads or writes of X.

- The types of source access, either read or write, can also be specified:

  ```
  #pragma no_dependence X write read 3
  ```

  In this example, any write of X does not alias with the next three reads of X. Similar dependences maybe specified for read-to-write pair as well as write-to-write pair.

- You can optionally specify an event by its `$SCC` comment label:

  ```
  #pragma no_dependence X read:e1 write:e2 3
  ```

  Reads of X labeled with e1 do not alias with next three writes labeled with e2. For information about labeling, see "$SCC Labeling" on page 7-3.

The dependence distance (the value 3 in the foregoing examples) is in terms of the next *n* references of the array in a completely unrolled view. The compiler maps this dependence distance into the corresponding pipelined iteration separation in a manner that depends on how this code was mapped to iterations of the pipelined loop.

For example,

```
int X[N];
#pragma no_dependence X 3

X[i1] = … // W1
for (k=0; k<2; k++) {
   X[i2] = … // W2
}
… = X[i3] // R
```

In the following examples, to show the relationship between user-specified distances and the compiler translation of them into its internal representation, the following notation is used:

- Each row of text represents an "iteration" with respect to an imaginary outer loop around the pair of references in question.
- Aliasing references are highlighted in bold.
- If the inner loop was unrolled each row would map to the iterations of the pipelined loop and if not each row maps to multiple iterations of the pipelined loop shown in alternating colors

`no_dependence 3` for W1 to R means a pipelined iteration separation of 3 if the inner loop is unrolled or 7 if not unrolled:

```
iteration 0:  W1 W2 W2 R(1)
iteration 1:  W1 W2 W2 R(2)
iteration 2:  W1 W2 W2 R(3)
iteration 3:  W1 W2 W2 R(4)
```

`no_ dependence 3` on W2 to R means a pipelined iteration separation of 3 if the inner loop is unrolled or 6 if not unrolled:

```
iteration 0:  W1 W2 W2 R(1)
iteration 1:  W1 W2 W2 R(2)
iteration 2:  W1 W2 W2 R(3)
iteration 3:  W1 W2 W2 R(4)
```

In the representation, aliasing references are highlighted in bold and “iteration” in the first column is shown with respect to an imaginary outer loop around the pair of references in question. If the inner loop is unrolled, each row maps to the iterations of the pipelined loop. Otherwise, each row maps to two iterations of the pipelined loop shown in alternating colors.

For simpler counting, the tool assumes that conditions around the reads and writes do not matter (they are counted as if both arms were executed), and any variable-bound loop executes exactly the minimum number of iterations specified by `#pragma num_iterations` for such loops.

For example,

```
for(i=0;i<m;i++) {
   X[] <- W
   #pragma num_iterations(3,,5)
   for(j=0;j<n;j++) {
      R <- X[]
   }
}
```

If you want to specify that the `W in i=k` cannot alias with `R in i=k, i=k+1`, but can alias with `i=k+2` or more:

```
iteration 0: W R R R    // i=k
iteration 1: W R R R    // i=k+1
iteration 2: W R R R    // i=k+2
```

then you can specify the `no_dependence` distance spec as 6.

While doing hierarchical synthesis, Synphony C Compiler labels provide the kind of control needed to differentiate references in the container from the references done inside the TCABs. A function call is counted as one operation in the caller. Any specification given using an `$SCC` label treats the call as a single unit. It is independent of how many times the array is read or written inside the function or whether there was a loop inside the function and whether the function was inlined.

For example, in a typical double buffering case, to specify that all reads and writes to a given memory passed to TCABs do not alias in the same iteration of the container, the following specification can be used:

```
type mem[2][SIZE];
#pragma no_dependence mem write:L1 read:L1 1

/*$SCC label_begin L1*/
for(k=0;k<n;k++) {
   tcab1(mem,k&1);       // writer
   tcab2(mem,1-(k&1));   // reader
}
/*$SCC label_end L1*/

iteration 0: L1 tcab1 tcab2
iteration 1: L1 tcab1 tcab2
```

Alternatively, to specify that reads of the same iterations alias but writes in the next iterations do not alias with reads of the current iteration, you can break the read-to-write dependence for one iteration using the following:

```
type mem[2][SIZE];
#pragma no_dependence mem read:L1 write:L1 1

/*$SCC label_begin L1*/
for(k=0;k<n;k++) {
   tcab1(mem,k&1);       // writer
   tcab2(mem,k&1);       // reader
}
/*$SCC label_end L1*/

iteration 0: L1 tcab1 tcab2
iteration 1: L1 tcab1 tcab2
iteration 1: L1 tcab1 tcab2
```

Breaking False Dependencies

The following example shows how to break false dependencies. It is an FFT implementation with the following code structure:

```
#pragma ii 2
#pragma num_iterations(1,8,8)
  for (i = 0; i < m; i++) {
    n1 = n2;
    n2 = n2 + n2;
    n3 = n3 / 2;
#pragma num_iterations(1,1,128)
    for (j = 0; j < n1; j++) {
      s_complex<fft_scc_table_data> r = fix[j * n3];
#pragma num_iterations(1,128,128)
       for (k = j; k < n; k = k + n2) {
        s_complex<fft_scc_data> tmpz = z[k];
```

```
        t.x = r.x * z[k + n1].x - r.y * z[k + n1].y;
        t.y = r.y * z[k + n1].x + r.x * z[k + n1].y;

        z[k + n1].x = tmpz.x - t.x;
        z[k + n1].y = tmpz.y - t.y;

        z[k].x = tmpz.x + t.x;
        z[k].y = tmpz.y + t.y;

      }
    }
  }
```

This design does not achieve the intended II=2 performance because the tool is unable to prove that writes to z[k+n1] do not alias with writes to z[k].

To break this dependence, you can do the following.

1. 1. Label the different operations of z[k] and z[k+n1] as shown

```
/*$SCC label_begin k_n1_write*/
                z[k + n1].x = tmpz.x - t.x;
        z[k + n1].y = tmpz.y - t.y;
                /*$SCC label_end k_n1_write*/
        /*$SCC label_begin k_write*/
        z[k].x = tmpz.x + t.x;
        z[k].y = tmpz.y + t.y;
        /*$SCC label_end k_write*/
```

2. 2. Apply the no dependence pragma as follows.

```
#pragma no_dependence z write:k_n1_write  write:k_write  1
```

   This pragma tells the compiler that no writes of z[k+n1] alias with z[k] within one iteration. This is valid because n1 is never 0 in this design.

As a result, after this pragma is applied, Synphony C Compiler can achieve an II of 2 for this design.

Note:

If optimization scalarizes the array, it is no longer memory in generated hardware. In that case, the `no_dependence` pragma does not apply to that array.

no_inter_loop_memory_analysis Pragma

Array

Use this pragma to suppress memory dependency analysis across loop nests. Memories that are shared across loop nests can cause the loop nests to execute sequentially because of various dependencies. The tool cannot prove the dependencies do not exist, and this

limits its power to execute parallel loops. Use this pragma when you can guarantee that each loop nest works on non-aliasing locations of memory, and the loops can be parallelized.

```
#pragma no_inter_loop_memory_analysis arrayName
```

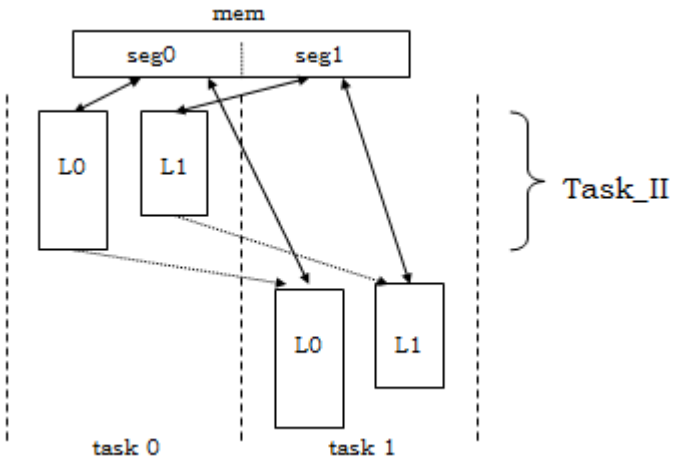
You can specify this pragma on a typedef, or members of a class or struct.

Example

```
int mem[100];

#pragma no_inter_loop_memory_analysis mem
```

The following figure shows that this pragma allows loop nests L0 and L1 to execute in parallel, even when they share the same array mem. One common use is double buffering by hand. Two loops read and write the same memory, but for each task there is a bank flip in the addressing that prevents aliasing.



Note that the Synphony C Compiler tool does not support linting for this feature and it is your responsibility to ensure that the dependencies through this array are not violated. If the pragma is incorrectly used, the Synphony C Compiler behavior is undefined.

no_inter_loop_stream_analysis Pragma

Stream

The Synphony C Compiler tool sequentializes the execution of loop nests which read from or write to the same stream. You can avoid this by suppressing stream dependence analysis

between loop nests within a PPA task using this pragma. This pragma can be applied to both internal and external streams. This pragma can be applied to only one end (either input or output) of a stream or both the ends by specifying the input and output function name appropriately with the pragma.

```
#pragma no_inter_loop_stream_analysis streamFunctionName
```

streamFunctionName can be either of the following:

- Input/output simulation function names in C designs
- obj:input or obj:output for read and write ports of the scc_stream class object obj.

You can specify this pragma on a typedef, or members of a class or struct.

Example

```
#include "scc.h"
int mode;
#pragma bitsize mode 1
FIFO(x,int); // Internal Stream x.
#pragma no_inter_loop_stream_analysis x.get()

void ppa()
{
      int i;
      int var1,var2, var3;
      for (i=0;i<10;i++){ // Loop L0
         if (mode==0) scc_stream_output_x(var1);
      }
      for (i=0;i<10;i++){ // Loop L1
         if (mode==0) var2=x.get();
      }
      for (i=0;i<10;i++){ // Loop L2
         if (mode==1) x.put(var2);
      }
      for (i=0;i<10;i++) { // Loop L3
         if (mode==1) var3=x.get();
      }
}
```

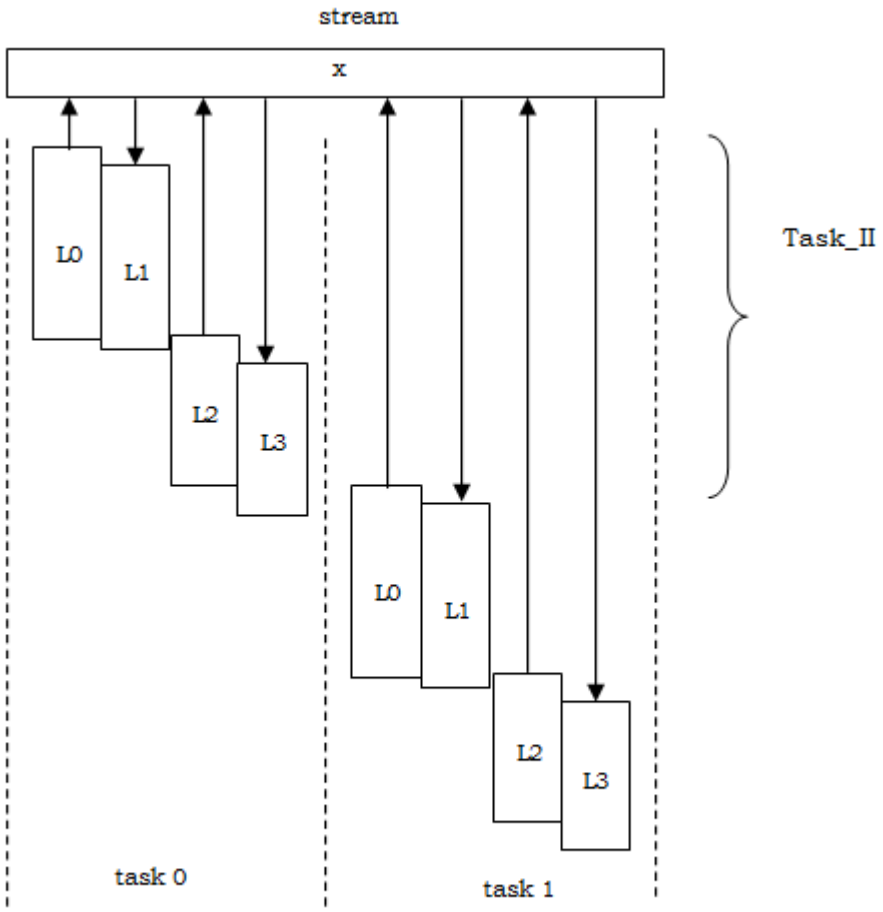
This example uses an internal stream x. The Synphony C Compiler tool would normally analyze the shared stream x.get() and x.put() usage and sequentialize the loops L0-L2 (write to same stream x) and L1-L3 (read from same stream x). This example uses a input scalar (mode) which guarantees that only one loop can access the stream at any time, so there is no need to sequentialize the two loops.

This pragma can be used to avoid sequentialization of loops L0-L2 and L1-L3 caused by the use of shared stream and the loops can start in parallel. Hardware for distinct FIFOs that are read and written in a non-interleaved fashion can be shared by naming them with the same

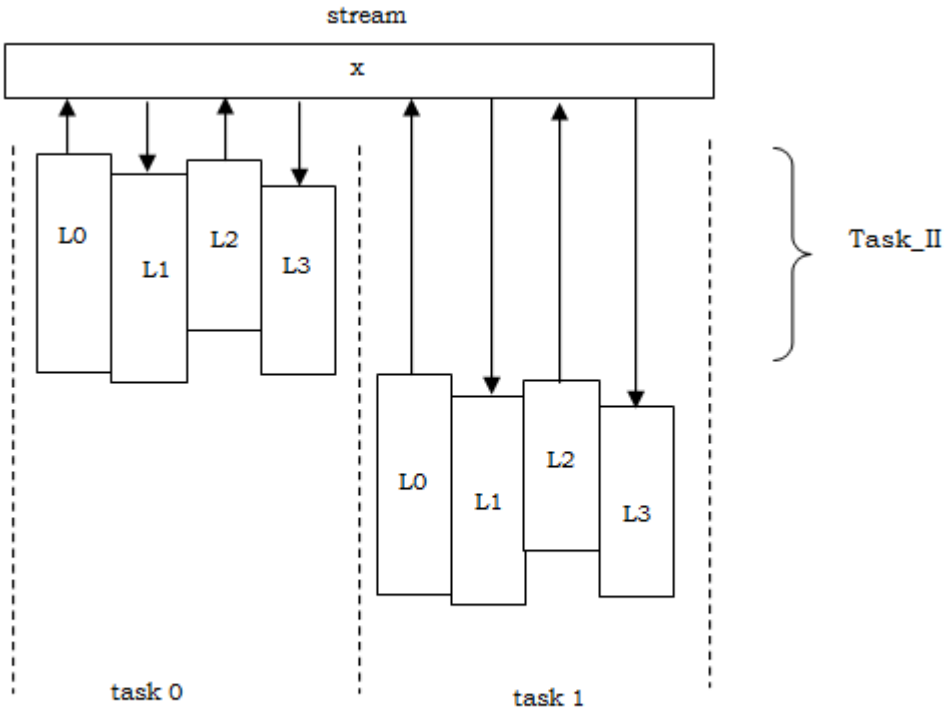
name and yet allowing them to execute in parallel using this pragma. In this example, the logic for x is shared between the loops L0 and L1. The order of execution of the loops is as shown in the following two figures.

Note:

The Synphony C Compiler tool does not support linting for this feature and it is your responsibility to ensure that the stream dependencies are not violated. The behavior of the Synphony C Compiler is undefined for incorrect usage of the pragma.



Without Pragma



With Pragma

no_inter_task_memory_analysis Pragma

Array

Memories that are shared across loop nests cause the subsequent tasks to execute sequentially due to data dependencies. This limits the tool in its ability to pipeline tasks. There may be situations when you guarantee that each task works on non-aliasing locations of memory and the tasks can be pipelined.

```
#pragma no_inter_task_memory_analysis arrayName
```

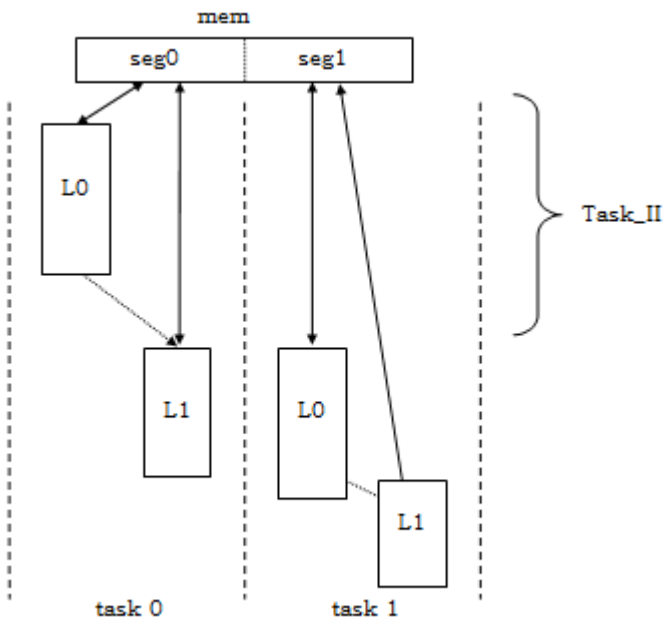
You can specify this pragma on a typedef, or members of a class or struct.

Example

```
int mem[100];

#pragma no_inter_task_memory_analysis mem
```

The next figure shows that this pragma allows loop nest L0 of task 1 to execute in parallel with loop nest L1 of task 0, even though they share the same array mem. Even though the array is accessed as read-only in the PPA code, the Synphony C Compiler tool has to conservatively assume that there may be a potential write to the array in the driver between tasks, and it does not start the executing the next task until the current task finishes using the memory. If you know there are no intermediate writes to the array between tasks, you can specify this pragma to allow the tasks to be pipelined.



Note that the Synphony C Compiler tool does not support linting for this feature and it is your responsibility to ensure that the dependencies through this array are not violated. If the pragma is incorrectly used, the Synphony C Compiler behavior is undefined.

no_inter_task_stream_analysis Pragma

Stream

Streams that are shared across loop nests cause the subsequent tasks to execute sequentially due to data dependencies. Use the following pragma to ignore the dependencies:

```
#pragma no_inter_task_stream_analysis streamFunctionName
```

streamFunctionName can be either of the following:

- Input/output simulation function names in C designs
- obj:input or obj:output for read and write ports of the scc_stream class object obj.

You can specify this pragma on a typedef, or members of a class or struct.

The Synphony C Compiler tool does not support linting for this feature and it is your responsibility to ensure that the stream dependencies are not violated. If you use the pragma incorrectly, the Synphony C Compiler behavior is undefined.

Example

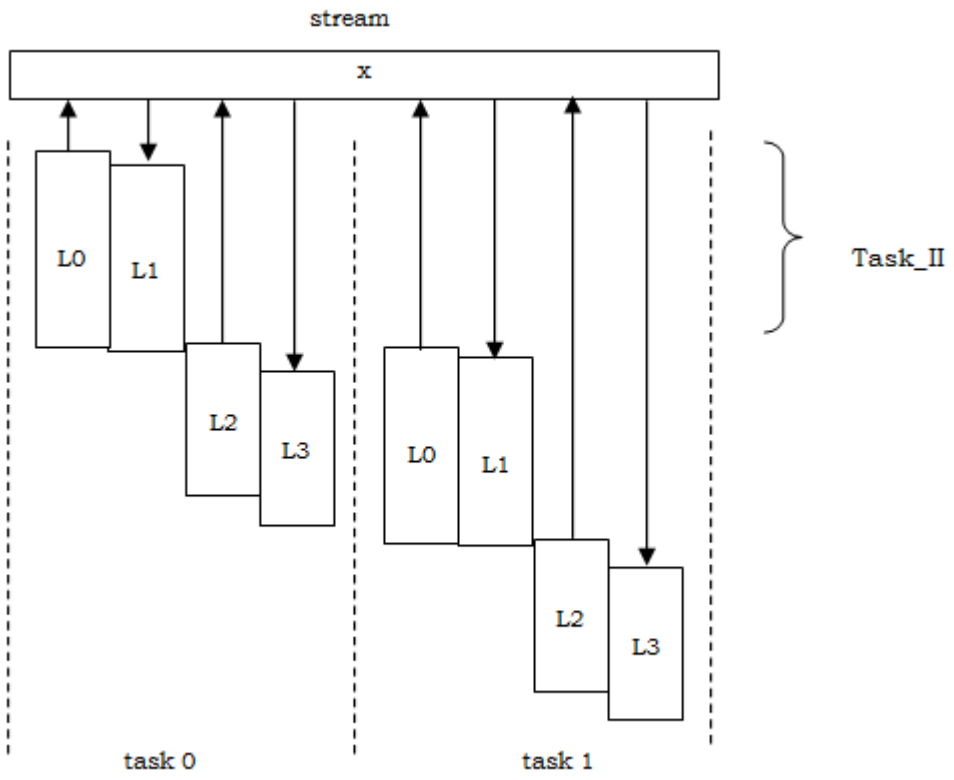
This example shows stream x is accessed by loops L0-L2 for write operation and L1-L3 for read operation.

```
#include "scc.h"
int mode;
#pragma bitsize mode 1
FIFO(x,int); // Internal Stream x.
#pragma no_inter_task_stream_analysis x.get()
#pragma no_inter_task_stream_analysis x.put()
void ppa()
{
   int i;
   int var1,var2, var3;
   for (i=0;i<10;i++) { // Loop L0
      if (mode==0) scc_stream_output_x(var1);
   }
   for (i=0;i<10;i++) // Loop L1
   {
      if (mode==0) var2=x.get();
   }
   for (i=0;i<10;i++) // Loop L2
   {
      if (mode==1) x.put(var2);
   }
   for (i=0;i<10;i++) // Loop L3
   {
      if (mode==1) var3=x.get();
   }
}
```

Since x is shared by L0-L2 and L1-L3, the two loop nest sets are sequentialized and this stream resource is considered occupied until all the loops of a task complete. This means a new task cannot start on loop L0 until loop L2 completes the old task. The pragma no_inter_task_stream_analysis can break this dependency by allowing a new task to start on loop L0 the moment the previous task completes on loop L0 and starts executing on loop L2.

You retain control over the loop that gets the current stream data by using a condition, like the input variable shown in the example, even when both the loops execute in parallel across tasks.

This figure shows the order of execution of the loops:



no_inter_tcab_subtask_memory_analysis Pragma

Array

Specify no_inter_tcab_subtask_memory_analysis pragma on the array variable to allow pipelining of subtasks on the TCAB. It is your responsibility to ensure that each call to the TCAB function works on a non-aliasing segment of the memory, and that memory dependencies are not violated by the pipelining of TCAB operations.

When a TCAB accesses a memory, the scheduling of the TCAB operations in the caller is constrained by other accesses to the same memory. The Synphony C Compiler tool does not perform memory analysis across procedure boundaries, so it conservatively assumes that the memory is occupied from the first access (read or write) time until the last access time of the memory. This prevents the pipelining of subtasks on the TCAB for the duration.

```
#pragma no_inter_tcab_subtask_memory_analysis arrayName [procs]
```

You can specify this pragma on a typedef, or members of a class or struct.

The pragma specification shown here allows TCAB tcab_fn to be pipelined in ppa:

```
int datain[N], W[M];
int dataout[M*N];
void tcab_fn(int *in, int *out, int off, int W_coeff)
{
   #pragma no_inter_tcab_subtask_memory_analysis in
   #pragma no_inter_tcab_subtask_memory_analysis out
   int i;
   #pragma unroll
   for (i = 0; i < N; i++) {
      out[off + i] = in[i] * W_coeff;
   }
}

void ppa()
{
   int j;
   for (j = 0; j < M; j++) {
   tcab_fn(datain, dataout, N*j, W[j]);  // each call works on a
                                         //different slice of
                                         //datain and dataout
   }
}
```

If the pragma is incorrectly used, the Synphony C Compiler behavior is undefined.

not_input, not_output Pragmas

Host Access

The testbench code and synthesizable code communicate using global variables, procedure parameters, and class member variables (when the synthesized procedure is a class method). Use the following pragmas to identify variables that are not inputs to, or outputs of, the procedure call:

```
#pragma not_input variableName [procs]
#pragma not_output variableName [procs]
```

The Synphony C Compiler tool detects some input and output communications automatically. For example, local variables are treated as “not input” and “not output.” If you identify other variables that are not inputs or not outputs, the tool can optimize the hardware further by omitting interfaces that are not needed.

For example, if your application uses a global memory called `scratchpad` that is both written and read during the procedure call but does not need its contents to be kept across calls, you can specify:

```
#pragma not_input scratchpad
#pragma not_output scratchpad
```

Note:

The `not_input` or `not_output` pragma refers to the communication into or out of the software procedure call, not the hardware connections into or out of the PPA. For example, if `scratchpad` is written in one call and that data is used in a subsequent call, it is both an input and an output of the procedure, and you should not use these pragmas. Instead, use the `localize_access` pragma to eliminate any external hardware connections.

The `not_input` or `not_output` pragma can include a list of procedures. If you provide a list, the pragma only applies during the synthesis of the listed procedure.

In the following example, if `fir` is called by procedure `ppa`, the `not_input` pragma applies during the synthesis of `fir` and `ppa`. Similarly, the `not_output` pragma only applies during the synthesis of `ppa`.

```
#pragma not_input scratchpad fir ppa
#pragma not_output scratchpad ppa
```

You can specify also this pragma on a typedef, or members of a class or struct.

num_iterations Pragma

Miscellaneous

The Synphony C Compiler tool finds the number of times a loop will execute when the loop bounds are statically known constants. This iteration count is used to determine the performance for each loop during rate matching. In addition, this information is also useful in analyzing the array addressing patterns and for performing optimizations. In cases where the loop bounds cannot be statically determined or the number of iterations varies for different tasks, use the num_iterations pragma to specify the number of iterations.

```
#pragma num_iterations ([min], [target], [max])
```

This pragma specifies that the loop executes between min and max number of iterations and that the hardware must be designed with the assumption that the loop executes target number of iterations. The pragma must be specified immediately before the loop to which it applies. All three arguments are optional; supply the ones that apply. The Lint simulation phase verifies that the loop executes between the min and max iteration counts specified in the pragma.

In the example, the loop i must execute between 10 and 20 iterations and the rate-matching phase will consider 15 iterations for performance.

```
#pragma num_iterations (10, 15, 20)
for (i=0; i<n; i++) {
     for (j=0; j<10; j++) {
     ...
     }
}
```

Loop Nesting

For a complex nesting of loops, the Synphony C Compiler tool interprets the iteration specification as follows:

```
#define I_MIN 1
#define I_TARGET 10
#define I_MAX 20
#define J_MIN 1
#define J_TARGET 20
#define J_MAX 40
#define K_MIN 10
#define K_TARGET 50
#define K_MAX 100
#pragma num_iterations (I_MIN, I_TARGET,I_MAX)
for(i=0;i<n;i++)
     #pragma num_iterations (J_MIN, J_TARGET, J_MAX)
     for (j=0;j<m;j++)
          #pragma num_iterations (K_MIN, K_TARGET, K_MAX)
          for (k=0;k<p;k++){
}
```

The Synphony C Compiler tool considers the total number of iterations for the loop nest based on the product of the respective fields for min, target and max iteration values as (I_MIN * J_MIN * K_MIN, I_TARGET * J_TARGET * K_TARGET).

Conditions Around Loops

If the loops are embedded in conditional statements, the minimum number of iterations of the loop nest is computed as the product of min values of iterations at each level of nesting. The target and max iterations of the loop nest are computed as the product of max iterations at each level of nesting.

Original loop structure:

```
#define I_MIN 1
#define I_TARGET 10
#define I_MAX 20
#define J1_MIN 1
#define J1_TARGET 20
#define J1_MAX 40
```

```
#define J2_MIN 10
#define J2_TARGET 50
#define J2_MAX 100
#pragma num_iterations (I_MIN, I_TARGET, I_MAX)
for(i=0;i<n;i++){
     if (cond1) {
          #pragma num_iterations (J1_MIN, J1_TARGET, J1_MAX)
          for (j1=0; j1<m1; j1++) {
      }
}
else{
     #pragma num_iterations (J2_MIN, J2_TARGET, J2_MAX)
          for (j2=0; j2<m2; j2++) {
               ...
     }
   }
}
```

The number of iterations of the loop nest are computed as (I_MIN * MIN(J1_MIN, J2_MIN)), (I_TARGET * MAX(J1_TARGET, J2_TARGET)), (I_MAX * MAX(J1_MAX, J2_MAX)) for min, target and max iterations respectively.

Multiple Parallel Sub-Loops

If multiple parallel loops are embedded in outer loops, the minimum, target and max number of iterations of the loop nest are computed as the sum of the respective values of iterations of each of the parallel loops.

Original loop structure:

```
#define I_MIN 1
#define I_TARGET 10
#define I_MAX 20
#define J1_MIN 1
#define J1_TARGET 20
#define J1_MAX 40
#define J2_MIN 10
#define J2_TARGET 50
#define J2_MAX 100
#pragma num_iterations(I_MIN, I_TARGET, I_MAX)
for (i=0;i<n;i++) {
   #pragma num_iterations(J1_MIN, J1_TARGET, J1_MAX)
   for (j1=0;j1<m1;j1++) {
   ...
   }
   #pragma num_iterations(J2_MIN, J2_TARGET, J2_MAX)
   for(j2=0;j2<m2;j2++) {
   }
}
```

The number of iterations of the loop nest are computed as (I_MIN * (J1_MIN+J2_MIN)), (I_TARGET * (J1_TARGET+J2_TARGET)), (I_MAX * (J1_MAX+ J2_MAX)) for min, target and max iterations respectively.

packed Pragma

Arrays, Streams, and Scalars

A stream of C struct type results in a wide or packed data port containing all fields of the struct. This is the default behavior. To create multiple, field-wise data ports on the PPA interface, specify the following pragma for the stream:

```
#pragma packed streamname no
```

Note that this pragma on streams does not change the internal hardware. It only changes the ports on the PPA interface.

A scalar of C struct type results in multiple, field-wise data ports on the PPA interface. This is the default behavior. To create a wide or packed data port (with fields of the struct) on the PPA interface, specify the following pragma for the scalar:

```
#pragma packed scalarname yes
```

Note that this pragma on scalars does not change the internal hardware. It only changes the ports on the PPA interface.

An array of C struct type results in multiple field-wise arrays (unpacked). This is the default behavior. To create a single wide or packed memory (with fields of the struct), specify the following pragma for the array:

```
#pragma packed arrayname
```

Unlike scalars and streams, this pragma has a significant effect on the internal hardware implementation and is applicable to TCABs and PPAs.

Packed and Unpacked Arrays

The SCC tool allows packed elements of a struct or class. But arrays of such objects cannot be used in synthesizeable code. Packed pragma can be specified on the individual fields:

```
#pragma pragmaName variableName [.fieldName [.fieldName […]]]
```

The following table lists the effects of using this pragma.

	Unpacked Memory	Packed Memory	Comments
Implementation	Separate memories are created for each kind of struct.	One combined wide memory.	
Access mechanism	Read and write accesses to fields of a struct memory are allowed.	Read: Any read to individual struct fields retrieves the entire struct from memory. Write: Illegal to write to a single field. Need to update the entire struct.	For packed memory, each field cannot be written, so it might require caching of data read from memory. This might lead to an increased number of registers.
Porting	Each memory is analyzed separately, and so porting might be better.	The number of accesses to a memory might increase due to the limitation of packed memory. This might lead to a higher number of ports.	
Cost	Each memory would get its own addressing logic; therefore, more addressing overhead. Less caching of memory data leads to fewer state or pipeline registers.	Lesser addressing overhead More registers are needed as register's values are read before their intended use.	
Feedback analysis across iterations	Flexible	Complex	Wider memories can increase the number of dependencies to be analyzed.

Packing Order

Field packing is in the same as the order of the fields appearing in the complex struct. The order of individual components at the memory interface is as follows: the high-level ordering spec follows the order of definition of the struct entries but within individual fields of the nested struct, the packing order keeps individual elements of the nested struct together.

The exact order of packing is reported in the Reports/report_interface.txt file generated at the end of the synthesize step.

Example 1:

```
struct S
{
int a;
short b;
};
```

Packed data order is:

{b15, b14, … b1, b0, a31, a30, … a1, a0};

Example 2:

```
typedef struct {
int Re;
int Im;
} ComplexData;
typedef struct {
int header[2];
ComplexData Info[3];
int priority;
} packet;
packet InterleaveMem[100];
#pragma packed InterleaveMem
```

Packed data order is:

```
[31:0] InterleaveMem.header[0]
[63:32] InterleaveMem.header[1]
[95:64] InterleaveMem.Info[0].Re
[127:96] InterleaveMem.Info[1].Re
[159:128] InterleaveMem.Info[2].Re
[191:160] InterleaveMem.Info[0].Im
[223:192] InterleaveMem.Info[1].Im
[255:224] InterleaveMem.Info[2].Im
[287:256] InterleaveMem.priority
```

The Synphony C compiler allows packed elements of a struct or class. But arrays of such objects cannot be used in synthesizeable code.

Packed Memory Interfaces

The packed memory of a complex `struct` is instantiated as follows:

```
typedef struct {
     int Re;
     int Im;
```

```
} ComplexData;

typedef struct {
     int header[2];
     ComplexData Info[3];
     int priority;
} packet;

packet InterleaveMem[100];
#pragma packed InterleaveMem
```

The order of individual components at the memory interface is as follows. The high-level ordering follows the order of definition of the struct entries but within the individual fields of the nested struct. The packing order keeps individual elements of the nested struct together.

```
[31:0]      InterleaveMem.header[0]
[63:32]     InterleaveMem.header[1]
[95:64]     InterleaveMem.Info[0].Re
[127:96]    InterleaveMem.Info[1].Re
[159:128]   InterleaveMem.Info[2].Re
[191:160]   InterleaveMem.Info[0].Im
[223:192]   InterleaveMem.Info[1].Im
[255:224]   InterleaveMem.Info[2].Im
[287:256]   InterleaveMem.priority
```

For the exact order of packing for other cases, check the `report_ppa_interface.txt` file generated at the end of synthesize step.

Packed Stream Interfaces

The Synphony C Compiler tool supports packed interfaces. Consider the following structure declaration:

```
a struct S
{
     int a;
     short b;
};
```

A stream of type S is always packed. For an array of type S that maps to a memory in the hardware, you may specify the array to be packed. The packed stream or memory interface for a variable of type S is as shown, where, a0…a31 are the 32 bits of field a and b0…b15 are the 16 bits of field b.

```
{ b15, b14, … b1, b0, a31, a30, … a1, a0 };
```

The following is a more complex stream created of nested `structs`:

```
typedef struct {
int Re;
int Im;
} ComplexData;
```

```
typedef struct {
     int header[2];
     ComplexData Info[3];
     int priority;
} packet;

packet MacToPHY.get();
```

The field packing is the same as the order of fields appearing in the complex `struct` for streams:

```
[31:0]        header[0]
[63:32]       header[1]
[95:64]       Info[0].Re
[127:96]      Info[0].Im
[159:128]     Info[1].Re
[191:160]     Info[1].Im
[223:192]     Info[2].Re
[255:224]     Info[2].im
[287:256]     priority
```

Synphony C compiler does support any packed elements to be part of a struct or class.

preserve_array Pragma

Array

Preserves an array and prevents the automatic banking and scalarization of arrays, scalarization of TCAB parameters, ROM value substitution, automatic ROM bitsizing, and the copying in or out of reference parameters.

This is the syntax for preserve_array:

```
int x[10][20];
#pragma preserve_array x
```

You can specify this pragma on a typedef, or members of a class or struct.

The Synphony C Compiler tool performs automatic scalarization and banking of arrays. You can suppress these optimizations and ensure that the tool implements internal_blockram, internal_fast, or user_supplied memories when you specify preserve_array along with the corresponding hardware implementation pragma. When used for ROM arrays, this pragma disables the constant substitution of initialization values and bit resizing optimizations based on initialization values.

If force_constant_addressing is also specified, preserve_array does not override it, because force_constant_addressing applies to the references, not the array, while preserve_array controls the properties of the array. In this case, it preserves the arrays while converting the references to constants.

read_write_ports Pragma

Array

The following pragmas specify the memory ports for user-supplied memories. The Synphony C Compiler tool tries to meet the specified porting requirement. If it does not meet the requirement, it instantiates an arbitrator.

```
#pragma read_write_ports arrayName combined [n]
#pragma read_write_ports arrayName separate [n]
                              [r readonly [w writeonly]]
#pragma read_write_ports arrayName combined [n] [no_arbitration]
#pragma read_write_ports arrayName separate [n]
                              [r readonly [w writeonly]] [no_arbitration]
#pragma read_write_ports arrayName [no_arbitration]
```

combined	Indicates that the memory for the array arrayname has combined read-write ports. If you mark an array for combined read-write ports but the application requires either read-only or write-only ports, the tool generates read-only or write-only ports as required for the application. It is your responsibility to make the necessary updates to the memory connections so that it performs only the read or write operations.
separate	Indicates that the memory for the array arrayname has separate read and write ports. Specify the kind of ports with the readonly and writeonly options.
n	Optional. Specifies the maximum number of ports. For separate read write ports, you can specify 0 readonly or 0 writeonly. This specifies that no read (or write) ports are expected for that array. If the Synphony C Compiler tool finds the unexpected ports, it produces the RTL but exits with error.
n readonly *n* writeonly	Specifies whether the port is read or write only when you use the separate keyword. The tool tries to meet this porting requirement, and instantiates an arbitrator if it cannot meet it.
no_arbitration	Specifies a hard constraint for the tool to create a memory without instantiating any arbitrator. If the constraint cannot be met, the tool generates an error message during the synthesize phase, relaxes the constraint, and generates the RTL.

You can specify this pragma on a typedef, or members of a class or struct.

You can globally specify a memory type for all ASIC and FPGA user-supplied memories with the `set_params` command. The pragma specification on individual arrays overrides the global specification.

Examples

Here are some examples of the `read_write_ports` pragma:

```
#pragma read_write_ports foo combined 3

#pragma read_write_ports foo combined  // unconstrained //readwrite

#pragma read_write_ports foo separate 2 readonly 1 writeonly

#pragma read_write_ports foo separate 1 writeonly 2 readonly

#pragma read_write_ports foo separate 2 readonly no_arbitration

#pragma read_write_ports foo separate 1 writeonly //unconstrained read

#pragma read_write_ports foo separate 3 readonly //unconstrained write

#pragma read_write_ports foo separate // unconstrained read //and write
```

This pragma specification applies to the entire design hierarchy where the memory is accessed. At each level where the pragma is applied, the Synphony C Compiler tool generates a schedule and tries to meet the required port specifications. At any level, it only instantiates an arbitrator when all accesses to a memory are self-contained at that level, and any imported TCABs inside it are not instantiated outside the design.

To prevent the implementation of arbitration, use the `no_arbitration` option. In that case, the tool takes alternative actions to handle multiple accesses and produce a design that does not instantiate an arbitrator.

In the following example, if you synthesize functions tcabA, tcabB, and tcabC as TCABs and function ppa1 as a PPA, the pragma is applied in all the entities except on the TCAB of function tcabC( ), because there is no access to the memory memA inside the TCAB. Also, if function ppa1( ) is the topmost hierarchy, then arbitration is instantiated at this level, because there is access to the memory memA from this level.

```
int memA [100];
#pragma read_write_ports combined 2
void tcabA(){
   // contains several accesses to memA.
}
void tcabB(){
   // contains several accesses to memA and some calls to tcabA().
}
void tcabC(){
   // contains no calls to either tcabA or tcabB nor accesses to
   // memory memA.
}
void ppa1(){
   // calls to memory memA and calls to tcabB }
```

Contrast this with the following ppa2( ) function. The top level of the design does not separately access memA but only calls tcabB. Effectively, all accesses to memA are restricted to tcabB and its child TCABs, so any arbitration would be instantiated in the implementation for the tcabB( ) function.

```
void ppa2(){
// no calls to memory memA but calls tcabB.
}
```

rom Pragma

Array

This pragma specifies that an array be treated as a ROM. This is the syntax for the pragma:

```
#pragma rom arrayName
```

You can specify this pragma on a typedef, or members of a class or struct.

Example

```
unsigned char D[20] = {2, 1, 3, 2, };
#pragma rom D
```

Alternatively, you can specify a ROM in the source code without using the pragma by declaring the array with the const modifier.

signal_name_prefix, signal_name_identifier, and signal_name_suffix Pragmas

Miscellaneous

The `signal_name_prefix`, `signal_name_identifier`, and `signal_name_suffix` pragmas can be used to rename the stream, scalar, and memory ports on a PPA interface. This is the pragma syntax:

```
#pragma signal_name_prefix entity prefix_string
#pragma signal_name_identifier entity identifier_string
#pragma signal_name_suffix entity suffix_string
```

The *entity* specified in the pragma is the name of a variable, stream, or type; and the specified naming string must be able to serve as a valid name for a signal in Verilog.

The tool constructs the entity name according to the following rules:

- If you do not use any of these pragmas, the tool constructs the entity name according to its own internal naming scheme.
- If you specify one or more of the prefix, identifier, and suffix strings with these pragmas, the entity name is a concatenation of the specified strings in that order, with any unspecified strings determined as follows:
  - Prefix string – an empty string (no prefix used if none specified)
  - Identifier string – the top-level variable or object name
  - Suffix string – obtained from the hierarchy of its fields for structs or class members, or an empty string for other types of entities

Similar to any other pragma, these pragmas can also be specified on a type or field declaration. In that case, all variables of that type get the same string as the prefix, identifier, or suffix, as specified in the pragma. You can override the general string specification given on a type with a more specific one by providing a separate pragma on the variable or derived type. Make sure that the pragmas assign a unique name to each signal at the interface. Otherwise, the tool inserts additional qualifier strings into the names to make them unique.

Scalarization of arrays and banking or multi-buffering of memories can result in the creation of multiple ports. The tool names such ports uniquely by using array element numbers or buffer ID numbers in the names. Ports of multi-ported memories are also made unique in a similar manner. You cannot control the naming of individual elements in these cases.

A memory, stream, or scalar interface is a port-bundle containing multiple signals (for example, `datain`, `req`, and `address`). To control the name of a port, you specify the port function by its keyword in the pragma, along with the name prefix, identifier, or suffix. Table 7-1, Table 7-2, and Table 7-3 show the keywords, default identifiers, and descriptions of the ports created for memory, stream, and scalar interfaces. The examples following the tables show how the keywords are used in the pragmas.

Table 7-1 Memory Interface Keys and Default Port Names

Interface	Keyword	Default identifier	Description
Memory	req	cen	chip enable
	mode	wen	write enable
	wdata	d	data write
	rdata	q	data read
	addr	a	address
	ack	ack	data valid
Direct memory	direct_req	req	chip enable
	direct_ack	ack	data enable
	direct_mode	mode	data read or write
	direct_rdata	rdata	data read
	direct_wdata	wdata	data write
	direct_addr	addr	address

Table 7-2 Stream Interface Keys and Default Port Names

Interface	Keyword	Default identifier	Description
Input stream	get_req	ready	chip enable
	can_get	valid	data valid
	datain	data	data
Output stream	put_req	valid	data valid
	can_put	ready	chip enable
	dataout	data	data

Table 7-3 Scalar Interface Keys and Default Port Names

Interface	Key	Port in hardware	Description
Scalar	datain	scalar_in	input data
	dataout	scalar_out	output data
	detain_enable	scalar_in_enable	chip enable

You can control the port names using the signal naming pragmas as shown in the following examples.

```
struct rd_st {
    uint32  araddr;
    uint4   arlen;
};

scc_stream<rd_st> x;
scc_stream<rd_st> y;

... = x.get();

y.put(data);
```

The *default* port names for these streams are:

```
stream_in_x_data
stream_in_x_ready
stream_in_x_valid

stream_out_y_data
stream_out_y_ready
stream_out_y_valid
```

You can explicitly specify the name identifier strings used for these ports as follows:

```
scc_stream<rd_st> x;
#pragma signal_name_identifier x vs
```

In that case, the generated port names are:

```
vs_data
vs_ready
vs_valid
```

You can also create per-field data ports using the packed pragma:

```
scc_stream<rd_st> x;
#pragma packed x no
#pragma signal_name_identifier x vs
```

In that case, the generated port names are:

```
vs_araddr
vs_arlen
vs_ready
vs_valid
```

Note that you cannot specify pragmas on the struct data members of an scc_stream object using the object name:

```
scc_stream<rd_st> x;
#pragma packed x no
#pragma signal_name_identifier x.araddr vs # Not valid
#pragma signal_name_suffix x.araddr _addr  # Not valid
#pragma signal_name_suffix x.arlen _len    # Not valid
```

Instead, you can specify the pragmas on the fields of the struct data type, and these names are applied to the signals in the stream interface:

```
struct rd_st {
    uint32  araddr;
    #pragma signal_name_suffix araddr _addr
    uint4   arlen;
    #pragma signal_name_suffix arlen _len
};

scc_stream<rd_st> x;
#pragma packed x no
#pragma signal_name_identifier x vs
```

In that case, the generated port names are:

```
vs_addr
vs_len
vs_ready
vs_valid
```

You can explicitly specify the individual signals in the stream port bundle as follows:

```
struct rd_st {
    uint32  araddr;
    uint4   arlen;
};
```

```
scc_stream<rd_st> x;
#pragma signal_name_identifier x vs
#pragma signal_name_suffix x:datain _din
#pragma signal_name_suffix x:get_req _rdy
#pragma signal_name_suffix x:can_get _vld

scc_stream<rd_st> y;
#pragma signal_name_identifier y hs
#pragma signal_name_suffix y:dataout _dout
#pragma signal_name_suffix y:put_req _vld
#pragma signal_name_suffix y:can_put _rdy

... = x.get();

y.put(data);
```

In that case, the generated port names are:

```
vs_din
vs_rdy
vs_vld

hs_dout
hs_rdy
hs_vld
```

Similarly, these pragmas can be used for scalars and memories, as follows:

```
struct rd_st z_var;
#pragma signal_name_identifier z_var zv
#pragma signal_name_suffix z_var.araddr:dataout _addr
#pragma signal_name_suffix z_var.arlen:dataout _len

z_var = data;   // scalar output
```

In that case, the generated names for scalar output ports are:

```
zv_addr
zv_len
```

Note:

Pragmas in the form of ":*keyword*" notation (`can_get`, `can_put`, `get_req`, `put_req`, `datain`, and `dataout`) applied on stream types are ignored by the tool while constructing signal names for the stream.

The following additional examples show how you can create different signal naming schemes by combining signal name properties in different ways.

Common Prefix for All Entities in a Class

In this example, the same prefix applies to all entities in a class.

```
struct rd_st {
    uint32  araddr;
    uint4   arlen;
};

class my_class {
    scc_stream<rd_st> in;
    #pragma signal_name_suffix in:get_req _rdy
    #pragma signal_name_suffix in:can_get _vld
    #pragma signal_name_suffix in:datain  _din

    rd_st d;
    #pragma signal_name_suffix d.araddr:dataout _addr
    #pragma signal_name_suffix d.arlen:dataout _len
};
#pragma signal_name_prefix my_class es_

my_class v1, v2;

… = v1.in.get();          // input stream
v1.d = data1;             // scalar output
… = v2.in.get();          // input stream
v2.d = data2;             // scalar output
```

In this case, the generated port names for the input streams and scalar outputs are:

```
es_v1_in_rdy
es_v1_in_vld
es_v1_in_din

es_v1_addr
es_v1_len

es_v2_in_rdy
es_v2_in_vld
es_v2_in_din

es_v2_addr
es_v2_len
```

Hierarchical Name as a Prefix Instead of Suffix

In this example, the hierarchical name is used as the naming prefix:

```
struct rd_st {
    uint32  araddr;
    #pragma signal_name_prefix araddr addr_
```

```
    uint4   arlen;
    #pragma signal_name_prefix arlen len_
};

#pragma signal_name_suffix rd_st ""

scc_stream<rd_st> x;
#pragma packed x no
#pragma signal_name_identifier x vs
#pragma signal_name_prefix x:get_req rdy_
#pragma signal_name_prefix x:can_get vld_
```

In this case, the generated port names for the input stream are:

```
addr_vs
len_vs
rdy_vs
vld_vs
```

signal_polarity Pragma

Stream

You can specify the polarities of stream interface control signals by using the `signal_polarity` pragma:

```
#pragma signal_polarity stream_name:key high | low
```

This pragma sets the signal polarity of the stream control port associated with the key to either active-high or active low. For a list of stream interface keywords, see Table 7-2 on page 7-62.

For example, the following pragma specifies an active-low signal for the valid port of input stream mystream1 in the hardware interface:

```
#pragma signal_polarity mystream1:can_get low
```

signal_production Pragma

Stream

You can specify whether to produce particular stream interface control signals by using the `signal_polarity` pragma:

```
#pragma signal_production stream_name:key yes | no
```

When this pragma is set to no, it prevents creation of the port associated with the key; the port is omitted from the hardware interface. However, there is no change to the internal core data path. For a list of stream interface keywords, see Table 7-2 on page 7-62.

For example, the following pragma prevents the generation of the `valid` signal for `mystream1` in the hardware interface.

```
#pragma signal_production mystream1:can_get no
```

The following example further demonstrates the usage of this pragma. A design has a `video_in` input stream and a `video_out` output stream with data generated in every clock cycle. In this case, there is no need for `valid` and `ready` ports in the interface. To prevent these ports from being created, you can use code similar to the following:

```
scc_stream<T> video_in;
#pragma signal_production video_in:get_req no
#pragma signal_production video_in:can_get no

scc_stream<T> video_out;
#pragma signal_production video_out:put_req no
#pragma signal_production video_out:can_put no
```

Using this type of pragma specification, you can generate various configurations of signal interfaces, such as the following types of streams:

- Handshake: no, Data=yes
- Handshake:1-way, Data=yes
- Handshake:1-way, Data=no
- Handshake:2-way, Data=yes
- Handshake=2-way, Data=no

scalarize Pragma

Array

This pragma specifies whether to dissolve an array and map it to wires and/or registers in the hardware.

```
#pragma scalarize arrayName [yes | no]
```

Depending on how you specify the pragma, you can implement one of three modes, infer, scalarization, or no scalarization:

Specification	Implementation
No pragma specified	Infer mode. The tool determines whether to map the elements to wires an/or registers, based on heuristics.
Pragma specified without yes or no Pragma specified with yes	The default is yes. The tool maps all elements to wires and/or registers.
Pragma specified with no	The tool prevents the mapping of array elements to wires and/or registers. Note that the array may still be banked. To prevent banking and scalarization, use the preserve_array pragma ("preserve_array Pragma" on page 7-57).

The hardware generated from a scalarized array depends on where the array was declared:

Declaration	Hardware
Procedure scope	Internal wires/registers
TCAB/PPA parameter	Ports for each element that is read and each element that is written
Global and memory mapped	Each element that is read and each element that is written is placed in the memory map
Global and direct ports	Ports for each element that is read and each element that is written

See "Scalarization" on page 8-21 for more information about scalarization.

Examples

```
int X[8][64][4];
#pragma scalarize X
int Y[8][64][4];

#pragma scalarize Y no
```

unroll Pragma

Miscellaneous

The Synphony C Compiler tool supports automatic unrolling of for loops as directed by the unroll pragma. Unrolling can be useful for coding convenience and for increasing performance.

This is the syntax:

```
#pragma unroll [factor]
```

If you specify the optional factor the loop is unrolled by the specified factor, which must be a positive compile time constant. For example:

```
#pragma unroll 2
```

The Synphony C Compiler tool propagates constants through direct initialization, loop unrolling and function inlining to enable further loop unrolling.

Specify the unroll pragma just before the for loop that needs to be unrolled. There should be no intervening C statements between the pragma and the loop to be unrolled. In this example, the second j loop is unrolled:

```
for (i=0; i<10; i++) {
      sum = 0;
      #pragma unroll
      for (j=0; j<10; j++) {
         ...
      }
}
```

You can mark multiple loops within a loop-nest or across loop-nests for unrolling. If an outer loop is marked for unrolling, then its inner loops are not automatically unrolled. Each loop to be unrolled must be individually marked.

Example

Here is a complete example showing multiple uses of the unroll pragma.

Here is a complete example showing multiple uses of the unroll pragma.

```
extern int yin.get();
extern void yout.put(int);
extern int c0,c1,c2,c3,c4;
void filter5() {
   int i, j, i1, j1, k, f[5];
```

```
   // perform a 5-tap horizontal filter
   for (j= 0; j < ROWS; j++) {
      for (i= 0; i < COLS; i++) {
         int output;

         // iscc_streamnitializing
         if(i==0) {
            #pragma unroll
         for(k=1;k<5;k++) f[k]=0;
         }
   f[0] = yin.get();

   // filtering
   output = (f[4]*c4 + f[3]*c3 + f[2]*c2 + f[1]*c1+f[0]*c0)>> 8;

   // shifting
   #pragma unroll
   for(k=4;k>0;k--) f[k]=f[k-1];
      x.put(output);
      }
   }
}
```

This simple 5-tap filter uses unrolling in two places: to initialize the shift register taps, and then to shift them with each incoming data.

Loops with non-constant bounds cannot be unrolled even if they have a constant trip count, unless they have the maximum number of iterations specified through #pragma num_iterations. For example:

```
for (i=0; i< 20; i++) {
#pragma unroll j
for(j = i + 5; j < i + 10; j++) {
...
```

The j loop cannot be unrolled unless loop i is unrolled as well.

However, this loop can be unrolled, because #pragma num_iterations is specified:

```
#pragma num_iterations (10, 15, 20)
for(j = 0; j < N; j++)
...
```

This example has three parameters: the minimum number of iterations, the targeted number of iterations, and the maximum number of iterations.

Unrolling a loop nest might increase runtime because unrolling increases the number of operations. It might also depend on the style of coding. Take this code for example:

```
index = 0;
#pragma unroll
for (i = 0; i < 128; i++) {
   #pragma unroll
   for (j = 0; j < 8; j++) {
      out[index] = in[table[index%16]];
      index ++;
   }
}
```

The code resulting from unrolling will be of this form:

```
index = 0;
out[index] = in[table[index%16]];
index++;
out[index] = in[table[index%16]];
index++;
…
```

This code would require multiple successive passes of constant folding and constant propagation to pass the constant index values to the array indexes. This would result in increased runtime. This is a better coding approach if you want to save runtime:

```
#pragma unroll
for (i = 0; i < 128; i++) {
   #pragma unroll
   for (j = 0; j < 8; j++) {
      out[8*i+j] = in[table[(8*i+j)%16]];
   }
}
```

Note that the resulting code after optimizations in these two examples will be the same.

user_supplied, user_supplied_fpga Pragmas

Array

There are two pragmas, one for ASIC and one for FPGA designs. When you specify the pragma on an array, the results in the memory corresponding to the array are placed outside the PPA. The syntax of this pragma is shown:

```
#pragma user_supplied array (ASIC only)
#pragma user_supplied_fpga array (FPGA only)
```

You can specify this pragma on a typedef, or members of a class or struct.

You can prevent the array from being scalarized with #pragma scalarize *arrayName* no, which is described in “scalarize Pragma” on page 7-68.

8

Synthesizable Code

You can generate better hardware from C or C++ code by specifying the constraints to Synphony C Compiler tool. This chapter describes how to get the Synphony C Compiler tool to generate the hardware you need. See the following for more information:

- Non-Standard Variable Widths
- Memory Support
- Stream Support
- Mutual Exclusion
- Pipeline Flushing
- Multi-Level Break Statements
- Pre-Defined Custom Operations

Non-Standard Variable Widths

Standard ANSI C/C++ provides char, short, and int/long as the means to declare 8-bit, 16-bit, and 32-bit integers. The Synphony C Compiler tool does extensive value analysis and bit width optimization to prune the bit size of variables to their necessary minimum.

For example, if x is of type char, the Synphony C Compiler tool can infer that (x + 10) can be represented with 9 bits, while (x + 1000) requires 11 bits. Similarly, the tool infers that (x & 0x1F) can be represented with only 5 bits.

The Synphony C Compiler tool uses this information to reduce the size of registers, local memories, FIFOs, function units, and data paths, which in turn reduces the area of the resulting RTL.

To increase the efficiency of width optimizations, you can use a pragma to specify a bit size for any scalar, array, field of a structure, or stream procedure call. This allows the Synphony C Compiler tool to further optimize the size of the hardware generated for the expressions involving that variable.

bitsize Pragma Examples

You can specify bitsize pragmas on variables of all the Synphony C Compiler supported data types, including typedefs, SystemC data types, and enumerations. You can also use the pragma on streams and memories, typedefs, and fields of struct type. If you specify the bitwidth of an individual variable, it overrides the bitwidth specification on the data type of that variable.

Example 1

```
int x;
#pragma bitsize x 19
```

Example 2

```
typedef int int19;
#pragma bitsize int19 19
      int19 x,y; //The bitwidth of x is set as 19,
         // similar to the previous example
      #pragma bitsize y 10  // This sets the width of y as 10
         // overriding the specification of 19
         // through the typedef.
```

Example 3

```
char y[10];
#pragma bitsize y
```

Example 4

The following example shows a function-based stream interface. For a C++ program, where you use the scc_stream class to represent the interface, #pragma bitsize would not be applicable to the stream methods or objects. See "Stream Support" on page 8-24 for details.

```
short mystream.get();
   #pragma bitsize mystream 3
```

Bitsize for Data Types

A bitsize pragma on non-SystemC data types does not change the C or C++ semantics. Do not treat it as a way to create a new data type, and alleviate the need for truncation and masking operations. Instead, think of the bitsize pragma as an assertion that the application output depends only on the specified bits. It is an error for an application to depend on any more bits than the lower-order bits specified in its bitsize.

The tool also supports parameterized integer data types s_int< > and s_uint< >. These types can be sized from 1 to 64 bits using standard SystemC notation, for example, s_uint<38> for a 38-bit unsigned variable.

Note that these types have semantics different from C types of the same width. Using parameterized integer data types has effects different from using the `#pragma bitsize` specification on variables with other types. Bitsize pragmas are really assertions that are verified during linting whereas parameterized integer data types cause the compiler to do the masking or truncation on the values of the variables with these types. As a result, these types have a different effect during linting and in the rest of the system in the sense that no overflow or underflow errors are produced.

Struct Bitsize

Bitsizing of struct fields is supported using the bitsize pragma. The following example specifies bitsizing by adding pragmas to the struct and struct variable fields.

```
struct s1 {
int x1;
#pragma bitsize x1 4
short x2;
#pragma bitsize x2 9
} A[10], B;
```

- Bitsize pragma can be specified on all base data types. For struct and class type elements, any base type members of the struct/class can be bitsized as follows:
  `#pragma bitsize A.x1 3`
- Struct type defaults are allowed for bitsize

- Bitsize can be greater than the default: `#pragma bitsize A.x2 12`
- Bitsize cannot be greater than variable native type
- Bitsize pragma cannot be applied to a struct variable. It can be applied on fields of struct variable.

```
#pragma bitsize B 37      //invalid
#pragma bitsize B.x1 5    //This specification is legal and
                          //overrides bitsize specification
                          //already present on the field
                          //"x1" inside the struct.
```

Note:
Bit size pragmas are used by the compiler at its own discretion to optimize the hardware.

As an example of proper usage of bitsize pragma consider the following global declarations of inputs x, y, and output z:

```
unsigned short x,y;
unsigned int z;
#pragma bitsize x 11
#pragma bitsize y 9
```

Case #1 - Masking:

```
z = (x & 0x7ff) * (y & 0x1ff);
```

Case #2 - No masking:

```
z = x * y;
```

Case #1 has correct bitsize semantics. Golden simulation will match RTL simulation for all values of x, y.

Case #2 does not have correct bitsize semantics for all values of x, y. Golden simulation is guaranteed to match RTL simulation for x < 2048 and y < 512. For other values of x and y, the results are unpredictable. It is an acceptable code if the driver guarantees that x < 2048 and y < 512.

Note:
Lint simulation will catch cases in which the data sets violate the implied assumptions of the bitsize pragmas. For more details on linting, see the Chapter 12, "Synphony C Compiler Linting."

The Synphony C Compiler tool performs bitwidth operation according to the semantics of operation such as:

- multiplication, where output width = sum of input widths
- addition, subtraction, where output width = larger input width + 1

- constant AND, where output width = width of larger input
- assignments, where output width = input width

The tool understands the width of constraints and applies the rules accordingly, for example ANDing an unsigned int with 0xff results in an output of 8-bits.

It is generally not necessary to specify widths for most variables in the code, as they can be inferred from other widths as well as the semantics of the code. However, it is important to specify the widths of the data when the tool cannot prove that they are different from the native types. This is true of the following objects:

- Inputs
- Outputs
- Accumulators or other points where you have knowledge about the width that the compiler is not able to prove
- The output of operation such as saturation, where the width of the result may not be visible

Example

In the following code, we set bitwidths for inputs, outputs, and the accumulator. We need not specify widths for the variables as they will be correctly calculated by the tool.

```
extern short in(void);
#pragma bitsize in 7
extern void out(short);
#pragma bitsize out 7
extern void acc(int);
#pragma bitsize acc 11

void ppa(void) {
      int i;
      int acc=0;
      for(i=0;i<10;i++) {
         short out = in.get();
         acc += out;
         out.put(out);
         acc.put(acc);
      }
}
```

The compiler propagates bitwidths in both directions. For forward propagation, the compiler knows that stream scc_stream_input_in is 7 bits wide, and can use this information to prove that the variable out requires only 7 bits.

The compiler is unable to prove some properties of the variable acc. By examining the code it ascertains that it only accumulates ten times, so the value of acc should not grow beyond

11 bits. However, you do need not specify this explicitly as you also need to specify the width of the output stream scc_stream_output_acc, which is 11 bits. The reverse bitwidth propagation pass by the compiler shows that acc also requires only 11 bits.

In most cases, these specifications of bitwidths for the boundary of the application are sufficient to yield good results, but in few cases, variables like acc have to be directly constrained if neither the forward nor backward propagation can successfully determine the correct bitwidth.

Memory Support

This section describes various aspects of memory structures, and how they are handled by the Synphony C Compiler tool. See the following for details:

- Array Mapping
- Local Memory
- ROMs
- Packed and Unpacked Memories
- Memory Arbitration
- Banking and Multi-Buffering of Memories
- Scalarization

Array Mapping

Synphony C Compiler tool allows you to target the placement of arrays in the memory structures of the system. However, the behavior is different for ASIC and FPGA devices.

ASIC Array Mapping

In ASIC, Synphony C Compiler maps arrays to local SRAM, internal register-based RAM, external ROM, synthesized ROM or registers. The choice depends on the array's declaration in the PPA file:

- Declared with const modifier or with “`#pragma rom`” - external or synthesized ROM
- Declared without const modifier and without “`#pragma rom`” - local memory or registers
- By default, arrays of C structures/C++ objects are unpacked, each field of the structure mapped to an independent memory. An array of C structure/C++ class can be packed to a single memory using #pragma packed. In such a case, the structure becomes one

word of the “wide memory” synthesized. In particular, addressing refers to all the fields of the structure unlike the unpacked case where each field gets an independent address space.

Synphony C Compiler supports two types of local memories and ROMs for arrays:

- Declared with “user_supplied” - external (user-supplied) SRAM or ROM
- Declared with “internal_fast” - flip-flop- or register-based (synthesized) internal memory

If you have not specified the type of the memory for an array, the Synphony C Compiler tool decides on the type of the memory based on the number of words and the total number of bits the memory requires. Thresholds on the number of words and number of bits can be modified through the Command Line Interface using the Tcl commands `set_params -ram_synthesis_threshold` and `set_params -rom_synthesis_threshold`.

For more information on Tcl commands, see the Chapter 3, “Synphony C Compiler Commands.”

Here are some ASIC examples:

- ```
  // external SRAM
  int X[200];
  ```
- ```
  // internal register-based RAM
  int Y[10][10];
  #pragma internal_fast Y
  ```
- ```
  // external ROM
  const int Z[200] = {10, 20,};
  ```
- ```
  // synthesized ROM
  const int C[10][10] = {{5, 3, 2, 1,}, {3, 0,},};
  #pragma internal_fast C
  ```

FPGA Array Mapping

Synphony C Compiler tool maps arrays to either RAMs or ROMs. For both RAMs and ROMs the following types of memories are options: global memory, external local memory (EM), internal register-based memory (RM), internal block RAM or register.

If the array is declared with const modifier or with #pragma rom then the corresponding memory is a ROM. Write access to such memories are illegal. In all other cases, the corresponding memories have both read and write access.

By default, arrays of C structures or C++ objects are unpacked, each field of the structure mapped to an independent memory. An array of C structure or C++ class can be packed to a single memory using #pragma packed. In such a case, the structure becomes one word of

the “wide memory”. In particular, addressing refers to all the fields of the structure unlike the unpacked case where each field gets an independent address space.

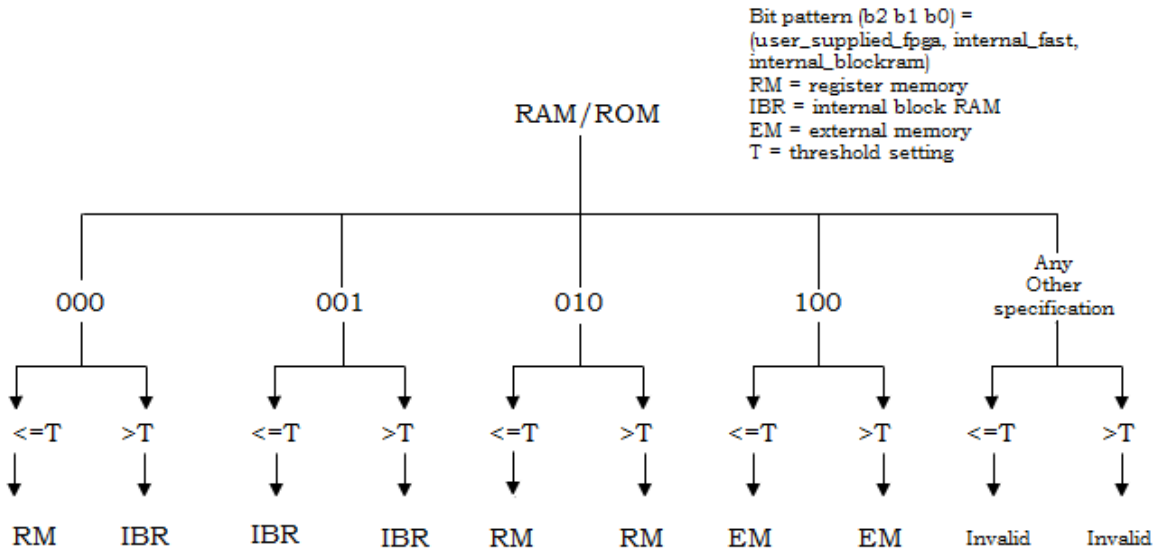
There are other factors that influence the type of memory used:

- The following #pragma specifications influence the type of memory:
  - #pragma user_supplied_fpga
  - #pragma internal_fast
  - #pragma internal_blockram
- You can set a thresholds on the number of words and total number of bits, using these commands:

set_params -ram_synthesis_threshold (for RAMs) and set_params -rom_synthesis_threshold (for ROMs).

For more information on Tcl commands, see the Chapter 3, “Synphony C Compiler Commands.”

The following figure specifies the rules according to which the type of memory is determined:



Here are some FPGA examples:

- ```
  //internal block RAM
  int X[200];
  ```
- ```
  //external SRAM
  int X[200];
  #pragma user_supplied_fpga X
  ```
- ```
  // internal register-based RAM
  int Y[10][10];
  #pragma internal_fast Y
  ```
- ```
  // internal block RAM
  const int Z[200] = {10, 20, };
  ```
- ```
  //synthesized ROM
  const int C[10][10] = {{5, 3, 2, 1, }, {3, 0, },};
  #pragma internal_fast C
  ```

Local Memory

Synphony C Compiler tool determines the size and porting of local memories based upon array access patterns and PPA performance targets. For information about specifying performance targets and writing code to reduce memory size and porting, see “Optimizing Memories for Rate Matching” in the *Synphony C Compiler User Guide*.

Synphony C Compiler tool generates an external SRAM interface compatible with standard third party SRAM generators. For more information on SRAM interfaces, see “Local Memory Interfaces” on page 14-12. The Synphony C Compiler tool automatically generates the register-based memories (internal_fast).

These are some examples of local memories:

```
unsigned char W[100][100];
int X[2000];
unsigned char Y[20];
#pragma internal_fast Y
```

These declarations tell Synphony C Compiler tool to map W and X to SRAM and Y to register-based RAM.

For ASIC, the Synphony C Compiler tool maps non-rom arrays declared in the PPA file to local memory. If the array is defined with #pragma internal_fast, then it is mapped to an internal register-based RAM. Otherwise, it is mapped to an external SRAM.

For FPGA, if the array is marked internal block RAM, then it is mapped to a block RAM on FPGA device. If the array is defined with user_supplied_fpga, it is mapped to external
```

memory. If the array is marked to internal_fast, it results in the memory corresponding to the array getting mapped to registers and placed inside the PPA.

Note:

In Synphony C Compiler tool, the mapping of arrays to memory is done on a one-to-one basis. That is, Synphony C Compiler tool does not pack multiple arrays to a single memory.

ROMs

The Synphony C Compiler tool maps array declared with const modifier or #pragma rom to ROM. A write to such array is not permitted. ROM arrays declared as internal_fast are automatically synthesized by the Synphony C Compiler tool. External ROMs give the flexibility of modifying the values of ROMs without the need to resynthesize, whereas synthesized ROMs have a lower latency of access.

Here are some examples.

```
const unsigned char C[10][10] = {{10, 20,}, {0, 33, 21},};
unsigned char D[20] = {2, 1, 3, 2, };
#pragma rom D
const unsigned int E[20][10] = {{112, 220,}, {20, 332, 121},};
#pragma internal_fast E
const int TEST[] = {
#include "data.txt"
};
```

where the file data.txt contains the ROM initialization data.

ROM arrays can be specified in local or global scope. Global scope will allow host access to the ROM through the PPA. Partial initialization of ROMs is allowed and some values may be specified as don't cares using the #pragma dont_care_code.

```
#pragma dont_care_code 99
const unsigned char F[10][10] = {{99, 20,}, {0, 99, 21},};
```

In the example, location F[0][0] and F[1][1] will be treated as don't care values. Access to uninitialized or don't care locations is permitted, but the value returned by such access is undefined. The linting phase of Synphony C Compiler reports warning on such access.

Synphony C Compiler tool automatically resizes the ROM arrays based on the values specified at the initialization. The ROM array F in the example would be resized to 5 bits. The bitsize of the ROM is not influenced by the don't-care value.

Synphony C Compiler will also try to replace the ROM accesses with their initialization values in the PPA code. If the ROM value is intended to be supplied externally or modified later, this optimization can be suppressed through the preserve_array pragma. The Synphony C Compiler tool will also suppress the automatic bit resizing of ROM arrays under this pragma. However, you can resize the ROM using a bitsize pragma.

```
const unsigned char ext_D[20] = {2, 1, 3, 2, };
#pragma preserve_array ext_D
#pragma bitsize ext_D 7
```

A ROM port is never arbitrated for both synthesizable and external ROMs regardless of the bandwidth. However, sequential or mutually exclusive memory requests can be multiplexed into a single port. Multi-ported synthesizable ROMs are created as many single ported ROMs inside the RTL. For external ROMs, number of memory load ports on the top module is equal to the number of parallel requestors to the ROM. You can create a multi-ported ROM or create many single ported ROMs to be wired to these ports.

Packed and Unpacked Memories

The Synphony C Compiler tool supports the C aggregate type (structs and arrays of structs) and C++ aggregate type (objects and array of objects). Union types are not supported.

Aggregate arrays are mapped onto a collection of independent memories (unpacked) or a single wide memory with one row per aggregate element (“packed” memory).

The Synphony C Compiler tool supports both packed and unpacked memory variables. By default, array variables are unpacked. To generate packed memories, you must specify packed pragma on array variables. In the following example, A results in a packed memory of 64-bit and B results in two unpacked memories of 32-bit each.

```
class s1 {
   public:
      int x1;
      int x2;
}  A[10], B[10];
#pragma packed A
```

Scalar variables of struct type with array fields inside structure are mapped to memories. For example,

```
class s {
   public:
      int x;
      int y[10];
}  r;
```

r.y is mapped to a memory of its own.

- All memories such as local SRAM, internal register-based RAM, external ROM, block RAM, and synthesized ROM can be packed.
- Packed pragma can be specified on struct or class, as shown in the code.

```
typedef class s2 {
   public:
   short x1;
   short x2;
   mytype;
   mytype C[10];
   #pragma packed C
```

- Like bitsize, default struct type is a property of the variable and not the property of the type. Array variables of default struct type results in packed memories.
- The SCC tool allows packed elements of a struct or class. But arrays of such objects cannot be used in synthesizeable code.

Write to the fields of a packed memory is not permitted. To do this, you must create unpacked memories by removing the packed pragma or inserting read-modify-write operations to avoid field writes.

Example 1: Packed Array That Permits Field Writes

The way the following code is written permits writes to fields of the packed array, which is an error:

```
typedef class {
   public:
      int a, b, c;
      int d[10];
} ABC;
ABC abc[20], y2;
#pragma packed ABC
#pragma internal_fast abc.d    //This is an error, cannot
                               //specify any pragma
#pragma internal_fast y2.d
int c;
void foo()
{
   int i,j;
   for(i = 0; i < 10; i++) {
      // Array member RHS occurrence
      c = abc[2*i].b + 5; // Not Error: Reading elements from
                          //packed array automatically reads
                         //the entire packed element
      // Array member LHS occurrence

      abc[2*i].a = i + j + 5; //Error: Writing to fields of packed
                              //array is not permitted.
```

```
      // Array update
      y2.d[i]=i;
      // Array aggregate operation
      abc[2*i + 1] = y2;
   }
}
```

Example 2: Packed Array that Does Not Permit Field Writes

To avoid writing to the fields of the packed array, rewrite the code in "Example 1: Packed Array That Permits Field Writes" on page 8-12 as shown here:

```
typedef class {
   public:
   int a, b, c;
   int d[10];
} ABC;
ABC abc[20], y2;
#pragma packed abc
#pragma internal_fast abc       //Specification of internal_fast on
                                //entire packed array is permitted
int c;
void foo()
{
   ABC y2, tmp2;
   #pragma internal_fast y2.d
   #pragma internal_fast tmp2.d
   int i,j;
   for(i = 0; i < 10; i++) {
      // Array member RHS occurrence
      c = abc[2*i].b + 5;
      // Array member LHS occurrence
      tmp2 = abc[2*i];
      tmp2.a = i + j + 5;
      tmp2.d[i]=i;
      abc[2*i] = tmp2;

     // Array aggregate operation
     abc[2*i + 1] = y2;
   }
}
```

The two preceding code examples also have different memory access patterns:

- In Example 1, without the packed pragma for abc, the array maps to four memories: abc.a, abc.b, abc.c (each 20 elements deep and 32 bits wide) and abc.d (32 bits wide and 200 elements deep). For every iteration abc.a has two writes, abc.b has one read and one write, abc.c has one write and abc.d has 10 writes to different elements of the array.

- In the packed view of Example 2, the array maps to a single 416-bit wide memory (3x32+10x32), and there is one read and two writes to this memory for every iteration.

Memory Arbitration

Synphony C Compiler tool can arbitrate memory ports other than `internal_fast` memory ports. You can specify the maximum allowed ports to a memory using `read_write_ports` pragma and the Synphony C Compiler tool tries to meet the porting requirements. If the constraints cannot be met, the resulting ports are arbitrated. To prevent the implementation of arbitration, use the `no_arbitration` option of the `read_write_ports` pragma. For more information on this pragma, see "read_write_ports Pragma" on page 7-58.

The Synphony C Compiler tool implements multi-stall domain arbitration. Stall domains are PAs and independent stall domain TCABs inside the PAs. Consider a m:n arbitrator where m > n (m requests and n ports for a memory). Whenever the number of requests is greater than n, the arbitrator grants utmost 'n' requests per cycle. Requestors that are not granted access to memory are stalled appropriately to maintain the latency requirements. The arbitrator is called multi-stall domain as it only stalls the stall-domains of the requestors that were not granted access to memory.

The Synphony C Compiler tool uses a fixed priority arbitrator. Priority{arbitrator_port_0} > Priority{arbitrator_port_1}. The tool assumes a default priority of stall domains (PAs and independent stall domain TCABs) for wiring to the arbitrator.

Recommended Use Model for Arbitration

Event simulation is a cycle accurate simulation used to determine the performance of generated RTL for a given FIFO sizing and SRAM porting. The tool uses an event trace produced by C simulation to determine overall RTL runtime, FIFO high water marks (that is, the maximum FIFO lengths reached during simulation), stall cycle counts for each loop, and the presence of deadlock. Event simulation models multi-stall domain arbitration. Event simulation can be used to specify porting constraints on memories and determine the porting that gives acceptable performance. After determining this, the design can be built by specifying the porting constraint using the `read_write_ports` pragma.

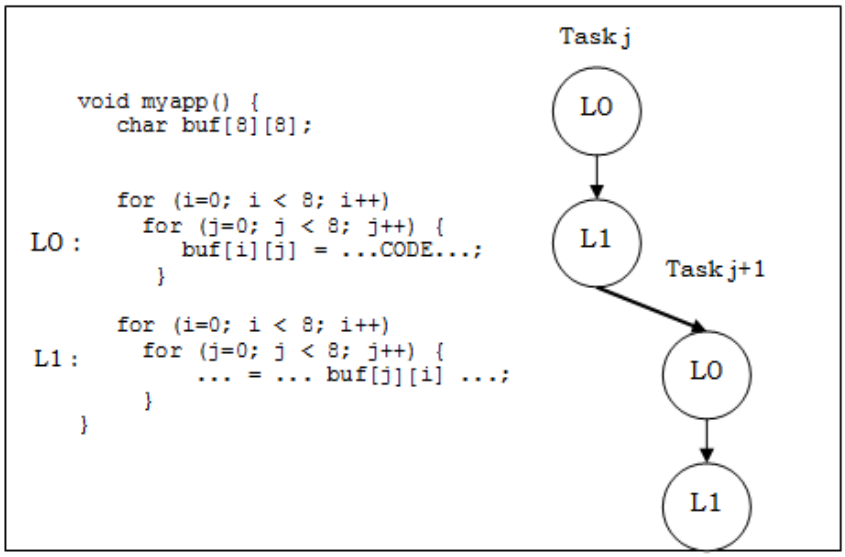
In some cases, it is possible that the final performance numbers that are achieved could be different from the model. This happens mainly as the Synphony C Compiler tool finds a different schedule when the constraints are specified. For more information on arbitration, see "Memory Arbitration" on page 15-21.

Interconnect to Shared Memory

When an external SRAM is shared, multiplexing or arbitration is inserted to yield maximum number of ports required by PAs that access the SRAM concurrently. However, each PA still includes the load and store units required for its own access to the memory.

Banking and Multi-Buffering of Memories

In many applications an array is declared in procedure scope and used as a scratchpad to store temporary values. Loops that share this array must be sequentialized within a task. In addition, they must also be sequentialized across tasks to ensure that the previous task is done with the array before the next task begins using it.



For example, the PPA procedure myapp in the figure uses a procedure-scope declared array buf. Because L0 writes to buf and L1 reads from buf, L0 and L1 are sequentialized within a task. In addition, L1 of the current task must finish reading buf before L0 of the next task can start. Therefore, the tasks are also sequentialized.

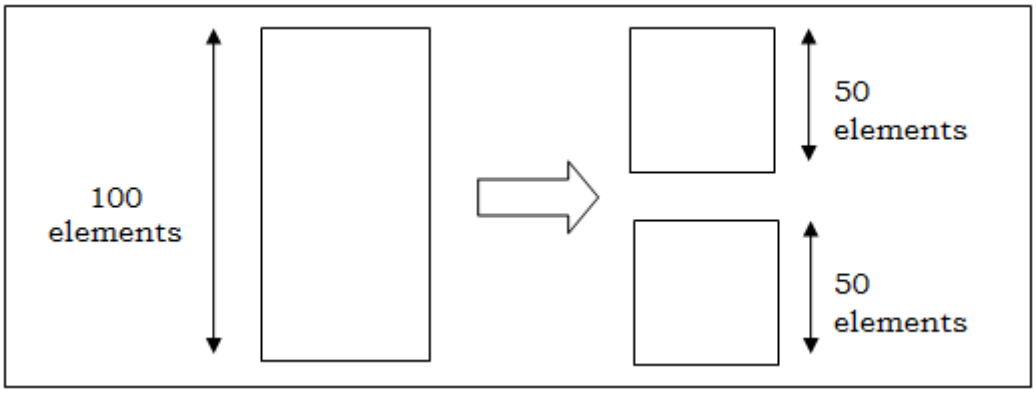
The Synphony C Compiler tool can remove the sequentialization between tasks by multi-buffering the memory. Multi-buffering is an optimization that enables more task overlap by storing the shared array in different physical locations during subsequent tasks. Synphony C Compiler automatically creates multi-buffers for arrays to meet the performance target.

Note that multi-buffering differs from banking. Before proceeding, we first define precisely what we mean by banking and buffering.

Banking

The next figure shows the simplified concept of banking, where a single array with 100 elements is implemented as two SRAM banks of 50 elements each. This kind of

transformation is used to avoid multi-ported SRAM. This improves bandwidth and performance within a single task.



Banking is typically done by hand. For example, to reduce the porting to an array A, you could replace the definition int A[100] with two definitions A0[50] and A1[50].

The tool automatically performs banking if all of the following are true:

- The array is declared in procedure scope.
- The number of banks desired is the extent of the first array dimensions.
- All references to the first array dimension are compile-time constants.

For example, Synphony C Compiler generates two banks for A with this code:

```
void myapp() {
   char A[2][100][64];  int i, j;
   for (i=0; i < 100; i++) {
      for (j=0; j < 64; j++) {
         A[0][i][j] = …;
         A[1][i][j] = …;
         …
         if (i >= 2) …  = A[0][i-2][j] * A[1][i-1][j];
      }
   }
}
```

Banking is not restricted to first array dimension, but is implemented left to right. For example, if the first two dimensions are addressed by a constant, a[2][2][100] results in 4 banks of 100 elements each. However, if the second dimension has variable addressing, the tool implements it as two banks, even if the third dimension is addressed by a constant.

Local Array Banking Example

This is an example of banking:

```
#define MAX_ROWS 128
#define MAX_COLS 128
int rom_coef[3][5];
int image[MAX_ROWS][MAX_COLS];
int ind_i, ind_j;
int ind_rows, ind_cols;
int sum=0;
for (ind_cols=0;ind_cols<MAX_COLS;ind_cols++){
   for (ind_rows=0;ind_rows<MAX_ROWS;ind_rows++){
      #pragma unroll
      for (ind_j=0;ind_j<3;ind_j++){
         for (ind_i=0;ind_i<5;ind_i++){

            sum+=rom_coef[ind_j][ind_i]*image[ind_rows][ind_cols];
         }
      }
   }
}
```

If the ind_j loop is not unrolled as shown in the example, the array is synthesized as a single memory of 15 elements. If the ind_j loop is unrolled, this code is implemented as a banked solution, as shown in the following code.

```
for (ind_cols=0;ind_cols<MAX_COLS;ind_cols++){
   for (ind_rows=0;ind_rows<MAX_ROWS;ind_rows++){
      for (ind_i=0;ind_i<5;ind_i++){
         sum+=rom_coef[0][ind_i]*image[ind_rows][ind_cols];
      }
      for (ind_i=0;ind_i<5;ind_i++){
         sum+=rom_coef[1][ind_i]*image[ind_rows][ind_cols];
      }
      for (ind_i=0;ind_i<5;ind_i++){
         sum+=rom_coef[2][ind_i]*image[ind_rows][ind_cols];
      }
   }
}
```

The Synphony C Compiler tool recognizes that the first dimension is always a constant index and banks the rom_coef array into three smaller arrays of 5 elements each, thus implementing a multi-banking solution for this array. The tool can now further optimize each of the slices based on the usage pattern, like reducing ports for example, which it could not do without the banked implementation.

The tool only implements a banked solution when the outer dimensions are accessed with constant indexes. In the weighted sum calculation, this would only happen in the following cases:

- Only the ind_j loop is unrolled. This coding style generates 3 banks of 5 elements each for memory rom_coef.
- Both ind_j and ind_i loops are unrolled. This coding style generates 15 individual scalars for the rom_coef.

Any other combination of unrolling (for example, unroll ind_i only), does not result in scalarization because the outer dimension (ind_j) accesses will not be compile time constants.

User-Controlled Banking

Local arrays are banked automatically, based on the usage of constant reference values for the outermost index. Synphony C Compiler also supports the banking of array parameters and global arrays. However, for these object types, you must explicitly specify the desired level of banking by using the `memory_structure` pragma. In the absence of this pragma, no banking of array parameters and global arrays is performed.

For example, the following statements create a three-level array with the first level banked:

```
int x[2][2][4];
#pragma memory_structure x bank
```

The pragma invokes first-level banking, producing two banks, `x_0[2][4]` and `x_1[2][4]`.

To invoke second-level banking beyond the first level, repeat the `bank` keyword:

```
int x[2][2][4];
#pragma memory_structure x bank, bank
```

This pragma invokes first-level and second-level banking, producing four banks: `x_0_0[4]`, `x_0_1[4]`, `x_1_0[4]`, and `x_1_1[4]`.

The number of `bank` keywords can be no more than the number of array dimensions minus 1. Multiple `bank` keywords invoke banking for each array dimension in sequence, starting from the outermost index.

You can also control banking in this manner for locals, with both constant and nonconstant addressing.

You can use the `memory_structure` pragma to invoke banking of the outermost dimension or dimensions, even when nonconstant index references are used in those dimensions. In that situation, the tool introduces an if-then-else structure to reference the banks. For example,

```
void tcab (int a[3][10])
{
#pragma memory_structure a bank
    tmp = a[k][i];
}
```

The tool implements the TCAB array parameter as follows:

```
if (k == 0) load_a = a[0][i];
else if (k == 1) load_a = a[1][i];
else load_a = a[2][i];
tmp = load_a;
```

This structure is later inferred as parallel hardware at the time of scheduling.

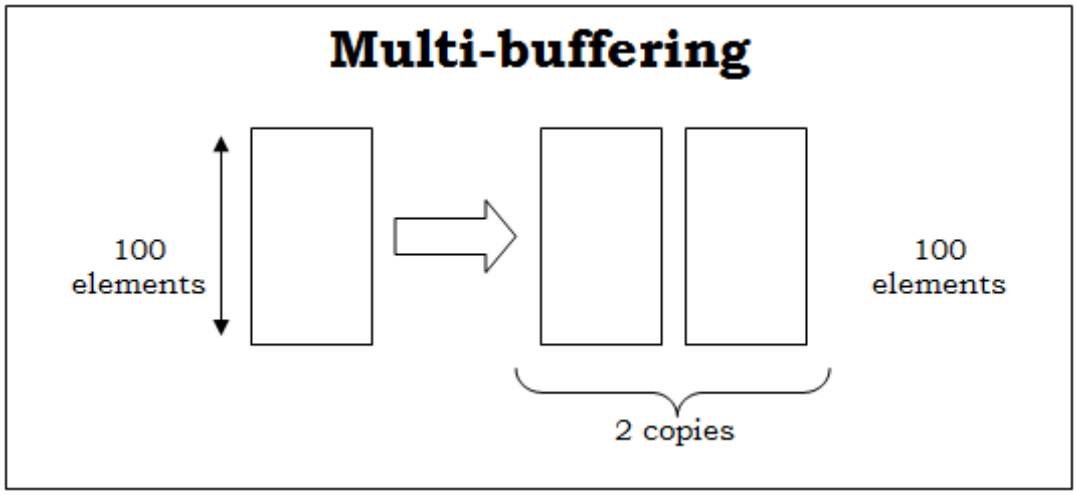
The if-then-else construction for a nonconstant index requires information about the array bounds. If no bounds are provided, the tool will error out.

Banking must be consistently applied across TCAB boundaries. In other words, banking of a TCAB parameter must be consistent with the actual array. This is also true for global banking.

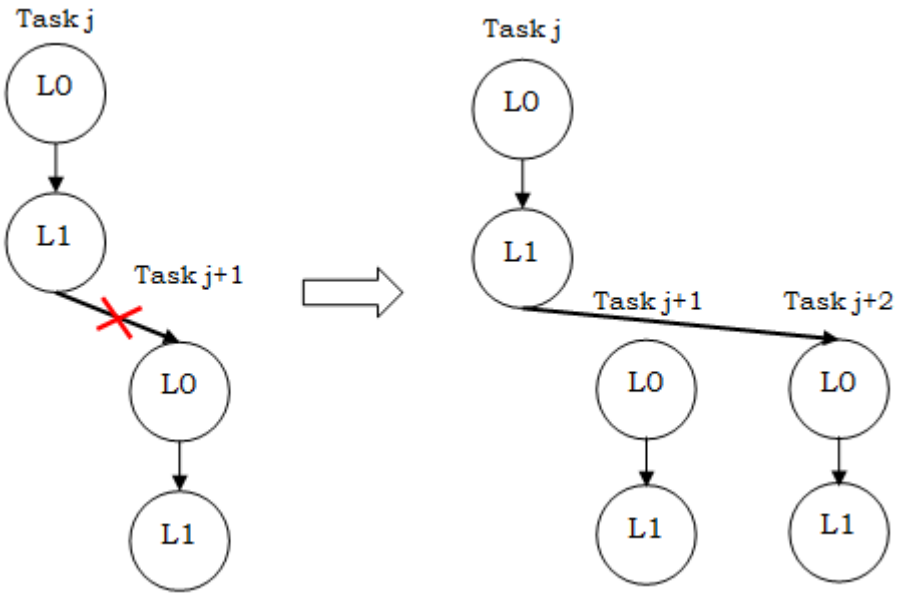
Banks inherit the properties you specify for the whole array. There are no user controls available to give different properties to different banks. Passing array slices to TCABs is not allowed, even if each slice can be mapped to a different memory bank.

Multibuffering

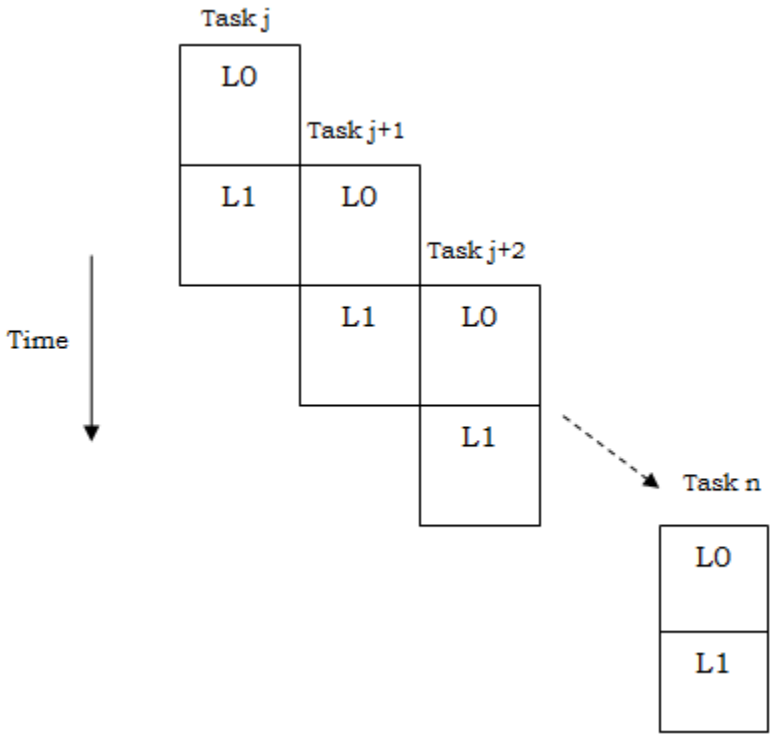
The next figure simplifies the concept of multi-buffering, where the physical storage for an array with 100 elements is replicated into 2 SRAMs of 100 elements each. Use this kind of transformation to achieve task overlap.



For example, in the myapp code shown in "Banking and Multi-Buffering of Memories" on page 8-15, loops L0 and L1 share an array. By double buffering the array buf, the dependence between tasks can be removed as shown in the following figure.



If Synphony C Compiler allocates two SRAMs to store the buf array from alternate tasks to meet performance, the execution timeline for multiple invocations of myapp looks like the following diagram.



The Synphony C Compiler tool does not limit this optimization to double buffering. Many copies of physical SRAM can be used to store the shared array from different tasks, so that parallelism is exploited as much as possible.

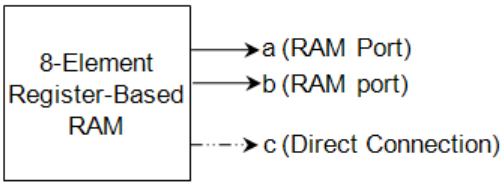
Scalarization

Scalarization is an optimization technique, which allows arrays in your code to be dissolved and implemented as wires or registers. This gives you additional control over how the array is implemented. Scalarization allows each of the smaller arrays to be analyzed and optimized individually, or individually eliminated if unused.

Scalarization requires constant references. All accesses to the array must be made with compile time constant references. Use the scalarize pragma to do this; it automatically enforces constant addressing.

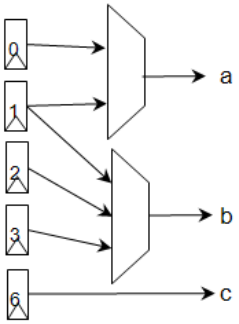
This figure shows code that is implemented as a RAM because the scalarize pragma is not specified. Each RAM port can access all eight elements:

```
int x[8];
...
if (i < 2) a = x[i];
if (j > 0 && j < 4) b = x[j];
c = x[6];
```



When the scalarize pragma is applied, the array is scalarized, and addressing is optimized:

```
int x[8];
#pragma scalarize x
...
if (i < 2) a = x[i];
if (j > 0 && j < 4) b = x[j];
c = x[6];
```



If a pragma is applied to an array that is scalarized, the pragma is automatically propagated to the scalarized elements. For example,

```
int a[4];    //Array that gets scalarized due to usage pattern
#pragma bitsize a 10
```

The Synphony C Compiler tool recognizes that the array gets scalarized and applies the bitsize pragma to all the elements of array a after scalarization. You should apply the pragma only on the array name and not on the scalarized components. The following statement would be an example of wrong specification.

```
#pragma bitsize a[0] 10
```

See the following for details:

- Pragmas for Scalarization
- Scalarization of Global Arrays
- Hardware Implementation of Scalarized Arrays

Pragmas for Scalarization

The SCC tool provides some pragmas that can influence scalarization:

scalarize	Use to enable or disable scalarization. See "scalarize Pragma" on page 7-68 for syntax details.
preserve_array	Use this to preserve the array and disable everything: array: scalarization, banking, ROM value substitution, value-based ROM bitsizing, and the copying in or out of reference parameters. preserve_array does not override force_constant_addressing, because force_constant_addressing applies to the references, not the array, while preserve_array controls the properties of the array itself. See "preserve_array Pragma" on page 7-57 for more information.
force_constant_addressing	Use this when you want to retain the memory but generate efficient addressing logic based on referenced addresses. When combined with preserve_array, it makes all references of an array constant while preserving the physical structure as a memory. This is most commonly used with internal_fast arrays. When used without preserve_array, it scalarizes the array, just like the scalarize pragma. See "force_constant_addressing Pragma" on page 7-28 for more information.
internal_fast internal_blockram user_supplied user_supplied_fpga	Use this to specify hardware implementations when there is no scalarization. If the array is automatically scalarized, these pragmas are ignored. If there is a preserve_array pragma, the pragmas are processed and the hardware implemented. If force_constant_addressing and preserve_array are both specified, the references are converted to constant references, but the array is preserved and not scalarized. See "internal_blockram Pragma" on page 7-32, "internal_fast Pragma" on page 7-33, and "user_supplied, user_supplied_fpga Pragmas" on page 7-72 for more information about these pragmas.

Scalarization of Global Arrays

When the scalarize pragma is specified for an array, all the array references have constant addressing and the array has host access. The host has a contiguous address space relative to the base address in the memory map, but the scalars may be placed anywhere in the memory map.

For direct access arrays, the tool creates one port for each input and output. If a TCAB accesses a global array, scalarization is treated independently from addressing in the container.

Hardware Implementation of Scalarized Arrays

The hardware generated from a scalarized array varies, depending on where the array was declared:

Array Declaration	Hardware Implementation
Procedure	Internal wires or registers
TCAB/PPA parameter	Ports for each element that is read, and each element that is written
Global and memory-mapped	Memory map entries for each element that is read and each element that is written
Global and direct ports	Ports for each element that is read and each element that is written

Stream Support

Streaming interfaces are very common in embedded architectures. Some common streaming interfaces are described here:

- Streams are typically used to pass data between two PAs (loops), but they can also pass data within the same PA if both ends of the stream are in the same loop. The following examples describe some other scenarios:
- Many system architects have an existing IP block, like a DMA controller, that performs a sequence of global memory accesses using the system bus and producing or consuming a stream of data based on pre-configured addressing patterns. Such IP blocks are already optimized for the global memory organization and for the overall demands on the system bus. The resulting flow-controlled stream interface provides new IP blocks generated by the Synphony C Compiler tool with a convenient mechanism to access global memory without having to be aware of the details of the system architecture.

- Another common use of streams is to hook multiple hardware blocks together in a pipeline. The data is produced and consumed within the pipeline, so there is no need to use memory to store intermediate results. Instead, the data can flow directly from one block to the next block through the streaming interface.

The Synphony C Compiler tool supports two types of streams: external and internal.

External streams	Transmit data between the hardware being synthesized by the Synphony C Compiler tool and any other block in the system.
Internal streams	Stream data within the hardware synthesized by the SCC tool. Typically, you use internal streams for streaming data between PAs inside the PPA synthesized by the Synphony C Compiler tool. However, you can also use internal streams for streaming data within a PA.

You can define these streams using global functions, or by using the SCC streaming class. The latter approach is recommended for C++ programs. See the following topics for details:

- Class-Based Stream Definition
- Function-Based Stream Definition
- External Streams
- Internal Streams
- User Controls for Streaming Interfaces

Class-Based Stream Definition

The SCC tool provides a standard streaming class called scc_stream<T> which you can use to implement streams instead of functions. The global function approach for internal and external streams is still supported for C designs, but the class object method is preferred for C++ designs.

The SCC streaming class provides blocking or non-blocking stream methods; blocking methods can cause a stall and non-blocking methods do not stall but return a Boolean value to indicate the success of the transaction.

For details of the streaming class, see the following topics:

- scc_stream<T> Class Definition
- scc_stream<T> Streaming Methods
- User-Defined Simulation Methods

scc_stream<T> Class Definition

This is the basic class definition for the SCC streaming class:

```
template <typename T>
class scc_stream {
public:
   // constructors
   explicit scc_stream( void );
   virtual ~scc_stream();
   // tlm get interface
   virtual T get( void );
   virtual bool nb_get( T& );
   virtual bool nb_can_get( void ) const;
   // tlm peek interface
   virtual T peek( void ) const;
   virtual bool nb_peek( T& ) const;
   virtual bool nb_can_peek( void ) const;
   // tlm put interface
   virtual void put( const T& );
   virtual bool nb_put( const T& );
   virtual bool nb_can_put( void ) const;
}
```

The class has nine streaming methods to support blocking and non-blocking streaming, which are described in "scc_stream<T> Streaming Methods" on page 8-27. In addition, the implementation has a constructor, destructor, and various functions to maintain the state of the stream FIFO.

The following declaration creates a stream object called x, with all nine stream methods defined. The tool only synthesizes stream methods that are actually used.

```
struct Complex {
   int r,i;
};
scc_stream<struct Complex> x;
```

Note the following:

- Width is inferred from the data type associated with the stream.
- You cannot use the bitsize pragma with the stream methods or objects.
- You can use the fifo_length pragma with an scc_stream object. See "fifo_length Pragma" on page 7-27 for the syntax.

scc_stream<T> Streaming Methods

The scc_stream<T> class supports blocking and non-blocking stream transactions. Accordingly, it supports nine stream methods for get, put, and peek operations. This table shows them listed according to blocking or non-blocking method:

Method	Description
Blocking Methods	
get	Waits for stream to have data, returns the data, and then dequeues data.
put	Waits for stream to be empty, and then queues data.
peek	Waits for stream to have data, and returns that data.
Non-Blocking Methods	
nb_get	If the stream contains data, returns the data, dequeues the data and returns true; otherwise returns false.
nb_put	Returns true and queues data if the stream is ready; otherwise it returns false.
nb_peek	Returns data and true if there is data in the stream; otherwise it returns false.
nb_can_get	Returns true if the stream has data; otherwise it returns false.
nb_can_put	Returns true if the stream is ready; otherwise it returns false.
nb_can_peek	Returns true is stream has data; otherwise it returns false.

Note the following:

- The tool synthesizes any of the nine scc_stream<T> methods listed in the table, and directly translates them to the associated hardware interface. For synthesis, the scc_stream object is a handle to the stream and its interface; any objects of the class or a derived class are stateless.
- All nine methods can also be called by the testbench code.

User-Defined Simulation Methods

You can override the functionality of one or more of the interface functions for simulation or testbench integration, without affecting synthesis. To do this, you create a derived class from scc_stream, and override one or more of the streaming methods with your own implementations. The prototypes must match the requirements of the corresponding stream methods, but they must not be virtual:

```
// tlm get interface
T get( void );
bool nb_get( T& );
bool nb_can_get( void ) const;
// tlm peek interface
T peek( void ) const;
bool nb_peek( T& ) const;
bool nb_can_peek( void ) const;
// tlm put interface
void put( const T& );
bool nb_put( const T& );
bool nb_can_put( void ) const;
```

The basic form is as follows:

```
template <typename T>
class CLASSNAME : public scc_stream<T> {
   // member variables (not synthesizable)
public:
   // constructor/destructor (ignored by synthesis)
   // helping methods that access members (not synthesizable)
   // functions that do not access any
  // members (inlined for synthesis)
  // One or more of the 9 streaming methods
  // (direct translation to HW ports for synthesis)
};
```

Note that class member variables of the derived class are not synthesized.

Blocking Streams Example

The following example defines two counters that are both synthesized. The constructor, input_stream, and output_stream methods are inlined. The calls to put and get on the _local variable (scc_stream<T> variable) are synthesized as hardware ports.

```
template <typename T>
class datain_with_counter  {
   scc_stream<T> _local;
   int counter_in, counter_out;
public:
   datain_with_counter() : counter_in(0), counter_out(0) {}
   T input_stream() {
      counter_in++;
      return _local.get();
   }
   void output_stream(const T &val) {
   counter_out++;
   local.put(val);
   }
};
```

Non-Blocking Streams Example

The following simple example illustrates how to specify non-blocking streams in the code. It does not have any significant functionality, but shows a loop nest that uses non-blocking streams to implement a drainable loop.

```
#include <scc.h>
// External streams
scc_stream<int> yin, yout;
void scc_procedure( )
{
    int i, j;
    // Read from external input stream 'yin' and
    // write to external output stream 'yout'.
    for (i= 0; i < 100;) {
        int temp;
        if(yin.nb_get(temp)) {
            temp *= temp;
            yout.put(temp);
            i++;
        }
    }
}
```

The loop reads data from one stream `yin` (only when data is available) and writes data to another stream `yout`. This example increments the loop counter only when there is a successful read from the stream. The usage of `yin.nb_get(temp)` ensures that the loop does not stall when there is no new data. However, usage of `yout.put(temp)` introduces a blocking call and the loop stalls if the `yout` FIFO is full.

Function-Based Stream Definition

Although you can use global functions to define streams in C and C++ designs, it is recommended that you use the scc_stream class for C++ designs ("Class-Based Stream

Definition" on page 8-25). Using function-based stream definitions hardwires the stream names, which makes code encapsulation and reuse harder, but it is the easiest method to use. It is also the only method you can use in C designs.

External Streams

The following sections describe two ways to define external streams. Use the class-based approach unless you are working in plain C, or if code encapsulation and reuse are not a consideration. You cannot define non-blocking streams unless you use the class-based approach.

See User Controls for Streaming Interfaces for additional information.

Class-Based External Stream Definition

This approach uses the scc_stream class described in "scc_stream<T> Class Definition" on page 8-26. You can use it to implement blocking or non-blocking streams.

External Blocking Streams

Use the scc_stream<> class to define an external stream and pass stream objects to your PPA function. The tool then instantiates a FIFO for the external stream. You must also specify how the driver code is to handle writing and reading this FIFO. There are two ways to do this:

- One way is to do all the loading before the PPA is called, and all the unloading after it completes. In the driver code, you preload input streams and then read them back using the put() and get() methods respectively. (See "scc_stream<T> Streaming Methods" on page 8-27 for details.) This method assumes you know how much data needs to be put into the input. On the output side, you either need to know how much data is expected, or you can use nb_can_get() to test the FIFO.
- If you want the code to do something other than reading or writing the scc_stream<> FIFO while the PPA is running, you must derive a new class from scc_stream<> and write your own put() and/or get() functions.

External Non-Blocking Streams

You can only implement non-blocking streams in C++, using the scc_stream class. By their nature, non-blocking streams affect simulation behavior, because their data can vary.

- verify -golden
  The default verify -golden behavior is based on the actual data in the internal FIFO, so nb_can_get() returns true if there is data and false if not. get() errors out if there is no data in the FIFO. For designs with non-blocking streams, you should test with different behaviors. For example, have nb_can_get() return false for some calls to mimic

unavailability of data in a stream, even though you might have pre-loaded the FIFO in golden. To do this, derive a class from scc_stream<> and override the methods.

Function-Based External Stream Definition

This method is the easiest to use, but it does not offer the flexibility of reuse or encapsulation, because the names are hardwired.

For external streams, you must explicitly write the C or C++ procedures and they must functionally emulate receiving from and transmitting to external streams. One end of the FIFO macro must be in the synthesizable code, and the other end must be in the testbench. The Synphony C Compiler tool synthesizes a standard streaming interface for each external stream in your code.

Internal Streams

The following sections describe two ways to define external streams. Use the class-based approach unless you are working in plain C, or if code encapsulation and reuse are not a consideration. You cannot define non-blocking streams unless you use the class-based approach.

- Class-Based Internal Stream Definition

See "User Controls for Streaming Interfaces" on page 8-32 for additional information.

Class-Based Internal Stream Definition

The class-based approach lets you define blocking or non-blocking internal streams in C++ designs.

Use scc_stream<> and call the put() and get() methods to define internal streams. See "scc_stream<T> Class Definition" on page 8-26 and "scc_stream<T> Streaming Methods" on page 8-27 for details.

For non-blocking streams, note that the tool does not support the playback of non-blocking transactions on internal streams.

Passthru and Non-Passthru FIFOs

There are two kinds of FIFOs the tool can create, passthru and non-passthru.

- Passthru FIFOs have timing paths between the read and write ends of the FIFO, and a depth of 0.
- Non-passthru FIFOs have no timing paths between the read and write ends of the FIFO.

The tool only instantiates non-passthru FIFOs for the following cases:

- For FIFOs intercepting non-blocking stream ports
- For boundary FIFOs created to insulate the timing of external stream ports.

User Controls for Streaming Interfaces

The tool provides you with some controls over the stream interface, which are described in this section.

FIFO Name

The FIFO name must be a valid C identifier.

FIFO Length

You can specify the length of the FIFO using the fifo_length pragma on the stream object, as shown, or on the stream procedure name.

```
scc_stream<int> A;
#pragma fifo_length A 10
```

The default FIFO length (number of storage elements) is zero. Length zero indicates a simple boundary register between PAs, as well as a register to handle stalling cases. Increasing the length will add FIFO storage elements in addition to the boundary register. You can use event simulation to simulate various sizes to optimize the overall cost; however you cannot use it for non-blocking streams.

FIFO Width

Apply the bitsize pragma to the data type of the stream class object. For example:

```
typedef int int17;
#pragma bitsize int17 17
scc_stream<int17> B;
```

FIFO Implementation

The default implementation of FIFO storage is register-based for ASIC designs. For FPGA devices, you can specify the implementation of FIFO storage using the internal_fast or internal_blockram pragmas, or by setting a threshold with fifo_synthesis_threshold and letting the tool infer the FIFO implementation. For more details on Tcl commands and pragmas, see Chapter 3, “Synphony C Compiler Commands” and Chapter 7, “Pragmas and Constraints.”

Non-blocking internal streams can result in race conditions in certain situations. In these cases, the FIFO is implemented as a non-pass through FIFO. A non-pass through FIFO has a depth of at least one, which means that it has at least one register. You can check if non-pass through FIFOs were implemented by checking the report_summary file.

FIFO Sharing

You can share FIFO hardware across different PAs by using the same stream name. Note that the Synphony C Compiler tool sequentializes the execution of PAs which read from or write to the same stream. You can avoid this by suppressing stream dependence analysis between PAs within the same task or across different tasks using these two pragmas:

```
#pragma no_inter_loop_stream_analysis streamFunctionName

#pragma no_inter_task_stream_analysis streamFunctionName
```

For more information on these pragmas, see “no_inter_loop_stream_analysis Pragma” on page 7-42 and “no_inter_task_stream_analysis Pragma” on page 7-46.

Struct Type Streams

Streams of struct type or class type are always packed. The stream is as wide as the size of the class; in the following example, it as wide as s1.

```
class s1 {
   public:
   int x1[2],x2;
   class {
      public:
      int y1,y2;
   } x3[10];
   } a, b, c[100], d[10];

FIFO (X, class s1);
#pragma packed c
scc_stream_output_X(a)
scc_stream_output_X(b)
scc_stream_output_X(c[i])
scc_stream_output_X(d[j])
a = scc_stream_input_X();
b = scc_stream_input_X();
c[i] = scc_stream_input_X();
d[j] = scc_stream_input_X();
```

The next example uses the C++ scc_stream class interface:

```
class s1 {
   public:
   int x1[2],x2;
   class {
```

```
        public:
        int y1,y2;
    } x3[10];
} a, b, c[100], d[10];
#pragma packed c

scc_stream<class s1> fifoX;

bool status;
fifoX.put(a);
fifoX.put(c[i]);
status = fifoX.nb_put(b);
status = fifoX.nb_put(d[j]);
a      = fifoX.get();
c[i]   = fifoX.peek();
status = fifoX.nb_get(b);
status = fifoX.nb_peek(b);
```

Passing scc_stream Objects to Functions

You can pass C++ scc_stream objects to TCABs or inlined functions, but you can only pass them by reference. This means you cannot pass scc_stream objects by value, and the following code is invalid:

```
int func_A(scc_stream<int> x) {
   ... = x.get();
}
```

You cannot copy scc_stream objects, so the following example is also invalid:

```
scc_stream<int> x,y;
...
x = y;
```

You can only pass scc_stream objects to TCABs or inlined functions by reference, as shown:

```
int func_A(scc_stream<int> &x) {
   ... = x.get();
}
```

Passing Stream Functions to Other Functions

In C designs, use pragmas to pass streams as parameters to TCAB or inline functions, as shown:

```
int ext_stream_input_yin ();
#pragma input_stream ext_stream_input_yin
…
int TCAB_A (int (*ext_stream_input_yin) ())
{
   #pragma input_stream ext_stream_input_yin
   // pragma required for formal parameter
   …
}
```

In C++ designs use scc_stream objects to pass a parameter to a TCAB:

```
void TCAB_A(scc_stream<int>& in)
   {
      ...
      int x = in.get();
      ...
   }
   void TCAB_B(int val, scc_stream<int>& out)
   {
      bool status;
      ...
      status = out.nb_put(val);
      ...
   }
```

Mutual Exclusion

There are different ways to indicate mutual exclusion:

- Parallel Hardware
- Switch Case Syntax
- Full-Case Syntax
- The scc_assert(0) Construct

Parallel Hardware

Within a PA, the Synphony C Compiler tool can generate parallel hardware if it can prove that the conditional blocks are mutually exclusive. This can improve timing on some designs that had mutually exclusive conditional blocks. The following are some examples of mutually exclusive conditional blocks.

Example 1

```
for (i = 0; i < 10; i++) {
```

```
     if (X == 0) {
          a[i] = b[i];
     }
     else if (X == 1) {
          a[i] =c[i];
     }
     else if (X == 2 ) {
          a[i] = d[i];
     }
}
```

In the example, since only one of the conditions (0==x), (1==x), (2 ==x) will be true on a given iteration, these conditions are evaluated in parallel instead of sequentially in a nested fashion. This would produce the same result while reducing the control flow depth.

Example 2

```
for (i = 0; i < 10; i++) {
   switch (x) {
   case 0:
      a[i] = b[i];
      break;
   case 1:
      a[i] =c[i];
      break;
   case 2:
      a[i] = d[i];
      break;
   }
}
```

Switch Case Syntax

The following are some guidelines for using switch case syntax to obtain efficient hardware:

- All case labels must be at the highest scoping level inside the switch statement. This means that the “case statements” within a switch-case block should not be enclosed in an if-block or any other block. The following example illustrates this point.

  Note the if-block around the following example. This code may not result in parallel hardware:

```
for (i = 0; i < 10; i++) {
   int temp;
   temp = 5;
   switch (x) {
      case 0:
      case 1:
         a[i] = b[i];
         break;
      case 2:
         a[i] =c[i];
         break;
      if (temp == 5) {
      case 3:
         a[i] = d[i];
         break;
      }
   }
}
```

- Case blocks must end with an unconditional break for generating parallel hardware. This point is illustrated in example 2. In the following examples, Synphony C Compiler would not generate parallel hardware.

  In the following example, the condition, if (i == (packetlen_delay>>4))", can be true if i is 1 and (packetlen_delay >> 4) is 1. Since both the conditions if (i == (packetlen_delay>>4) and if (i == 1) cannot be evaluated together to decide which block within the condition has to be executed, the example would result in a priority encoder kind of a structure.

```
if(i==0) {
   packetlen = scc_stream_input_plen2();
}
else if (i == (packetlen_delay>>4)) {
   packetlen_delay = 32;
}
else if (i == 1) {
   packetlen_delay = packetlen;
}
```

  Similarly in the example 5, the conditions (p < q), (r < t) and (0 == x) can all be true. Therefore, parallel hardware would not be generated.

```
for (i = 0; i < 10; i++) {
   if (0 == x) {
      a[i] = b[i];
   }
   else if (p < q) {
      a[i] =c[i];
   }
   else if (r > t) {
      a[i] = d[i];
   }
}
```

Full-Case Syntax

The term “full-case” comes from the synthesis community and defines an analogous view in C. Say that statement s1 is a switch statement. We say that s1 is full if at least one of the blocks within s1 executes whenever s1 executes. Therefore, if it is possible that s1 could execute but none of the blocks within s1 would execute, then s1 is not full.

Knowledge that a statement is full-case can potentially improve QoR. For example, consider the following:

```
for (i=0; i < 10; i++) {
   if (mode2) {
      switch (x[i]) {
         case 0: y = a[i]; break;
         case 1: y = b[i]; break;
         case 2: y = c[i]; break;
      }
      z[i] = 3*y;
   }
   else z[i] = x[i];
}
```

Without any additional information, the Synphony C Compiler tool must assume that x[i] could take on a value other than 0, 1, or 2 when the switch statement executes. Therefore, on a given iteration, y may not be defined in the switch statement. This means that the Synphony C Compiler tool must create a register for y to hold its value from a previous iteration (or as a live-in to the loop). In addition, the multiplexer on the input to the multiplier must be increased from 3 inputs to 4 (a[i], b[i], c[i], and the register for y). If you had a way of specifying that the switch statement was full-case, then the Synphony C Compiler tool could reduce this hardware.

A switch-case statement containing a default block is trivially full since the default block always executes if no other case condition is true. Therefore, one mechanism to turn a non-full switch-case into a full switch-case is to just add a default-block. However, you must also let Synphony C Compiler know that the default block is unreachable to obtain the QoR benefits.

A new Synphony C Compiler function called scc_assert, is added for this purpose. It has no return value and always takes 0 as its single argument. It has the same semantics as the C macro assert( ). The scc_assert function requires the `scc.h` file, so it must be specified when you use this function.

```
#include "scc.h"
for (i=0; i < 10; i++) {
   if (mode2) {
      switch (x[i]) {
         case 0: y = a[i]; break;
         case 1: y = b[i]; break;
         case 2: y = c[i]; break;
         default: scc_assert(0);
      }
      z[i] = 3*y;
   }
else z[i] = x[i];
}
```

The presence of the default block makes the switch statement full-case. The scc_assert(0) call informs the Synphony C Compiler tool that the default block is unreachable, and therefore the hardware for variable y can be optimized.

Note that specifying the switch statement as full-case does not restrict the values of x[i] to be `0, 1,` or `2`. On the contrary, x[i] can take on any value when `mode2` is false. To emphasize, a switch statement is full if at least one of the blocks within the switch statement executes whenever the switch statement executes. In this example, the switch statement only executes when `mode2` is true.

In addition, it is important to understand that specifying a full-case does not necessarily eliminate registers and reduce hardware. For example:

```
#include "scc.h"
for (i=0; i < 10; i++) {
   if (mode2) {
      switch (x[i]) {
         case 0: y1 = y2 = a[i]; break;
         case 1: y1 = y2 = b[i]; break;
         case 2: y1 = c[i]; break;
         default: scc_assert(0);
      }
      z[i] = y1*y2;
   }
else z[i] = x[i];
}
```

In this example, the full-case allows the Synphony C Compiler tool to optimize away the register for y1 but not y2. As y2 is not defined under case 2, the Synphony C Compiler tool must still create loop-carried state for it. As a result, full-case is most beneficial for variables that are defined in every block of the switch statement.

One of the benefits of using the scc_assert(0) mechanism is that it is not restricted to switch-case statements. It can be used to tell the Synphony C Compiler tool about the reachability of any block in the code. The obvious place to use this is with if-then-else statements in which one of the blocks in unreachable. For example:

```
#include "scc.h"
if(i==0) {
   packet len = scc_stream_input_plen2();
}
else iffy == (packetlen_delay>>4)) {
   paedogenetically = 32;
}
else if (i == 1) {
   paedogenetically = packetlen;
}
else scc_assert(0);
```

This example cannot be expressed with switch-case syntax since the condition i==(packetlen_delay>>4) depends on an scc_stream_input call.

The scc_assert(0) Construct

The scc_assert(0) construct indicates that the block containing the expression is not reachable. It has the same semantics as the C macro assert( ).

The scc_assert construct requires that the `scc.h` file be specified.

The scc_assert(0) construct is not mapped onto any kind of synthesizable artifact (like error hardware). If this specification is incorrect, and the block is reached during a C simulation, the Synphony C Compiler tool fails with an error. However, such failure is again dependent on the test coverage. The synthesized hardware can fail in unpredictable ways if simulations are not comprehensive enough (for example, some functional unit might be shared and have two simultaneous requests without arbitration).

Pipeline Flushing

To achieve high performance, the Synphony C Compiler tool builds a pipeline of operations. At any given cycle, there could be many loop iterations in flight. Each pipeline is in its own stall domain and any stall condition causes the whole pipeline to stall. Due to constraints on the external environment, it is often necessary to flush the pipeline of old iterations prior to issuing an operation that might cause a stall. The Synphony C Compiler tool provides you with pragma flush that can be used to flush the pipeline as explained for some of the scenarios.

- Avoiding pipeline stalls across outer loop iterations

  Consider the following piece of code.

```
for(rows=0; rows < 720; rows++) {
   for(cols=0; cols < 1280; cols++) {
      x = scc_stream_input_in();
      filter = x * weight;  // assume latency 4 multiplier
      scc_stream_output_next_block(filter);
      }
}
```

The example inputs a 720p video (720x1280 pixels), does a one-tap filter, and then outputs the result to the next block in a video pipeline. The input is received in bursts of 1280 consecutive values and the input stream could stall waiting for the next burst of 1280 values. As the inner loop is not flushed before the next iteration of the outer loop, the loop stalls before the 1280 values are sent to the downstream block (the downstream block expects the output to be burst of 1280 consecutive values). We can add #pragma flush to the inner loop to ensure that the pipeline is flushed before the next iteration of the outer loop as:

```
for(rows=0; rows < 720; rows++) {
   #pragma flush
   for(cols=0; cols < 1280; cols++) {
   x = scc_stream_input_in();
   filter = x * weight;  // assume latency 4 multiplier
   scc_stream_output_next_block(filter); }
}
```

The Synphony C Compiler tool adds the necessary cycles to flush the inner loop and the downstream block would be able to receive the burst of 1280 values.

- Relaxing constraints on feedback paths

  Consider the following piece of code segment:

```
…
unsigned int x, y;
for (i=0; i < 10; i++) {
   x = scc_asm_MPYL4C_W(x,y); // 4 cycle multiplier
   for (j = 0; j < 1000; j++) {
      x = x + scc_stream_input_in();
   }
}
…
```

  In the loop, the latency of the multiplier constrains the achievable Initiation-Interval (II). When j==0, the result of multiplication is used in the addition. Therefore, the addition operation must be scheduled at least three cycles after the multiplication. When j==999, the result of the addition is used in the multiplication of next iteration. Given that the addition must be scheduled 3 cycles after multiplication, we can achieve an II>=3 for the loop. This constraint on pipelining due to operations in the outer loop can be relaxed by flushing the pipeline at the end of inner loop iterations. This can be accomplished by using the pragma flush as:

```
for (i=0; i < 10; i++) {
   x = scc_asm_MPYL4C_W(x,y); // 4 cycle multiplier
   #pragma flush
   for (j = 0; j < 1000; j++) {
      x = x + scc_stream_input_in();
   }
}
```

The Synphony C compiler tool introduces idle cycles necessary to completely flush the pipeline and we can get an II=1 for the loop after adding #pragma flush.

- Creating opportunity for sharing using pragma flush

  A common occurrence in some algorithms doing frequency/time or frequency/spatial domain transformations (example, DCT) is to handle DC and AC components differently. Sometimes this is expressed as a first step for DC and then a loop for AC:

```
for (blocks=0; blocks < nBlocks; blocks++) {
   encode_DC();
   for(i=1;i<64;i++) {
      encode_AC(i);
   }
}
```

  Suppose that encode_DC( ) and encode_AC( ) share a memory -- DC value is written to element 0 and AC values to elements 1-63. In addition, they may have common operations such as multiplications, divisions, etc.

  Assume that encode_DC( ) and encode_AC( ) each make one read and one write access to a shared memory m[ ], and that each performs one multiplication. Furthermore, assume that performance target is Initiation-Interval of 1 for the loop.

  In the first iteration, the loop makes four memory accesses for encode_DC( ) and encode_AC( ). It also performs two multiplications on the first iteration. At II=1 we require four ports to m[ ]. We can modify the code by adding #pragma flush as:

```
for (blocks=0; blocks < nBlocks; blocks++) {
   encode_DC();
   #pragma flush
   for(i=1;i<64;i++) {
      encode_AC(i);
   }
}
```

  This coding will have one load, one store and one multiply on each iteration. These will be shared between the encode_DC( ) and encode_AC( ) functions.

  The compiler does not consider flush cycles when it determines initiation intervals for each loop to achieve the throughput target (task_II). You must verify the achieved throughput by checking simulation.

Multi-Level Break Statements

The Synphony C Compiler tool supports normal C or C++ break and continue statements for applications that require data dependent termination or continuation of loops. In addition, it also provides a mechanism to break from multiple loops in a loop nest.

Syntax and Semantics

A multi-level break requires two function calls: scc_break_loop(i) within the loop-nest and scc_end_loop(i) after the loop-nest. The following are requirements for scc_break_loop or scc_end_loop:

- The argument to scc_break_loop and scc_end_loop must be a unique label in the procedure body.
- scc_end_loop must be first instruction after end of loop to exit.

Example

```
   for (i = 0; i < 100; i++) {
      for (j = 0; j < 200; j++) {
         int cond = scc_stream_input_val();
         if (cond > 100) scc_break_loop(Li);
         if (i < MAX) x[i] = y[i] * z[i];
      }
   }
scc_end_loop(Li);
```

This is an equivalent example using C or C++ break:

```
   for (i = 0; i < 100; i++) {
      int cond;
      for (j = 0; j < 200; j++) {
         cond = scc_stream_input_val();
         if (cond > 100) break;
         if (i < MAX) x[i] = y[i] * z[i];
      }
      if (cond > 100) break;
   }
```

Outer Loop Controller

The Synphony C Compiler tool consolidates all the hardware it infers in a loop nest into one module (PA). A loop can consist of any levels of nesting and any number of parallel sub-loops at any level. The Synphony C Compiler tool perfectizes this loop nest into a consolidated loop, which automatically takes into account any data dependences between

different sub-loops at different levels of this loop nest. This allows you to generate the required hardware easily. Consider the following piece of code.

```
for (i...) {
   statement_list1;
   for (j...) { loop body 1; }
   statement_list 2;
   for (k...) { loop body 2; }
   statement_list 3;
   for (l...) { loop body 2; }
   statement_list 4;
}
```

This will get transformed into the following in the Synphony C Compiler tool after an inner-loop-merge operation as shown.

```
for (i...) {
   for (m = 0; m <= 2; m++) {
      if (m == 0) { statement_list 1; }
      else if (m == 1) { statement_list 2; }
      else { statement_list 3; }
      if (m == 0) { ub_x = ub_j; }
      else if (m == 1) { ub_x = ub_k; }
      else { ub_x = ub_l; }
      for (x = 0; x < ub_x; x++) {
      if (m == 0) { loop body 1; }
      else if (m == 1) { loop body 2; }
      else { loop body 3; }
   }
   if (m == 0) { }
   else if (m == 1) { }
   else { statement_list 4; }
   }
}
```

This allows you to create outer for loop control and also to build applications with multiple loops as TCABs. The performance constraints for the resulting loop can be different from the contributing loops. For more information on loops, see “Multiple Parallel Sub-Loops” on page 7-52.

The Synphony C Compiler tool does not support reading from the same loop and writing into the same FIFO. If two inner loops that use streams to communicate with each other are merged using the transformation mentioned, then the resultant loop would have both ends of the stream in the same loop. In this case, you should try other coding techniques to get the required performance.

Custom Opcodes

Many applications use operations that are not available as native C operators. To facilitate these operations to be easily expressed in the application source code and to be efficiently

mapped to hardware, the Synphony C Compiler tool provides several pre-defined operations and a standard way to integrate them into the application. These custom operations may be invoked in the application C code in the form of specialized procedure calls, and are mapped to hardware automatically by the Synphony C Compiler tool. The complete list and the semantics of all the custom operations supported by the Synphony C Compiler tool are defined in section "Pre-Defined Custom Operations" on page 8-45. The general syntax applicable to all custom operations is described in the subsequent section.

Custom Opcode Usage

Custom opcodes may be used in the application C code using special C procedure call syntax with a scc_asm prefix. For each pre-defined custom opcode, a function prototype of the following form has been defined:

```
<type> scc_asm_<opcode>(<arg>, …);
```

By including the "`scc.h`" in the application these prototype declarations also get included. These functions are implemented in a pre-defined library that is automatically linked during C simulation run (including golden run).

Example:

```
#include "scc.h"
void myapp (void) {
   int i, j, y[20][20];
   for (i=0; i<20; i++)
      for (j=0; i<20; j++) {
      y[i][j] = scc_asm_MIN_W(i, j);
      }
}
```

Note:

You can also create a TCAB for the function and integrate into this PPA. For more information on TCABs, see the Chapter 10, "Hierarchical Synthesis."

Pre-Defined Custom Operations

This section describes the syntax and semantics of the pre-defined custom operations in the Synphony C Compiler tool that are not native to C and are made available as procedure calls. In hardware, these opcodes translate directly into a single operation executing on a specialized function unit.

The following section provides a list of custom opcodes. The arithmetic operation details are described in the subsequent sections.

```
scc_asm_MIN_W              scc_asm_MIN_D
scc_asm_MINL_W             scc_asm_MINL_D
scc_asm_MAX_W              scc_asm_MAX_D
scc_asm_MAXL_W             scc_asm_MAXL_D
scc_asm_MEDIAN3_W          scc_asm_MEDIAN3_D
scc_asm_MEDIAN3L_W         scc_asm_MEDIAN3L_D
scc_asm_MPYL_W             scc_asm_MPYL_D
scc_asm_RANGER_D
scc_asm_RANGEL_D
scc_asm_CRYPT03GPPS7_W
scc_asm_CRYPT03GPPS9_W
scc_asm_MPY1C_W            scc_asm_MPY1C_D
scc_asm_MPY2C_W            scc_asm_MPY2C_D
scc_asm_MPY3C_W            scc_asm_MPY3C_D
scc_asm_MPY4C_W            scc_asm_MPY4C_D
scc_asm_MPYL1C_W           scc_asm_MPYL1C_D
scc_asm_MPYL2C_W           scc_asm_MPYL2C_D
scc_asm_MPYL3C_W           scc_asm_MPYL3C_D
scc_asm_MPYL4C_W           scc_asm_MPYL4C_D
```

Arithmetic Operations

The Synphony C Compiler tool provides the following custom arithmetic operations.

Absolute Value

This is the user function prototype for the absolute value of a signed number:

```
unsigned int scc_asm_ABS_W(int);
unsigned long long scc_asm_ABS_D(long long);
```

This function may be used in the application source code, and it gets mapped automatically to the appropriate hardware block.

Minimum Value

These are the user function prototypes for the minimum value of two signed, unsigned or long long numbers:

```
int scc_asm_MIN_W(int, int);
long long scc_asm_MIN_D(long long, long long);
unsigned int scc_asm_MINL_W(unsigned int, unsigned int);
unsigned long long scc_asm_MINL_D(unsigned long long,
   unsigned long long);
```

MIN_W is used for signed numbers and MINL_W for unsigned ones.

Maximum Value

These are the user function prototypes for the maximum value of two signed, unsigned, or long long numbers:

```
int scc_asm_MAX_W (int, int);
long long scc_asm_MAX_D(long long, long long);
unsigned int scc_asm_MAXL_W(unsigned int, unsigned int);
unsigned long long scc_asm_MAXL_D(unsigned long long,
   unsigned long long);
```

Median of Three Values

These are the user function prototypes for the median (middle) value of three signed, unsigned or long long numbers:

```
int scc_asm_MEDIAN3_W(int, int, int);
long long scc_asm_MEDIAN3_D(long long, long long, long long);
unsigned int scc_asm_MEDIAN3L_W(unsigned int, unsigned int,
   unsigned int);
unsigned long long scc_asm_MEDIAN3L_D(unsigned long long,
   unsigned long long, unsigned long long);
```

The median function can be used for clipping a number between a lower-bound and an upper-bound value.

Examples

```
x = scc_asm_ABS_W(diff);
x = scc_asm_MIN_W(a,b);
x = scc_asm_MEDIAN3L_W(pixval, 0, 255);
```

Unsigned Multiply

Some applications perform an operation where two unsigned variables are multiplied. In such a case, using the native-C * operator (which is a signed multiply operator) results in multiplier hardware whose input port widths are 1 bit wider than the original unsigned variables widths. To get cheaper hardware, an unsigned multiplier may be used. The Synphony C Compiler tool supports this with the following operation:

```
unsigned int scc_asm_MPYL_W(unsigned int, unsigned int);
unsigned long long scc_asm_MPYL_D(unsigned long long, unsigned
   long long);
int scc_asm_MPY1C_W(int, int);
long long scc_asm_MPY1C_D(long long, long long);
unsigned int scc_asm_MPYL1C_W(unsigned int, unsigned int);
unsigned long long scc_asm_MPYL1C_D(unsigned long long, unsigned
   long long);
```

Example

```
x = a * b;
```

where a and b are unsigned char, will generate a signed hardware multiplier with 9-bit inputs, while

```
x = scc_asm_MPYL_W(a, b);
```

will generate an unsigned hardware multiplier with 8-bit inputs. However, note that signed and unsigned multiply operators map to separate hardware macrocells. Therefore, hardware sharing is possible only among signed multiply operators, and likewise, only among unsigned multiply operators, and not between a signed and an unsigned multiply operator.

Internally-Wide Multiply

Some applications perform an operation where two values are multiplied to produce a value that is greater than 32 bits, of which a certain number of MSBs (most significant bits) are extracted as the desired result. The Synphony C Compiler tool supports this with the following set of operations:

```
int a, b, x;
#pragma bitsize a 31
#pragma bitsize b 27
#pragma bitsize x 30
x = ((long long) a * (long long) b) >> 28;
```

Here, a 31-bit number a is multiplied with a 27-bit number b to produce a 58-bit intermediate result, which is then arithmetic-right-shifted by 28-bits to produce a 30-bit result x.

Bit Extraction and Setting

The Synphony C Compiler tool provides custom opcodes that extract and set the value of a range of bits of a variable. The conventions followed are:

- All indexes used in specifying bit-select indexes and ranges are zero-based.
- The integer pair, (a, b), where 'a' >= 'b', represents an inclusive range of values; that is, it represents the sequence of 'a-b+1' values [a, a-1, ..., b+1, b].

```
unsigned long long scc_asm_RANGER_D(unsigned long long a, unsigned int

h, unsigned int l);
```

This custom opcode selects the bits in the range (h, l), where 0 <= l <= h <64, from within the unsigned 64-bit parameter, a, and returns it as a zero-extended 64-bit value with the selected h-l+1 bits as the LSB bits.

```
unsigned long long scc_asm_RANGEL_D(unsigned long long a, unsigned int h,

unsigned int l, unsigned long long value);
```

This custom opcode selects the following sequence of bits:

1. bits in the range (63, h+1) from the 64-bit parameter, ‘a’
2. bits in the range (h-l, 0) from the 64-bit parameter, 'value'
3. bits in the range (l-1, 0) from the 64-bit parameter, 'a'

and returns the 64-bit value resulting from the sequential concatenation of the 3 sequences of bits.

CCode	Verilog Code
c=scc_asm_RANGER_D(a, h, l)	assign c={{64-(h-l+1) {1'b0}}, a[h:l]};
c=scc_asm_RANGEL_D(a, h, l, b)	assign c={a[63:h+1], b[h-l:0], a[l-1:0]};

Example

```
Expression                                  Result
scc_asm_RANGER_D(0x9977, 11, 4)             0x97
scc_asm_RANGEL_D(0x998877, 15, 8, 0xff)     0x99ff77
```

1. All MSB extensions are zero-extensions, in accordance with SystemC v2.1 semantics; SystemC v2.0.1 semantics are not supported.
2. Synphony C Compiler opcode pattern-matching converts all calls to the *RANGER* and *RANGEL* opcodes that have constant values for both the h and l parameters, into corresponding *RANGERK* and *RANGELK* internal opcodes, which are implemented as zero-cost RTL macrocells.

Cryptography

These opcodes are used in the security architecture of a 3GPP system. The security and the confidentiality algorithms in a 3GPP system are based on an algorithm called the Kasumi algorithm. The Kasumi algorithm is a block cipher algorithm that produces a 64-bit output from a 64-bit input under the control of a 128-bit key. As part of the Kasumi algorithm uses three sub-functions called FL, FO and FI. The FI sub-function uses what are called S-boxes.

The first S-box is called S7 and it maps a 7-bit input to a 7-bit output. This S7 function is implemented as the opcode:

```
unsigned int scc_asm_CRYPTO3GPPS7_W(unsigned int x);
```

The argument passed to the function is the input to the S7 S-box and the returned value is the value that it is mapped to.

The second S-box is referred to as S9 and it maps a 9-bit input to a 9-bit output. This S9 function has been implemented as the opcode:

```
unsigned int scc_asm_CRYPTO3GPPS9_W(unsigned int x);
```

The argument passed to the function is the input to the S9 S-box and the returned value is the value that it is mapped to.

Pipelined Arithmetic

Typically, the Synphony C Compiler tool generates 1-cycle (that is, combinational) multipliers for multiplies in the application code using the * C-operator. This may be a bottleneck for high-clock frequency designs, in which case, pipeline some of the multiplies in the application code. The Synphony C Compiler tool provides a mechanism for the application developer to selectively force a multiply in the application code to take 2 cycles or 3 cycles using the following custom operations:

```
int scc_asm_MPY2C_W(int, int);
unsigned int scc_asm_MPYL2C_W(unsigned int, unsigned int );
long long scc_asm_MPY2C_D(long long, long long );
unsigned long long scc_asm_MPYL2C_D(unsigned long long,
unsigned long long );
int scc_asm_MPY3C_W(int, int);
unsigned int scc_asm_MPYL3C_W(unsigned int, unsigned int);
long long scc_asm_MPY3C_D(long long, long long);
unsigned long long scc_asm_MPYL3C_D(unsigned long long,
  unsigned long long);
int scc_asm_MPY4C_W(int, int);
long long scc_asm_MPY4C_D(long long, long long);
unsigned int scc_asm_MPYL4C_W(unsigned int, unsigned int);
unsigned long long scc_asm_MPYL4C_D(unsigned long long,
unsigned long long);
```

Example

```
#include "scc.h"
int in0[5], out0[4];
char in1[5], out1[4];
void test(void) {
   int i;
   for (i=0; i<4; i++) {
      out0[i] = scc_asm_MPY2C_W(in0[i],in0[i+1]);
      out1[i] = in1[i] * in1[i+1];
   }
}
```

Single-Cycle Delay

The Synphony C Compiler tool provides the application developer with the following functions for introducing a single-cycle delay between the output and the input.

```
int scc_asm_DELAY_W(int);
long long scc_asm_DELAY_D(long long);
unsigned int scc_asm_DELAYL_W(unsigned int);
unsigned long long scc_asm_DELAYL_D(unsigned long long);
```

If the input argument of any of the functions is bit-sized, then it is propagated to the output. The function can have reset facility based on the implementation parameter (reset_data_registers) described in Chapter 3, “Synphony C Compiler Commands.”

Example

```
int four_cycle_mpy(int a, int b)
{
   int c;
   c = scc_asm_MPY3C_W(a, b);
   c =scc_asm_DELAY_W(c);
   return c;
}
```

9

Fixed-Point Library

This chapter describes the Synphony C Compiler fixed-point library. The following topics describe the details:

- Fixed-Point Library Support
- Quantization and Overflow Modes
- Fixed-Point Arithmetic Library Usage
- Fixed-Point Division
- Fixed-Point Library Classes
- SystemC s_fixed / s_ufixed Conversion
- Limitations of the Fixed-Point Library

Fixed-Point Library Support

This chapter describes the Synphony C Compiler fixed-point arithmetic library. It uses C++ syntax for implementation. Include the scc_sfixed.h header file in the source files (driver as well as the synthesizable code) to make use of the fixed-point library.

The Synphony C Compiler fixed-point arithmetic library derives its semantics from the SystemC fixed-point library. For additional information refer to the sc_fixed and sc_ufixed classes in the I*EEE Standard SystemC Language Reference Manual version 2.1*. The major difference is that the Synphony C Compiler implementation is limited to 64-bit precision for fixed point variables and expressions. To avoid potential errors due to this restriction, the library dynamically checks and asserts that all fixed-point expressions use no more than 64-bits of precision.

There are two separate classes for handling fixed-point operations: s_fixed for signed numbers and s_ufixed for unsigned numbers. The classes s_fixed and s_ufixed use template parameters with the following defaults:

```
template <int wl=32, int iwl=32, s_q_mode Q=S_TRN, s_o_mode O=S_WRAP,
 int n_bits=0> s_ufixed;
template <int wl=32, int iwl=32, s_q_mode Q=S_TRN, s_o_mode O=S_WRAP,
 int n_bits=0> s_fixed;
```

wl or word length	Total number of bits in the fixed-point format. wl must be greater than zero and less than or equal to 64.
iwl or integer word length	Number of integer bits in the fixed-point format. iwl can be either positive or negative.
s_q_mode	Quantization mode.
s_o_mode	Overflow mode.
n_bits	Number of saturated bits used in the S_WRAP and S_WRAP_SM overflow modes. n_bits must be greater than or equal to zero.

The quantization and overflow modes are specified by the enumerated types. There are two enumerated types used:

- s_q_mode
- s_o_mode

The quantization modes, overflow modes and the classes s_fixed, s_ufixed are qualified by the *using directive* under a compile time flag S_INCLUDE_FX. If you do not intend to use these directives, you can define this value to be zero as follows:

```
#define S_INCLUDE_FX 0 or #define S_INCLUDE_FX false
```

Quantization and Overflow Modes

The enumerated types used in the Synphony C Compiler fixed-point library are as follows:

- s_q_mode:

```
typedef enum {
   S_RND,
   S_RND_ZERO,
   S_RND_MIN_INF,
   S_RND_INF,
   S_RND_CONV,
   S_TRN,
   S_TRN_ZERO,
   S_Q_NONE = 7
} s_q_mode;
```

s_q_mode is an enum that represents the possible quantization modes supported by the fixed-point signed or unsigned data types in the SCC tool. The functionality of each of the quantization modes matches the corresponding SystemC quantization modes as defined in the section 7.10.9 of the *SystemC LRM version 2.1*.

S_Q_NONE is Synopsys' quantization mode that can be used if quantization is not required. This mode will avoid hardware for quantization and will improve QoR and timing of the design.

The quantization mode specification provides corrective action when the precision of a fixed-point value exceeds the precision of a fixed-point variable. The different modes mentioned above control the final representation of the value in the variable according to the following table.

Quantization Mode	**Signed +ve value s_fixed <4,2> to s_fixed <3,2, [Quantization]> 01.01 (1.25)**	**Signed -ve value s_fixed <4,2> to s_fixed <3,2, [Quantization] > 10.11 (-1.25)**	**Unsigned value s_ufixed <16,8> to s_ufixed <12,8,[Quantization]>000100110.01001000 (38.28125)**	**Comment**
S_RND	01.1(1.5)	11.0 (-1)	00100110.0101 (38.3125)	Rounding to plus
S_RND_ZERO	01.0 (1)	11.0(-1)	000100110.0100 (38.25)	Rounding to zero
S_RND_MIN_INF	01.0 (1)	10.1(-1.5)	000100110.0100 (38.25)	Rounding to minus infinity
S_INF	01.1(1.5)	10.1 (-1.5)	000100110.0101 (38.25)	Rounding to infinity
S_RND_CONV	01.0 (1)	11.0 (-1)	000100110.0100 (38.25)	Convergent rounding (LSB of remaining bits 0: round towards minus infinity 1: round towards plus infinity
S_TRN	01.0 (1)	10.1 (-1.5)	00100110.0100 (38.25)	Truncation
S_TRN_ZERO	01.0 (1)	11.0 (-1)	00100110.0100 (38.25)	Truncation to zero
S_Q_NONE				User guarantees that no truncation will be required (allows the tool to build more optimal hardware)

- s_o_mode:

```
typedef enum {
   S_SAT,
   S_SAT_ZERO,
   S_SAT_SYM,
   S_WRAP,
   S_WRAP_SM,
   S_O_NONE
}  s_o_mode
```

s_o_mode is an enum that represents the possible overflow modes supported by the Synphony C Compiler tool's fixed-point data types. The functionality of each of the overflow modes matches the corresponding SystemC overflow modes as defined in section 7.10.9 of the *SystemC LRM*.

S_O_NONE is Synopsys' overflow mode that can be used if overflow mode is not required. This mode will avoid hardware for quantization and will improve QoR and timing of the design.

The following table gives the details about different saturation modes:

Overflow Mode	**Signed +ve 3 bit number [Input = 01110 (14)]**	**Signed -ve 3 bit number [Input = 10010 (-13)]**	**Unsigned 3 bit number [Input = 11100(28)]**	**COMMENTS**
S_SAT	011 (3)	100 (-4)	111 (7)	Saturate to the maximum value possible in case of overflow in both +ve and -ve cases
S_SAT_ZERO	0	0	0	Make all overflow cases as 0.
S_SAT_SYM	011 (3)	101 (-3)	111 (7)	Same as S_SAT except that the saturation for -ve values is minus the positive maximum value
S_WRAP (n_bits = 0)	110 (-2)	010 (2)	100 (4)	Just passes the lower bits through

Overflow Mode	**Signed +ve 3 bit number [Input = 01110 (14)]**	**Signed -ve 3 bit number [Input = 10010 (-13)]**	**Unsigned 3 bit number [Input = 11100(28)]**	**COMMENTS**
S_WRAP (n_bits > 0)	010 (2)	110 (-2)	100 (4)	Keeps the sign bits in the MSB n_bits and just rolls, just passes the lower bits
S_WRAP_SM (n_bits = 0)	110 (-2)	010 (2)	011 (3)	Discard all the MSBs and if LSB among the bits discarded is different from the MSB of remaining bits (signed numbers)/ 0 (unsigned numbers), invert all the remaining bits
S_WRAP_SM (n_bits > 0)	010 (2)	101 (-3)	001 (1)	The first bit in the resulting number retains the sign of the original number and the remaining bits in the resultant number are formed by xoring the old sign bit, new sign bit and the original bit in that position
S_O_NONE				User guarantees that no quantization will be required (allows the tool to build more optimal hardware)

Fixed-Point Arithmetic Library Usage

You can use the Synphony C Compiler fixed-point library classes, s_fixed or s_ufixed for any other built in C++ data types. You use the template parameters to indicate the following:

- Word length
- Integer word length
- Saturation mode
- Quantization mode
- Number of saturated bits for overflow modes S_WRAP and S_WRAP_SM

You can declare objects of the class s_fixed for signed fixed-point number representation or s_ufixed for unsigned fixed-point number representation. After declaring the objects of appropriate type, you can do arithmetic on these objects.

The following are some examples of fixed-point library.

Examples

- Fixed point representation and arithmetic

s_fixed <10,2> a, b, c;	a, b, c are signed with 10 bits word, 2 integer bits, S_TRN for overflow mode and S_WRAP for Quantization mode. a, b, and c are numbers of the form bb.bbbbbbbb
s_ufixed <10,0, S_RND, S_SAT> d, e;	d, e are unsigned with 10 bit words, 0 integer bits, S_RND for overflow mode and S_SAT for saturation mode. d and e are numbers of the form 0.bbbbbbbbbb
c =(a+b)+(d+e);	d+e is s_ufixed <11,1> 0b.bbbbbbbbbb a+b is s_fixed <11,3> bbb.bbbbbbbb00 (a+b) + (d+e) is s_fixed <14,4> bbbb.bbbbbbbbbb The result is saturated, quantized to bb.bbbbbbbb

- Quantization and Overflow

  Consider the assignment:

```
s_fixed <28,12>b;
s_fixed <13,5> a = b;
```

- Overflow logic is inserted to determine how discarded integer bits affect the final value
- Quantization logic is inserted to determine how discarded fractional bits affect the final value
- Quantization and overflow can also be triggered by type casting, for example:

```
b = (s_fixed<13,5>)b;
```

- Coding Techniques

  It must be noted that unnecessary quantization or overflow are bad for Synphony C Compiler runtime, QoR and timing. It is recommended to avoid intermediate variables in expression trees, use S_Q_NONE and S_O_NONE.

  - Avoid intermediate variables

    Intermediate variables could insert quantization or overflow logic and might prevent some optimizations in the compiler. For example, consider the following code segment:

```
 …
t1 = a*b;
t2 = a*c;
t3 = a*d;
t4 = a*e;
x = t1+t2+t3+t4;
```

    You can write this code as shown:

```
x = (a*b) + (a*c) + (a*d) + (a*e);
```

    This makes it possible for the compiler to optimize it as follows:

```
x = a*((b+c)+(d+e));
```

  - Use S_Q_NONE and S_O_NONE modes

    It is recommended to use Synopsys' no quantization and no overflow modes if possible. In these modes there are runtime assertions if there is any loss of bits. You can get an error if the assertions are violated during golden simulation (verify -golden).

    Consider the following code segment:

```
…
s_fixed <5,2> a;
…
s_fixed <8,5>x = a;
for (i = 0; i < 7; i++) {
if (c1) x++; // overflow logic
}
```

If you know that x will not overflow, re-write the code as follows:

```
…
s_fixed <5, 2> a;
…
s_fixed<8,5, S_Q_NONE, S_O_NONE> tmp = a;
for (i = 0; i < 7; i++) {
if (c1) tmp++; // no overflow logic
}
s_fixed <8,5> x = tmp;
```

Fixed-Point Division

The Synphony C Compiler fixed point division semantics differs from the SystemC division. The Synphony C Compiler fixed point division implements what is good for the hardware. The format of the quotient is based on the format of the dividend and divisor in the following way:

```
Integer bits of quotient = Integer bits of dividend + fractional bits of

divisor
```

The quotient word length is defined as follows:

```
Word length of quotient = word length of dividend + MAX (fractional bits

of divisor, 0)
```

MAX (a, b) is the maximum of a and b.

If you want additional precision, cast the dividend to a type with more fractional bits.

Fixed-Point Library Classes

The s_fixed class implements the fixed point operations for signed numbers. It contains methods for type conversion between the built-in types as well as the s_fixed and s_ufixed types. Various operators (relational, logical, arithmetic) are overloaded in this class to perform the operations on the Synphony C Compiler tool's fixed-point types as you may do any other C or C++ -Language's built in types. The methods defined in this class implement the different quantization and saturation modes defined in the previous section.

The s_ufixed class implements the fixed-point operations for unsigned numbers. It contains methods for type conversion between the built in types as well as the s_fixed and s_ufixed types. Various operators (relational, logical, arithmetic) are overloaded in this class to perform the operations on the Synphony C Compiler tool's fixed point types as one would do on any other C or C++ - Language's built in types. The methods defined in the class implement the different quantization and saturation modes defined in the previous section.

The various fixed-point library operators that are supported for synthesizable code are listed in the following tables. In the tables, “a” and “b” are two fixed-point numbers (signed or unsigned).

Arithmetic Operators

Operator Name	Syntax
Addition	a+b
Subtraction	a-b
Multiplication	a*b
Division	a/b
Unary plus	+a
Unary minus	-a
Post increment	a++
Pre increment	++a
Post decrement	a--
Pre decrement	--a
Assignment	a=b

Relational Operators

Operator Name	Syntax
Equal to	a == b
Not equal to	a != b
Greater than	a > b
Less than	a < b
Greater than or equal to	a >= b
Less than or equal to	a <= b

Logical Operators

Operator Name	Syntax
Logical AND	a && b
Logical OR	a || b

Bitwise Operators

Operator Name	Syntax
Bitwise NOT	~a
Bitwise AND	a & b
Bitwise OR	a | b
Bitwise XOR	a ^ b
Bitwise left shift	a << b
Bitwise right shift	a >> b

TypeCast Operators

Operator Name	Syntax
operator bool ( )	(bool) a
operator char ( )	(char) a
operator short ( )	(short) a
operator int ( )	(int) a
operator long ( )	(long) a
operator long long ( )	(long long) a
operator unsigned char ( )	(unsigned char) a
operator unsigned short ( )	(unsigned short) a
operator unsigned int ( )	(unsigned int) a
operator unsigned long ( )	(unsigned long) a
operator unsigned long long ( )	(unsigned long long) a
operator char ( ) const	(char) a
operator short ( ) const	(short) a
operator int ( ) const	(int) a
operator long ( ) const	(long) a
operator long long ( ) const	(long long) a
operator unsigned char ( ) const	(unsigned char) a
operator unsigned short ( ) const	(unsigned short) a
operator unsigned int ( ) const	(unsigned int) a
operator unsigned long ( ) const	(unsigned long) a
operator unsigned long long( ) const	(unsigned long long) a

Constructors in Fixed-Point Library

Constructor description	Syntax
Construct from one of the fixed types	s_fixed<32,32> a = b; s_ufixed<32,32> c = b;
Construct from constant	s_ufixed<32,32> a = 45; s_fixed<16,12> b = -5.56;

Operator Support in Non-Synthesizable Code

To facilitate integration with floating-point testbenches, the tool supports double and float typecast operators in non-synthesizable code. In the following example, the `to_double()` typecast operator converts a value to double precision:

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include "scc_sfixed.h"
using namespace std;

int main() {
  const s_fixed<32, 3, S_RND, S_SAT> MY_PI = M_PI;
  cout << setprecision(12) << "MY_PI = " << MY_PI.to_double() << endl;
  return 0;
}
```

This allows the setprecision(12) to properly set the value to a desired precision, which can be used for checking whether there are sufficient bits to quantize a value. Note that typecasting operators such as this are not synthesizable, so they should be used only for simulation and diagnostics.

SystemC s_fixed / s_ufixed Conversion

You can convert from SystemC s_fixed/s_ufixed types to the corresponding SCC types, and vice versa. To convert from a SystemC type to a Synphony C Compiler fixed-point type, directly assign the SystemC fixed-point type variable to the SCC fixed-point type variable.

To convert from a SCC fixed point type to a SystemC type, use the to_sc_fixed( ) or to_sc_ufixed( ) methods:

```
s_fixed a;
sc_ufixed b;
a = b;                // convert from sc_ufixed to s_fixed
b = a.to_sc_ufixed(); // convert from s_fixed to sc_ufixed
```

These conversion functions are not synthesizable and should only be used in driver code. You must define the compile time flag SC_INCLUDE_FX to use these conversion functions and enable SystemC source in the implementation parameters with this command:

```
set_params -systemc_source yes
```

Limitations of the Fixed-Point Library

The SCC fixed-point library has some limitations, when compared with the SystemC fixed-point library.

1. A precision of 64-bits is supported for the fixed-point operation. This is dynamically checked. If the result of an operation exceeds 64-bits, an assertion failure is issued.

2. Floating-point types are not supported inside the synthesizable code. Therefore, you would not be able to synthesize code that has the following form:

   ```
   s_fixed var;
   double d;
   s_fixed var = (double) d; //not supported in synthesizable code,
                             // but supported in driver code
   ```

   Floating-point literals are supported inside the synthesizable code only for the Synphony C Compiler fixed-point initializations. Therefore, you can assign a floating-point literal to an s_fixed or s_ufixed type. That is, you can do the following:

   ```
   s_fixed var;
   var = 10.5;               // supported
   ```

3. Shift value for the shift operator must be greater than or equal to zero and less than or equal to 64. An assertion failure is issued if the shift value is not within the range.

   The C-based implementation of fixed-point library is obsolete. Contact Synopsys if the earlier version of the C-based fixed-point library is being used in the designs.

10

Hierarchical Synthesis

Most hardware designs consist of hierarchy of instantiated modules. The Synphony C Compiler tool also allows you to create modular hardware that facilitates lower level modules to be instantiated in higher levels automatically. This chapter includes the following topics:

- Hierarchical Block Overview
- Hierarchical Block Operations
- Specifying Constraints for Hierarchical Blocks
- Sharing Hierarchical Blocks
- Importing Hierarchical Blocks
- Physical Instantiation of Hierarchical Blocks
- Overlapped Execution of TCABs and IPPAs
- Specifying the Properties of TCAB and IPPA
- Target Performance Specification for TCABs
- TCAB Input/Output Controls
- Uses of TCABs
- TCAB Sharing and Inlining
- TCAB-Specific Parameters

- Classifications of TCABs

Hierarchical Block Overview

The default behavior of the Synphony C Compiler tool is to inline all the functions at the point of call. Alternatively, you can synthesize one or more procedures as building blocks called Tightly Coupled Accelerator Blocks or TCABs. TCABs can be built as standalone modules or to can be used as building blocks that are instantiated in higher level designs.

You can import the TCABs into higher level PPA or TCAB implementations to generate hierarchical designs. Higher-level designs are referred to as TCAB containers. After building functions as TCABs and importing them into a higher-level design, all the code in the higher-level design which is not marked to be built as a TCAB becomes inlined into the original function. This is illustrated in the following code:

```
// Original Call tree

A() {
   B();
}

B() {
   C();
}

C() {
   D();
}
```

```
// Realized hardware where only
// C() is built as a TCAB
A() {
   { // B() is inlined
      C();
   }
}

C() {
   {
      // D is inlined.
   }
}
```

You might want to use a TCAB for a hierarchical design in any of the cases listed. For more information about how these can be accomplished, see “Uses of TCABs” on page 10-26.

- Enhanced Datapath Sharing
- Reconfigurable Pipeline of Logic
- Target Performance based logic partitioning
- Power Savings
- Hide Complexity of Lower Blocks at Higher Levels
- Outer loop control

TCABs use the same command syntax as PPAs. See Chapter 6, “C/C++ Synthesizable Syntax” for details.

Hierarchical Block Operations

This section describes the build, import, and verification commands for TCABs or IPPAs:

- Building Hierarchical Blocks
- Importing Hierarchical Blocks
- Verifying Hierarchical Blocks

Building Hierarchical Blocks

You can build a function into a building block as shown:

Batch Mode
set_params -type TCAB | IPPA | CHANNEL -target "*procedure_name*"
GUI
File > *Set Parameters* > *Basic* > *Target Type* > *TCAB* | *IPPA* | *CHANNEL*. For more information on building hierarchical designs using the GUI, see “Building Hierarchical Designs” in the *Synphony C Compiler User Guide*.

Importing Hierarchical Blocks

You can also import a pre-synthesized TCAB or an IPPA implementation in both the batch mode and in the GUI.

Batch Mode
set_params -import *imp1 imp2 imp3* …
imp is the implementation name for the hierarchical block.
GUI
File > *Set Parameters* > *Hierarchy* > *List*
For more details about importing hierarchical blocks, see “Importing Hierarchical Blocks” on page 10-9.
Only single-source projects are supported (this means that the source code for all the TCAB procedures as well as the top-level PPA must be in the same project as the Synphony C Compiler project.

Verifying Hierarchical Blocks

After a function is built as a TCAB or IPPA, all the other logic inside the application becomes part of the testbench. In the code segment shown in "Hierarchical Block Overview" on page 10-3, when the function C( ) is built as a TCAB, all the code in functions A( ) and B( ) become part of the testbench. The same testbench can be used to verify the functionality of the function C( ) as well as the TCAB hardware created for the function C( ).

Specifying Constraints for Hierarchical Blocks

There are no mandatory constraints required to build a TCAB or IPPA(other than specification to build the hardware as a TCAB). However, most designs being implemented as TCABs or IPPAs set the following parameters.

- Target clock frequency
- II, if the function body contains a loop
- Pipelinability

The optional constraints that can be applied when building the TCAB are listed in the following table.

Constraint	Description
Clock	
`set_params -clock frequency`	Target clock frequency
`set_params -clock_gating`	Creates clock gating hardware.
Performance	
`set_params -ii`	Number of cycles between successive iterations of the top most loop.
`set_params -architectural_pipelinability`	Pipelinability. Number of cycles required between different tasks on the TCAB. See "Architectural Pipelinability Specification" on page 10-19.
Interface	

Constraint	Description
`set_params -tcab_input_registers <option> and set_params -tcab_output_registers <option>`	TCAB inputs and outputs to be registered.
`set_params -raw_scalar_input_external _delay <delay>` `set_params -raw_scalar_output_external _delay <delay>`	Timing of the external logic connecting to an input port or taking inputs from a TCAB output port.
`set_params -tcab_deployment <options>`	Type of ports for different interfaces.
Input/Output Cycle Controls	
`set_params -input_sample_time`	Forces the scalar inputs to be read at cycle 0 or let the tool decide the cycle when the input is read. See "Input Sample Time Specification" on page 10-19.
`set_params -resource_first_use_time`	Specifies when a memory or stream resource is valid to be used by TCAB. See "Resource First Use Time Specification" on page 10-21.
`set_params -output_production_time`	The last cycle before which a scalar variable is produced at the output of the TCAB. See "Output Production Time Specification" on page 10-20.
`set_params -resource_last_use_time`	Specify the cycle number before which a memory/stream resource needs valid data from TCAB. See "Resource Last Use Time Specification" on page 10-21.

Sharing Hierarchical Blocks

A TCAB and an IPPA can be shared just like any other functional unit. For more information on different methods to share functional units, see the Chapter 5, "Programming Model Basics." Additionally, TCABs and IPPAs can be shared within a PA or across PAs if the calls

to the TCAB or IPPAs are mutually exclusive in time. The following considerations determine the way a TCAB or an IPPA is shared:

- TCABs or IPPAs can be shared only if the PAs have no overlap in execution either in the same task, or across different tasks. The Synphony C Compiler tool supports only multiplexed sharing and no arbitrated sharing. TCABs and IPPAs are shared only if the calls are mutually exclusive.
- A TCAB and an IPPA can be shared only among a group of independent stall domain blocks, or among a group of shared stall domain blocks. A TCAB or an IPPA cannot be shared among a mixture of both shared and independent stall domain blocks. For information about stall domains, see “Specifying Stall Domains for Hierarchical Blocks” in the *Synphony C Compiler User Guide*.
- Sharing is done only if such sharing does not create any combinational loops in the logic.

Hierarchical TCAB or IPPA sharing makes it possible to share TCABs and IPPAs that are called at different levels of hierarchy. The following examples illustrate the different kinds of sharing that is described.

Example 1: Mutual exclusion across hierarchy in the same loop.

The following code segment shows TCAB_A, TCAB_C and TCAB_B built as TCABs. The Synphony C Compiler tool can share TCAB_B if the calls are mutually exclusive.

```
void TCAB_A()
{
   …
   TCAB_B();
   …
}
void TCAB_C()
{
   …
   TCAB_B()
   …
}
void TCAB_B()
{
   …
}
void PPA1(){
   // PA 0
   for (L0 = 0; L0 < n; L0++) {
      if (cond1) {
         TCAB_A();
      else {
         TCAB_C();
      }
   }
}
```

In PPA1, TCAB_B can be shared as calls to TCAB_A and TCAB_C are mutually exclusive.

Example 2: Hierarchical sharing across different PAs.

In this code segment, TCAB_A, TCAB_C and TCAB_B are built as TCABs. For PPA2, loops PA0 and PA1 are sequential with a task due to accessing the same memory A[ ]. However, the PAs can execute simultaneously across tasks due to task overlap. This design shares the TCAB_B hardware only if you disable task overlap.

```
void TCAB_A( )
{
   …
   TCAB_B( );
   …
}
void TCAB_C( )
{
   …
   TCAB_B( );
   …
}
void TCAB_B( )
{
   …
}
void PPA2( ){
   int a[10];

   // PA 0
   for (L0 = 0; L0 < n; L0++) {
      …
      TCAB_A( );
      …
      A[L0] = x;
      …
   }

   // PA1 - sequential with respect to PA0 (mutually exclusive)
   for (L0 = 0; L0 < n; L0++) {
      …
      TCAB_C( );
      …
      Y = A[L0];
      …
   }
}
```

Example 3: Hierarchical sharing across multiple levels of hierarchy.

In the following code segment, TCAB_A, TCAB_C and TCAB_B are built as TCABs. TCAB_B can be shared in PPA3.

```
void TCAB_A()
{
   …
   TCAB_B();
   …
}

void TCAB_B()
{
   …
}

void PPA3(){
   // PA 0
   for (L0 = 0; L0 < n; L0++) {
      …
      if (cond)
         TCAB_A();
      else
         TCAB_B();
      …
   }
}
```

To disable sharing of the TCABs across different PAs, you can use the following command while building the TCAB:

```
set_params -sharable_across_blocks never
```

This implementation parameter can potentially affect the hardware in the container, but does not affect the hardware in the TCAB itself. The other options to this implementation parameter are as follows:

`always:` share regardless of benefit in cost

`infer`: share only if there is a benefit in cost

Importing Hierarchical Blocks

The Synphony C Compiler tool requires the following:

- All TCAB or IPPA function definitions must be available in the PPA file during the import
- The TCAB or IPPA code and its interfaces must be unchanged between the TCAB or IPPA synthesis and its import into another TCAB or PPA.

If your design has a complex hierarchy of TCABs or IPPAs, then while building the current implementation you must specify only the first level TCABs or IPPAs imported (into the

current implementation). All other TCABs or IPPAs which are instantiated inside the first level TCABs or IPPAs are automatically imported into the current level as TCABs or IPPAs.

You can build static functions as TCABs or IPPAs and import them as TCABs or IPPAs, without inlining. However, you cannot have multiple static functions defined in `-appfiles`.

The Synphony C Compiler tool expects all calls of a function to be either inlined or built as TCABs or IPPAs throughout the implementation. If you need to inline a function in one instance and build it as a TCAB or IPPAs in another, you must use wrapper functions, as shown in the following code snippet:

```
void A() {
   B();
   C();
}
void T() {
   A();
   B();
}
```

B( ) and C( ) are built as TCABs and imported into A( ) which is another TCAB. Importing A( ) as a TCAB would automatically import B( ) as a TCAB and so the function call to B( ) inside T( ) would automatically be inferred as the built TCAB. Instead, if you want to inline the B( ) function inside T( ), do the following:

1. Create a TCAB wrapper around the function.

```
void B_TCAB(){
   B();       // This call is inlined into the B_TCAB()
}
```

2. Use the TCAB wrapper function where the function is needed to be built as a TCAB and use the original call where inlining is required.

```
void A() {
   B_TCAB();  // Call to the TCAB version of the function B()
   C();
}

void T() {
   A();
   B();       // Inlined version of function B()
```

When you import a TCAB or IPPA into a higher-level block, the function definition of the TCAB or IPPA is not imported and only a limited amount of linting support is provided during the import. It is your responsibility to rebuild the TCAB or IPPA and import it again if the source code for TCAB or IPPA functions or its interfaces are modified.

Physical Instantiation of Hierarchical Blocks

To facilitate hierarchical TCAB or IPPA sharing, most TCABs or IPPAs are physically instantiated outside their parent TCAB or IPPA by default. The following implementation parameter can be used to control the place where the TCAB or IPPA is instantiated.

```
set_params -integrate_block
       localize/externalize/infer|TCAB_proc:localize/externalize/infer
```

This specifies the location (that is, the place where the TCAB or IPPA is instantiated) of the imported TCABs or IPPAs. The default is infer. The options to this implementation parameter are as follows:

`localize`	Creates all instances of the imported TCAB in the current container. This prevents any further sharing of the imported TCAB with other instances outside the current container.
`externalize`	Ports all instances of the imported TCAB out of the current container. The TCAB is eventually created wherever it is localized. Porting out allows for hierarchical sharing at a higher level. For example: TCAB A calls TCAB B, which in turn calls TCAB C. If TCAB B is externalized out of TCAB A, then TCAB C will also be externalized out of TCAB A. In other words, if a child is externalized, all the hierarchy below the child is externalized.
`infer`	Localize if the TCAB is not sharable; otherwise, externalize.

The following examples demonstrate how you can control the implementation of TCABs or IPPA in the design hierarchy.

- To localize all imported TCABs or IPPA at the current implementation level:

  ```
  set_params -integrate_block localize
  ```

- To externalize all imported TCABs or IPPAs out of the current implementation level:

  ```
  set_params -integrate_block  externalize
  ```

- To selectively control localization of imported TCABs or IPPAs at the current implementation level:

  ```
  set_params -integrate_block \
   "externalize A1:localize B1:localize C1:externalize D1:infer"
  ```

  In this example, the tool externalizes all the TCABs by default except A1, B1, and D1. TCABs A1 and B1 are localized at this level and the localization of D1 is based on whether TCAB or IPPA is sharable across blocks.

Two reports provide information about the implementation hierarchy, report_hierarchy.pdf and report_hierarchy_detailed.pdf, in the implementation/Reports/ directory. For detailed information about these reports, see Chapter 13, "Analysis Views and Reports."

Sometimes you may not get TCAB or IPPA sharing in spite of meeting the conditions described. This could be because the never option was specified for the `sharable_across_blocks` implementation parameter, or because the tool determined that there was no benefit to sharing. For the latter case, you can force sharing by specifying `-sharable_across_blocks always`.

Localization is a step in synthesis that moves TCABs and other structures into the appropriate physical hierarchy closer to the blocks that access them. For details, see "Localization" on page 15-17.

Overlapped Execution of TCABs and IPPAs

You can groups tasks to be run in a pipelined fashion on a TCAB or IPPA and overlap the execution in certain situations.

In the following example a TCAB tcab( ) is called in a two different loops, that are separated by some code which modifies the variables used by the TCAB.

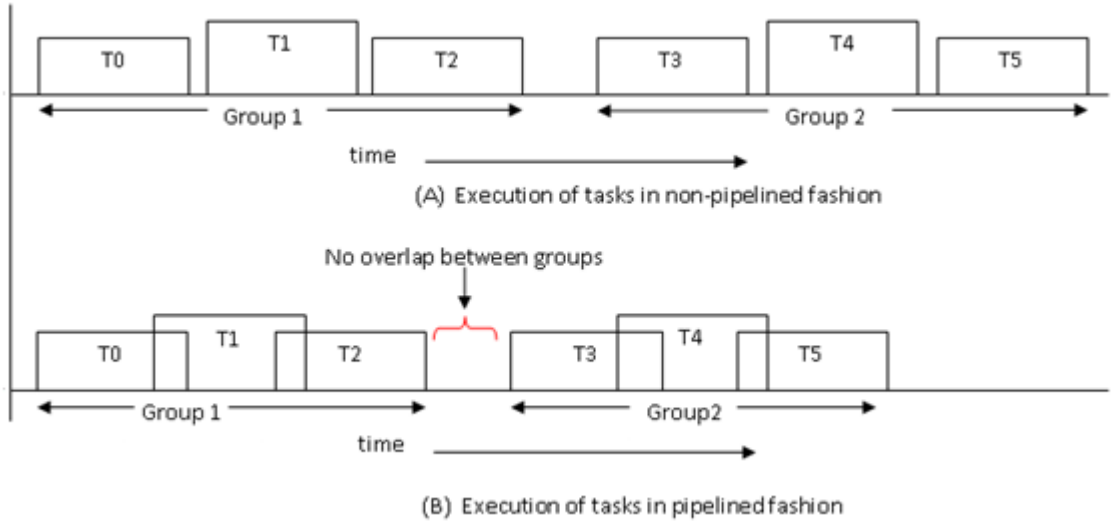
```
void ppa
   {
            // Group1: Tasks T0, T1, T2
      for (i=0;i<3;i++)
      tcab(); // this is a tcab and not ppa.

            // Code block which accesses/modifies inputs and
            // outputs of the tcab( ).
      <CODE>
           // Group2: Tasks T3, T4, T5
     for (i=0;i<3;i++)
     tcab(); // this is a tcab and not ppa.
   }
```

The Synphony C Compiler tool creates the required Verilog and SystemC testbenches to test this TCAB or IPPA. It infers that the inputs and outputs of the TCAB tcab( ) have changed between the two loops which call the tcab( ), and treats the tcab( ) calls from the two loops as different groups of tasks. Within the two loops, there is no code which changes the TCAB input data between TCAB calls.

The Verilog testbench executes such groups of tasks in a pipelined fashion. In the following figure, (A) shows the execution of different tasks executing in a nonpipelined fashion and (B) shows the execution in a pipelined fashion. The extent of overlap between tasks is determined by the number of draining stages reported in the report_summary.txt file for the

TCAB implementation. For more information on this report, see "report_summary.txt" on page 13-6.



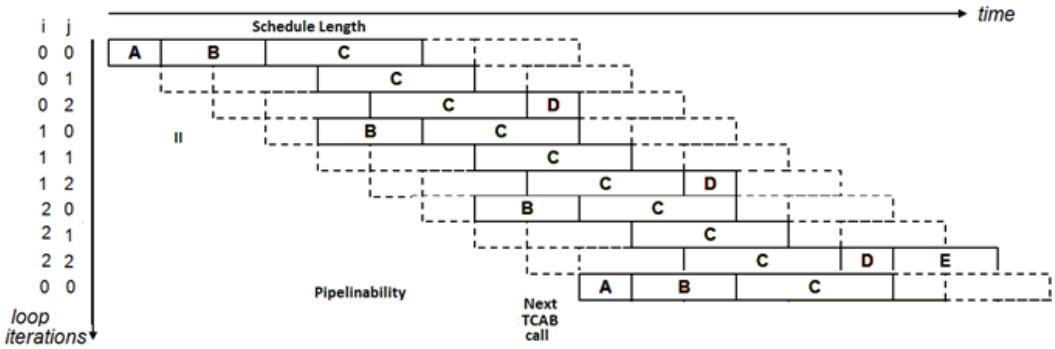
Specifying the Properties of TCAB and IPPA

This section defines the TCAB properties and helps you understand how TCAB operations are scheduled and how they influence the properties of the TCAB.

The following function is built as a TCAB, with five blocks of code (A, B, C, D and E) that execute under different conditions. Each block has a data dependence with other blocks within an iteration. Therefore, none of these operations can co-execute within the same iteration.

```
void tcab_T(){
   A;
   for (i = 0; i < 3; i++) {
      B;
      for(j = 0; j < 3; j++) {
         C;
      }
      D;
   }
      E;
   }
```

The scheduling of different iterations of the inner and outer loops is shown in the following diagram. The operations that execute in each cycle are the solid boxes. The operations that do not execute in the current iteration have dotted lines.



The following rules apply when iterations inside the TCAB and successive calls to the TCAB (from higher level designs) are scheduled:

1. Successive iterations of the outer loop are scheduled only after all the iterations of the inner loops have started execution.
2. Similarly, successive calls to the TCAB (from higher level designs) can be scheduled only after all the iterations of the entire loop nest have already started execution.

Details of TCAB Property

The following TCAB properties can be explained using the previous example.

Schedule Length (SL)	Total number of cycles taken per iteration of the outer loop inside the TCAB.
II	The minimum number of cycles between starting successive iterations of the outer most loop (loop index I in the example).
Stall Domain	Often times, a set of interconnected modules have similar data dependencies, causing them to either stall or execute together. Such a collection of modules which operate in similar fashion for all sets of inputs are called a stall domain. If a module co-executes with its external environment, such a module is called a shared stall domain TCAB. If a module can either stall or execute independent of its caller, such a module is called an independent stall domain TCAB.

Pipelined tasks	All the new sets of tasks which start when the previous task starts draining out are called pipelined tasks.
Nonpipelined tasks	Successive tasks which start only after the previous tasks' have completed execution completely are called nonpipelined tasks.
Pipelined Cycles	The tool can schedule operations across several iterations in the same cycle. In the TCAB code, the extent of such overlapped execution of iterations is signified by the pipelined cycles.
Draining Cycles	The number of cycles for the hardware pipeline to drain completely when no new inputs are being sampled by the TCAB.
Total cycles (TC)	The latency of one task execution, and includes the time till the last iteration of a TCAB completes.
Input Sample Time	Scalar variable inputs in a TCAB module can be sampled either when they are actually required for data processing or at the time when the TCAB is started itself. The cycle at which any input is sampled relative to the start of the task on the TCAB is the input sample time of the scalar variable. See "Input Sample Time Specification" on page 10-19.
Resource First Use Time	Array and stream inputs in a TCAB module can be utilized when they are actually required for data processing or at the time of TCAB start. The cycle at which a resource is first used after the start of the task on the TCAB is the input sample time of the scalar variable. See "Resource First Use Time Specification" on page 10-21.
Output Production Time	Output production time is the delay (in cycles) from the TCAB start to the time a resource is last updated by the within a task. See "Output Production Time Specification" on page 10-20.
Resource Last Use Time	Resource last use time is the delay (in cycles) from the TCAB start to the last time a resource is accessed by the TCAB within a task. See "Resource Last Use Time Specification" on page 10-21.
Pipelinability (P)	Pipelinability is the minimum number of between successive calls to the TCAB function from a higher level design. This is also referred to as Physical Pipelinability. This number is dependent on when the TCAB completes execution in the absence of any stalls due to data flow or arbitration. A new task on the TCAB can be started only after all the iterations of the current TCAB task have started. See "TCAB Pipelinability" on page 10-16 for figures.

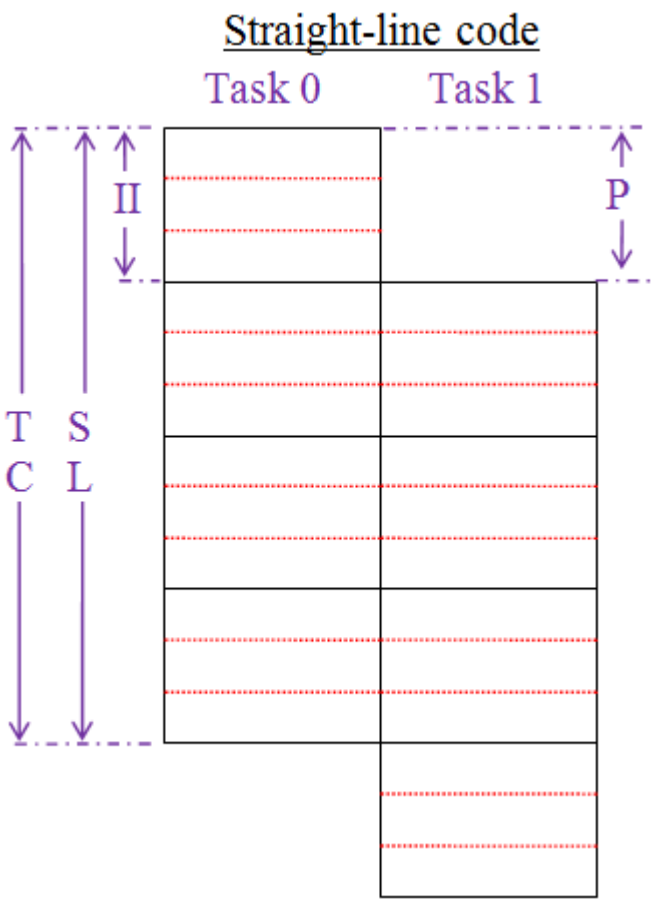
Architectural Pipelinability	This is the pipelinability of a TCAB as seen by higher-level designs. It can be specified as an input parameter when building a TCAB and the tool tries to achieve a physical pipelinability equal to the specified architectural pipelinability. If it is unable to do so, it either errors out or synthesizes an independent stall domain TCAB (see "Shared Stall Domain and Independent Stall Domain TCABs" on page 10-33). You can control stall domains with the set_params -allow_latency_violation command.

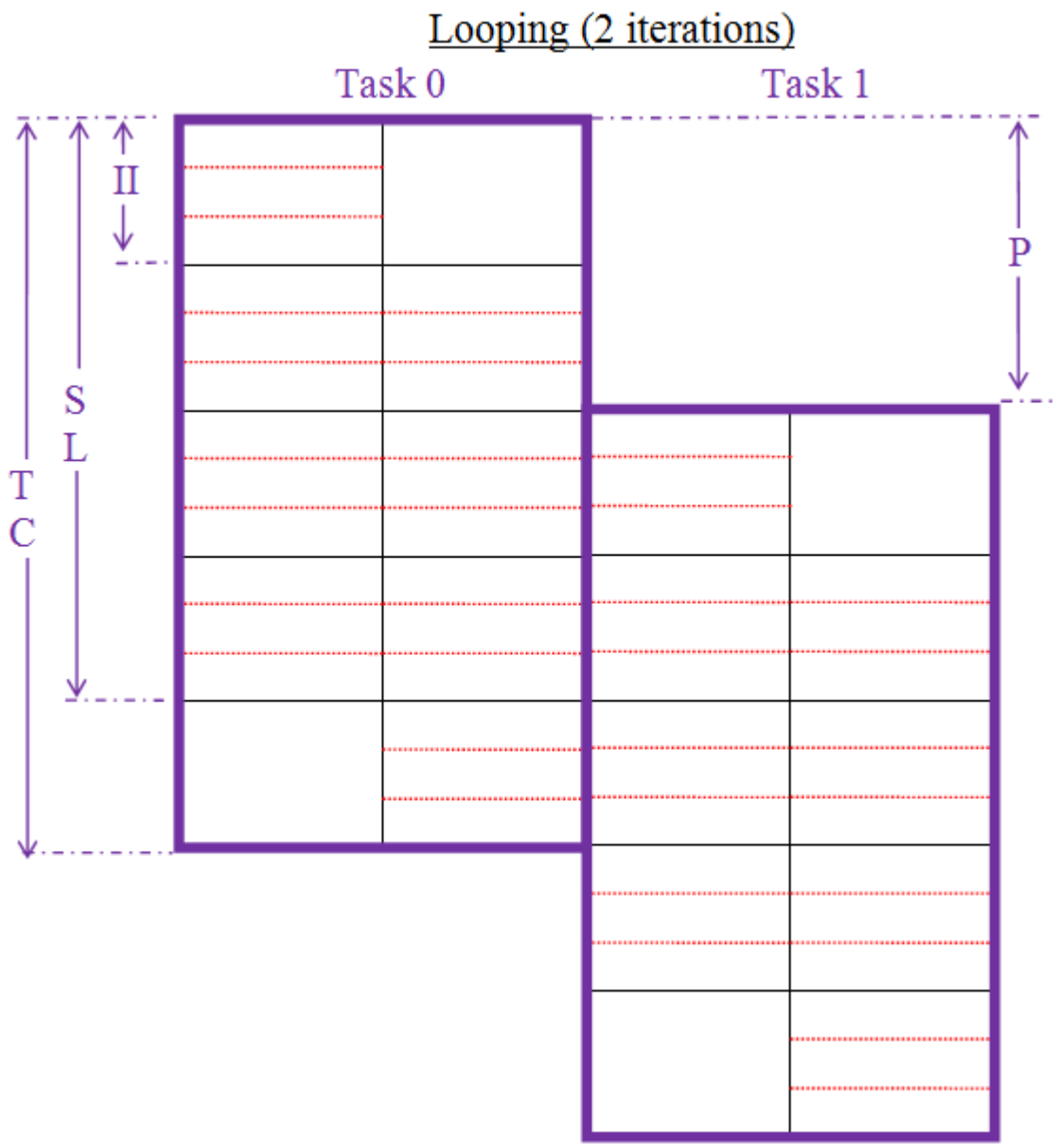
Details of IPPA Property

Constraints	Behavior
`output_production_time`	The specified value is the latency. The PPA returns the scalar outputs to the container in the same cycle as the done handshake
`resource_last_use_time`	The specified value + 1 cycle is the latency. The PPA uses its internal and shared resources (such as memory and streams) from the start cycle until the cycle before the done handshake.

TCAB Pipelinability

The pipelinability of the design for straight line code is same as the II of the TCAB. The TCABs with looping code can start the next task only after the last iteration in the current task has already started. This is shown in the following figures.


Straight-line code
Task 0
Task 1
II
P
T
C
S
L



Target Performance Specification for TCABs

Task_II and clock frequency can be specified for the TCAB synthesis similar to PPA synthesis through the implementation parameter settings. In addition, you can also specify the architectural pipelinability, input sample time, resource first use time, output production

time for the scalar outputs, and the desired resource last use time for memory and stream resources of the TCAB. These specifications can be applied globally to all the interfaces of the TCAB or specifically to a selected few. These specifications help the Synphony C Compiler tool to ease the tight feedback around TCABs wherever possible.

The architectural pipelinability is a high level specification to control the behavior of the TCAB. You can further control the TCAB performance by specification of the input sample time and output production time for scalar inputs and outputs and resource first and last use times for memories and FIFOs. For more details see the following sections:

- Architectural Pipelinability Specification
- Input Sample Time Specification
- Output Production Time Specification
- Resource First Use Time Specification
- Resource Last Use Time Specification

Architectural Pipelinability Specification

You can specify the architectural pipelinability for a TCAB using the following command.

```
set_params -architectural_pipelinability num
```

num must be greater than 0.

The default behavior of the tool is to generate a shared stall domain TCAB that satisfies the specified pipelinability requirement. If that is not achievable, an independent stall domain TCAB with the specified architectural pipelinability is generated. You can force the tool to generate a shared stall domain TCAB with the specified architectural pipelinability constraint or fail if not possible by specifying:

```
set_params -allow_latency_violation no
```

Input Sample Time Specification

The Synphony C Compiler tool allows you to specify whether certain scalar inputs to a TCAB can be sampled after the TCAB has started execution on a task. This helps in achieving higher TCAB pipelinability. It also helps in scheduling tighter feedback paths involving the TCAB in the container. By reducing the register lifetime of the scalar inputs in the TCAB, it gives better costs.

You can specify the input sampling time for a number of selected TCAB variables as follows:

```
set_params -input_sample_time "name1:num1 name2:num2…"
```

The implementation parameter can also be globally specified for all the TCAB scalar inputs as follows.

```
set_params -input_sample_time option
```

The option can be either 0 (default) or infer. In the latter case, the container gets the flexibility to schedule the particular input to the TCAB with more relaxed constraints and therefore eases tight feedback.

Output Production Time Specification

When the output production time is specified for an output, the Synphony C Compiler tool tries to produce a schedule for the operations in the TCAB such that the production time for the output can be met. If such a schedule is not achieved by the scheduler due to resource or data dependences, an independent stall domain TCAB is created that will stall the caller for the required number of cycles to match the output production time specification.

The output production time specification is a "<=" specification, that is, if the Synphony C Compiler tool can schedule the operations in less number of cycles than specified by you, it will do so. The output production time seen by the compiler for such operation will be the actual achieved output production time.

Output production time number always includes a register write-back component. For example, a combinational output requirement is specified as output production time 0. Output production time can be specified through an implementation parameter. Consider the following example,

```
int glob,  mem[10], d;
int  f (int a, int *b, int *c) {
   …
   glob = …
   mem[i] = …
   *b = …
   *c = …
}
d = f(a,&b,&c);
```

You can specify the output production time as

```
set_params -output_production_time x (x in cycles)
```

This specification applies to all scalar outputs of the TCAB. It is incorrect to specify output production time for any item other than scalar outputs. There is no default value assumed for the output production time. When you do not specify the output production time, the output production time seen by the compiler will be the actual output production time.

```
set_params -output_production_time "b:3 c:2 glob:2"
```

The specification implies that the output production time for b is 3 cycles, for c is 2 cycles, and for glob is 2 cycles.

The output production time on the function return value is associated with the name of the function. For example, if the TCAB procedure is named “hadamard,” the output production time on return value can be specified as follows:

```
set_params -output_production_time "hadamard:2"
```

Resource First Use Time Specification

The Synphony C Compiler tool allows you to specify whether certain memories or streams to a TCAB can be sampled after the TCAB issue. This helps in achieving higher TCAB pipelinability. It also helps in scheduling tighter feedback involving the TCAB in the container.

You can specify the resource first use time for a number of selected TCAB variables as follows:

```
set_params -resource_first_use_time "[<array1| stream1>
 <0 | infer>] [<array2 | stream2> <0 | infer>] .."
```

The implementation parameter can also be globally specified for all the TCAB resources globally as follows.

```
set_params -resource_first_use_time option
```

The option can be either 0 (default) or infer. In infer mode, the container has the flexibility to schedule the input to the TCAB with more relaxed constraints, and thus can ease tight feedback.

Resource Last Use Time Specification

Output production time applies only to scalar outputs. For memories and streams, resource last use time is specified. It is incorrect to specify last use time for scalar outputs. Resource last use time is the delay (in cycles) from the operation issue to the resource release. When the resource last use time is specified for a resource, the Synphony C Compiler tool tries to produce a schedule for the operations in the TCAB such that the resource can be released at a desired time. If such a schedule is not achieved by the scheduler due to resource or data dependences, an independent stall domain TCAB is created that stalls the caller for the required number of cycles to match the resource last use time specification.

The last_use_time specification is a "<=" specification, that is if the Synphony C Compiler tool can schedule the operations such that the output can be produced in less number of cycles than specified by you, it can do so. The last use time seen by the compiler for such an operation is the actual achieved last use time. In addition, the last use time is always specified 1 less than the number of clock cycles of the last usage desired.

Last use time can be specified through an implementation parameter. Consider the following example,

```
int glob,  mem[10], cache[10], d;
int  f (int a, int *b, int *c) {
   …
   mem[i] = ...
   *b = …
   *c = …
}
d = f(a,&b,&c);
```

You can specify the resource last use time as

```
set_params -resource_last_use_time x (x in cycles)
```

This specification applies to all memory/stream resources of the TCAB. There is no default value assumed for the resource last use time. The resource last use time seen by the compiler is the actual release time for the memory/stream when you do not specify the last use time value.

```
set_params -resource_last_use_time "mem:2 cache:1"
```

This specification implies that the last use time for mem is 2 cycles, and for cache is 1 cycle. Note that last use time cannot be specified on the function return value as the memory or stream return from functions is not allowed.

Example

The following example shows a DCT transformation function that has been synthesized as a TCAB and then integrated into the PPA.

```
void dct()
{
   …
   for(block_pair= 0; block_pair < NBLOCKS/2 + 1; block_pair++){
      for (odd = 0; odd < 2; odd++) {
         for (row_col = 0; row_col < 8; row_col++) {
            if(block_pair < NBLOCKS/2) {
               // read initial data in row major order
               dct_transform(r_in,r_out);
              // write data to memory in column major order
            }
            if(!(first | last)) {
               // read stored data in row major order
               dct_transform(c_in, c_out);
            }
         }
      }
   }
}
```

```
void dct_transform(short in[8],short out[8])
{
// actual transformation
}
```

The Synphony C Compiler commands used to build and use the TCAB are given in sequence as follows:

```
   # Project parameter settings
   set_params -sources "dct_main.c dct.c streamproc.c"
   set_params -headers "dct.h"

   # TCAB implementation creation
   new_implementation exp1

   #TCAB settings
   set_params -appfiles dct.c -target dct_transform
   set_params -task_II 1 -resource_ last_use_time 0

   # Specification to build the current implementation as TCAB
   set_params -type "tcab"

   # Build the TCAB
   build
   create_rtl_package

   #Create top level implementation
  new_implementation exp2

   # Set implementation parameters for the top level function
   set_params -appfiles dct.c -target dct
   set_params -task_II 2400

   # Import the TCAB implementation
   set_params -import_tcab "exp1"

   # Build the top level hardware
   build
   verify
```

Target Performance Specification for IPPAs

An importable PPA is treated as a scheduled operation in the container PA of the parent implementation. To achieve a certain performance level in the container and/or the imported PPA, it might be necessary to specify constraints. These constraints are related to importing a PPA into another implementation and are specified while building or importing the importable PPA. Constraints specified at build time are treated as defaults and can be overridden at import time. These constraints do not impact the importable PPA hardware; however it impacts how the block is treated while importing the block.

The control interface between the importable PPA and its container consists of a start handshake and a done handshake. When the importable PPA operation is scheduled, the schedule time specifies the time of the start handshake. While importing the PPA, you can only control the number of container schedule cycles between the start handshake and the done handshake, which is also referred as latency.

You can control the latency of the importable PPA by specifying either of the following parameters.

```
set_params -resource_last_use_time value (default is 0)
set_params -output_production_time value (default is 1)
```

These are the same parameters that you use to control the TCAB. These parameters are the properties of TCABs because they are built into the hardware. In case of PPAs these parameters are not part of the hardware; the parameters only control the latency. So, you can control the scheduled latency of an importable PPA at import time and experiment with different latency values without rebuilding the importable PPA. You can specify the parameters to control latency either while building the PPA or while importing the PPA.

The tool interprets these parameters follows:

- When you specify only the -resource_last_use_time option, the tool uses the specified value + 1 cycle as the latency.

  The PPA uses its internal and shared resources (such as memory and streams) from the start cycle until the cycle before the done handshake.

- When you specify only the -output_production_time option, the tool uses the specified value as the latency

  The PPA returns the scalar outputs to the container in the same cycle as the done handshake.

- When you specify both the -resource_last_use_time and the -output_production_time options, the tool uses the larger of the two latency values specified.

Use one of the parameters consistently.

TCAB Input/Output Controls

When a TCAB is synthesized at a target frequency, the Synphony C Compiler tool ensures that the logic delay at the boundary of the TCAB does not exceed the clock period. However, when such TCABs are imported into their containers, it is sometimes desirable to have an upper bound on such delays to enable logic chaining at the boundary of the TCAB in the container. The Synphony C Compiler tool allows you to control the timing of the TCABs through external delay specifications on raw inputs and outputs. This ensures that the TCAB

uses only the remaining portion of the clock period for any logic beyond any sequential element.

You can specify the external delay constraint on scalar inputs as

```
set_params -scalar_input_delay x
```

x is the delay in nano-seconds.

This specification applies to all scalar inputs. You can specify delay constraints on individual inputs as well. For example consider the procedure synthesized as a TCAB

```
int tcab(int a,int &b) {
   b=b+1;
   return a+b;
}
```

If you want to specify an external delay constraint of 2ns on the input 'a' then the following specification can be used:

```
set_params -scalar_input_delay "a:2"
```

The Synphony C Compiler tool uses this timing constraint during scheduling and generates hardware that honors it. A boundary register is inferred if the delay constraint can be satisfied only by registering the input. You can control this behavior using the implementation parameter

```
set_params -tcab_input_registers always | never | infer
```

With '`always`' setting, a boundary register is created for all inputs irrespective of the external delay constraint. With '`infer`', the external delay constraints specified on the input if any, is used to determine whether to create a boundary register. With '`never`' the Synphony C Compiler tool tries to generate a hardware that meets the external delay constraint specified without inserting a boundary register and if it cannot, fails.

You can control timing on the outputs as well, in a similar manner using the implementation parameters

```
set_params -scalar_output_delay x
(where x is delay in nanoseconds)
```

This specification applies to all the outputs. If you want to specify an external delay constraint of 2ns on the output 'b' only, then the following specification can be used:

```
set_params -scalar_output_delay "b:2"
```

You can control the registering of outputs using the implementation parameter

```
set_params -tcab_output_registers always|never|infer>
```

Uses of TCABs

As mentioned earlier in this chapter, hierarchy of modules can be built to serve several needs. This section gives details about how TCABs can be used to serve specific requirements. The utilities shown here show some possible uses.

- Enhanced Datapath Sharing
- Reconfigurable Pipeline of Logic
- Performance-Based Logic Partitioning
- Power Savings
- Hide Complexity of Lower Blocks at Higher Levels
- Outer Loop Control

Enhanced Datapath Sharing

The Synphony C Compiler tool automatically shares expensive functional units (multipliers, dividers, etc) and introduces the right multiplexing and demultiplexing logic at the interface of the shared units. Several applications have repeating pieces of computation in which the actual functions being computed remain the same but only the input and output variables differ. In such cases, you can reduce hardware cost by sharing the entire data path (corresponding to this computation across the repeating instances) rather than just share the most expensive functional units in it. You can build the entire datapath as a separate function and first build that function as a TCAB. When you build higher level designs after importing the TCAB, the entire data path inside the TCAB gets shared. For more information on different TCAB sharing scenarios see “Sharing Hierarchical Blocks” on page 10-6.

Reconfigurable Pipeline of Logic

You can build a flexible pipeline of logic with execution order being different for different blocks of code. The execution order depends on the mode of operation. Consider the example of a design where filtering needs to be applied in both horizontal and vertical dimensions but the order of filtering is not known ahead of time. You can build the horizontal and vertical filter blocks as TCABs and import it in a higher level design as follows.

```
void filter(unsigned int rows, unsigned int cols, unsigned char mode)
{
   unsigned char k;
   for(k = 0; k < 1; k++) {
      if(mode==1)
         filter_hz(rows, cols, mode); break;
      else
         filter_vt(rows, cols, mode); break;
   }
   for(k = 0; k < 1; k++) {
      if (mode==1)  filter_vt(rows, cols, mode);
      else  filter_hz(rows, cols, mode);
      }
   }
```

The following steps need to be done to accomplish the reconfigurable pipeline.

1. Build the filter_vt and filter_hz as TCABs.
2. Import the horizontal and vertical filter implementations into the implementation which builds hardware for filter( ) function.

Performance-Based Logic Partitioning

You may want to synthesize a part of a procedure at a particular II and a different part of the same procedure at a different II. In this case, it would be useful to encapsulate these different parts into your own procedures and synthesize them as TCABs at different IIs.

Power Savings

Several designs contain functional blocks which are used infrequently. You can isolate such blocks from the rest of the design and save power by gating the clock to the infrequently used module when it is not in use. You can use the following implementation parameter at the time of building a TCAB.

```
set_params -clock_gating yes
```

This creates additional clock_gate_enable signals at the interface of the TCAB which can be used to determine when the TCAB is not active. A TCAB is considered active only if it is processing any data or if it is stalling due to some data dependency. When a TCAB which has a gated clock is imported into a higher level design, the additional clock gating port for the TCAB is propagated to the higher levels automatically. Each instance of the TCAB inside the design gets separate clock gate enable ports at the higher level design interface.

Hide Complexity of Lower Blocks at Higher Levels

Your program may require a lot of inlining and unrolling to achieve the desired performance. Inlining and unrolling increases the program size, thereby increasing the problem size for various optimizations done by the tool and the scheduling of operations. This may in turn lead to increased runtime and memory requirements of the tool. By synthesizing some of the procedures as independent TCABs and then integrating them inside a bigger TCAB or PPA, the problem size can be limited and better runtimes can be achieved. The higher level design treats the entire TCAB as one functional unit with specific input and output relationship at its interface. This effectively masks the complexity of the lower level modules (built as TCABs) when higher level designs are being synthesized.

Outer Loop Control

If you have a design with several parallel sub-loops, the Synphony C Compiler tool creates hardware which executes all those inner loop nests sequentially even though there are really no data dependences between the sub-loops. In such cases, you may build the several parallel loops as TCABs and import them into the higher level design. If there are no data dependences between the TCABs, all the code in the TCABs execute in parallel and thus improve performance.

Consider the following code segment where the inner loops encode data and pack the data separately.

```
void processPacket(){
   for (int i=0;i<10;i++){ // Outer loop
      for (int j=0;j<10;j++) {  // Code block 1
         encode_block();
      }
      for (int j=0;j<10;j++) { // Code block 2
         pack_data();
      }
   }
}
```

This kind of code executes all the encode_block() functionality before starting execution of pack_data() functionality. This sequential execution introduces additional memory requirements to buffer additional data between the inner loops. You can improve QoR and performance of this unit by synthesizing the inner loops separately as two different TCABs and just importing them into the processPacket() function as follows.

```
void encode_loop(){
   for (int j=0;j<10;j++) {  // Code block 1
      encode_block();
   }
}
void pack_data_block(){
   for (int j=0;j<10;j++) {  // Code block 2
      pack_data();
   }
}
```

And then import this TCAB into the top-level design as follows.

```
void processPacket(){
   for (int i=0;i<10;i++){ // Outer loop
      encode_block(); // TCAB call
      pack_data_block(); // TCAB call
   }
}
```

This code transformation and design processing to include TCABs cause different TCAB blocks executing in parallel to increase performance.

TCAB Sharing and Inlining

The following points describe some aspects of TCAB sharing and inlining:

1. Only TCABs are hierarchically shared. Other expensive functional units like multipliers, dividers, etc. inside a TCAB are shared across hierarchies only if they are built as TCABs.
2. Hierarchical sharing of a block has some restrictions with respect to the stall domains. In the following description, a child refers to the child block shared hierarchically across different containers (call trees). SSD refers to shared stall domain and ISD refers to independent stall domain.
   - A child is an ISD and can be shared without any restriction of container stall domains.
   - A child is an SSD and is called from two call trees C0 and C1:
     - C0 and C1 contain no ISDs and have only SSDs—a child can be shared.
     - C0 and C1 have an ISD somewhere in the call tree—a child can be shared.
     - One of the call trees has an ISD and other does not—a child can be shared; you can achieve sharing by either making the child or some container an ISD.
     - One of the call trees has an ISD and other is a direct call from container—a child cannot be shared. You can achieve sharing by either making the child an ISD or introducing an ISD in the path of direct call from the container.

- One of the call trees has only SSDs and other is a direct call from container—a child can be shared.

3. TCABs which take in array parameters need to satisfy additional constraints in order to be shared.

   - If a TCAB T has an externally instantiated child TCAB and that child TCAB takes in the formal array parameter of T as actual argument, such TCAB cannot share operations with different array actuals. Consider the following code segment where the functions Q and R are built as TCABs.

     ```
     Q(int f[])
     {
        R(f);
     }
     P()
     {
        // mutually exclusive calls to TCAB Q.
        If (mode) {
           Q(A);
        } else {
           Q(B);
        }
     }
     ```

     In this example, if TCAB R is externalized out of TCAB Q, you cannot get a single instance of TCAB Q since TCAB R which is externalized takes in the same parameter as Q.

   - If a TCAB T has a child TCAB to which its formal array parameter is passed as an argument and the calls to the T overlap, this TCAB cannot be shared. Consider the following code segment where the functions P, Q and R are built as TCABs.

     ```
     Q(int f[])
     {
        R(f);
     }
     P()
     {
        for (i=0;i<10;i++) {
           …
           if (mode) {
              Q(A);
           } else {
              Q(B);
           }
           …
        }
     }
     ```

     In this example, multiple iterations of loop in TCAB P are executed in parallel, causing TCAB Q to be called with different array parameters in overlapped fashion. This

example creates two instances of TCAB Q in TCAB P (P could be anything. TCABs of interest are Q and R only).

This restriction applies even for arrays marked with the `constant_across_pipelined_calls` pragma.

4. TCABs provides hierarchical synthesis capability which allows maximum sharing, inlining a function instead enables optimization of the design in software itself, and simplifies the design in some cases. If a function is called with constant parameters in several places in your code, consider inlining such functions.

TCAB-Specific Parameters

The Synphony C Compiler tool provides the following additional implementation parameters for TCABs:

- `set_params -force_independent_stalldomain_tcab yes|no`

  This implementation parameter specifies whether the TCAB should be forced to be in its own stall domain or not. This helps in achieving better dynamic performance due to overlapped stalls of TCAB and container. This increases the cost and therefore should be judiciously used.

- `set_params -latency_independent_schedule yes|no`

  This implementation parameter specifies whether the scheduling should be aware of output production time or resource last use time or not. Latency unaware schedule could result in cheaper hardware but would result in stalls.

- `set_params`
  `-input_stall_path_insulation always | never | infer` *(default)*

  This implementation parameter applies only to independent stall domain TCABs. The TCAB contains a stallbar_in input port to receive container stalls. The stall path is typically a high-fanout net and can therefore impact timing. This option allows the tool to register the stallbar_in port inside the TCAB before distributing into TCAB internals, thereby insulating the stall input path. The setting `always` or `never` causes the tool to always or never insulate the stall path on the input of the TCAB. The `infer` setting causes the tool to insulate the stall path on the input if there is no performance or cost overhead.

- `set_params`
  `-output_stall_path_insulation always | never | infer` *(default)*

  This implementation parameter applies only to independent stall domain TCABs. The TCAB contains a stallbar_out output port to stall the container. The stall path is typically a high-fanout net and can therefore impact timing. This option allows the tool to register the stallbar_out port inside the TCAB before sending it out to the container, thereby

insulating the stall output path. The setting `always` or `never` causes the tool to always or never insulate the stall path on the output of the TCAB. The `infer` setting causes the tool to insulate the stall path on the output if there is no performance or cost overhead.

Classifications of TCABs

This section describes the different kinds of TCAB classifications. Any TCAB module can be classified under one of the following category:

- Combinational and Sequential TCABs
- Pipelined and Nonpipelined TCABs
- Shared Stall Domain and Independent Stall Domain TCABs

Combinational and Sequential TCABs

Combinational TCABs are capable of producing its outputs in the same cycle of valid inputs. There are no sequential elements (registers, flip-flops, etc.) between any input and output ports. Combinational units can be created only for procedures with acyclic code. If a combinational function unit is desired, Task_II =1 can be specified to prevent the II based sharing. In addition, you can specify 0 for the output production time or the resource last use time (depending on whether it has scalar output or not) of 0 may be specified to tell the Synphony C Compiler tool that a combinational unit is desired.

Sequential TCABs have at least one memory element in any of the paths between inputs and outputs. Most TCABs that contain non-trivial operations usually are synthesized as sequential TCABs.

Even if a TCAB is not purely combinational, there could be paths between inputs and outputs which do not have any sequential element. If you want to force all such paths to be sequential, you can do any of the following.

- Register inputs by using the following implementation parameter

  ```
  set_params -tcab_input_registers always
  ```

- Register outputs by using the following implementation parameter

  ```
  set_params -tcab_output_registers always
  ```

Pipelined and Nonpipelined TCABs

The Synphony C Compiler tool allows pipelined execution of the sequential TCABs. You can specify Task_II to drive the desired pipelinability of the TCAB. The pipelinability of the TCAB

depends on the II at which the TCAB is designed, output production time, resource_last_use_time constraints specified by you for the outputs and the access patterns of resources like memories and streams by the TCAB. For II>1 design, resources can be shared between each of the II time slots. Therefore, the generated TCAB can accept new operations only at II boundaries.

When creating TCAB for a procedure that contains a loop nest (after unroll), as the loop iterations themselves are issued in a pipelined manner on the TCAB, task pipelining is possible only in the epilog stage of this loop pipeline, that is, only after the last iteration of the loop nest has finished executing its first stage. The pipelinability in this case depends on the loop iteration count as well. For a variable trip count loop, where the maximum iteration count is unspecified, the pipelinability computation assumes a single trip count loop. The hardware generated is an independent stall domain TCAB that stalls the container if the loop executes for multiple iterations. The synthesized TCAB is automatically designed for epilog stage pipelining unless the unit is required to be in an independent stall domain

If a TCAB uses a module in both the first and the last cycles of execution, the next task cannot be started till the current TCAB task completes execution. Such TCABs cannot pipeline a set of tasks on the hardware.

Shared Stall Domain and Independent Stall Domain TCABs

A shared stall domain TCAB mimics the stalling behavior of its caller. On the other hand, an independent stall domain TCAB is not stalled due to the stalls of its caller and it does not stall the caller for its own stalls. The following rules describe the stalling behavior of independent stall domain TCABs.

- This TCAB stalls itself when an output is ready and the caller is not yet ready to accept it
- This TCAB stalls the caller when the caller is ready to accept data but the TCAB output is not yet ready to be sampled.

Typically, the TCABs created by the Synphony C Compiler tool execute in the same stall domain as the caller. An independent stall domain TCAB gets created under the following conditions:

- While synthesizing a TCAB, you have the flexibility to specify a latency (and therefore a degree of pipelinability) that is not achievable by hardware.
- If a TCAB has an unknown number of iterations or if there are conditional break statements which cause loops to terminate earlier than their final iteration
- If you explicitly specify to the Synphony C Compiler tool that the hardware needs to be built as a independent stall domain TCAB using the following implementation parameter.

  ```
  set_params -force_independent_stalldomain_tcab yes | no
  ```

In such cases, the Synphony C Compiler tool produces a TCAB artifact that conforms to the desired latency and degree of pipelinability, at the cost of dynamic performance and lack of dynamic-pipelinability.

This is possible because the TCAB artifact executes in its own stall domain that is independent of the stall domain of the artifact instantiating it. The specified latency is met by means of stalling the instantiating artifact at the specified latency, and holding the stall until the TCAB is ready with the output or release event. For the duration of the stall, the TCAB continues execution by virtue of being in its own stall domain.

Only an output production time of greater than 1 is possible with independent stall domain TCABs. If you had specified output production time or resource last use time that leads to a deadlock, then a conservative modification of the latencies are done to avoid deadlocks with a user warning.

As explained, an independent stall domain unit is produced as a result of output production time or resource last use time violations. In the absence of latency violations, you can also choose to force an independent stall domain TCAB, which could be useful to get improved performance using overlapping of stalls between the container and the TCAB.

If you had specified output production time or resource last use time that cannot be achieved, the generated TCAB would be an independent stall domain unit that issues stalls to the container at the specified output latency and holds the stall until the achieved latency.

11

Channel Synthesis

Channel synthesis is a feature that can help you implement a custom interface having a specified protocol. You can model complex interface at the transaction level without being concerned about the low-level implementation and signal handshaking details.

The chapter is organized as follows:

- Introduction
- Predefined SCC Classes for Channel
- Channel Examples
- Channel Simulation
- Channel Build Flow
- Implementation Parameter Support
- Limitations

Introduction

Using the channel synthesis feature of Symphony C Compiler, you can design a specific internal or external interface and hide the implementation within a C++ class abstraction. You can write your own channel derived from a provided SCC `scc_channel_if` class. You specify the implementation of the channel in a special method called `execute`, which is translated into a free-running hardware block by SCC.

Channels interact with PAs, other channels, or PPA ports primarily via stream interfaces. The channel may also connect to `stable` scalars. It can access memory and scalars internal to that channel (class member variables of the channel and local variables of the `execute` method). Channels are imported into a PPA.

Predefined SCC Classes for Channel

Synphony C Compiler provides following predefined classes that can be used for modeling:

- `scc_channel_if`
- `scc_instream_if<`*`data type T`*`>`
- `scc_outstream_if<`*`data type T`*`>`
- `scc_port<`*`stream_interface_type`*`>`
- `scc_in<`*`data type T`*`>`

SCC provides additional stream classes that can be used for connecting channels and simulating them:

- `scc_stream_with_get_trigger<`*`data_type T`*`>`
- `scc_stream_with_put_trigger<`*`data_type T`*`>`

These classes are derived from `scc_stream` and provide a trigger mechanism with get/put methods to execute the channel connected to these streams.

The predefined classes and additional stream classes are described in the following subsections.

scc_channel_if

Any user-defined class derived out of the predefined class `scc_channel_if` is treated as an SCC channel. `scc_channel_if` is an abstract class with two pure virtual methods, `init` and `execute`. A user-defined channel should define these two methods.

The body of the `init` method should contain the synthesizable code to be executed on system initialization. You should call this `init` method from the channel constructor. The body of the `execute` method should contain the actual implementation of the channel functionality. The `init` and `execute` methods should contain only synthesizable code.

```
class my_channel: public scc_channel_if {
public:
     void execute() {
     }
     void init() {
     }
};
```

scc_port and scc_in

The `scc_port` and `scc_in` classes are provided to define the interfaces of the channel. `scc_port<stream_interface_type<T>>` can be used to represent the stream interface of the channel and `scc_in<data_type T>` can be used to represent the constant input port to the channel and considered as `stable` by default. The CAPC value can change only after the channel drains completely. The imported channel's CAPC input can read a value from a loop or host.

`scc_port<stream_interface_type<T>>` can be connected to `scc_stream<T>` of the same data type, and the stream port direction is defined by the *`stream_interface_type`*.

```
class my_channel: public scc_channel_if {
public:
     scc_port<interface_type> datain;
     scc_port<interface_type> dataout;
     scc_in<int> const_param;
};
```

scc_instream_if and scc_outstream_if

`scc_instream_if` and `scc_outstream_if` are the interface classes provided to represent the type of input and output ports of a streaming interface. `scc_instream_if` provides the operations `get`, `nb_get`, `nb_can_get`, `peek`, and `nb_peek` on the port, while `scc_outstream_if` provides the operations `put`, `nb_put`, and `nb_can_put` on the output stream port. Stream ports of a channel must have the `scc_instream_if` or `scc_outstream_if` interface type.

```
class my_channel: public scc_channel_if {
public:
     scc_port<scc_instream_if<unsigned_char> > datain;
     scc_port<scc_outstream_if<unsigned_int> > dataout;
};
```

scc_stream_with_get_trigger and scc_stream_with_put_trigger

The `scc_stream` with-trigger classes are provided to call the `execute` method of `scc_channel`. In sequential simulation, the `execute` method of `scc_channel` is called via the `get` or `put` functions of the `scc_stream` connected to the ports of that channel. For correct simulation, you must provide the binding of `scc_stream` with `scc_channel` in your code.

When `scc_stream_with_get_trigger<T>` is bound to `scc_port<T>` of `scc_channel`, the `execute` method is called whenever `get/nb_get/nb_can_get/peek/nb_peek` is called for that `scc_stream`.

When `scc_stream_with_put_trigger<T>` is bound to `scc_port<T>` of `scc_channel`, the `execute` method is called whenever `put/nb_put/nb_can_put` is called for that `scc_stream`.

Channel Examples

When you create a channel, you need to consider the following issues:

- Channels must be derived from the `scc_channel_if` class and should have the definition for the `execute` method.
- Channels can have only `scc_stream` and PPA `stable` scalars at the interface.
- Channels can connect to SRAM provided it is a local variable of the `execute` method or a member of the channel class. Channels cannot connect to SRAMs that are accessed by other channels or PAs.
- All functional implementations for which hardware is expected should be put in the `execute()` method.
- All initialization-related behavior should be put in the `init()` method.
- Multiple loops in the `execute` method are merged by SCC into a single block, with the original loops running sequentially. This is the same as the treatment of multiple loops inside a TCAB or inside the outer loop of a PA.
- In the software, it is recommended you instantiate the channel variable and any variables bound to the channel ports outside the PPA procedure. These variables could be in global scope or passed by reference to the PPA. Note that the channel hardware is still synthesized within the PPA hardware.

  The reason for this recommendation is that most channels implement external interfaces and need to interact with the testbench. The channel's execute method may be triggered before or after the PPA call, at which point variables local to the PPA procedure have no allocated state during simulation.

For cases where the channel variable and all variables bound to the channel ports are completely isolated by PAs, the variables can be instantiated within the PPA because the channel implements an internal interface and needs no testbench access, either directly or indirectly.

- The channel can be a member of a class, and such a class can be passed as a reference parameter to the PPA. The definition of the class containing the channel must be in the synthesizable code. Binding of a channel can be done in the class constructor or inside the PPA.
- Similarly, the class with the channel member can be passed as a reference parameter to a TCAB; the TCAB must not access the channel member directly. Channel members can be accessed indirectly through the stream objects bound to the channel.

Example: Basic Channel With PPA

The following channel example performs packing of data from two different streams and inserts `START` and `STOP` code for every ten data items in the data stream. Order of packing is decided by the constant input mode. The PPA always gets the data stream with valid `START` and `STOP` codes for further processing.

```
#main.cpp
#-------------------------------------------------
#include "scc.h"
#define START 0x11
#define STOP  0xFF

using namespace scc;
class my_channel: public scc_channel_if {
  int _cnt;
  public:
  scc_port<scc_instream_if<unsigned char> > data1;
  scc_port<scc_instream_if<unsigned char> > data2;
  scc_port<scc_outstream_if<int> > dataout;
  scc_in<int> mode;

  my_channel(void) {init();}
  my_channel(scc_instream_if<unsigned char>  &in1,
      scc_instream_if<unsigned char> &in2,
      scc_outstream_if<int> &outif,
      int &const_in)
  {
    data1(in1);
    data2(in2);
    dataout(outif);
    mode(const_in);
    init();
  }
  void init() {
```

```
    _cnt = 0;
  }

  void execute() {
    int byte1, byte2;
    if(_cnt==0) {
        byte1 = START;
        byte2 = 0x0;
      _cnt++;
    } else if(_cnt==10) {
      byte1 = STOP;
      byte2 = 0x0;
      _cnt = 0;
    } else {
      _cnt++;
      byte1 = data1->get();
      byte2 = data2->get();
    }
    if((*mode)==1)
      dataout->put(byte1 | (byte2<<8));
    else
      dataout->put(byte2 | (byte1<<8));
  }
};

my_channel packer;
scc_stream <unsigned char> packer_in1;
scc_stream <unsigned char> packer_in2;
scc_stream_with_get_trigger<int> packer_out(packer);
scc_stream <int> ppa_out;

void ppa(int &mode) {
  SCC_IMPORT(&packer);
  packer.data1(packer_in1);
  packer.data2(packer_in2);
  packer.dataout(packer_out);
  packer.mode(mode);
  for(int x=0; x<100; x++) {
    ppa_out.put(packer_out.get());
  }
}

int mode;
int main() {
  for(int x=0; x<100; x++) {
    packer_in1.put(x);
    packer_in2.put(x);
  }
  mode = 1;
  ppa(mode);
  while(ppa_out.nb_can_get()) {
    printf("OUT : %.2X\n", ppa_out.get());
  }
```

```
  return 0;
  }
#----------------------------------------
```

Example: Class With Channel Member as Parameter to PPA/TCAB

In the following example, a `wrapper` class is used and a channel object is instantiated in the class. Accesses to the channel are done using helper methods. The class with the channel member is passed as a parameter to the PPA. The binding of the stream ports for the channel member of the class must be done inside the default constructor or inside the PPA, and the constructor should be included in the application files for SCC.

```
#main.cpp
#----------------------------------------

#include "scc.h"
#define START 0x11
#define STOP  0xFF

using namespace scc;
class my_channel: public scc_channel_if {
  int _cnt;
  public:
  scc_port<scc_instream_if<unsigned char> > data1;
  scc_port<scc_instream_if<unsigned char> > data2;
  scc_port<scc_outstream_if<int> > dataout;
  scc_in<int> mode;

  my_channel(void) {init();}
  void bind (scc_instream_if<unsigned char>  &in1,
      scc_instream_if<unsigned char> &in2,
      scc_outstream_if<int> &outif
      )
  {
    data1(in1);
    data2(in2);
    dataout(outif);
  }
  void bind_mode(int &const_in){
    mode(const_in);
  }
  void init() {
    _cnt = 0;
  }

  void execute() {
    int byte1, byte2;
    if(_cnt==0) {
        byte1 = START;
        byte2 = 0x0;
      _cnt++;
```

```
    } else if(_cnt==10) {
      byte1 = STOP;
      byte2 = 0x0;
      _cnt = 0;
    } else {
      _cnt++;
      byte1 = data1->get();
      byte2 = data2->get();
    }
    if((*mode)==1)
      dataout->put(byte1 | (byte2<<8));
    else
      dataout->put(byte2 | (byte1<<8));
  }
};

class wrapper {
  scc_stream <unsigned char> packer_in1;
  scc_stream <unsigned char> packer_in2;
  scc_stream_with_get_trigger<int> packer_out;
  my_channel packer;
public:
  wrapper() {
    packer.bind(packer_in1, packer_in2, packer_out);
    packer_out(packer);
  }
  void bind_mode(int &mode) {
    packer.bind_mode(mode);
  }
  int get_data() {
    return packer_out.get();
  }

  void put_in1(unsigned char in) {
    packer_in1.put(in);
  }
  void put_in2(unsigned char in) {
    packer_in2.put(in);
  }

};

scc_stream <int> ppa_out;
void ppa(int &mode, wrapper &wrap_obj) {
  wrap_obj.bind_mode(mode);
  for(int x=0; x<100; x++) {
    ppa_out.put(wrap_obj.get_data());
  }
}

int mode;
wrapper wrapper_obj;
int main() {
```

```
  for(int x=0; x<100; x++) {
    wrapper_obj.put_in1(x);
    wrapper_obj.put_in2(x);
  }
  mode = 1;
  ppa(mode, wrapper_obj);
  while(ppa_out.nb_can_get()) {
    printf("OUT : %.2X\n", ppa_out.get());
  }
  return 0;
}
```

Channel Simulation

As explained earlier, stream trigger classes are used to get the correct simulation behavior from channels, similar to free-running hardware blocks. To get the expected C simulation behavior, you need to use the stream trigger classes correctly. Incorrect usage can lead to mismatching results between C simulation and RTL simulation.

To avoid common mistakes in using stream trigger classes, observe the following points:

- When a channel is built and imported into a PPA, it becomes a part of the PPA hardware. All inputs to the channel that are not connected to any PA become input ports of the PPA. All stream outputs from a channel that are not connected to any PA become output ports of the PPA.
- Output streams from a channel connected to an input of the PA/channel should be of the `get_trigger` type so that when the PA reads from the stream, it can trigger the `execute` function inside the channel.
- Input streams to a channel connected to an output of the PA/channel should be of the `put_trigger` type so that when the PA writes to the stream, it can trigger the `execute` function inside the channel.
- An input or output stream not connected to the PA/channel should be a normal `scc_stream`.
- Channels are intended to generate free-running hardware blocks with stream interfaces, so you should have a fair idea of the blocking and nonblocking stream usage. Any incorrect use of a nonblocking call can lead to incorrect behavior at RTL simulation. It is advisable to use guarded non-blocking calls inside a channel.

Channel Build Flow

SCC channels are considered free-running hardware blocks. Most implementation parameters that are applicable to PPAs are also applicable to channels. To implement a channel, you must build it first and then import it into a PPA.

Building the Channel

To build the channel, use:

```
set_params -target channel_name
set_params -type channel
```

where *channel_name* is the class name or an `$SCC` label_type used for the instance name when the channel has a template.

In the case of the foregoing example, you should use:

```
set_params -target my_channel
set_params -type channel
```

For template type channels, you need to provide a `$SCC` label_type to the channel object and use that label in the `set_params` command. For example, if you have the `my_channel<T>` template type, then the channel object must be labeled with a unique instance name and that name must be used with the `-target` implementation parameter:

```
/* $SCC label_type my_chnl_a */
my_channel<unsigned_int> packer;
set_params -target my_chnl_a
set_params -type channel
```

If the channel is a member of a template class, you must label each template channel member explicitly and synthesize it.

You can also build the channel by using the following settings in the GUI:

- Implementation > Configure New > Application > CHANNEL Name: *channel_name*
- Implementation > Configure New > HW Settings - Build as: CHANNEL

Importing the Channel Into a PPA

After the channel is built successfully, you can import that channel into a PPA by specifying the `SCC_IMPORT(&channel_object)` macro in the PPA source and by using the `-import_channel` implementation parameter in PPA synthesis:

```
set_params -import_channel "my_channel"
```

You can import more than one channel by specifying them with `-import_channel` implementation parameter:

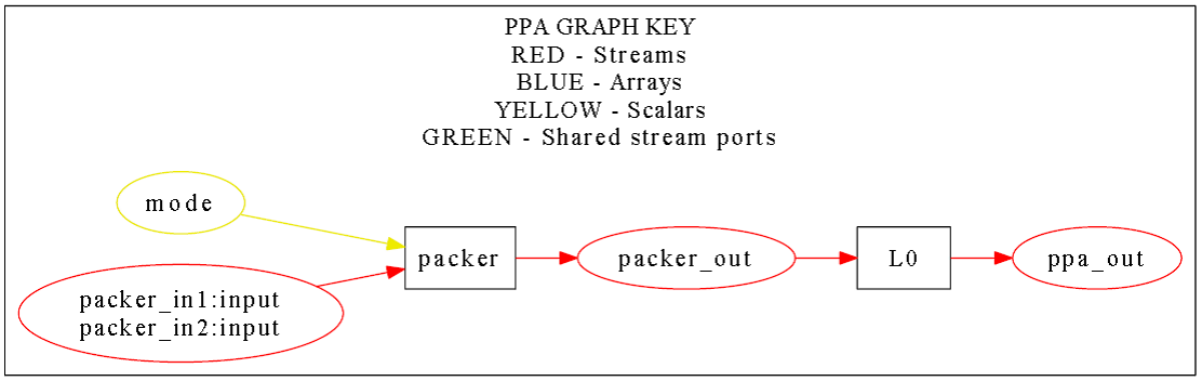
```
set_params -import_channel "my_channel instance_a"
```

You can also import a channel into a PPA implementation by using the following settings in the GUI:

Implementation > Configure New > Application > CHANNEL Import List: *channel_implementation_name*

After you successfully import the channel into the PPA, you can see the PPA graph with a channel packer as part of the generated hardware, as shown in Figure 11-1.

Figure 11-1 PPA Graph With Channel



All the reports generated during the SCC build flow are the same for channels as for PPAs.

Example: run.tcl File

The following script example demonstrates usage of the channel synthesis feature.

```
set_params -source_dir ./
set_params -sources main.cpp

set imp channel_test
if [file exists $imp] { delete_implementation $imp }
new_implementation $imp

set_params -type channel
set_params -appfile main.cpp
set_params -target my_channel
set_params -clock_freq 200

verify -golden
build
```

```
create_rtl_package

set imp ppa_with_channel
if [file exists $imp] { delete_implementation $imp }
new_implementation $imp

set_params -appfile main.cpp
set_params -proc ppa
set_params -clock_freq 200
set_params -import_channel "channel_test"

verify -golden
build
create_rtl_package
```

Implementation Parameter Support

The following implementation parameters are valid for channel synthesis:

```
always_enabled_ppa
clock_freq
edge_clock
false_path_handling
generate_abort_logic
import_tcab
init_data_registers
macrocell_prefix
memory_access_latency
memory_forward_boundary_register
memory_forward_path_external_delay
memory_return_boundary_register
memory_return_path_external_delay
no_annotate_hdl
ram_synthesis_threshold
reset_data_registers
reset_level
reset_sensitivity
rom_synthesis_threshold
system_port_name
system_port_polarity
system_port_production
techlib
user_supplied_memory_read_write_ports
wait
```

TCAB-specific implementation parameters and the following implementation parameters cannot be used in the channel synthesis flow:

```
auto_start
disable_ppa_on_host_access_to_memory
host_memory_access
```

```
host_scalar_access
hostlm_access_latency
infinite_run_ppa
integrate_block
sharable_across_blocks
task_ii
task_overlap
```

Limitations

The following limitations apply to channel synthesis:

- The `$SCC label_type` label works only when applied on a variable. It does not work on a type or a class member.
- Binding of channel ports to `scc_stream` objects or scalars must be visible inside the PPA for SCC to generate the right hardware. When a class with a channel data member is passed as a parameter to the PPA, SCC tries to infer the binding information from the default constructor of the class in the absence of any explicit binding inside the PPA.
- Data members of channels cannot be accessed inside a PPA or TCAB implementations.
- Host access is not allowed for memories and scalars inside a channel. They cannot be accessed inside the PPA or in the driver code.
- Globally scoped scalars and arrays cannot be accessed inside a channel. Only global constant arrays that map to ROMs can be accessed in a channel.
- A channel cannot have localized or static variables. Localized global variables and any variables localized to the execute method are not supported. All the channel data members are automatically localized.
- Scalar ports other than `stable` type ports for a channel are not allowed.
- Scalar ports of a channel cannot be bound to pass-by-value parameter variables inside a PPA or to a field of a struct or class type.
- Channels cannot be instantiated inside the `execute` method of a channel or any other function called by it.
- Channels cannot be imported into TCAB or channel implementations.
- Channels can be instantiated inside an unrolled loop only.
- Arrays of channels are not supported.
- Many-to-one, one-to-many, and many-to-many types of port binding for channel hookup are not supported.
- Channels cannot be shared across different objects.

- Object-specific pragmas or implementation parameters cannot be applied on channel objects. The `#pragma packed` option can only be applied on channel data members of struct or class type.
- SCC does not infer the banking of array members of a channel. Use `#pragma memory_structure` to bank the array members of a channel.
- Local channel objects inside the PPA retain state across the tasks in the hardware, unlike sequential C simulations. For the hardware and C simulations to match, the local channel objects should be marked as static if the data on the stream ports is not completely drained at the end of the task.

12

Synphony C Compiler Linting

The SCC tool performs extensive linting during the verify -lint command to make the PPA code. This chapter describes the linting process to make the PPA code compatible with the Synphony C Compiler tool and to design efficient hardware.

The chapter is organized as follows:

- Overview of Linting
- Compile-Time Syntax Checking
- Simulation-Based Semantic Checking

Overview of Linting

The Synphony C Compiler tool performs extensive linting during the `verify -lint` command to make the PPA code compliant with the requirements to synthesize efficient hardware.

The tool goes through two phases of linting:

- Compile-time syntax checking

  This phase checks that you have followed the syntax guidelines described in Chapter 6, "C/C++ Synthesizable Syntax."

- Semantic checking through simulation

  The major focus of simulation-based linting is to test the correctness of bitsize pragmas, to identify cases in which variables are used before being initialized, and to identify out-of-bounds array accesses. Since it is simulation based, this phase requires a working driver in addition to the PPA code. This phase provides quick feedback by identifying the line numbers of potential semantic errors.

  The tool treats all pragma specifications as directives to the compiler while building hardware. During the `verify -lint` command, the Synphony C Compiler tool checks for violations in the pragma specifications.

  By default, it runs linting on only those application files that were marked when you specified the implementation parameters (`set_params -appfiles`). The other unmarked source files are treated as part of the driver and are not linted. These application files can contain some driver code as well.

  If the current implementation is being built as a TCAB, the Synphony C Compiler tool limits the linting to only the code within the TCAB function and other TCABs or inlined functions called from that TCAB. If the implementation is synthesized as a PPA, the tool extends the linting to all the code in the application files.

  If you have specified a memory access dependence, the tool reports any violations of that specification during linting.

Compile-Time Syntax Checking

The first phase of linting is syntax checks performed at compile time. The tool does the following checks:

- Standard C syntax checks
- Synphony C Compiler-specific syntax checks, like checks of pragma statements

- PPA code verification, to make sure that the code follows the restrictions on traditional ANSI-C, as outlined in Chapter 6, “C/C++ Synthesizable Syntax.”

Simulation-Based Semantic Checking

This section describes the simulation-based semantic checks that the tool performs during the linting phase:

- Bitsize Pragma Checks
- Uninitialized Variable Checks
- Out-of-Bounds Array Access Checks
- Array Usage Distance Specification

Bitsize Pragma Checks

This check identifies overflow situations in the bitsize pragma and determines when a variable needs more bits to represent its value during simulation.

However, overflow does not always imply a programming error. The bitsize pragma indicates that high order bits (above the specified width) will not affect the final program output. So, a variable can overflow, if the high order bits of the overflow do not propagate to the output. If it does affect the output, the golden simulation and lint simulation outputs would not match.

After bitsize pragma checks, the tool can issue the following:

- Warnings

  Overflows might not always imply an error, but the tool prints warnings for all overflows found during simulation. This helps you isolate real overflow errors when results from golden simulation do not match the ones from linting simulation.

- Error messages

  The tool prints error messages if the overflow affects the final output and there is a mismatch between the golden simulation and lint simulation outputs.

Overflow Example that Does not Affect Final Output

The following code does not imply an error. The variable a can overflow past 9 bits as a result of the multiply, but the overflow does not affect the final output because the multiply is immediately followed by a masking operation.

```
#pragma bitsize a 9
#pragma bitsize b 7
```

```
#pragma bitsize c 7
a = b * c;
a = a & 0x1ff;
```

To reduce the number of warnings, you can rewrite the example as follows so that there is no overflow:

```
a = (b * c) & 0x1ff;
```

Overflow Example that Generates a Warning

In this example, the variable a needs only 7 bits. As a is a signed variable, the Synphony C Compiler tool assumes that the range is always between -64 and 63.

```
Driver code: main.c
#include "pico.h"
char a = 0;
extern void add( );
int main() {
   add();
   printf("a is %d\n", a);
   return 0;
}
PPA code: add.c
extern char a;
#pragma bitsize a 7
void add() {
   int i;
   for(i=0;i<10;i++)
      a += 10;
}
```

When the lint command is executed, simulation-based linting finds a time when a overflows the expected width (a is 70), and prints a warning message as opposed to an error message, because the overflow itself is not an error:

```
At iteration '7' of loop at add.c:6 in function 'add'

WARNING:add.c:7:3: Width overflow for "a", type 7 signed bits (i7),
value 70 (0x46).

preprocess.output vs. golden.output:FILES DIFFER!
```

The error occurs when the overflow affects the final output. For this example, the final output depends on the overflow value, so the tool displays the FILES DIFFER message.

Overflow Example that Generates an Error Message

Overflow Example that Generates an Error Message

The following code packages several values into a single 28-bit output:

```
PPA loop-nest
     int ci, cj;
     for (ci=0; ci<LINES; ci++) {
          for (cj=0; cj<PIXELS; cj++) {
               signed char cb = scc_stream_input_cb_bc();
               signed char cr = scc_stream_input_cr_rc();
               unsigned short int ys = scc_stream_input_ys_yc();
               unsigned char smode = ys & 0x01;
               unsigned short int y = (ys >>1) & 0x7FF;
               unsigned int out = (y<<17) | (cr <<9) | (cb<<1) | smode;
               #pragma bitsize out 28
               scc_stream_output_out(out);
     }
}
```

```
PPA loop-nest
   int ci, cj;
   for (ci=0; ci<LINES; ci++) {
      for (cj=0; cj<PIXELS; cj++) {
         signed char cb = scc_stream_input_cb_bc();
         signed char cr = scc_stream_input_cr_rc();
         unsigned short int ys = scc_stream_input_ys_yc();
         unsigned char smode = ys & 0x01;
         unsigned short int y = (ys >>1) & 0x7FF;
         unsigned int out = (y<<17) | (cr <<9) | (cb<<1) | smode;
         #pragma bitsize out 28
         out.put(out);
   }
}
```

Simulation-based linting catches common ANSI-C coding errors. In this case, the error was not allowing for sign extension. When a signed variable is used as an argument to a | operation, the variable is first sign-extended to 32-bits. This caused an overflow in bits 29-32 of the output.

You can fix the error and have simulation pass by rewriting the assignment to out as shown:

```
out = (y<<17) | ((cr &0xff) <<9) | ((cb & 0xff) <<1) | smode;
```

Uninitialized Variable Checks

The linter checks uninitialized variables. The tool treats any ROM array offsets marked with #pragma dont_care_code as uninitialized array locations. If you specify that variables are not inputs and outputs with #pragma not_input and #pragma not_output the linter treats them as uninitialized variables at the entry to the PPA, and garbage values at the exit.

There are two main reasons to check uninitialized variables:

- Default compiler settings for uninitialized variables

In ANSI-C, the initial value of a variable is undefined. Many compilers set uninitialized variables to zero by default. As a result, automatically assuming zero initialization, either on purpose or by accident, can lead to unpredictable results. This error might not be caught until RTL simulation.

- Reentrant scopes with local declarations, like the body of a loop

  In ANSI-C the initial value of a variable is undefined on each entry into the scope in which it is declared, as in the case of each loop iteration for a variable declared in the inner-most loop. However, many compilers maintain the values of such variables across multiple executions of the same scope. Making an assumption about loop-carried state for such variables can lead to unpredictable results.

Out-of-Bounds Array Access Checks

The linter checks for out-of-bounds array references at runtime. For example, consider the declaration int X[10][10];. In ANSI-C, a dynamic reference to X[2][10] is the same as a reference to X[3][0], but in the Synphony C Compiler tool a dynamic reference to location X[2][10] is undefined. The linter catches this and reports it as an error.

Array Access Check Example

The following example is a simple image filter that performs pair-wise averaging of all pixel columns of a 480x576 image.

PPA File: smooth.c

```
unsigned char image_in[480][576], image_out[480][576];
void smooth() {
   int i,j;
   for(i=0;i<480; i++) {
      for(j=0;j<576; j++) {
         image_out[i][j] = (image_in[i][j-1] + image_in[i][j]) / 2;
      }
   }
}
```

Driver Code: main.c

```
#include "pico.h"
extern unsigned char image_in[480][576], image_out[480][576];
int main( ) {
   int i,j,ppaid;
   for(i=0;i<480; i++) {
      for(j=0;j<576; j++) {
         image_in[i][j] = (2*j) % 256;
      }
   }
smooth( );
for(i=0;i<480; i++) {
   printf("%3d: ", i);
   for(j=0;j<576; j++) {
      printf("%d ", image_out[i][j]);
   }
   printf("\n");
}
return 0;
}
```

In this case, the boundary cases are not handled correctly, so the linter finds an out-of-bounds reference to image_in[0][-1]. This is a true error, because the out-of-bounds reference leads to a difference between the golden and preprocess outputs, as shown in the following comparison of the first few bytes of each of them:

Golden Output	Preprocess Output
0: 0 1 3 5 7 9 11 …	0: 63 1 3 5 7 9 11 …

The out-of-bounds reference is the cause of the big discrepancy in the first byte: 0 vs 63. The linter does not access location image_in[0][-1] but instead returns garbage data. In this case the garbage data affected the final output and simulation failed, so the linter generated the error message.

You can rewrite the inner loop as follows:

```
image_out[i][j] = (image_in[i][j] + image_in[i][j+1]) / 2;
```

However, this too is incorrect, and results in a different error being generated after the lint command is run:

```
smooth.c:6:Warning: Out of bounds error, ignoring reference to
“image_in[480][0]”.

preprocess.output vs. golden.output:FILES DIFFER!
```

In this case, the actual out-of-bounds reference is image_in[479][576], which is treated as location image_in[480][0] by both the linter and ANSI-C. The simulation fails because this is out of bounds.

The following illustrates one correct way of writing the inner-loop code for this example. If you write the code in this way, the lint command and all levels of simulation pass.

```
image_out[i][j]=(image_in[i][j]+((j<575) ? image_in[i][j+1] : 0)) / 2;
```

Array Usage Distance Specification

When you use the `no_dependence` pragma to specify memory access dependences, the tool reports any violations. This checking finds any user errors in the distance specification.

Consider the following example.

```
for (i = 0; i < 8; i++) {
  x[i]   = d*i;
  x[i+d] = (d*i)>>2;


i = 0  Write x[0], Write X[d]
i = 1              Write x[1], Write X[d+1]
i = 2                          Write x[2],   Write X[d+2]
```

This code has two writes to the array `x` in each loop iteration. The variable `d` is a parameter to the PPA function and is always nonzero. This means that in each iteration of the loop, each access to `x[]` is to a different location. However, because `d` is a function parameter, the tool cannot prove that these writes are to different locations. Consequently, the tool cannot achieve an iteration interval of 1 (II=1) for this loop.

Because `d` is always nonzero, it is clear that both writes in each iteration are nonaliasing and the distance between successive writes to same location is one iteration.

To achieve II=1, use the `no_dependence` pragma to specify the correct distance between the two writes to `x[]`. This helps improve scheduling of memory accesses. For example, you can use the pragma as follows:

```
unsigned int x[16];
#pragma no_dependence x write:w1 write:w2 1
void ppa(unsigned int d) {
    int i;
    for (i = 0; i < 8; i++) {
        /* $SCC label_begin w1 */
        x[i]   =  d*i;
        /* $SCC label_end w1 */
        /* $SCC label_begin w2 */
        x[i+d] = (d*i)>>2;
        /* $SCC label_end w2 */
```

```
    }
}
```

If you simulate the design with the wrong input “`d=0`”, the `no_dependence` specification is proven wrong and `verify -rtl` generates an error message similar to the following:

```
ERROR
At iteration '1' of loop at main.cpp:11 in function 'ppa'
ERROR: Dependence violation
 From: write at x[0] at main.cpp:13:12:
 To: write at x[0] at main.cpp:16:12:
 Actual distance is 0; user specified distance is 1
END ERROR
```

13

Analysis Views and Reports

To obtain better performance with good QoR, it is important to understand the design and the choices made by the tool. The tool generates several reports and GUI views during various build and simulation steps to help you analyze the design. This chapter discusses the following:

- Overview of Generated Reports
- Design Summary Reports
- User Code Reports
- Internal Timing Report
- Integration Reports
- QoR Analysis Reports
- Performance Analysis Reports
- C to RTL Mapping
- Power Estimation Reports
- Verilog Simulation Analysis Reports

Overview of Generated Reports

This overview contains an alphabetical list of the reports, and information about how the reports are organized.

Alphabetical List of Reports

The following table lists the Synphony C Compiler reports in alphabetical order, along with equivalent analysis views and the phase in which they were generated.

Report	Generated Phase	Equivalent View
report_address_map.txt	synthesize	-
report_constraints.txt	preprocess	-
report_cost.txt	synthesize	Resource Browser
report_false_paths.txt	synthesize	
report_feedback_path.txt	synthesize	Feedback Path viewer
report_hierarchy.pdf and report_hierarchy_detailed.pdf	synthesize	Hierarchy Graph viewer
report_interface.txt	synthesize	-
report_interface_timing.txt	synthesize	-
report_memory_access_trace.txt	verify -rtl	-
report_original_loop_to_line_number.txt	preprocess	-
report_perfectized_loop_info.txt	schedule	-
report_performance.txt	synthesize	-
report_power_theater_offline.txt report_power_theater_sv.txt	RTL offline and SV simulation	-
report_pseudo_code.txt	synthesize	Operation Schedule viewer
report_rate_match.txt	schedule	-

Report	Generated Phase	Equivalent View
report_register.txt	synthesize	-
report_spyglass_offline.txt	RTL offline simulation	-
report_summary.txt	Updated throughout Build and Simulation (RTL simulation)	-
report_verify_rtl_performance.txt	SV simulation	Dynamic performance graph (SV simulation)

Organization of the Reports

All reports can be generated using the report_implementation Tcl command, which is described in "report_implementation" on page 3-14. The reports in this chapter are organized by their intended purpose and are as follows:

- Design Summary Reports
- User Code Reports
- Internal Timing Report
- Integration Reports
- QoR Analysis Reports
- Performance Analysis Reports
- C to RTL Mapping
- Power Estimation Reports
- Verilog Simulation Analysis Reports

Design Summary Reports

For design summary information, see these two reports:

report_constraints.txt	Captures the input constraints for the application.
report_summary.txt:	Contains the summary of the hardware generated and simulation performances.

report_constraints.txt

This report captures the constraints you specified in the design (code, memories, and FIFOs). It also lists any implementation parameters and pragmas that affect the code, memories and FIFOs. The pragmas that you specify here override any application-wide you might have set with implementation parameters. This report consists of the following sections:

- General Section
- Memory Specification Section
- Stream Specification Section

General Section

This section describes your general specifications for the entire application, and includes the following:

- Procedure to by synthesized
- Operational frequency
- Target performance (Task_II specified in number of cycles)

  If you do not specify a Task_II for the implementation, the tool tries to achieve maximum performance while meeting all the other constraints. In such cases, the Task_II value in this report will show as 0.
- Base address of the PPA.

A sample of the general section is shown.

```
START OF CONSTRAINTS
====================
General Specifications:
   proc: wlan_top_level
   miti: 0
   : 200
   base_address: 0x1000
```

Memory Specification Section

The next section describes the constraints on different memories in the application. It first lists the application-wide settings, followed by the overriding specifications for individual arrays. This section includes information about the RAM or ROM synthesis thresholds, memory-porting constraints, boundary register specifications, interface protocol for host access, external delay specifications on forward and return paths, specification of any flags that disable any compiler analysis on the memory, and so on. A sample of this section is shown.

```
Memory Specifications:
all arrays
hardware

ram_synthesis_threshold: 16:256
   rom_synthesis_threshold: 256:4096
   user_supplied_memory_read_write_ports: combined
   memory_forward_boundary_register: never
   memory_return_boundary_register: never

interface
   host_memory_access: infer,direct,,readwrite
delay
memory_forward_path_external_delay: 0
memory_return_path_external_delay: 0
RATE_PARAMS ./wlan_top.h:98:10
type
rom

interface
host_memory_access: none,
x ./wlan_top.h:127:9

hardware
return_boundary_register: datapath
forward_boundary_register: datapath

analysis
no_dependence

general
bandwidth: 2
```

```
   interface
   host_memory_access: none
```

Stream Specification Section

The next section describes the constraints on different streams in the application. Application wide settings are first mentioned followed by stream-wise overriding specification. The properties include boundary register and external delay specifications on forward and return paths of the FIFOs, FIFO lengths, specification of any flags which disable any compiler analysis on a stream, specification of coscheduling of some streams, active edges of request and ready signals on the FIFO, etc. If the design is being synthesized using the FPGA product, any BRAM synthesis threshold specification and any EDK integration details (names of request/ready/data signals etc), are also reported. A sample of this section is shown. Here the streams outstream_zout and outstream_yout are coscheduled.

```
Stream Specifications:
all streams
     hardware
          instream_boundary_register: never
          outstream_boundary_register: never
          fifo_synthesis_threshold: 32:1024
     delay
          instream_forward_path_external_delay: 0
          outstream_forward_path_external_delay: 0
          instream_return_path_external_delay: 0
          outstream_return_path_external_delay: 0
instream_MacData ./wlan_top.h:295:21
     interface
          request_level: high
          ready_level: low
          control_mode: passive
          bus_standard: MAX_STD
          bus_type: MAX_BUS
          bus_name: MAX_NAME
          data_name: MaxNumData
          data_name: 1
          request_name: d_req
          ready_name: d_rdy
```

report_summary.txt

This report is updated through the build and simulation flows. It contains a performance summary and QoR information. It has the following sections:

General Information	Provides general information about the implementation
TCAB Properties	Provides information if this implementation is being built as a TCAB

Local Memory Information	Provides information on local memories in the design
Streams Information	Provides information on streams in the design
Relaxed Constraints	Provides information about any constraints which were relaxed to generate the hardware
Cost Estimate	Provides information on estimated cost of the design
RTL Synthesis Cost	Provides information on cost of the design after RTL is synthesized to netlist using Design Compiler
Power Consumption	Provides information on power consumption

General Information

This section provides information about the Synphony C Compiler, C compiler, and simulation tools used, the top level function being synthesized in this design, performance specifications, and the technology library used to create the implementation. For FPGAs, it also displays the device information (part number, package and speed grade), along with the technology library.

Some of these values, like the version of the Synphony C Compiler tool, are properties of the tools and are known from the beginning while others, such as the Synphony C Compiler tool -estimated cost and delivered task latency, become available as hardware is built, and simulated. For example, this section lists information about the C simulators after you run a C simulation, and information about the RTL simulator after you run a Verilog simulation.

A sample of this section is shown:

```
Implementation: fgt_fri_vtest_0802_ppa
Created: Tue Jan 13 11:02:56 AM IST 2009
Synphony C Compiler version: 10.09-0 (Build Time 01/12/09 22:20:59)

C compiler used: /usr/bin/gcc
C++ compiler used: /usr/bin/g++
SystemC compiler used: /usr/bin/g++

Application function: otfn
Task_II: 0 (unconstrained)
Loop 6 II: 1
Loop 7 II: 1
Clock: 280MHz
Techlib: artisan_tsmc65gp-rvt_adv10
Base address: 0x10000
```

The following table lists commands that affect the values of some items in this section. Note that the delivered Task_II, task latency, and throughput are Synphony C Compiler outputs and depend on several factors, including the code and constraints used while building hardware.

Item	Command
Implementation name	new_implementation
Compilers	set_default_params and set_params
Task II	set_params -task_ii

TCAB Properties

If the implementation is synthesized as a TCAB, the report first lists properties for the TCAB. For more information about these properties, see Chapter 10, "Hierarchical Synthesis."

A sample of this section is shown.

```
TCAB Properties:

Maximum physical output production/resource last use time: 1
Architectural pipelinability (*): 1 cycle(s)
Physical pipelinability (**): 1 cycle(s)
Speeded up during scheduling: no
Stall domain: Shared
Clock gated: yes
Stall path insulation: input
Last Cycle pipelining: no

(*) Minimum cycles needed between issue of successive tasks from
the container
(**) Minimum cycles between acceptability of successive tasks by
the TCAB
```

This table describes the user controls over the properties listed here.

Feature	User Controls
Maximum physical output production or resource last use time	This depends on several factors. Refer to Chapter 10, "Hierarchical Synthesis" for more details.

Feature	User Controls
Architectural pipelinability	set_params -architectural_pipelinability <cycles> set_params -output_production_time <cycle> and set_params -resource_last_use_time <cycle>.
Physical pipelinability	This depends on several factors. Refer to Chapter 10, “Hierarchical Synthesis” for more details.
Speeded up during scheduling	Any constraint or code change which can influence the scheduling of operations can affect this field.
Stall domain	This depends on several factors. Refer to Chapter 10, “Hierarchical Synthesis” for more details.
Clock gated	set_params -clock_gating <option>
Stall path insulation	set_params -input_stall_path_insulation always | never | infer *(default)* set_params -output_stall_path_insulation always | never | infer *(default)*

The “Last cycle pipelining” field indicates whether or not the TCAB has pipelining in the last cycle.

Next in the TCAB summary are four tables that summarize architectural and physical latencies. This is useful to understand whether a specified architectural latency has been achieved at the physical (actual) level on a per-variable basis. The following table shows the implementation parameters that can be used to control the properties described in the architectural and physical latency tables.

Properties	Implementation Parameters
Input sample time	set_params -input_sample_time
Output production time	set_params -output_production_time
Resource first use time	set_params -resource_first_use_time
Resource last use time	set_params -resource_last_use_time

Scalar Input Sample Times

The first table specifies the cycles at which each of the inputs into the TCAB are sampled when the TCAB executes. It also lists the C name of the variable and whether it has a boundary register. Here is an example of this table.

```
Scalar Input Sample Times:
+---------------+-------------+-----+-------------------------------+
| Architectural | Physical    | Reg | Item(width, C name)           |
| Cycle #       | Cycle #     |     |                               |
|+--------------+-------------+-----+-------------------------------+
|         0     |        0    |  no |_pico_tcab1_t1_a_in_0(8, t1_a) |
|         0     |        0    |  no |_pico_tcab1_t1_b_in_1(8, t1_b) |
|         0     |        0    |  no |_pico_tcab1_t1_c_in_2(8, t1_c) |
+---------------+-------------+-----+-------------------------------+
Cycle # is the cycle during which the data is needed by the TCAB.
```

Scalar Output Production Times

The next table specifies the cycles in which the TCAB produces the scalar variables. This table lists adjusted and specified values for the architectural latency and an initial value for the physical latency. The specified or initial value is the one you specified or the one the tool attempts to achieve. The adjusted latency is the latency value that was modified from the specified or initial value during the design process. A sample of such a table is shown here.

```
Scalar Output Production Times:
+----------------+----------------+-----+---------------------+
| Architectural  | Physical       | Reg | Item(width, C name) |
| Cycle # [Spec] | Cycle # [Init] |     |                     |
+----------------+----------------+-----+---------------------+
|            1[] |           1[1] |  no | _pico_ret_tcab1(8)  |
+----------------+----------------+-----+---------------------+
Cycle # is the cycle during which the data is available to the container.
```

Resource First Use Time

If the TCAB accesses any arrays or FIFOs, then the TCAB summary lists a table lists with information about when those resources were first sampled.

```
Resource First Use Times:
+---------------------+---------------------+----------+-------------+
| Architectural       | Physical            | Type     | Item        |
| Cycle #             | Cycle #             |          |             |
|     Read |    Write |     Read |    Write |          |             |
+----------+----------+----------+----------+----------+-------------+
|        0 |       -- |        0 |       -- | Memory   | _alias_mem  |
|        0 |       -- |        0 |       -- | Memory   | _ave_vals   |
|       -- |        1 |       -- |        1 | Memory   | _ave_vals   |
+----------+----------+----------+----------+----------+-------------+
Cycle # is the cycle at the beginning of which the TCAB starts using
the resource
```

Resource Last Use Time

If arrays or FIFOs are updated inside the TCAB, the report prints a separate table listing the cycles when those resources were last updated by the TCAB.

```
Resource Last Use Times:
+---------------------+---------------------+----------+-------------+
| Architectural       | Physical            | Type     | Item        |
| Cycle # [Spec]      | Cycle # [Init]      |          |             |
|     Read |    Write |     Read |    Write |          |             |
+----------+----------+----------+----------+----------+-------------+
|      6[] |       -- |     6[6] |       -- | Memory   | _alias_mem  |
|      6[] |       -- |     6[6] |       -- | Memory   | _ave_vals   |
|       -- |      7[] |       -- |     7[7] | Memory   | _ave_vals   |
+----------+----------+----------+----------+----------+-------------+
Cycle # is the cycle at the end of which the TCAB stops using the
resource.
```

Local Memory Information

The report_summary.txt file contains a section that summarizes the different memories in the design. For each memory, it gives the name and memory ID (MemID). If there are no memories, this section is empty.

ASIC:

```
Local memories:

---------------------+-------------+---------------------+-------------+-----------------+-----------------+----------------+
Memory name (ID)       Description       Parameters        Indexed ports    Arbitration        Boundary Reg    External Delay
                                   (words,width,#buf,lat) (#RO,#WO,#RW) (#reqstrs, #ports) (Forward, Return) (Forward, Return)
                                                                          RO    WO    RW
---------------------+-------------+---------------------+-------------+-----------------+-----------------+----------------+
RATE_PARAMS      (10)  (SROM,Int)       (8,28,1,0)            (1,0,0)      --    --    --       (no,no)          (0.0,0.0)
Interleaver[0]    (7)  (RAM,Ext)        (288,1,1,1)           (0,0,1)      --    --    --       (no,no)          (0.0,0.0)
Interleaver[1]    (8)  (RAM,Ext)        (288,1,1,1)           (0,0,1)      --    --    --       (no,no)          (0.0,0.0)
xyz[0][0]         (3)  (RAM,Ext)        (64,16,1,1)           (0,0,2)      --    --  (4,2)     (yes,yes)         (0.0,0.0)
xyz[0][1]         (4)  (RAM,Ext)        (64,16,1,1)           (0,0,2)      --    --  (4,2)     (yes,yes)         (0.0,0.0)
HeaderBitSerialTx (9)  (SRF,Int)        (24,1,1,0)            (0*,1,0)     --    --    --       (no,no)          (0.0,0.0)
sin_cos_fix[0]   (12)  (SROM,Int)       (128,16,1,0)          (1,0,0)      --    --    --       (no,no)          (0.0,0.0)
sin_cos_fix[1]   (13)  (SROM,Int)       (128,16,1,0)          (1,0,0)      --    --    --       (no,no)          (0.0,0.0)
xyz[1][0]         (5)  (RAM,Ext)        (64,16,1,1)           (0,0,2)      --    --  (4,2)     (yes,yes)         (0.0,0.0)
xyz[1][1]         (6)  (RAM,Ext)        (64,16,1,1)           (0,0,2)      --    --  (4,2)     (yes,yes)         (0.0,0.0)
---------------------+-------------+---------------------+-------------+-----------------+-----------------+----------------+
```

FPGA:

```
Local memories :
---------------------+-------------+---------------------+-------------+-----------------+-----------------+----------------+---------+
Memory name (ID)       Description       Parameters        Indexed ports    Arbitration        Boundary Reg    External Delay  Est. #brams
                                   (words,width,#buf,lat) (#RO,#WO,#RW) (#reqstrs, #ports) (Forward, Return) (Forward, Return)
                                                                          RO    WO    RW
---------------------+-------------+---------------------+-------------+-----------------+-----------------+----------------+---------+
RATE_PARAMS      (10)  (SROM,Int)       (8,28,1,0)            (1,0,0)      --    --    --       (no,no)          (0.0,0.0)        -
Interleaver[0]    (7)  (BRAM,Int)       (288,1,1,1)           (0,0,1)      --    --    --       (no,no)          (0.0,0.0)        1
Interleaver[1]    (8)  (BRAM,Int)       (288,1,1,1)           (0,0,1)      --    --    --       (no,no)          (0.0,0.0)        1
xyz[0][0]         (3)  (BRAM,Int)       (64,16,1,1)           (0,0,2)      --    --  (4,2)     (yes,yes)         (0.0,0.0)        1
xyz[0][1]         (4)  (BRAM,Int)       (64,16,1,1)           (0,0,2)      --    --  (4,2)     (yes,yes)         (0.0,0.0)        1
HeaderBitSerialTx (9)  (SRF,Int)        (24,1,1,0)            (0*,0,1)     --    --    --       (no,no)          (0.0,0.0)        -
sin_cos_fix[0]   (12)  (SROM,Int)       (128,16,1,0)          (1,0,0)      --    --    --       (no,no)          (0.0,0.0)        -
sin_cos_fix[1]   (13)  (SROM,Int)       (128,16,1,0)          (1,0,0)      --    --    --       (no,no)          (0.0,0.0)        -
xyz[1][0]         (5)  (BRAM,Int)       (64,16,1,1)           (0,0,2)      --    --  (4,2)     (yes,yes)         (0.0,0.0)        1
xyz[1][1]         (6)  (BRAM,Int)       (64,16,1,1)           (0,0,2)      --    --  (4,2)     (yes,yes)         (0.0,0.0)        1
---------------------+-------------+---------------------+-------------+-----------------+-----------------+----------------+---------+
```

The report also includes a legend to understand the contents of the table.

```
'Memory description' is a tuple (RAM/ROM/BRAM/BROM/SRF/SROM, Int/Ext):
    RAM=>User-supplied RAM, ROM=>User-supplied ROM
    BRAM=>RAM mapped to BRAM, BROM=>ROM mapped to BRAM
    SRF=>Synthesized register file, SROM=>Synthesized ROM
    Int=>Inside current design, Ext=>Outside current design
'RO'=>Read-only, 'WO'=>Write-only, 'RW'=>Read-write.
'*'=>Existence of raw ports
'#bufs' w.r.t ROM => multi-port ROM requirement met by multiple
 single-port ROM instances.
```

Description: This column shows how the memory is physically implemented as a set of two entries, consisting of the type of array implementation (`RAM`, `ROM`, `BRAM`, and so on) followed by the physical location of the memory (`Int` for internal or `Ext` for external to the design boundary).

You can use the following implementation parameters or pragmas to control the implementation of a memory.

`set_params`: `-ram_synthesis_threshold` and `-rom_synthesis_threshold`

pragmas: `user_supplied`, `user_supplied_fpga`, `internal_fast`, `internal_blockram`

You can use the `localize_access` pragma to control the placement of the memory. See "localize_access Pragma" on page 7-33.

Parameters: This column shows the organization of the memory: Number of words (sometimes called entries or rows), width in bits, the number of buffers, and the latency of memories. If the number of buffers is not as you expected, you can control it via the Task_II settings and/or the `#pragma num_buffers`.

Ports: This column lists the exact number and type of ports for each memory. The signals for these ports will match the types specified -- separated or combined read/write, and Artisan/Native. If a memory is multi-buffered and user-supplied, all ports will be duplicated according to the number of buffers, and that number of physical memories must be supplied. The PPA Ports field gives the number of ports for each buffer.

Arbitration: This column specifies the arbitration of memories. For some memories synthesize may find that arbitration is useful to reduce the number of ports on the memory while still meeting your performance specification. The number of simultaneous requests as well as the final porting is shown.

Note:
Arbitration is not applied to memories synthesized as registers.

Boundary Reg: The next two columns indicate whether boundary registers are placed on the forward/return paths of the array. Boundary registers are applied only for user-supplied arrays and block RAMs (for FPGA designs). You can use the following implementation parameters and pragmas to control whether any boundary registers are to be inserted.

`set_params`: `-memory_forward_boundary_register` and `-memory_return_boundary_register`

pragmas: `forward_boundary_register`, `return_boundary_register`

External Delay: The next two columns indicate the delay outside the design assumed on the forward and return paths of the memory. You can use the following implementation parameters and pragmas to control those numbers.

`set_params`: `-memory_forward_path_delay` and `-memory_return_path_delay`

pragmas: `forward_path_external_delay`, `return_path_external_delay`

Est. #brams: For FPGA, if instantiated memories are expected to be mapped to blockrams, the Synphony C Compiler tool estimates the number of blockrams for that memory and lists that in the final column. This gives you more information about the BRAM usage due to different arrays in the design.

Streams Information

The next table gives details of the FIFOs used in the design. Properties like the Name, Location, width, length, and source position of each FIFO is listed.

ASIC

The first column gives the stream name. The second column gives the stream identifier that was used in some of the reports. The third column classifies the streams as being input/output and internal/external. The fourth column gives the stream width. The next column gives the length of the FIFO that is produced. The FIFO length should be set to zero if the stream is an external one.

```
Streams:
+-------------------+--+----------------+------+------+--------------------+
| Name              ID Property          Width Length Source Position
+-------------------+--+----------------+------+------+--------------------+
InterleaveMem        0  internal(passthru)   2     0   wlan_top_level.c:11:1
streamInterleave     2  internal(passthru)   9     0   wlan_top_level.c:14:1
MACTXVEC             1  input(passthru)      36    --  ./wlan_top.h:281:21
MacData              3  input(passthru)      8     --  ./wlan_top.h:279:21
FFTData              4  output(passthru)     16    --  ./wlan_top.h:280:13
```

FPGA

```
Streams:
+---------------+--+----------------+-----+-------+-------------+---------+-------------------+
| Name           ID Property          Width  Length Impl. type    Est #brams Source Position
+---------------+--+----------------+-----+-------+-------------+---------+-------------------+
InterleaveMem     0 internal(passthru) 2    0       internal_fast -         wlan_top_level.c:11:1
streamInterleave  2 internal(passthru) 9    0       internal_fast -         wlan_top_level.c:14:1
MACTXVEC          1 input(passthru)    36   --      internal_fast -         ./wlan_top.h:281:21
MacData           3 input(passthru)    8    --      internal_fast -         ./wlan_top.h:279:21
FFTData           4 output(passthru)   16   --      internal_fast -         ./wlan_top.h:280:13
```

For FPGA designs, the Implementation type and the Estimate number of BRAMs columns follow the stream length. The Implementation type column describes the implementation of the memory, and whether the memory element inside the stream is synthesized as a set of registers (internal_fast). If the implementation of the memory in a stream is a BRAM, then the Esti #r BRAMs column lists the number of BRAMs required for the stream implementation. The final column of the stream table gives the source position of the FIFO or stream declaration.

Loop Statistics and Task Processing

You can use the `report_summary.txt` report for understanding the performance of the PPA. The report shows achieved task iteration interval, which can be used to determine the iteration pipelining within a task. There is another column called “Pipelining Loss Between Tasks”, which can be used to determine the intra-PA task overlap across tasks.

If the pipelining loss is reported as zero, then it means that new tasks are started without any dead cycles and without losing any pipelining. If the pipelining loss is 1, then it means that you lost one cycle between the current task and the start of next task. After running RTL simulation, you should ensure that the pipelining loss, if any, is acceptable before checking the dynamic simulation time performance.

In the `reports_summary.txt` file, the following conditions can be reported as the cause of pipelining loss between tasks:

- PA Task pipelining is turned off – Either task overlap is turned off on the PPA or continuous processing is not enabled.
- PA Scalar Input register optimization (1 cycle) – For the indicated PAs, the acquire-data registers are not present. Scalar data acquire by a PA contains a control register and a data register. A dedicated control staging register is always needed. However, a PA core datapath register can be reused as the data staging register in some cases to save area at the expense of performance.
- Pipelining Loss could be higher if dead iterations are spawned – SCC introduces dead cycles at the stage in which a break for the loop is scheduled. If break is scheduled in stage 1, then SCC introduces an iteration interval “II” dead cycle between tasks. Note that this can only happen if you explicitly coded the application using break statements in the loop. Here is an example of such coding:

  ```
  while (1){
  …
  If (condition) break;
  …
  }
  ```

  SCC tries to schedule the break in stage 0 for continuous processing hardware, but it is not always possible to do this. For example, if a condition to break comes from a blocking stream read in PA running at ii 1, then the break cannot be scheduled in cycle 0.

- PA Output handshakes are registered (1 cycle) – The output handshake is registered. This is generally done for good timing insulation.
- Resource accesses contribute to pipelining loss – This happens because a resource access window was higher than the pipelining. If a resource access window is 3, then the next task for the PA cannot start until the resource is available to the next task (which is three cycles).
- TCAB(s) pipelinability contribute to pipelining loss – This could happen if there is pipelining loss due to importing a higher ii TCAB into a lower ii container.

The following excerpts are taken from `report_summary.txt` for different designs.

Example 1:

```
Loop Statistics:
-------+----------+----------+------------+---------+-----------------
  Loop      II      Schedule     Target      Task     Pipelining Loss
  Name   Achieved    Length    Iterations   Latency    Between Tasks
-------+----------+----------+------------+---------+-----------------
   PA0          4          4      unknown   unknown                 2
   PA1          4          4           64      256*                 2
   PA2          1          4      unknown   unknown                 4
   PA3          8         16            8       72*                 9
   PA4          8         16            8       72*                 9
   PA5          8          8            5       40*                 1
-------+----------+----------+------------+---------+-----------------
* implies that the task latency is 1 more than the number shown due to
registered output handshake.

Task Processing:
   Task Overlap : No
   Continuous_processing : No
   Reasons for PA Pipelining Loss Between Tasks :
     PA Task pipelining is turned off
     PA Scalar Input register optimization (1 cycle): PA0 PA1
     Pipelining Loss could be higher if dead iterations are spawned: PA2

     PA Output handshakes are registered (1 cycle)
```

Example 2:

```
Loop Statistics:
-------+----------+----------+------------+----------+-----------------
  Loop      II      Schedule     Target       Task     Pipelining Loss
  Name   Achieved    Length    Iterations    Latency    Between Tasks
-------+----------+----------+------------+----------+-----------------
   PA0          1          2      unknown   unknown                 0
   PA1          1         12            1        12                 2
   PA2          1          1      unknown   unknown                 0
   PA3          1          2      unknown   unknown                 0
-------+----------+----------+------------+----------+-----------------
```

```
Task Processing:
   Task Overlap : Yes
   Continuous_processing : Yes
   Reasons for PA Pipelining Loss Between Tasks :
     Below resource accesses contribute to pipelining loss
-------+------------+--------+---------------
  Loop     Resource     Type   Per-Iteration
  Name         Name            Access Window
-------+------------+--------+---------------
   PA1   str_datain   STREAM               3
-------+------------+--------+---------------
```

Example 3:

```
Loop Statistics:
-------+----------+----------+------------+---------+-----------------
  Loop      II      Schedule     Target      Task     Pipelining Loss
  Name   Achieved    Length    Iterations   Latency    Between Tasks
-------+----------+----------+------------+---------+-----------------
   PA0          4          4      unknown   unknown                 0
   PA1          1          2      unknown   unknown                 0
   PA2          1          4            1         4                 1
   PA3          1          3           12        14                 0
-------+----------+----------+------------+---------+-----------------

Task Processing:
   Task Overlap : Yes
   Continuous_processing : Yes
   Reasons for PA Pipelining Loss Between Tasks :
     Below TCAB(s) pipelinability contribute to pipelining loss
-------+------------+---------------
  Loop         Tcab   Per-Iteration
  Name         Name   Access Window
-------+------------+---------------
   PA2   mpy_tcab_0               2
-------+------------+---------------
```

The loop statistics part shows a summary of the design at a PA level. Either you set the values for the II and Iteration fields using the `set_loop_params` command or the tool determines the value to achieve the required Task_II. The schedule length denotes the maximum pipelining depth for each of the PAs, whereas the task latency shows the lower bound of the total number of cycles each PA is expected to require, including the draining of the pipeline, but excluding the effects of any stalls or early exits. If the exact number of trip count of the loop is not known at compile time, the target iterations are displayed in the Loop Statistics table. Target iterations can be specified either through the Tcl command `set_loop_params` or by `#pragma num_iterations` *`min,target,max`*.

Note:

For variable trip counts, the number of iterations and latency cannot be determined during compile time. This is shown as “unknown” in the report.

There is a task processing section in the `report_summary.txt` report, which indicates the type of hardware built using the specified constraints:

- Task Overlap – This indicates whether the hardware was built for task overlap. Note that you can build hardware for task overlap using the `set_params -task_overlap always` command.
- Continuous_processing – This indicates whether the hardware was built for continuous processing. Note that you can build hardware for continuous processing using the `set_params -continuous_processing always` command.

Relaxed Constraints

If the Synphony C Compiler tool had to relax any constraints to be able to generate RTL, a summary of the relaxed constraints and the type of failure is presented in this section separately for each loop. There may be several categories of failures in each loop. You should identify the causes which cause the failures and fix them. In the following sample report, loops Loop2 and Loop3 could not achieve the specified frequency of 900MHz and additionally, loop Loop3 could not meet the specified II of 1.

```
Constraints Relaxed:
Name         Constraint     Specified Value   Achieved Value    Failure Type
-----------+--------------+-----------------+-----------------+-----------------+
Loop3*         II            1                  2                 F1
Loop2          Frequency     900 MHz           745 MHz           F1
Loop3          Frequency     900 MHz           530 MHz           F1
-----------+--------------+-----------------+-----------------+-----------------+
* Since the design was produced at a different II from the one specified,
  user specified task_ii, if present, may not be met.
Failure Categories
          F1 -> Feedback path failure
          F2 -> Higher pipelinability TCABs
          F3 -> Single static resource requirement
          F4 -> TCAB ii incompatible with container ii
```

TCAB, Channel, and Imported PPA Information

This section lists all the TCABs, channels, and PPAs imported into this design, showing the entity names and number of instances. A report example is shown.

```
+-------------------+--------------------+----------+
| Name              Entity names         Instances |
+-------------------+--------------------+----------+
E                   e_paw_0                1
C                   c_paw_0                2
D                   d_paw_0                1
B                   b_paw_0                1
A                   a_paw_0                1
```

Cost Estimate

Next is the estimated overall cost in terms of gates for ASIC and in terms of LUTs, FFs, DSPs and BRAMs for FPGA. Note that the cost reported here is only a coarse estimate.

ASIC

```
Synphony C Compiler estimated cost: 845 gates
```

FPGA

```
Synphony C Compiler FPGA estimated cost: (LUTs = 778 FFs = 733 DSPs = 2
 BRAMs = 0 )
```

RTL Synthesis Cost

If RTL synthesis has been run for this implementation (with the Synphony C Compiler rtl_synthesize command or, by hand, in the implementations package subdirectory), versions of the tools used and their results are included. Note that, if multiple tools have run successfully, the results are included for each one of them:

```
RTL synthesis with Synopsys ASIC (version B-2008.09): slack (MET) 0.57
Total cell area: 37027.599818 sq. microns (23142 gates)
```

Some of the simulations can produce task durations. If `verify -rtl` command has run successfully, the task durations are produced here. When the simulation has only one task, its duration is printed; for multiple tasks, the minimum/average/maximum durations are printed. Note that the task duration can be a few times larger than the user-supplied Task_II, up to as many times as the maximum task overlap. Simulations at different levels of detail produce different values of task latency. The most accurate is `verify -rtl` simulation, but it operates in an idealized environment and controller. The following is a sample of the task duration information:

```
Overall task duration (verify -rtl): 338 cycles
Overall task duration (verify -rtl)(min/avg/max):307/387/427 cycles
```

Power Consumption

In ASIC, when the power estimate tools such as PowerTheater and Spyglass-Power are run using the Synphony C Compiler estimate_power command or run manually in the implementation package subdirectory, a report will be generated. This report provides information of the tools and summary results. When power estimation is run successfully for the offline and SV simulations, the summary results will be included for each one of them, for example:

- `Estimated power (offline simulation) with PowerTheater (version R2006.2.1):internal power: 4.77mW (static=1.1mW, dynamic=3.67mW)`
- `Estimated power (offline simulation) with spyglass-power: Total power: 216uW (Leakage=22.7uW, Internal=187uW, Switching=6.12uW)`

User Code Reports

The following reports give information related to your code:

- report_original_loop_to_line_number.txt
- report_perfectized_loop_info.txt
- report_rate_match.txt
- PPA Loop Graph, Sequentialization Loop Graph

report_original_loop_to_line_number.txt

This report shows the location of your loop nests in the original source code. In a few cases, different sub loops within an outer loop contribute to the hardware of the outer for loop after a loop jamming transformation. In those cases, information about any loop jamming operations is also mentioned in this report. Only the loops that appear after any unrolling is done are referenced in this report. These loops are an exact match to the loops that may be referenced with the `set_loop_params` command in the user interface. Consider a sample report shown.

```
Loop at app1.c:8:5, with (0 0 0) iterations, jammed to form loop nest #0
Loop at app2.c:15:5, with (0 0 0) iterations, jammed to form loop nest #0
Loop nest #0: app.c:7: Outermost index variable: j
Loop nest #1: app.c:23: Outermost index variable: i
```

To force the II of the loop at app.c:23 (loop nest #1) to 3, you would specify:

```
set_loop_params 1 -ii 3
```

In the example, two such sub-loops are perfectized to form loop zero (0). These two loops do not have any num_iterations set for them and so the min iterations, max iterations and target iterations are reported as (0,0,0).

Also, note the different file names reported with each loop: the C code for this particular application was present in three C files, pulled together by the C standard pre-processor through #include directives.

report_perfectized_loop_info.txt

In the perfectization and rate matching process (in the schedule phase), additional loops may be introduced to deal with mapping of out-of-loop in your code to the PPA. For more information on these processes, see “Perfectization” on page 5-16 and “Rate Matching” on

page 5-25. Following these processes the tool reports an exact map of code to PAs in the hardware:

```
Loop nest #0 (II = 1 ): ppa.c:378: Created by perfectization
Loop nest #1 (II = 8 ): ppa.c:87: Was loop 0 before perfectization
Loop nest #2 (II = 1 ): ppa.c:76: Created by perfectization
Loop nest #3 (II = 8 ): ppa.c:141: Was loop 1 before perfectization
Loop nest #4 (II = 1 ): ppa.c:76: Created by perfectization
Loop nest #5 (II = 12 ): ppa.c:240: Was loop 2 before perfectization
```

This report distinguishes your loops in the original code from loops inserted by perfectization. At this point the numbering of the loops and mapping to PAs is finalized. For example, loop nest 3 in this report will become PA 3 in the RTL.

The files and line numbers listed are the locations of the first line of code that is mapped to this loop nest. You may also see the <implementation>/info/<appname>_perfected.c file to examine the exact code that is mapped to these loops.

Occasionally you may find opportunities for quality of results improvement by carefully examining this report as well as the perfectized code. Look at any loops inserted by perfectization and see if they match your intent in generating the hardware.

report_rate_match.txt

Rate matching produces a report intended to assist in debugging any issues with meeting Task_II. Typically, it is not necessary to look at this report if schedule is passing, as the relevant information can be gathered from the "report_perfectized_loop_info.txt" on page 13-19.

During perfectization, the Synphony C Compiler tool comes up with some II choices for loops with unspecified II. The computed II values are based on the specified Task II, code structure and any other performance constraints. This report starts with a list of those perfectization choices. Each perfectization choice may succeed or fail in meeting Task_II. Each one notes the PAs (loops) for the post-perfectization code, as well as their loop iterations and any of IIs that you specify:

```
**********************************************
** Starting rate matching with new Candidate **
**********************************************
   Candidate # 0
   General constraints
   -------------------
   Number of buffers for sin_cos_fix = 1;
   L0_iterations = 320; // Was originally 320 // Loop nest #0: app.c:48
   L1_iterations = 320; // Was originally 320 // Loop nest #1: app.c:64
   L2_iterations = 1; // Created by Perfectization
   L0_ii = unspecified;      (limit 1 - based on loop resource estimate)
   L1_ii = unspecified;      (limit 1 - based on loop resource estimate)
   L2_ii = unspecified;      (limit 1 - based on loop resource estimate)
```

```
The per-iteration stream bandwidth is listed next.
Per-iteration bandwlh of streams
---------------------------------
   L0 : output_SerStrm = 1
   L0 : input_ppa_in = 0.03125
   L1 : input_SerStrm = 1
   L1 : output_DiffData = 0.03125
   L2 : input_DiffData = 10
   L2 : output_ppa_out = 10
   L2 : output_ppa_out2 = 1
```

Stream constraints are listed next. When streams connect between PAs, the rate matching will attempt to balance the bandwidths. This may introduce artificial constraints if the bandwidths cannot be analyzed, so you should check the bandwidths as well as the constraints in this list to make sure that Task_II is not artificially limited.

```
Constraints due to streams
1 * L0_ii = 1 * L1_ii;
 //BW(input_SerStrm)*L0_ii=BW(output_SerStrm)*L1_ii
10 * L1_ii = 0.03125 * L2_ii;
 //BW(input_DiffData)*L1_ii=BW(output_DiffData)*L2_ii
```

Note:

The Synphony C Compiler tool may relax the stream constraints if the stalling introduced does not prevent Task_II from being met.

TCAB constraints are listed next. When a TCAB is instantiated inside a loop in a higher level design, the II of the PA or the TCAB corresponding to the outer-level block is constrained by the II achieved for the lower-level TCAB. An example snippet from the report is shown:

```
Constraints due to the TCABs
L2_ii >= 12; // Minimum cycles needed between issue of successive calls
to 'ConditionData'
```

Individual Loop Constraints are listed next.

```
Individual loop constraints
320 * L0_ii <= task_II; // L0_iterations * L0_ii <= task_II;
320 * L1_ii <= task_II; // L1_iterations * L1_ii <= task_II;
1 * L2_ii <= task_II; // L2_iterations * L2_ii <= task_II;
```

Finally, the solution for this rate matching choice is listed. This includes your Task_II, the final IIs, and the number of buffers used for any multi-buffered memories. A note shows whether this perfectization choice succeeded or failed.

```
Solution
task_II = 320
L0_ii = 1
L1_ii = 1 g
L2_ii = 12
```

Rate matching succeeded for current candidate.

After the reporting for all perfectization choices or candidates is done, the report contains a section that maps a perfectization choice to the candidate number as listed in this report. This mapping may be used to identify in this report the information corresponding to a given perfectization choice. For example, this mapping may be used to identify the rate-matching candidate corresponding to the perfectization choice being tried.

```
*********************************************************************
Mapping between the Sorted list of Perfectization choices and the
candidate list
*********************************************************************
Perfectization Choice # 0 -> Candidate # 0
Perfectization Choice # 1 -> Candidate # 2
Perfectization Choice # 2 -> Candidate # 1
Perfectization Choice # 3 -> Candidate # 3
**********************************************
** Schedule status of Perfectization choices **
**********************************************
Perfectization Choice # 0 -> passed
```

PPA Loop Graph, Sequentialization Loop Graph

Following a successful rate matching, the Synphony C Compiler tool produces both the PPA and the Sequentialization loop graphs. These graphs can be viewed in the GUI directly using the Analyze menu. Additional reports in .PDF and .PostScript formats are also generated in the <implementation>/Reports directory. The PPA graph shows the PAs in the design as well as their connectivity through memories, streams and scalars. The Sequentialization graph contains information about how loops execute in parallel or sequentially.

To produce the loop graphs, you must install the Graphviz package as detailed in the installation notes, and the dot command must appear in your PATH variable.

PPA Graph

PAs (loops) are shown as rectangular boxes. They are labeled by loop number (these numbers match the report_perfectized_loop_info.txt. New loops which are created during perfectization stage are shaded in gray color. All other loops which are present in your code are represented by unshaded boxes.

Memories are shown as blue ovals. The arrows connecting the PAs to these memories show the connectivity as well as the read or write relationships.

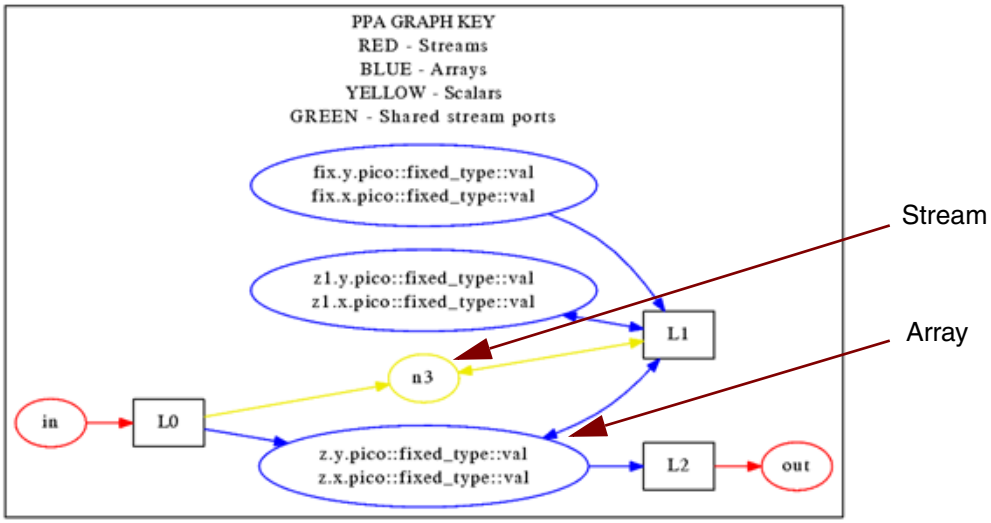
Streams are shown as red ovals. The arrows connecting them to the loops show the direction of data flow. If stream ports are shared between PAs, the ovals and arrows associated with those streams will show up as green rather than red.

Scalars are shown as yellow ovals. The arrows connecting them to the loops show the direction of data flow.

PPA Graph can be used to understand how different loops access resources like scalar variables, array and stream resources. This can be used to understand why some PAs execute either sequentially or in parallel.

- Any set of loops which only read from either a scalar variable or array can execute simultaneously within a task.
- Consider two loops Loop <j> and Loop <j> (where i and j are loop numbers and i<j) passing information from one to the other using a scalar variable. If Loop <j> accesses a scalar or array after Loop <i> updates the scalar, then Loop <j> would execute sequentially after Loop <i> the within a task.
- If two loops read the same stream, the loops cannot work simultaneously within a task and across tasks.
- If the resource is multi-buffered, the individual loops which access that resource may execute simultaneously on different tasks. All scalars are always multi-buffered and FIFO resources are not. Arrays can be either specified to be multi-buffered or be inferred to be so based on performance requirements
- Scalar variables which are only inputs into the PPA are not shown in the PPA graph.

The following figure shows an example of the PPA graph.



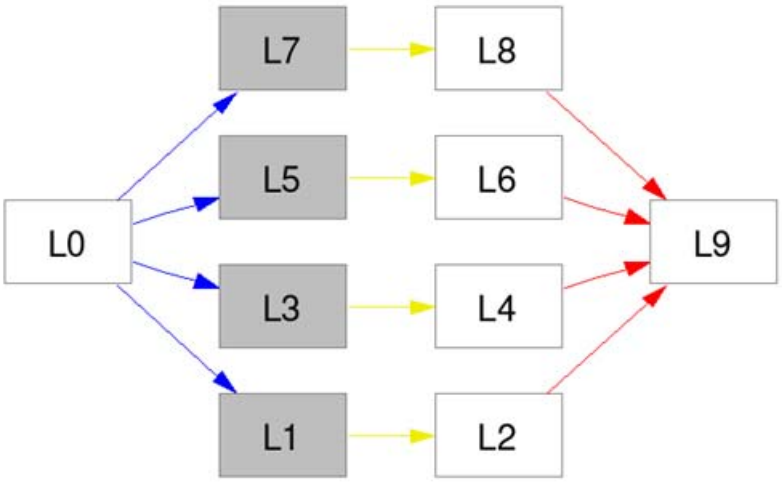
Sequentialization Graph

Sequentialization Graph shows the flow dependence which governs whether loops execute in sequence or in parallel within a task. This graph does not show the complete details about the resources which cause sequentialization. New loops which are created during perfectization stage are shaded in gray color. All other loops which are present in your code are represented by unshaded boxes.

A red arrow between two loops indicates that two loops can execute simultaneously if no other reason for executing the loops sequentially. Sequential execution of two loop can be due to a scalar dependence (yellow arrow), array dependence (blue arrow) or a shared stream (green arrow).

All the flow relations are listed only once, for example, even if there are multiple scalars which cause two loops to execute in sequence, there is only one (yellow) arrow in the graph.

There could be indirect flow dependences between two loops. The following figure shows an example of the Sequentialization graph.



Interpretation of the Sequentialization graph

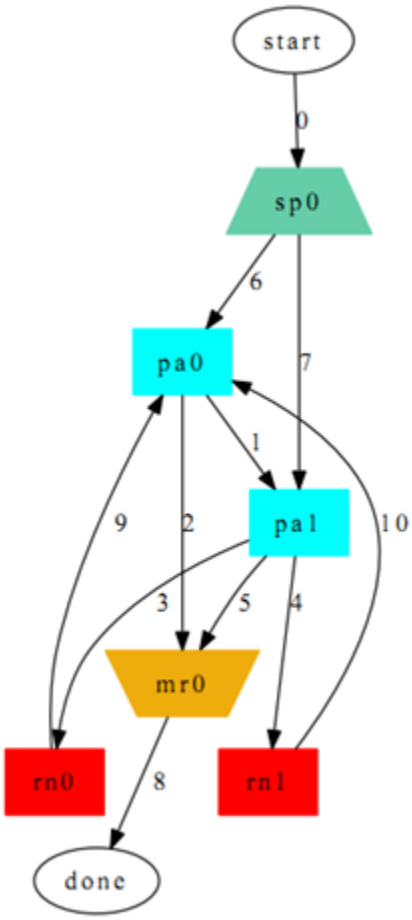
- The graph shows that there are four perfectized loops (L1, L3, L5 and L7) and six loops (L0, L2, L4, L6, L8 and L9) from your code.
- The perfectized loops (L1, L3, L5 and L7) execute only after loop L0 completes since each of the perfectized loops accesses a memory (blue arrow) updated by loop L0.

- The loops L2, L4, L6 and L8 execute only after loops L1, L3, L5 and L7 respectively due to the former set of loops using scalar variables (yellow arrow) after the latter set of loops update those variables.
- Loop L9 executes alongside loops L2, L4, L6 and L8 since loop L9 communicates with those loops using streams.
- Loops which are not connected with any data dependence edges (loops L0 and L9 for example) can execute in parallel.
- At a top level, you can infer that L0 executes first and then, loops L1, L3, L5 and L7 start almost simultaneously. At the completion of each of those loops, the sequentially dependent loop among L2, L4, L6 and L8 would start execution. In effect, there are four parallel streams of execution in the design which begin at loop L0 and converge at loop L9.

PPA FIFO Report

Synphony C Compiler generates two report files for the PPA that are useful for understanding task processing of your design: `ppa_all_fifos.pdf` and `ppa_sync_fifos.pdf`. The `ppa_all_fifos.pdf` file shows all the FIFOs in the design, whereas the `ppa_sync_fifos.pdf` file shows the graph without any user-specified FIFOs in the design.

The following is an example of a `ppa_all_fifos.pdf` report.



Blue Edge	User Fifo
Black Edge	Sync Fifo

pa	Loop
sp	Split
mr	Merge
rn	Resource Pool

FIFO Names	
0	fifo_7_host_sp0_len
1	fifo_0_pa0_pa1_len
	fifo_1_pa0_pa1_len
2	fifo_2_pa0_mr0_len
3	fifo_4_pa1_rn0_len
4	fifo_5_pa1_rn1_len
5	fifo_5_pa1_mr0_len
6	fifo_l1_sp0_pa0_len
7	fifo_l2_sp0_pa1_len
8	fifo_8_mr0_host_len
9	fifo_9_rn0_pa0_len
10	fifo_10_rn1_pa0_len
rn0	A [numbufs = 2]
rn1	B [numbufs = 2]

For information about the role of FIFOs in the hardware, see “Task Processing in PPAs” on page 5-4.

Timing Analysis Reports

The Synphony C Compiler tool provides reports and GUI analysis views to help you understand the different critical feedback paths that restrict any application from meeting your constraints like II and target clock frequencies. Usually a mixture of the analysis view and reports guide you towards the issues causing performance bottlenecks. This section describes the different reports and GUI analysis views and their corresponding reports

which help you to understand the scheduling of operations, the feedback paths in the design and combinational paths are listed in the following table.

Item	GUI	Text Reports
Generated schedules	Operation Schedule Viewer	report_psuedo_code.txt
Feedback paths	Feedback Path Viewer and Operation Schedule Viewer	report_feedback_path.txt
Timing paths	Operation Schedule Viewer	report_timing_path.txt & report_interface_timing.txt
False paths	Feedback path viewer	report_false_paths.txt

The following items describe the contents of the text reports and the GUI analysis views mentioned in the table.

Internal Timing Report

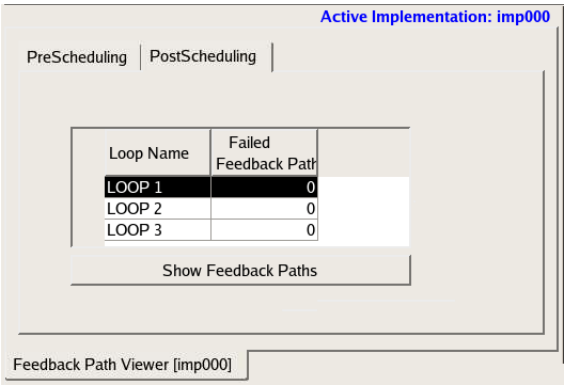
The following reports and views provide information to analyze timings:

- report_feedback_path.txt
- report_pseudo_code.txt
- report_false_paths.txt

In addition to the reports, there are two views you can use to analyze your results.

- Feedback Path Viewer

  This view lets you view and analyze critical feedback paths on a per-loop basis. The following figure shows the tabs to view the feedback paths.



- Operation Schedule Viewer

  This view shows the order of scheduled operations and their dependencies in different loops. The operations are displayed in the scheduled order for all the loops which have completed the synthesize phase successfully. For the loops which failed in synthesize step, the operation are listed in the program order.

report_feedback_path.txt

The report is generated during the synthesize phase and lists any critical feedback paths in the design. You can look at this report to identify any potential problem paths early in the flow. If this report shows any critical feedback paths with negative slack, there is a very good likelihood that the final synthesize step would fail. In such cases, the synthesize step fails if all the constraints are specified by you. However, if there are any constraints that are a result of rate matching, synthesize step may relax those constraints and proceed to build hardware. Absence of any critical feedback paths in this report however does not guarantee that the Synphony C Compiler hardware synthesis will be successful. In case the synthesize step fails, with a message indicating presence of critical feedback paths, then you can look at this report first to see if there are any obvious paths (that show negative slack) which may need to be looked at first.

The Synphony C Compiler tool schedules the operations that represent the logic in a input C/C++ application. However, scheduling may not be successful in a few scenarios where feedback paths across iterations do not meet timing. In such cases, the Synphony C Compiler tool would continue to synthesize the design by relaxing your constraints (where possible). Though RTL is produced after relaxing the constraints, this report lists the most critical feedback cycles (with the original constraints) in the input program. You can modify

your code suitably to eliminate these critical cycles. The following failing scenarios are possible:

II failure	This usually happens when there is a resource conflict or architecture limitation and no amount of decreasing the frequency can get you past the scheduling failure. You can only understand why this failure happened and try to re-architect the design to get past this issue.
Timing failure	This will happen in cases where the feedback paths can be re-timed after decreasing the frequency. A symptom of this failure would be to have a successful schedule after lowering the frequency. The synthesize log file (<implementation>/Logs/synthesize_*.txt) contains error messages indicating that report_feedback_path.txt contains information about the failing cycles. To fix this issue, you may have to re-code the logic in the application but the extent of changes may an architecture change in the design.

It is possible that different PAs in the design can have different failures. You should look at the corresponding section in each report for each loop where some constraints are relaxed.

The report_feedback_path.txt is also produced after the scheduling is successful. In this case, the cycles in the report still represent the most critical ones that were encountered in scheduling and are documented to enable you to get a deeper insight into the critical portions of the design.

The Synphony C Compiler tool produces report_feedback_path.txt when the Tcl command `report_implementation -feedback_path` is given.

Irrespective of the scenarios listed, the most important portions of this report are common and we describe these aspects here. The report starts with some acronyms for the various delay types that are handled. Each acronym has a brief description of the delay that it represents. It is important to understand these delays and are described in the following table.

Delay	Description
LIF (Logic + interconnect + fanout delay)	This delay is applicable to a combinational functional unit and to the first stage of a sequential functional unit. This delay adds the intrinsic logic delay of the unit or the stage, the multiplexer delay at the input of the functional unit and the delay due to the fanout from the output. For a sequential unit, since this delay is applicable only to the first stage, the fanout delay component is not present.
LSD (Pipelined unit last stage + fanout delay)	This delay is applicable only to sequential functional units. Also, it is applicable only to the last stage of the sequential unit. It includes both the intrinsic logic delay of the last stage as well as the fanout delay at the output of the last stage.
DSC (Delay due to scheduling constraints)	There are a few constraints imposed on the scheduler. Scheduling constraints: A scheduling constraint constrains the scheduler to schedule a pair of operations within a difference of a certain number of cycles. A scheduling constraint is imposed on the scheduler both for sharing resources as well as for scheduling multiple stream operations in the same cycle. Output production time constraint: In a TCAB, you can specify an architectural latency constraint on an output with respect to the start of execution of the TCAB. This constraint imposes a scheduling constraint, which translates to a feedback cycle type of constraint.
DFMS	It refers to Delay due to data flow merge selector. This is the delay introduced due to a multiplexer or demultiplexer for selecting between data flow paths which converge at a point.
ICD	It refers to Interconnect Delay.
SLACK (Slack within a clock period)	This represents the slack available within a clock boundaries (i.e. in the combinational part of the schedule). This could arise because this period is not sufficient to accommodate the next operation in the datapath.
SETUP (Setup time)	As the name indicates, this is the register setup time.
CKQ (Clock->q delay)	As the name indicates this is the delay from the active clock edge to the output of the register.

Delay	Description
FIMD (Forward internal memory path delay)	This delay appears on the forward path from the memory load/store unit to the memory. If there is a boundary register present on the artifact boundary then this delay measures the delay from the output of the memory unit to this boundary register. If there is no boundary register present, this delay represents the delay component from the output of the memory unit to the artifact boundary.
RIMD (Return internal memory path delay)	This delay appears on the return path from the memory to the memory load/store unit. If there is a boundary register present on the artifact boundary then this delay measures the delay from the boundary register to the memory unit. If there is no boundary register present, this delay represents the delay component from the boundary to the memory unit.
FED (Forward path external delay)	This is the delay from the boundary of the artifact to the memory. If there is a boundary register, then this delay is a whole cycle.
RED (Return path external delay)	This is the delay from the memory to the boundary of the artifact. If there is a boundary register, then this delay is a whole cycle.
PIPE (Pipeline delay)	This is the delay of an intermediate stage in a multi-cycle sequential unit. For example, consider a three-cycle multiplier, which has three pipeline stages. In this case, the delay of the second stage in the pipeline is represented by this delay.
EXINP (Raw scalar external input delay)	This delay is applicable only to a TCAB. This is the external delay constraint that you specified on an input port of the TCAB.
EXOPT (Raw scalar external output delay)	This delay is applicable only to a TCAB. This is the external delay constraint that you specified on an output port of the TCAB.
MDC / SDC (Delay enforced due to memory or stream dependence constraint)	If two memory (or stream) operations are sequentialized, there can be some unused part of the clock cycle in which the first operation occurs. This unused portion of the clock cycle is caused due to the scheduling of the next operation only on the next cycle. This unused portion of the clock is represented with the MDC or SDC flag.

Delay	Description
CST (Context Sensitive Timing)	Some operations are timed differently in the Synphony C Compiler tool based on surrounding context. The surrounding context consists of the operations that produce its inputs and consume its outputs. The context sensitive timing is typically lower than the timing of a unit without the context being applied. This flag is used to indicate that this operation was timed based on surrounding context.

The later portions of this report is a list of feedback cycles with each cycle being demarcated by a (BEGIN_CYCLE, END_CYCLE) pair. Each cycle also has the loop number that the cycle is associated with as well as the II at which the loop was scheduled/attempted to be scheduled. Each row in a feedback path represents an operation. For each row, there are 7 columns and the purpose of each column is described:

Opid	Operation identifier used internally by the Synphony C Compiler tool to refer to an operation.
Opwidth (out/in)	This denotes the width of the output/input operand for the current operation. If the operation is involved in data flow dependence, the width gives the width of the operand causing the dependence. For other dependences, this field represents the maximum width across all output/input operands.
Opcode	Denotes the functionality being provided by a particular operation. For example, the opcode OR_W represents an OR operation. f” lists most of the opcodes used in the compiler.
Sched time	This denotes the actual cycle in the schedule in which a particular operation is scheduled. If the report was produced before scheduling (after the check for II is satisfied), then this column is “not applicable” for every row. If the report has been produced after successful scheduling, then this column is populated for every operation or row. If the report was produced after a failed scheduling attempt, then this column may be populated or marked “not applicable” depending on whether the operation corresponding to that row was placed in the scheduled at the point of failure or not.
Delay (cycles)	This denotes the actual delay value as a fraction of the cycle.
Delay type	This column gives the acronym, which describes the type of delay. A detailed description of these acronyms is present in the earlier paragraphs.

Source position	This gives the source position in the original C application containing the C operation, which is implemented by the current compiler operation. The source position is represented by a set of triplets (where each of it is represented as C- file_name: line number: column number). This column also contains a snippet of C code, from the source application with a pointer below it that points to the exact source position. In some cases, the message no SourceInfo is printed. This means that the tool was unable to map this operation to the C source code.

The other terms used in this report are described:

CLOCK	This represents a clock boundary in the path represented by the cycle.
Iteration Distance	As mentioned earlier each row in a feedback path represents an operation. An entire cycle may be viewed as representing dependencies from one operation to another, with the dependence from the last operation leading back to the first operation, thereby completing a cycle. Thus, as we traverse a cycle from one operation to the next, the reader must understand that a dependency between the two operations is being traversed. Since, a cycle represents dependencies that are observed while scheduling a loop, a particular dependence may also cross loop iteration. In other words, going from operation A (row A) to operation B (row B) in a cycle, the dependence traversed may extend from an instance of operation A in one iteration to an instance of operation B in an entirely different iteration. The Iteration Distance entry signifies the number of iterations crossed for a particular dependence. This value is printed only if the distance is greater than one. In some sense, the iteration distance determines the amount of time available for scheduling the cycle and, thereby, is one of the factors that determine the criticality of a cycle. The higher the value of iteration distance, more is the time available for scheduling the cycle, and therefore, the less the criticality of the cycle.
Total Cycle Delay excluding SLACK	This represents the sum of the delays of the operations across the different clock period segments without including the SLACK component. Thus, this represents the total delays contributed by the elements in the cycle excluding the slack in each clock period.
Total Cycle Delay	This represents the sum of all the delays including SLACK in each clock period.

Available Time	As the name signifies, this represents the total available time for scheduling the cycle. As mentioned earlier, one of the factors affecting “Available time” is “Iteration Distance”. The other is II of the loop. Simply put, this entry is just II * S where S is the sum of all the Iteration Distance entries in the cycle.
Slack	This is just the Total Cycle Delay subtracted from the Available Time. If the result is negative, then this implies that sufficient time is not available for scheduling the cycle with the given delays. Therefore, this is a failing cycle. However, there is the following important note to be kept in mind. A positive value of slack does not necessarily imply that the cycle is not critical. When the report is produced after a failed attempt at scheduling, the cycles in the schedule where the operations are placed are not determined for all the operations. In such a situation, the tool can only estimate the likely cycles that the unscheduled operations could be placed in. Since this is an estimate, it can very well happen that the tool anticipates that a particular operation can be scheduled in a cycle earlier than would really be possible and therefore ends up under-estimating the overall cycle delay and, thereby, ends up overestimating the overall slack. Thus, in the case where scheduling has failed, a cycle with positive value of slack has also to be considered as critical cycle. The cycles are sorted in the increasing order of slack and therefore the cycles that come at the head of the corresponding slacks are negative or positive.
Speculated operation	This entry in a row indicates that the operation corresponding to that row has been speculated by the compiler. Typically, applications would contain operations that are control dependent on condition evaluations. Statements that are conditionally executed in an if-statement are such operations. In such a situation, the execution of the corresponding operation is control dependent on the condition evaluation. Speculation is transformation, which breaks this control dependence of an operation on its corresponding condition evaluation. This enables the operation to be executed in parallel with the condition evaluation with the results of the operation evaluation being actually committed under the condition. Such a transformation helps achieve timing more easily. This entry when present indicates that the corresponding operation has been speculated and provides more information to you on the nature of the dependence.

As mentioned earlier, a cycle in the report is really a record of the dependences from one operation to other spanning multiple iterations. The report also classifies the dependences into the following types:

Memory Dependence Edge	This is a sequentializing dependence between memory accesses to the same memory that can conflict. This is necessary to ensure the correct functionality of the application.
Stream Dependence Edge	This is a sequentializing dependence between stream accesses to the same stream.
Co-scheduling Edge	This dependence represents a constraint specified to the scheduler to schedule the operations involved a certain number of cycles apart. An example is the co-scheduling of stream accesses because you need the streams to be accessed together in the same cycle. A second example is the scheduling of operations such as multiply operations in the same cycle to share multiply functional units across mutually disjoint execution paths.
Anti Dependence Edge	There are several types of dataflow dependencies. One of them is anti dependence. This is a write-after-read dependence, which enforces that a value should be overwritten only after it has been consumed.
Output Dependence Edge	This is another type of dataflow dependence, which enforces an ordering of output producing operations. This is necessary to ensure the output order as per your application program.

See “Feedback Path Viewer” on page 2-51 for a detailed description of the view. The following figure shows an example of the feedback text report:

```
*********************************
* Feedback path: 5
* PA: jpeg_encode_scan_loop4
* II: 1
*********************************

 Opid    Opwidth    Opcode          Delay     Delay        Source Position
         (out/in)                   (ns)      Type
-------------------------------------------------------------------------------
 5421    (5/-)      LS_WIDE_0_4     0.131     LSD          jpeg_encode.c:594:40

 5565    (6/5)      ADD_W           0.203     LIF          jpeg_encode.c:673:34

 5571    (3/6)      SHRA3_W         0.000     LIF          jpeg_encode.c:674:21

 5678    (3/3)      MOVE            0.000     LIF          jpeg_encode.c:697:25

 5739    (3/3)      SUB_W           0.171     LIF          jpeg_encode.c:552:25


 47480   (1/1)      MOVE            0.000     LIF          jpeg_encode.c:699:5


 16307   (1/1)      CMPP_W_NEQ      0.001     LIF          jpeg_encode.c:699:5

 6940    (1/1)      MOVE            0.000     LIF          no SourceInfo

 16122   (1/1)      CMPP_W_EQ       0.001     LIF          jpeg_encode.c:591:5

 16162   (1/2)      CMPP_W_EQ       0.040     LIF          jpeg_encode.c:592:10

                                    0.099     SLACK
                                    0.171     SETUP
                                       [--------CLOCK--------]
                                    0.194     CKQ

                                    0.737     SLACK
 5421    (5/-)      LS_WIDE_0_4     0.148     LIF          jpeg_encode.c:594:40

                                       [--------CLOCK--------]
-------------------------------------------------------------------------------
                    Total Cycle Delay excluding SLACK =    1.637  ns (  1.309
                                 Total Cycle Delay =    1.735  ns (  1.388
                                    Available Time =    1.250  ns (  1.000
                                             Slack =   -0.485  ns ( -0.388
                                            Status =    FAILED
```

Any source positions indicated in the feedback path are listed in the bottom of the report under the header Source Positions, as shown in the following example:

```
-----------------------------------------------SOURCE POSITIONS---------
 Opid    Opcode                  Source Position
-------------------------------------------------------------------------

 5565    ADD_W                   jpeg_encode.c:673:34       unsigned char w

                                 jpeg_encode.c:735:12       jpeg_pack();
                                                                     ^

 5571    SHRA3_W                 jpeg_encode.c:674:21       nb_byte = width

                                 jpeg_encode.c:735:12       jpeg_pack();
                                                                     ^
```

report_pseudo_code.txt

The pseudo code report provides information on the schedule produced by Synphony C Compiler in the form of a pseudo code. This report is generated at the end of synthesize step. The report can be displayed using the command line option:

report_implementation -pseudo_code

This report contains the schedule of operations (the cycle in which each operation starts execution) in each loop and the corresponding hardware instance name for each operation. Information is printed in the scheduled order of operations for all loops which pass synthesize phase. For the first loop which fails in the synthesize step due to nonfeasibility of certain constraints, the tool prints the opcode information in the program order.

The report has two top-level components:

- Global Variables Section
- Loop Information Section

Global Variables Section

All the variables that are visible to the host (global variables in the PPA) are presented as a table, only once, at the top of the report. The global variables in the application are listed along with the bit width, signed/unsigned qualifier and the source position mappings for each global variable.

The source position mapping is in the format

file_name : *line_number* : *column_number*

Example

```
Name (RTL Name)                              Width and Sign       Source Position
-------------------------------------------------------------------------------------
pico_ret_tcab1_1 (--)                         7 signed             NA
pico_ua_ppa_nest1_nest1_i2 (rr_var_i)         5 signed             app.c:28:7
x (rr_var_x)                                  7 signed             app.c:6:5
y (--)                                        7 signed             app.c:6:7
pico_ret_tcab1_0 (--)                         7 signed             NA
pico_ua_ppa_nest2_nest2_i2 (rr_var_i)         5 signed             app.c:54:7
b (sr_var_b)                                  7 signed             app.c:5:7
c (sr_var_c)                                  7 signed             app.c:5:9
a (sr_var_a)                                  7 signed             app.c:5:5
d (sr_var_d)                                  7 signed             app.c:5:11
pico_ret_tcab2_1 (--)                         7 signed             NA
z (rr_var_z)                                  7 signed             app.c:6:9
pico_ret_tcab2_0 (--)                         7 signed             NA
```

Loop Information Section

For each loop, all the variables that are local to the loop are listed. Even the temporary variables which are created during different code transformations are printed here.

The local scope scalar variables section lists the same details as the global variables (name, bitwidth, signed or unsigned qualifier and source positions) for every variable local to the loop.

Example

```
Name (RTL Name)                      Width and Sign       Source Position
-----------------------------------------------------------------------
LC (--)                               4 unsigned                NA
ESC (--)                              1 unsigned                NA
vr2001174 (rr_func_brf_inst0)         1 unsigned                NA
```

If any variables are initialized before the execution of a loop, they are in the “Constant Initializations” section as shown. Note that this initialization could be due to loop transformations that are part of the tool flow.

```
---------------------------------------------------------------------
LOOP 0: Constant initializations
---------------------------------------------------------------------
RTL Name[omega] = value
-----------------------
```

Scalar values defined in an iteration can be used in a later iteration, which implies scalar data could flow across iterations. The omega value (number inside square braces) can be used to understand the number of iterations between the current access and its last update. It can be present for write and read operations. In an operation which updates this variable, the omega value is 0, implying that this update is the current iteration. In an operation where

this variable is read, the omega value represents the number of iterations spanned from the write to that particular read.

Details of the Loop Body

The body of the loop is listed in the following format:

“Schedule Time” “Pseudo Code” --> FU (OP #) [Source Positions]

Example

```
ST   PSEUDO-CODE --------------------------------> FU-NAME/OP-CODE (OP #) [Source Position]
-------------------------------------------------------------------------------------------
0   if (rr_func_moveii_inst1[1] != 0) { ----------> cmpp_neq_inst0  (op 1885)        [NA]
0    if (sr_var_a > 0) { -------------------------> cmpp_ineq_inst0 (op 1890)        [NA]
0     if (rr_var_i[1] < sr_var_b) { --------------> cmpp_ineq_inst1 (op 1895) [app.c:24:11]
0      rr_var_c[0] = scc_stream_input_X() ;------> instream_X_ldstr_sx_noreset_inst0
                                                                    (op 555)  [app.c:24:36]
     }
     else {
0      pico_lv_pico_tmp_2_2 = scc_stream_input_X() ; --> instream_X_ldstr_sx_noreset_inst0
                                                                    (op 558)  [app.c:25:32]
     }
   }
  }
  else {
0    rr_var_c[0] = rr_var_c[1] ; -----------------> MOVE            (op 3697)        [NA]
}
```

In the example, consider the line marked with (op 3697). In this line, you can see the write to the variable c represented by rr_var_c[0] and a read of the same variable represented by rr_var_c[1]. Note that write operation has omega of 0 and the read operation has a value of 1. This implies that this variable is updated in the current iteration and the value read is from a previous iteration. The difference between the indexes for this rotating register indicates the number of iterations in the past from which the stored value was referenced.

For operations, which are listed repeatedly, a “*” character is printed as the schedule time. This is to indicate that this operation has been printed earlier and the result of that operation is going to be reused here. Note that no new hardware is created for this repeated operation. This is shown in the example.

```
0  if (sq2_func_moveii_inst86[1] != 0) { --> cmpp_neq_inst0 (op 12070) [NA]
0   if (vr2009784 == 1) { --> cmpp_eq_inst0 (op 54680) [wlan_top_level.c:306:23]
0       vr2000462 = sq4_var_service_802_11a_tx[1]
        ; --> MOVE (op 462) [wlan_top_level.c: 308:42]
0       sq5_var_scrambleoutput[0] = vr2000462[0]
        ; --> MOVE (op 460) [wlan_top_level.c: 308:42]
    }
  }
0 vr2009785 = sq4_var_service_802_11a_tx[1] >> 1 ; --> shrkw_inst1 (op 466)
[wlan_top_level.c:312:27]
* if (sq2_func_moveii_inst86[1] != 0) { --> cmpp_neq_inst0 (op 12070) [NA]
*   if (vr2009784 == 1) { --> cmpp_eq_inst0 (op 54680) [wlan_top_level.c:306:23]
0       sq4_var_service_802_11a_tx[0] = vr2009785[0]
```

```
        ; --> MOVE (op 54681) [wlan_top_level.c:312:27]
     }
     else {
0       sq4_var_service_802_11a_tx[0] = sq4_var_service_802_11a_tx[1]
        ; --> MOVE (op 51166) [NA]
     }
}
```

Some operations are marked as *speculated operation*. For more details on speculated operations, see the section that describes report_feedback_path.txt.

report_false_paths.txt

The Synphony C Compiler tool infers and reports the false paths in the generated RTL in the <implementation>/Work/report_false_paths.txt file. This report is generated whether the -false_path_handling implementation parameter has been set or not. False paths can arise in the design due to sharing of different functional units (TCABs, adders, multipliers, memories, etc.). This sharing can occur within a loop and across different loops. This report gives a list of such false paths only for the cases where functional units are shared within a loop. False paths arising due to sharing of memory or stream hardware is not reported. A sample of the report_false_paths.txt is available.

```
REPORT_FALSE_PATHS
BEGIN FALSE PATHS for paw_0.pa.pe_0.
- Path: -through median3w_inst0.o0_enable
 -through mpyw_1_stage_inst0.o0[31:0]
 -through sq2_var_x_stage_0.i0[31:0]
  Comment: shared component is mpyw_1_stage_inst0
- Path: -through sr_var_d.o0[31:0]
 -through mpyw_1_stage_inst0.o0[31:0]
 -through sq1_var_y_stage_0.i0[31:0]
  Comment: shared component is mpyw_1_stage_inst0
- Path: -through sr_var_e.o0[31:0]
 -through mpyw_1_stage_inst0.o0[31:0]
 -through sq1_var_y_stage_0.i0[31:0]
  Comment: shared component is mpyw_1_stage_inst0
- Path: -through sr_var_f.o0[31:0]
 -through mpyw_1_stage_inst0.o0[31:0]
 -through sq2_var_x_stage_0.i0[31:0]
  Comment: shared component is mpyw_1_stage_inst0
END FALSE PATHS for paw_0.pa.pe_0.
END REPORT_FALSE_PATHS
```

Example of a Successful Schedule

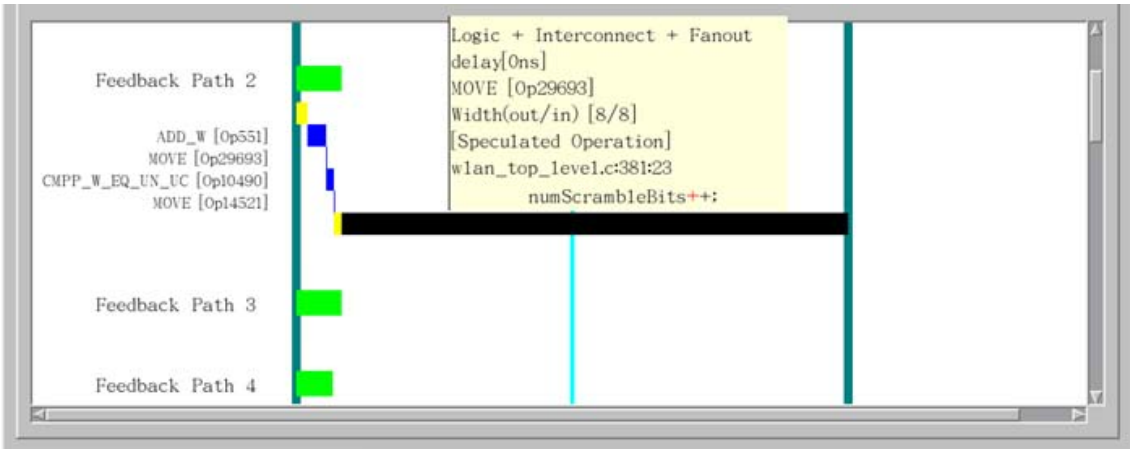
Consider the following code snippet which generated a successful schedule.

```
for (scrambler_i=0;;scrambler_i++)
  {
    ...
    numScrambleBits++;
    if (numScrambleBits==ndbps)
    {
      numScrambleBits = 0;
      break;
    }
    ...
  }
```

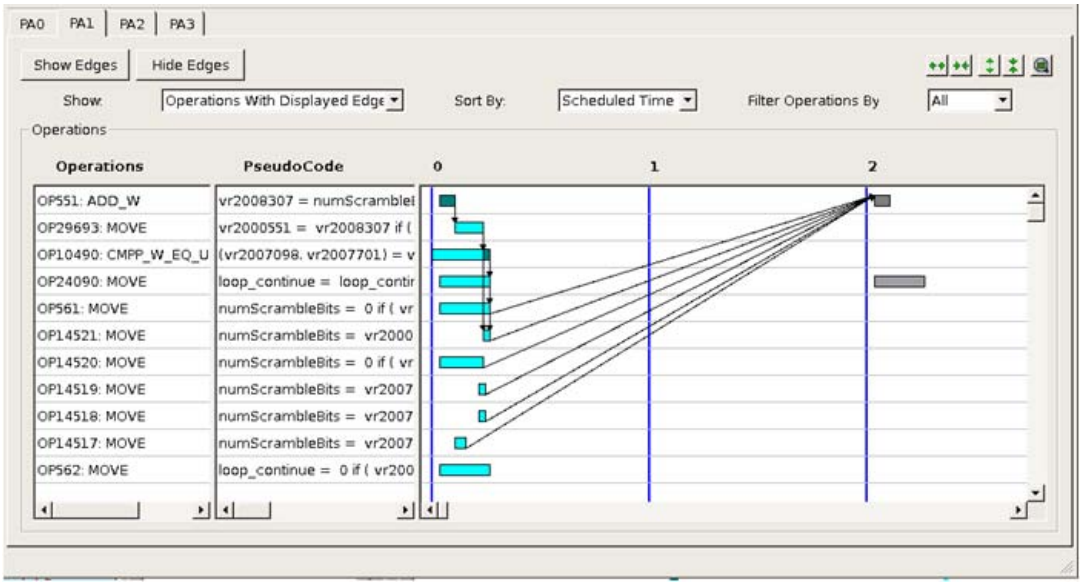
The report_feedback_path.txt snippet for this design which is successfully scheduled with an II of 2 for loop L1 and frequency of 200 MHz is shown.

```
BEGIN_CYCLE
loop: wlan_top_level_loop1, II=2
Opid   Opwidth   Opcode       Sched  Delay    Delay Source Position
                 (out/in)            time    (cycles)  Type
-------------------------------------------------------------------------------------
                                     [--------CLOCK--------]
                                      0.039    CKQ
551     (8/8)      ADD_W         0    0.068    LIF      wlan_top_level.c:381:23
                                                         numScrambleBits++;
                                                         (Speculated operation)
29693   (8/8)      MOVE          0    0.000    LIF      wlan_top_level.c:381:23
                                                         numScrambleBits++;
                                                        (Speculated operation)
10490   (1/8)      CMPP_W_EQ     0    0.030    LIF      wlan_top_level.c:382:27
                                                         if (numScrambleBits==ndbps) ^
14521   (8/8)      MOVE          0    0.000    LIF      no SourceInfo
                                                         numScrambleBits =
                                                         vr2000551 if  vr2007701
                                                        Iteration Distance = 1
                                      0.034    SETUP
-------------------------------------------------------------------------------------
                Total Cycle Delay excluding SLACK =   0.163      cycles ( 0.815  ns  )
                                Total Cycle Delay =   0.163      cycles ( 0.815  ns  )
                                   Available Time =   2.000      cycles ( 10.000 ns  )
                                            Slack =   1.837      cycles ( 9.185  ns  )
END_CYCLE
```

A snapshot of the Feedback Path Viewer for this implementation is also shown here.



The Operation schedule viewer for this design is shown. The entries shown here are only the operations which are involved in the feedback path and the arrow drawn shows the dependency across iterations.

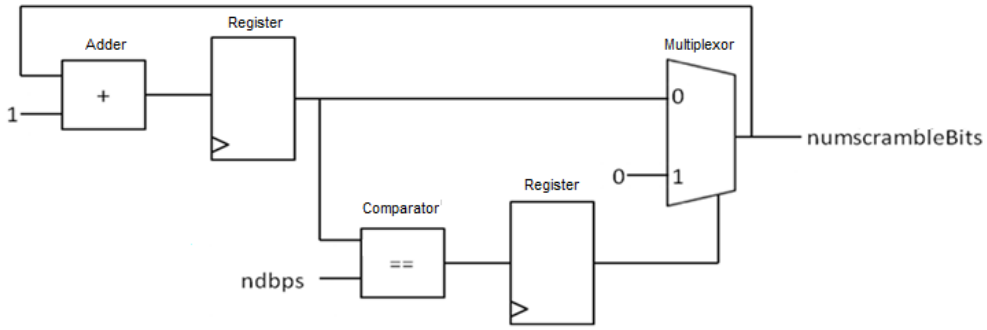


Interpretation of the feedback path

1. In loop L1, numScrambleBits variable gets incremented from a previous iteration. This is seen as Opid 551 in the text report, Operation Schedule Viewer and the Feedback Path Viewer.

2. Since this design is running at II=2, the next iteration would start at cycle 2. The II is indicated by the bottle green color in the Feedback Path viewer. In that view, the operation being started again in the next iteration is indicated by the gray colored box in cycle 2 for Op 551.
3. The incremented value is used in a comparison (numScrambleBits==ndbps) operation. This corresponds to the CMPP_* operations in the analysis views and the report segment.
4. Inside the condition where the compare is true, the numScrambleBits gets updated for the next iteration. This operation is shown as the MOVE operation in the analysis views and the report.
5. This report indicates that the scheduling of these operations was successful for loop 1 when constrained for II=2. This is indicated by the positive “Slack” value in the report and also the green box in front of “Feedback Path 2” in the Feedback Path Viewer.

   The hardware implied by this report is shown in the following diagram.



Example of a Failing Schedule

A failing schedule can happen due to several reasons listed in the section which talks about the “report_feedback_path.txt” on page 13-28.

To illustrate one such scenario, the same application which was used to generate the successful scheduling case was constrained further to synthesize all loops at II=1 at a frequency of 400 Hz. The report_summary.txt file contains the following entries confirming that the constraints were relaxed to generate the RTL.

```
Constraints Relaxed:
Name      Constraint      Specified Value  Achieved Value    Failure Type
----------------------+-----------------+-----------------+-------------+
Loop1*    II              1                  2                F1
----------------------+-----------------+-----------------+-------------+
```

* Since the design was produced at a different II from the value that you specified, the value of Task_II if you had specified, may not be met.

The following inferences can be drawn from the report_summary.txt snippet.

Loop L1 got scheduled at II=2 instead of II=1. There are no messages about a different frequency being achieved. This indicates an II failure scenario. For more information, see the section that describes report_feedback_path.txt.

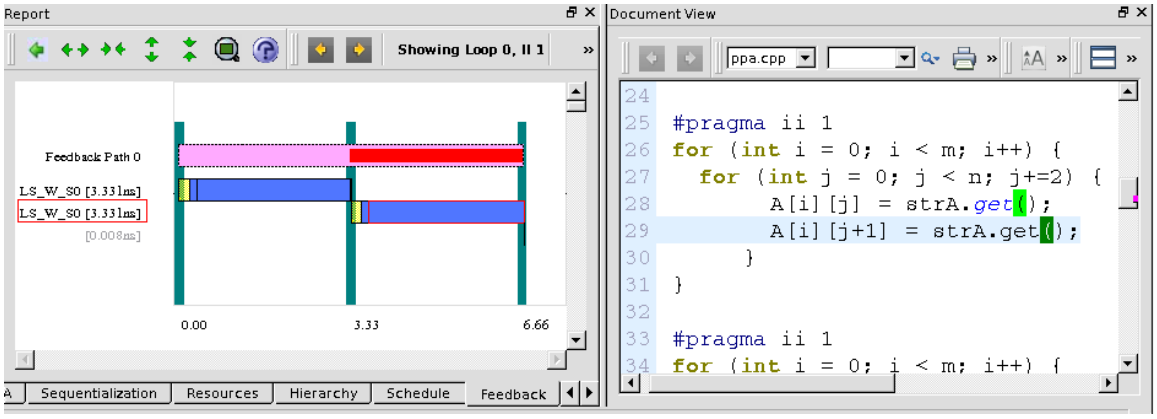
Even though the RTL gets generated in this design at increased II, it is important to understand why the constraints were not met. Let us look at the II failure first. The report_feedback_path.txt file snippet shows the following cycle.

```
Critical feedback path(s) in PA MMul_loop0 at II = 1

********************
* Feedback path: 1
* PA: MMul_loop0
* II: 1
********************

 Opid Opwidth Opcode  Delay Delay   Source Position
      (out/in)        (ns)  Type
-----------------------------------------------------------------------
                     0.194 CKQ
 204  (32/-) LS_W_S0  0.147 LIF  ppa.cpp:28:22 A[i][j] = strA.get();
                                                                ^
                     2.993 SDC
                        [Stream Dependence Edge]
                        [--------CLOCK--------]
                     0.194 CKQ
 214 (32/-) LS_W_S0  0.147 LIF  ppa.cpp:29:24 A[i][j+1] = strA.get();
                                                                 ^
                                                 Iteration Distance = 1
                              2.993    SDC
                              0.171    SETUP
                                 [Stream Dependence Edge]
                                 [--------CLOCK--------]
-----------------------------------------------------------------------
         Total Cycle Delay excluding SLACK = 6.666  ns (2.000 cycles)
                        Total Cycle Delay = 6.666  ns (2.000 cycles)
                           Available Time = 3.333  ns (1.000 cycles)
                                    Slack = -3.333 ns (-1.000 cycles)
                                   Status =   FAILED
```

The Feedback Path Viewer for the II failure shows failing paths:



Interpretation of the feedback path

- In Loop 0 running at ii 1, there are two calls to the same stream unit (`strA.get` in the source file). This is seen in the Document View. When ii = 1, it is not possible to have two calls to the stream in the same iteration.
- Understanding timing failures:

  A timing failure can occur if the cumulative timing of cascaded operations exceeds what can be fit within a single clock. A sample of such a code segment is shown here.

```
int a,b,c,d;
void ppa(){
    int i;
    for (i=0;i<10;i++){
     if (i>c){
         c += (a+b)>> d;
     }
    }
}
```

The code segment has several 32-bit global variables and the right shift operation is inferred by the tool as a variable right shift unit. This design was synthesized at 150 MHz targeting a FPGA device. The generated report_feedback_path.txt file is shown here.

```
Feedback path report prior to instruction scheduling:

Critical feedback path(s) in PA ppa_loop0 at II = 1

*******************
* Feedback path: 1
* PA: ppa_loop0
* II: 1
*******************

Opid Opwidth  Opcode    Delay    Delay  Source Position
```

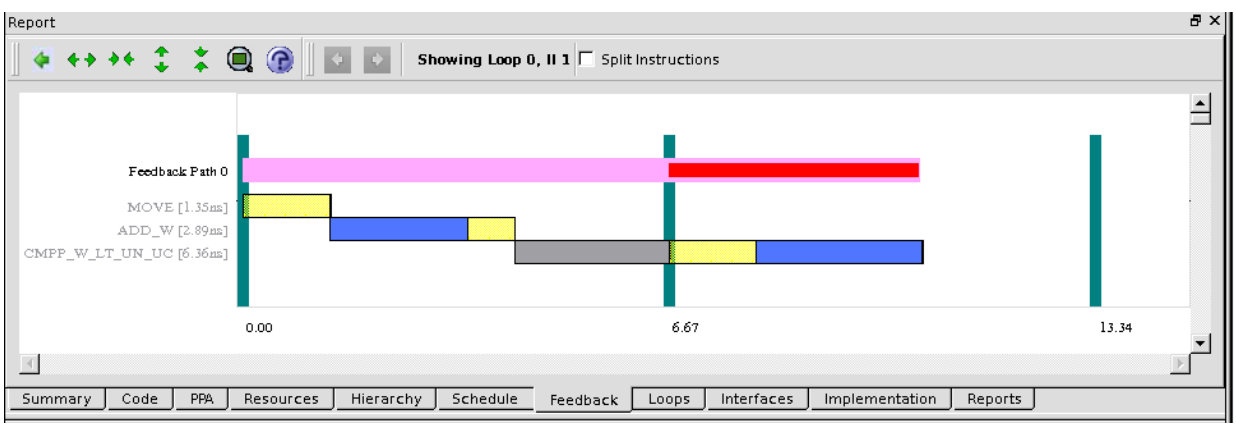
```
     (out/in)            (ns)      Type
-------------------------------------------------------------------
                         1.350     CKQ

475    (1/5)  CMPP_W_LT  2.910      LIF    ppa.c:4:1 for (i=0;i<10;i++){
                                                      ^
                         0.000     SETUP
                          [--------CLOCK--------]
                          1.350     CKQ

359    (2/1)  MOVE       0.000     LIF   no SourceInfo
                                         vp_upd_nest0_1 = 1 if  vr2000397
161    (5/5)  ADD_W      2.155     LIF   ppa.c:4:1  for (i=0;i<10;i++){
                                                      ^
                         0.740    DFMS(2)   Iteration Distance = 1
                         2.422     SLACK
                         0.000     SETUP
                         [--------CLOCK--------]
-------------------------------------------------------------------
Total Cycle Delay excluding SLACK =   8.195  ns (  1.229      cycles  )
                Total Cycle Delay =   10.617 ns (  1.593      cycles  )
                   Available Time =   6.667  ns (  1.000      cycles  )
                            Slack =  -3.950  ns ( -0.593      cycles  )
                           Status =   FAILED
```

Here loop L0 was able to meet II=1 constraint but was not able to meet the required frequency constraint and fails with a negative slack for some critical feedback path. This indicates the timing failure scenario. For more information, see the section that describes report_feedback_path.txt.

The feedback path viewer shows the feedback as follows.



This report shows that the scheduling fails with a slack of about -0.593 cycles when the clock is 150MHz. The report shows that at 150 Mz, Synphony C Compiler cannot fit the adder and the comparator in the same cycle.

Integration Reports

The following reports provide information on integration:

- report_address_map.txt
- report_interface.txt
- report_interface_timing.txt

report_address_map.txt

The report report_address_map.txt is generated at the end of synthesize step and contains a text report of the final address map. This report can be easily modified as a Tcl file and be used as inputs into the tool to always obtain the same address map.

You can do valid modifications to this address map (in the Tcl section) if required and feedback to Synphony C Compiler. Feedback process is as follows. Copy the address map file to “project” directory, edit it, and specify this new file using the command:

```
set_params -host_address_map fileName
```

Reset the implementation using `reset_implementation` command and re-run build flow.

This report contains two parts: a memory map and Tcl output.

The memory map section contains a detailed list of all the entries which are memory mapped and some other details about the size of the variable in the memory map. Since this file can be used as a Tcl input file for address map, this section of the report is prefixed with a '#' sign (all characters after '#' are treated as comments in Tcl). A sample of this section is shown here.

```
# PPA BASE ADDRESS    0X1000
#
# Start     End       Num       Bitsize  Read/  Load     Store   Name
# Address   Address   Elements           Write  Latency  Latency
# (Byte)    (Byte)
# 0x0       0x0       1         5        R+W    2        2   mode (CAPC Input)
# 0x200     0x200     1         8        R+W    2        2   rate (CAPC Input)
# 0x400     0x400     1         32       R+W    2        2   Processor Status Word (PSW)
# 0x600     0x65C     24        1        R+W    2        1   HeaderBitSerialTx (Memory)
# 0x800     0x87E     64        16       R+W    5        1   x (Memory)
# 0xA00     0xA7E     64        16       R+W    5        1   x1 (Memory)
# 0xC00     0xC7E     64        16       R+W    5        1   y (Memory)
# 0xE00     0xE7E     64        16       R+W    5        1   y1 (Memory)
# 0x1000    0x147C    288       1        R+W    3        1   Interleaver_1Stage_x (Memory)
# 0x1800    0x1C7C    288       1        R+W    3        1   Interleaver_2Stage_x (Memory)
#
```

The Tcl output section contains the address offsets for different symbols. A sample of this section for the same example shown in the Description section is shown here.

```
set HOST_ADDRESS_MAP {
{ {ADDR 0} {SYMBOL mode} }
{ {ADDR 512} {SYMBOL rate} }
{ {ADDR 1024} {SYMBOL PICO_PSW} }
{ {ADDR 1536} {SYMBOL HeaderBitSerialTx} }
{ {ADDR 2048} {SYMBOL x} }
{ {ADDR 2560} {SYMBOL x1} }
{ {ADDR 3072} {SYMBOL y} }
{ {ADDR 3584} {SYMBOL y1} }
{ {ADDR 4096} {SYMBOL Interleaver_1Stage_x} }
{ {ADDR 6144} {SYMBOL Interleaver_2Stage_x} }
}
```

report_interface.txt

This report is generated at the end of synthesis phase in the work directory. It describes the ports and their protocols that the hardware provides to the external environment. The ports that are created for the module can be described in two ways: logical and physical. Several terms described here are described in more detail in the Chapter 14, “System Integration and Verification.”

A logical port can be described as a collection of ports which can be grouped together to perform a particular functionality; for example, stream/SRAM/raw wire/memory mapped, etc. A physical port is a Verilog port in the top-level PPA interface of the Synphony C Compiler RTL. This section provides information about the logical ports in the design, how they map to the physical ports, and how they map back to the Synphony C Compiler C or C++ sources. Details about each of the interface connections, like direction of the port (Input/Output), width, any possible mapping to a C variable or memory etc., are described here for all the logical ports.

Logical port description syntax

Each logical port has a fixed number of fields. There are no optional fields. A description of logical port syntax which has N fields is as follows:

```
logical_port_name {
field_1_description
field_2_description
field_N_description
}
```

Field description syntax:

```
< INPUT | OUTPUT > field_name physical_port_name width
CNAME c_variable_or_function_name
```

The PPA has a set of predefined logical ports. Each logical port consists of a set of predefined fields that describe physical ports and their properties. Each logical port has a well-defined protocol, a mapping to physical ports, and wherever applicable, a mapping back

to the original C sources. An alphabetically sorted list of logical ports that may appear in this report and their functionality is listed in the following table.

Logical Port	Classification	Comments
ABORT_PORT	Control	Used to abort execution of task in the module
CONTROL_PORT	Control	Controls the execution of the module
DATAIN_PORT	DATA	Scalar input data into the module
DATAOUT_PORT	DATA	Scalar outputs data from the module
DIRECT_MEMORY_PORT	DATA	Memory element input and output
ENABLE_PORT	Control	Enable or disable the entire module
ERROR_PORT	Status	Signals error conditions in configuration of the module
HOST_PORT	Control or Status or Data (memory mapped)	All memory mapped transactions take place through this port
PA_ENABLED_CLK_GATE_PORT	Control	Signals to the external environment that the module is idle and clock can be gated to this module
PA_STALLING_PORT	Status	Conveys information about stalling behavior of different stall domains
STANDARD_MEMORY_PORT	DATA (Memory)	Memory element input and outputs
STATUS_PORT	Status	Conveys the whether the module is processing a task or not

Logical Port	Classification	Comments
STREAMIN_PORT	Data (Input Stream)	Flow controlled signal handshake to accept data into the module
STREAMOUT_PORT	Data (Output Stream)	Flow controlled signal handshake to send data outside the module.
SYSTEM_PORT	System	Clock and reset signals of the module

Detailed descriptions of the sections in the report:

- SYSTEM_PORT

  This section contains the physical-name and widths of the clock and the reset signal, along with the design attributes like clock edge, reset sensitivity and level. A sample of how this kind of ports would be reported is as follows.

```
SYSTEM_PORT {
   INPUT CLOCK clk 1 POSEDGE
   INPUT RESET reset 1 SYNC ACTIVE_LOW
}
```

- ENABLE_PORT

  This section contains information about the signal that enables the PPA, its width and active state. The enable signal is present only for PPA designs which are not always enabled and if the hardware is built to have raw_control for its control interface. This signal is not present by default on TCABs. The presence of this port on the PPA module can be controlled using the following command.

```
set_params -always_enabled yes|no
```

  A sample of the enable port is as follows.

```
 ENABLE_PORT {
     INPUT ENABLE enable 1 ACTIVE_HIGH
}
```

- CONTROL_PORT

  This section gives information about the control ports of the PPA. These ports are useful for controlling the flow of a task. These signals are present only for designs where the hardware is built to have raw_control for its control interface. Use the following implementation parameter to specify whether the control logic is memory mapped or not.

```
set_params -interface_type option
```

A sample of the control port is as follows.

```
CONTROL_PORT {
    INPUT START_VALID start_valid 1 ACTIVE_HIGH
    INPUT DONE_READY done_ready 1 ACTIVE_HIGH
}
```

- STATUS_PORT

  The signals in this section are useful for finding the status of the PPA at any point of time. For more information on these ports, see the Chapter 14, “System Integration and Verification.” Use the following implementation parameter to specify whether the status information is memory mapped or not.

  ```
  set_params -interface_type option
  ```

  A sample of the status port is as follows.

  ```
  STATUS_PORT {
      OUTPUT START_READY start_ready 1 ACTIVE_HIGH
      OUTPUT DONE_VALID done_valid 1 ACTIVE_HIGH
  }
  ```

- HOST_PORT

  This section contains the interface port, which can be used between the PPA and a host processor (often through a system bus). All the memory-mapped entities (control/scalars/arrays) in the PPA are accessible to the host through this port. Since this is a memory-mapped interface primarily, the type of access is a read-write. The names of the actual ports differ based on whether this port follows the ocp protocol or native protocol to exchange data. The representation of this report is different for native and ocp protocol in host access mechanism and is shown:

  Native Interface

  ```
  HOST_PORT {
     TYPE READ-WRITE
     INPUT REQ host_req 1 ACTIVE_LOW
     OUTPUT ACK host_ack 1 ACTIVE_HIGH
     INPUT MODE host_mode 1 ACTIVE_LOW
     INPUT ADDR host_addr 4
     INPUT WRITE_DATA host_wdata 32
     OUTPUT READ_DATA host_rdata 32
  }
  ```

OCP Interface

```
HOST_PORT {
   TYPE READ-WRITE
   INPUT REQ host_req 1 ACTIVE_HIGH
   OUTPUT ACK host_ack 1 ACTIVE_HIGH
   INPUT MODE host_mode 1 ACTIVE_HIGH
   INPUT ADDR host_addr 3
   INPUT WRITE_DATA host_wdata 32
   OUTPUT READ_DATA host_rdata 32
}
```

- SCALARS

  Unless specified otherwise, scalars in global scope are synthesized to be accessed by the host through ports. The kind of ports (input, output, or both) depends on the way the variable is accessed in the C/C++ source.

  If a scalar variable is written to unconditionally inside the synthesizable code before being used inside the synthesizable code, only output ports are created for such scalars. If the variable is used ahead of it being written or if the variable is written only conditionally before it is read, both input and output ports would be created for such scalar. If a variable is only read inside the design and not updated, only read ports get created for the variable. The "localize_access Pragma" on page 7-33 can be used on global variables to eliminate ports for those scalar variables at the hardware boundary.

  By default, all the input ports are registered if the hardware is built as a PPA. If the module is built as a TCAB, there are options to control whether the TCAB inputs are registered or not. If hardware is created for a scalar that is marked with the `stable` pragma, only an input port gets created for such scalar. If such a scalar is marked with an additional `hold_register` pragma, the input port is also qualified with an additional enable port to indicate when the data is to be sampled from the scalar input port. For more information about these pragmas, see "stable Pragma" on page 7-25 and "hold_register Pragma" on page 7-29.

- DATAIN_PORT

  This section contains information about different input signals into the system, like their width, whether it is registered internally immediately, active state and any possible source mapping to C-variable. A sample of the datain port is as follows.

```
DATAIN_PORT {
    TYPE HAS_REGISTER
    INPUT DATA scalar_in_a_0 32
    CNAME a verysimple.c:4:12
}
```

  The port has not been marked with `stable` pragma and so it is registered inside the generated hardware. This is indicated by the HAS_REGISTER type. A sample of the report for a scalar marked with `stable` pragma is as follows.

```
DATAIN_PORT {
       TYPE CONST_ACROSS_PIPELINED_CALLS
       INPUT DATA scalar_in_a1_0 32
       CNAME A1 app.c:3:7
}
```

- DATAOUT_PORT

  This section lists the signal used to communicate with an external world, their widths and the corresponding C source position. A sample of the dataout port is as follows.

```
DATAOUT_PORT {
    OUTPUT DATA scalar_out_b_0 32
    CNAME b verysimple.c:4:14
}
```

  If scalar outputs of a TCAB are specified to have output valid signal, then there is an additional output_enable signal as shown.

```
DATAOUT_PORT {
  OUTPUT DATA scalar_out_d_0 32
  OUTPUT ENABLE scalar_out_d_0_enable 1
  CNAME D app.c:3:17
}
```

- STREAMIN_PORT

  This section lists all the signals used to take input from an external world through the stream, their widths, active status, and the corresponding C source position. A sample of the control port is as follows.

```
STREAMIN_PORT {
     CONTROL_MODE ACTIVE
     INPUT DATA stream_in_rac_data 36
     OUTPUT REQUEST stream_in_rac_ready 1 ACTIVE_HIGH
     INPUT READY stream_in_rac_valid 1 ACTIVE_HIGH
     CNAME rac axi_easy.cpp:29:24
}
```

- STREAMOUT_PORT

  This section lists all the signals used to send output to an external world through the stream, their widths, active status, and the corresponding C source position.

```
STREAMOUT_PORT {
     CONTROL_MODE ACTIVE
     OUTPUT DATA stream_out_rdc_data 33
     OUTPUT REQUEST stream_out_rdc_valid 1 ACTIVE_HIGH
     INPUT READY stream_out_rdc_ready 1 ACTIVE_HIGH
     CNAME rdc axi_easy.cpp:35:24
}
```

- STANDARD_MEMORY_PORT

This section provides the names of signals that are used to access a particular user-supplied memory, their widths and active status along with the name and source position of the corresponding memory in the C file.

```
STANDARD_MEMORY_PORT {
     TYPE READ-WRITE
     OUTPUT CEN ram_memory_cen 1 ACTIVE_LOW
     OUTPUT WEN ram_memory_wen 1 ACTIVE_LOW
     OUTPUT ADDR ram_memory_a 5
     OUTPUT WRITE_DATA ram_memory_d 32
     INPUT READ_DATA ram_memory_q 32
     CNAME memory axi_easy.cpp:41:8
     BUFFER_ID 0 1
}
```

- PA_STALLING_PORT

  This section contains information about signal name, which conveys whether a particular stall domain is stalling (these ports are optionally generated). The PA_id in this report corresponds to the PA number for which this stalling port is reported.

```
PA_STALLING_PORT {
   OUTPUT psw_sa_0_stalling 1 ACTIVE_HIGH
   PA_ID 0
}
```

- PA_ENABLED_CLK_GATE_PORT

  This section contains information about the signal name of the gated clock for a particular stall domain (PA/TCAB) and its gating-enable signal, along with the bitsize of the port (usually 1), its sensitivity and direction. The stall domain number for which this signal applies is also listed in the PA_ID field. For more details refer to the Chapter 5, “Programming Model Basics.”

```
PA_ENABLED_CLK_GATE_PORT {
   INPUT CLOCK clk_verysimple_ppa_paw_0 1 POSEDGE
   OUTPUT ENABLE clk_gate_enable_verysimple_ppa_paw_01 ACTIVE_HIGH
   PA_ID 0
}
```

- ABORT_PORT

  This section contains the signal that handles the task of aborting the PPA. These do not have any C source information. The abort signal is a 1-bit input signal and the psw_aborting is a 1-bit output signal. Both of these are active high signals.

```
ABORT_PORT {
   INPUT ABORT abort 1 ACTIVE_HIGH OUTPUT ABORT psw_aborting 1
   ACTIVE_HIGH
}
```

- ERROR_PORT

This contains the name of the output Error signal, its width and active state.

```
ERROR_PORT {
    OUTPUT ERROR psw_err 1 ACTIVE_HIGH

}
```

- DIRECT_MEMORY_PORT

  This section lists the names of signals that are used by the host to access a user-supplied memory. These ports are present on the Host-PPA interface. For more details on these ports, see the Chapter 14, "System Integration and Verification." The path from PPA to the memory would be the same as the STANDARD_MEMORY_PORT. A sample of this port information in report_interface.txt is as follows:

```
DIRECT_MEMORYOUT_PORT {
     TYPE READ-WRITE
     INPUT REQ direct_ram_memory_req 1 ACTIVE_HIGH
     OUTPUT ACK direct_ram_memory_ack 1 ACTIVE_HIGH
     INPUT MODE direct_ram_memory_mode 1 ACTIVE_HIGH
     INPUT ADDR direct_ram_memory_addr 5
     INPUT WRITE_DATA direct_ram_memory_wdata 32
     OUTPUT READ_DATA direct_ram_memory_rdata 32
     CNAME memory axi_easy.cpp:41:8
     BUFFER_ID 0 0
}
```

Note:
If the scalar is of struct type, the packing order of individual bits in different fields at the interface is described in "Packed and Unpacked Memories" on page 8-11.

report_interface_timing.txt

This report is generated at the end of the synthesize phase and it shows the boundary timing aspects of the hardware created using the Synphony C Compiler tool. For all inputs, the IN-DELAYS section describes the logic delay in the path from a given port to the first register on the path. For all outputs, the OUT-DELAYS section describes the logic delay from the last register on a path to the external output port. Any logical paths which just go through the design without being registered are described in the section THRU-DELAYS. The information in this report is useful to make integration level decisions on whether the Synphony C Compiler generated input or output of a module needs to be registered. A sample report is shown.

```
IMPLEMENTATION  : axi_easy_ppa

 IN-DELAYS
  start_valid             ---> REG                      : 0.24008
```

```
 done_ready               ---> REG                      : 0.54666
 direct_ram_memory_req    ---> REG                      : 0.36875
 direct_ram_memory_mode   ---> REG                      : 0.32500
 ram_memory_q             ---> REG                      : 0.12000
 stream_in_rac_data       ---> REG                      : 1.65935
 stream_in_rac_valid      ---> REG                      : 1.60545
 stream_out_rdc_ready     ---> REG                      : 0.04610

OUT-DELAYS
 REG                      ---> start_ready              : 2.55589
 REG                      ---> done_valid               : 1.68433
 REG                      ---> direct_ram_memory_ack    : 0.17000
 REG                      ---> ram_memory_a             : 1.53775
 REG                      ---> ram_memory_cen           : 1.49159
 REG                      ---> stream_in_rac_ready      : 1.14858
 REG                      ---> stream_out_rdc_data      : 0.47610
 REG                      ---> stream_out_rdc_valid     : 0.40610

THRU-DELAYS
 direct_ram_memory_addr   ---> ram_memory_a             : 0.10000
 direct_ram_memory_wdata  ---> ram_memory_d             : 0.10000
 direct_ram_memory_req    ---> ram_memory_a             : 0.15665
 direct_ram_memory_req    ---> ram_memory_d             : 0.24375
 direct_ram_memory_req    ---> ram_memory_cen           : 0.24875
 direct_ram_memory_req    ---> ram_memory_wen           : 0.24875
 direct_ram_memory_mode   ---> ram_memory_wen           : 0.20500
 ram_memory_q             ---> direct_ram_memory_rdata  : 0.00000
 ram_memory_q             ---> stream_out_rdc_data      : 0.21124
 stream_in_rac_data       ---> start_ready              : 1.68853
 stream_in_rac_data       ---> done_valid               : 0.86117
 stream_in_rac_data       ---> ram_memory_a             : 0.58548
 stream_in_rac_data       ---> ram_memory_cen           : 0.66843
 stream_in_rac_valid      ---> start_ready              : 1.63463
 stream_in_rac_valid      ---> done_valid               : 0.80727
 stream_in_rac_valid      ---> ram_memory_a             : 0.66847
 stream_in_rac_valid      ---> ram_memory_cen           : 0.61453
```

QoR Analysis Reports

The following reports and the views in GUI provide information on QoR:

- report_cost.txt
- report_register.txt
- report_hierarchy.pdf and report_hierarchy_detailed.pdf

You can also use the following views for analysis. They are described in Chapter 2:

- Resource Browser for PPA Graph

- Relative Cost Viewer
- Hierarchy Graph

report_cost.txt

The cost report gives a breakdown of the estimated hardware cost of the PPA. It is organized in the following format:

Cost per PA

Cost per PAW

Cost for entire TCAB/PPA

```
For each PA:
processor_cost { // # gates
name = <appname>_pe_<loop_num>
total_processor_cost = <PE cost>
macrocell description rtl module name (#inst, #bits) = cost?
----------------------------------------------------------------------
<list of components and costs>
}
For each PA:
systolic_cost { // # gates
name = <appname>_pa_<loop_num>
total_array_cost = <PA cost>
array_shape = { 1 }
macrocell description rtl module name (#inst, #bits) = cost?
----------------------------------------------------------------------
<appname>_pe_<loop_num> ( 1, 0) = <PE cost>
<list of components and costs, possibly empty>
}
For each PA:
systolic_cost { // # gates
name = <appname>_paw_<loop_num>
total_cost = <raw PA cost>
macrocell description rtl module name (#inst, #bits) = cost?
----------------------------------------------------------------------
<appname>_pa_<loop_num> ( 1, 0) = <PA cost>
<list of components and costs, possibly empty>
}
```

The component lists have this form:

ASIC

```
macrocell description     rtl module name       (#inst, #bits) =  cost
----------------------------------------------------------------------
  systolic             <appname>_paw_<loop_num> (  1,     0) =  61983.00
  selector (1-hot)         select_3_1_wn        (  1,    32) =     64.00
  pred compare(<,<=,>,>=)  cmpp_ineq            (  8,    96) =   5496.00
```

```
  adder                      addw                ( 16,   277) =   3179.00
  pred compare (==)          cmpp_eq             (  8,   152) =   1008.00
  subtractor                 subw                (  4,    70) =    935.00
```

FPGA

```
macrocell description rtl module name (#inst,#bits)= LUT FF DSP BRAM
-----------------------------------------------------------------------
 adder             addw                (   1,     32) =  32  0   0   0
 and gate          (2-input)and2n(beh)(   2,      2) =   0  0   0   0
 or gate           (2-input)or 2n(beh)(   3,      3) =   0  0   0   0
 or gate           (2-input)or 2n(beh)(   1,      1) =   0  0   0   0
```

The cost is given in terms of gates for ASIC, while it is broken into LUT, FF, DSP and BRAM for FPGA.

In the report, rtl module name specifies the macrocell's module name (not instance name), #inst gives the number of instantiations of this macrocell at this level of the hierarchy, #bits gives the total bitwidth of these instantiations, and cost denotes the total estimated cost in gate equivalents for all instantiations of this macrocell at this layer of hierarchy.

The next section would contain information about aggregate information about the entire implementation. Based on whether the implementation is a TCAB or PPA, one of the following sections (PPA/TCAB) is visible.

For the entire PPA:

```
ppa_cost { // # gates
name            = <appname>_ml
  total_ppa_cost   = <total PPA cost>
  macrocell description     rtl module name    (#inst, #bits) =cost
-------------------------------------------------------------------
  <list of components and costs>
}
```

For the entire PPA:

```
Register (sregn/delayn) Usage {
  PURPOSE                                          #BITS
  Static Variable Registers                       <bits>
  Rotating Variable Registers                     <bits>
  Boundary Registers                              <bits>
  Stream/Fifo Handshake Registers                 <bits>
  Loop Initialization/Finalization Registers      <bits>
  Artisan Memory Ack Generation Registers         <bits>
  Multiloop Timing Controller Registers           <bits>
  FIFO Registers                                  <bits>
  Replay Buffer Registers                         <bits>
  Register-based Memory Registers                 <bits>
  Misc Registers                                  <bits>
  Registers inside TCABs                          <bits>
  ---------------------------------------------------------------
  Total Register Bits                            <total bits>
```

For the entire TCAB:

```
tcab_cost { // # gates
  name = <appname>_ml
  total_tcab_cost = <total TCAB cost>
  macrocell description rtl module name (#inst, #bits) = cost?
-----------------------------------------------------------------------
  <list of components and costs>
}
```

For the entire TCAB:

```
PURPOSE                                              #BITS
  Static Variable Registers                           <bits>
  Rotating Variable Registers                         <bits>
  Boundary Registers                                  <bits>
  Stream/Fifo Handshake Registers                     <bits>
  Loop Initialization/Finalization Registers          <bits>
  Artisan Memory Ack Generation Registers             <bits>
  Multiloop Timing Controller Registers               <bits>
  FIFO Registers                                      <bits>
  Replay Buffer Registers                             <bits>
  Register-based Memory Registers                     <bits>
  Misc Registers                                      <bits>
  Registers inside TCABs                              <bits>
  ------------------------------------------------------------------
  Total Register Bits                               <total bits>
```

All the different kind of registers in this report are implemented as flip-flops. The different types of registers are as follows.

- Static Variable Registers are single registers which are used when the schedule of operations guarantees that additional copies of the registers are not required to handle data dependences. Where possible, these registers use derivatives of the original source variable names.
- Rotating Variable Registers are registers used to handle data dependencies across iterations. They allow you to create copies of the values so that there are no violations of data dependences. They may also correspond to state variables in the source code. Where possible the registers are named with reference to the original variable names or derivatives of those names.
- Boundary registers are inserted by default for all external ports to facilitate timing closure and integration of the RTL. This line gives the total register bits used for SRAM, ROM, stream, host interface, and raw input or output ports.
- Stream or FIFO Handshake Registers
  - Loop Initialization or Finalization Registers
  - Artisan Memory Ack Generation Registers are used to translate between external synchronous read or write SRAM interfaces and the Synphony C Compiler Native

interface used internally. The Synphony C Compiler Native interface includes an Ack signal to indicate the return of data from the memory (or from the arbiter, if it is present). A small number of registers are used to create this Ack signal.

- Multiloop Timing Controller Registers give the bits used in the timing control of the PAs, for instance tokens that are passed to indicate when control of a memory buffer has passed from one PA to another.
- FIFO Registers are the bits contained in inter-loop stream FIFOs. This includes both data and valid bits.
- Replay Buffer Registers give the total bits used by replay buffers. Replay buffers are used in external memory load and other multi-cycle functional units to capture outstanding accesses during stall cycles.
- Register-based Memory Registers include all internal_fast memory bits.
- Misc Registers contain any other register bits used for control of the PPA.
- Registers inside TCABs include all registers used within all TCABs used in the current implementation.

For the entire PPA:

```
Aggregate cost report:
Macrocell Description Count Gates?% %(cumm)
----------------------------+-----------------+------+-------+-----+
<list of components, description, count and cost>
```

This component list sums all the macrocells in the PPA and is the best place to look for a quick summary of the content, e.g. number of multipliers, adders, load/store units, etc. A typical list might begin as follows:

```
For the entire PPA:
Aggregate cost report:

Macrocell              Description              Count   Gates    %
%(cumm)
----------------------+-----------------------+------+-------+-----+----+
cmpp_ineq              pred compare(<,<=,>,>=)     46   37500  25.2  25.2
sregn_noinit           register                   388   27365  18.4  43.5
sramff_2               register based sram         16   14076   9.4  53.0
mpyw_1_stage                                        9   12157   8.2  61.1
sramff_raw_2           reg sram with rawaccess      6    8918   6.0  67.1
addw                   adder                       54    7696   5.2  72.3
select_2_1_wn          selector (1-hot)          1265    5976   4.0  76.3
select_3_1_wn          selector (1-hot)           328    3958   2.7  78.9
and2n                  and gate (2-input)        3281    3281   2.2  81.1
```

All costs listed in the report are estimates based on characterized data from the chosen technology target. Each technology target has a gate count multiplier that corresponds to a

number of square microns. For example, for 0.13 micron technology the multiplier is typically about 5 square microns per gate.

The aggregate cost section shows the macrocells in order of their contribution to the total cost (highest first). The percentage contribution for each macrocell and cumulative contribution of all macrocells from the top of the list are shown in the last two columns.

You must always perform place and route to determine the final cost of the designs as the Synphony C Compiler tool cannot predict the exact optimizations that logic synthesis tools make.

report_register.txt

The register report contains the details of the registers used in the implementation. The report contains details such as the location of the register, type of the register, schedule stage where this register is used, width of the register, the name in C source code and the RTL name.

In certain cases, groups of registers are created to signify the same entity at different stages of module execution. For all the registers from the same group, the information provided is specific and with respect to the group instead of listing each individual register in it. There are some registers on the control path which provide timing insulation (and therefore classified as boundary registers even though they are in the control path).

The organization of the report is as follows:

The initial section of the report displays the legend of various types of registers used in the implementation.

```
START OF REGISTER REPORT
========================
Legend for Register Types
_________________________
Boundary Registers                                    BOUNDARY
Host Access Latency Uniformization Registers           LATENCY
Stream/Fifo Handshake Registers                      HANDSHAKE
Loop Initialization/Finalization Registers           INITFINAL
Artisan Memory Ack Generation Registers              MEMORYACK
Arbitration Support Registers                           ARBITR
Multiloop Timing Controller Registers                   TIMING
FIFO Registers                                            FIFO
Replay Buffer Registers                                 REPLAY
Register-based Memory Registers                         SRAMFF
Misc Registers                                            MISC
Multi-stage FU Registers                             MULTISTG
Registers inside imported TCABs                           TCAB
Counters                                               COUNTER
Memory Path Registers                                MEMORYPTH
FIFO Error Detection                                   FIFOERR
```

```
Stream Buffer                                          STREAMBUF
State/Pipeline                                              PIPE
Control                                                   CONTROL
Registers inside imported PPAs                               SPPA
Loop Carried                                                   LC
Input                                                          IN
Output                                                        OUT
```

Followed by the legend is the PA information section, which lists the different types of registers used in each PA in the application. All the columns in this section are listed with the following fields.

```
                  Num-     Num-    Bits/   Source-
Location  Type    Stages   Bits    Stage   Mapping     RTL-Instance
```

The following table describes each field.

Field	Description
Location	This field indicates the hierarchy at which the register is instantiated.
Type (# Registers in Group)	This field can be used to understand the functionality accomplished by the register.
Num-Stages	At each hierarchy, this field indicates the number of stages for which a value is required to be present. If a variable is being used across multiple stages in an iteration, this value can be more than one.
Num-Bits	This field shows the total number of bits for that register. For registers that are used in multiple stages in an iteration, a rotating register would get created for that variable and the “Bits” field contains the total number of bits in the rotating register.
Bits/Stage	This field shows the number of bits used by each stage.
Source-mapping	Whenever the information regarding the source structure is available the C variable names, file names and the source position may be mentioned.
RTL-Instance	This shows the exact name that is used in the generated RTL for the variable.

A summary section, which lists the different registers used in the PA immediately follows the corresponding PA information section. A sample of the PA information section and the summary section is shown:

```
Num-      Num-        Bits/                    Source-
Location  Type       Stages  Bits  Stage   Mapping    RTL-Instance
________  ____       ______  ____  _____   _________   ____________

PA_0      TCAB         1     218     218    ____        block_neighbordiff_paw_0_inst0
PA_0      TCAB         1     173     173    ____    cyclicprefix_and_zeropad_paw_0_inst0
PA_0      1 PIPE       1     32      32     i            sq1_var_i
PA_0      1 IN,LC      1     32      32     numPack      sr_var_numpack
PA_0      1 IN,LC      1     1       1      ____         sq0_func_brf_inst0
PA_0      1 PIPE       1     1       1      ____         sq2_func_moveii_inst8
PA_0      1 IN         1     1       1      ____         sr_var_loopcounter
PA_0      MISC         1     1       1      ____    controller_rsflipflop_noinit_inst0
PA_0      MISC         1     1       1      ____         controller_counter_inst0
PA_0      MISC         1     1       1      ____         firstiter_stagereg0
PA_0      MISC         1     1       1      ____         lastiter_stagereg0
PA_0      CONTROL      1     1       1      ____    shiftreg_stage0_sregn_noinit_inst0
PA_0      MISC         1     1       1      ____         statusreg
PAW_0     MISC         1     1       1      ____         saaslmentity_rsflipflop_inst0
Summary of Register Usage in PAW_0:
Functional Type                                                    Bits
_____________________________________________________________________
Registers inside TCABs                                              391
State/Pipeline                                                       67
Misc Registers                                                        6
Control                                                               1
_____________________________________________________________________
Total Register Bits                                                 465
```

The next section contains the information about the registers in the PPA. A summary section of the registers in the PPA follows the PPA register information. A sample of the PPA section and its summary section is shown:

```
                   Num-   Num- Bits/ Source
Location  Type     Stages Bits Stage -Mapping     RTL-Instance
________  ____     ______ ____ _____ _________   ___________________________
PPA       BOUNDARY   1    1    1     ____       registered_enable_sregn_noinit_inst0
PPA       BOUNDARY   1    1    1     ____registered_start_task_init_sregn_noinit_inst1
PPA       BOUNDARY   1    1    1     ____registered_start_task_final_sregn_noinit_inst2
PPA       BOUNDARY   1    1    1     ____registered_clear_init_done_sregn_noinit_inst3
PPA       BOUNDARY   1    1    1     ____registered_clear_task_done_sregn_noinit_inst4
PPA       BOUNDARY   1    1    1     ____      registered_port_sregn_noinit_inst5
PPA        MISC      1    1    1     ____       mlinterface_rsflipflop_noinit_inst0
PPA        MISC      1    1    1     ____       mlinterface_rsflipflop_noinit_inst1
PPA       CONTROL    1    1    1     ____       mlinterface_incdec_counter_inst0
PPA       CONTROL    1    1    1     ____       mlinterface_incdec_counter_inst1
PPA        MISC      1    1    1     ____       multiloop_rsflipflop_noinit_inst2
PPA        MISC      1    1    1     ____       multiloop_rsflipflop_noinit_inst3
PPA        MISC      1    1    1     ____       sregn_noinit_inst6_dly
PPA        MISC      1    1    1     ____       sregn_noinit_inst7_dly
PPA      MEMORYACK   1    1    1     ____       sregn_noinit_inst8_dly
```

```
Summary of Register Usage in PPA:
Functional Type                                                  Bits
_____________________________________________________________________
Boundary Registers                                                  6
Misc Registers                                                      6
Control                                                             2
Artisan Memory Ack Generation Registers                             1
_____________________________________________________________________
Total Register Bits                                                15
```

The final section of the report contains the summary of all registers used in the application. A sample of the summary of the complete register usage is shown:

```
Summary of Complete Register Usage:
Functional Type                                                  Bits
_____________________________________________________________________
Registers inside TCABs                                            391
State/Pipeline                                                     67
Misc Registers                                                     12
Control                                                             3
Boundary Registers                                                  6
Artisan Memory Ack Generation Registers                             1
_____________________________________________________________________
Total Register Bits                                               480
======================
END OF REGISTER REPORT
```

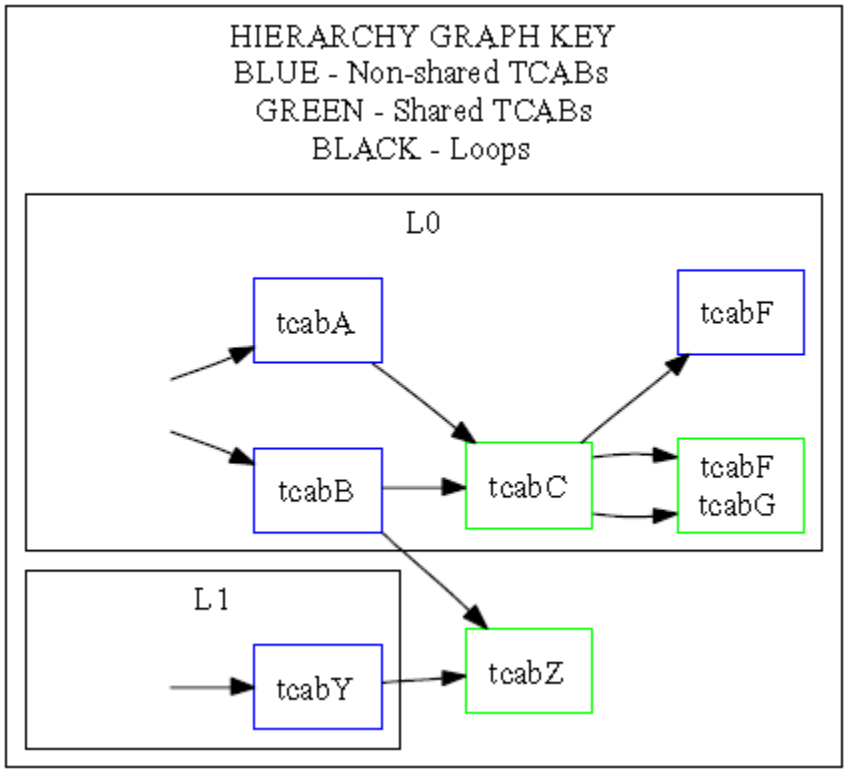
report_hierarchy.pdf and report_hierarchy_detailed.pdf

These reports show the physical location and sharing of TCABs in RTL. An edge from TCAB A to TCAB B indicates that TCAB B is called by TCAB A. TCABs are shown merged into one block wherever possible for readability. In a few cases, a TCAB can be shared from different hierarchies in the design. A box is colored either green or blue depending on whether the instance is hierarchically-shared or not. All loops in both the reports are shown in white color.

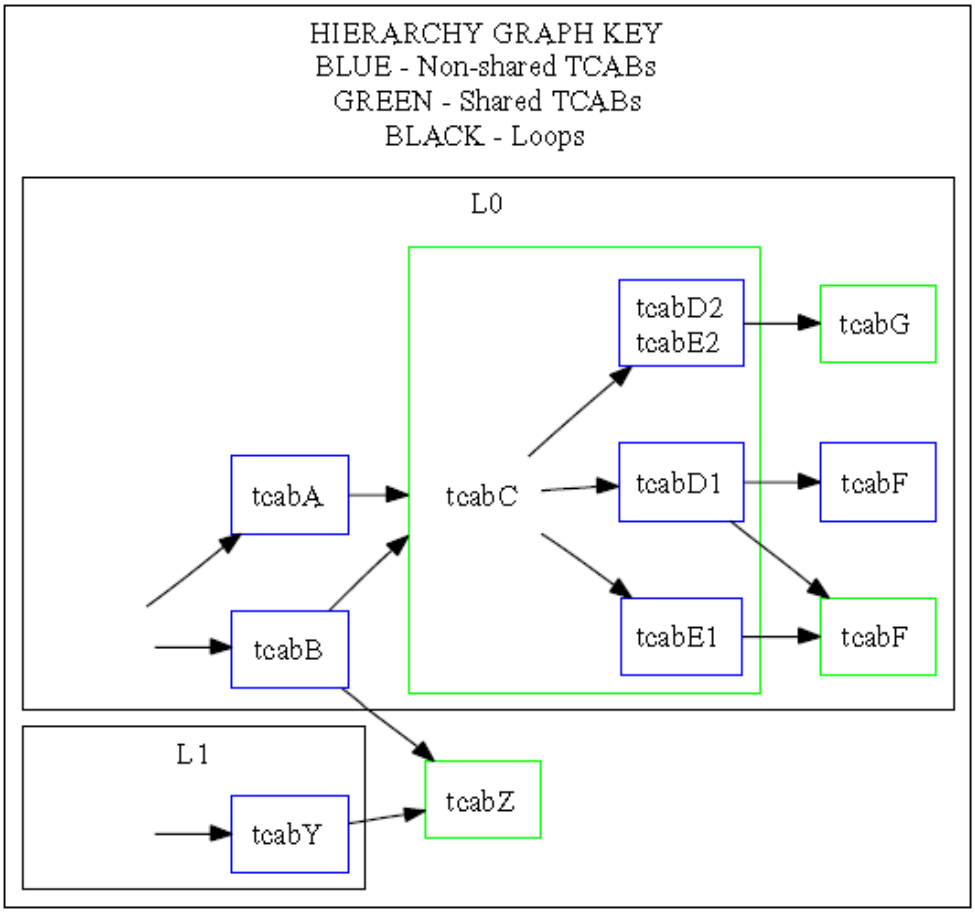
The “report_hierarchy_detailed.pdf” shows the complete hierarchy, with child-TCABs sitting inside parent-TCABs, while the “report_hierarchy.pdf” hides all child-TCABs and only shows first-level TCABs

The following are examples of these reports.

- report_hierarchy.pdf



- report_hierarchy_detailed.pdf



Performance Analysis Reports

Performance information is reported from `verify -rtl` command. Each of these reports captures loop start and stop times as well as some aggregate performance information. These reports are as follows:

- Dynamic Performance Graph
- report_performance.txt
- report_verify_rtl_performance.txt
- Support for Performance Charting

Dynamic Performance Graph

This analysis view helps you to understand the performance aspects of the design from the simulation flow. For more details, see "Dynamic Performance Graph" on page 2-65.

report_performance.txt

The `report_performance.txt` report file shows the synthesis decisions made by the tool that affect performance characteristics, such as clock rate, throughput, and latency. You can use the information in the report to determine the performance characteristics of the implementation and to set the performance constraints for subsequent synthesis runs.

The beginning of the report looks like this:

```
REPORT_PERFORMANCE

Clock period:
   10 ns (set_params -clock_freq 100)


Loop Performance:
   Loop: PA0:
      Schedule length: 7 (set_loop_params 0 -max_schedule_length 7)
      II: 1 (set_loop_params 0 -ii 1)
...
TCAB Performance:
   Pipelinability: 1
      (set_params -architectural_pipelinability 1)
   Loop: PA0:
      Schedule length: 8
...
Memory Architectures:
   Array : breg
```

```
Memory : ramff_breg
   Mapping : sramff    ( #pragma internal_fast breg (present) )
   Packing : yes    ( #pragma packed breg )
```

The performance report includes the following sections:

- Clock Period
- Loop Performance
- TCAB Performance
- Memory Architectures
- Perfectization
- FIFO Depths
- Hierarchical Blocks
- Constraint Report
- Operation Schedules
- Interface
- Host Address Map
- Interface Timing

In a hierarchical design, one report file is generated for each PPA and TCAB implementation.

When porting an implementation to a new release of Synphony C Compiler, the performance characteristics reported in the `report_performance.txt` file can be used as guidelines for specifying the constraints in the new release. It is not required to specify *all* the constraints shown in the report. You can use selected constraints to set the properties that are required to be consistent across multiple releases.

The various sections of the report are explained in the following subsections.

Clock Period

The Clock Period section shows the clock period used for constraining the design and the implementation parameter that can be used for constraining the frequency at which the RTL was generated. For example,

```
Clock period:
   10 ns (set_params -clock_freq 100)
```

Loop Performance

The Loop Performance section shows the different loops in the design and the parameters that affect the performance of the loop, schedule length and iteration interval. For example,

```
Loop Performance:
    Loop: PA0:
       Schedule length: 2 (set_loop_params 0 -max_schedule_length 2)
       II: 1 (set_loop_params 0 -ii 1)
```

You can use these parameters to constrain the design across different releases of the tool so that the performance does not degrade.

TCAB Performance

For TCAB implementations, the TCAB Performance section shows the performance parameters that can be used for constraining the TCAB. For example,

```
TCAB Performance:
   Pipelinability: 1 (set_params -architectural_pipelinability 1)
   Loop: PA0:
      Schedule length: 8
      II: 1
   I/O times:
      Source Name: invec.re
         Sample cycle: 0 (set_params -resource_first_use_time
          invec:0)
      Source Name: invec.im
         Sample cycle: 0
      Source Name: d_set_tap0
         Sample cycle: 3
      Source Name: DPD.re
         Production cycle: 7
         (set_params -resource_last_use_time DPD:7)
      Source Name: DPD.im
         Production cycle: 7
   Resource use:
      Source Name: sigdly
         Architectural first use cycle: 3
...
```

Memory Architectures

The Memory Architectures section shows the C/C++ arrays in the design and the parameters that can be used for constraining these arrays. Some arrays could be scalarized, synthesized as internal fast memory, or could be user-supplied.

This section shows the different memories in the design and the constraints that could be applied for keeping the properties same across different releases. For example,

```
Memory Architectures:
   Array : LUTCoef0
      Memory : No RTL-Name for user supplied memory
         Mapping : sram_artisan   ( #pragma user_supplied LUTCoef0 (present) )
         Packing : yes   ( #pragma packed LUTCoef0 )
```

```
        Banking : no
        Scalarize : no   ( #pragma preserve_array LUTCoef0 (present) )
        Multi-buffering : 1   ( #pragma multi_buffer LUTCoef0 1 (present) )
        Porting : 1   ( #pragma read_write_ports LUTCoef0 separate 1 readonly  )
        Arbitration : no   ( #pragma read_write_ports LUTCoef0 no_arbitration
       (present) )
   ...
   Array : lut_sig[0].im
   Scalarize : yes

   Array : lut_sig[0].re
   Scalarize : yes
```

Perfectization

The Perfectization section shows the region of the code that is perfectized by Synphony C Compiler. In case of SCC-generated loops, you can wrap the code into a single iteration loop if you need to keep the same perfectization choice across different releases.

Here is an example:

```
Perfectization:
   Straight Line Code Region :
      Start :
         Label : <labelname>
         Location : src/src.cpp:292:0
      End :
         Label : <labelname>
         Location : src/src.cpp:292:0
      Loop : PA0
```

FIFO Depths

The FIFO Depths section shows the various FIFOs in the design and the depth of each one. If a FIFO is used for passing resources between PAs, the report shows the corresponding resources. For example,

```
FIFO Depths:
    Name: filter_ils_fifo_filter_inter_len
       Source: PA0
       Destination: PA1
       Depth: 0 (`define filter_ils_fifo_filter_inter_len 0)
       Resources: NA
```

Hierarchical Blocks

The Hierarchical Blocks section shows information about blocks that are imported into the current implementation. It shows the path to the corresponding `report_performance.txt` file of each imported block. For example,

```
Hierarchical blocks:
   TCAB: Filter
      Implementation: filter_tcab
      Performance report: mainline/filter_tcab/Reports/
report_performance.txt
      Container: PA0
   TCAB: TappedDelayLine
      Implementation: tdl_tcab
      Performance report: mainline/tdl_tcab/Reports/
report_performance.txt
      Container: PA0
```

Constraint Report

The Constraint Report section shows the constraints that Synphony C Compiler ignored because the tool performed some kind of optimization that removed the operations associated with the constraint. If no constraints were ignored, the report shows `NONE`:

```
Constraint report: NONE
```

Operation Schedules

The Operation Schedules section shows scheduling of operations that could be constrained if you require some operations to be scheduled in fixed cycles across different releases. You can apply scheduling constraints on the resource accesses by using the commands in this section.

Here is an example:

```
Operation Schedules:
Loop : PA_0
  Source Name : filter::inter
    Type : Stream Output
       Location : file=fir.cpp line=77 col=27
          Cycle : 1   ( set_params -wait "<labelname>:1" )

   Source Name : filter::yin:input
     Type : Stream Input
       Location : file=fir.cpp line=63 col=23
         Cycle : 0   ( set_params -wait "<labelname>:0" )
       Location : file=fir.cpp line=72 col=36
         Cycle : 0   ( set_params -wait "<labelname>:0" )


Loop : PA_1
  Source Name : filter::inter
     Type : Stream Input
        Location : file=fir.cpp line=113 col=27
          Cycle : 0   ( set_params -wait "<labelname>:0" )
```

```
            Location : file=fir.cpp line=122 col=40
              Cycle : 0   ( set_params -wait "<labelname>:0" )

      Source Name : g
         Type : Memory Read
            Location : file=fir.cpp line=100 col=16
              Cycle : 0   ( set_params -wait
                          "<labelname>\[0\]\[0\]\[0\]\[0\]:0" )
         Type : Memory Write
            Location : file=fir.cpp line=144 col=8
               Cycle : 2   ( set_params -wait
                          "<labelname>\[3\]\[1\]:2" )
```

Interface

The Interface section gives information about the interfaces and the constraints that could be applied at the interface of the RTL if you require these to be fixed across releases of the Synphony C Compiler tool. For example,

```
Interface:
   System Ports :
      clock port :
            Portname : clk

      reset port :
            Portname : reset
            Level : low ( set_params -reset_level low )
            Sensitivity : async ( set_params -reset_sensitivity async )

      stall_container port :
            Portname : stallbar_out
      start port :
            Portname : start
      ...
   Non-System Ports:
      addr port :
            Portname : ram_lutcoef0_a0 ( #pragma signal_name_prefix LUTCoef0:addr "",
     #pragma signal_name_identifier LUTCoef0:addr ram_lutcoef0_a0, #pragma
     signal_name_suffix LUTCoef0:addr "" )

      cen port :
            Portname : ram_lutcoef0_cen0 ( #pragma signal_name_prefix LUTCoef0:req "",
     #pragma signal_name_identifier LUTCoef0:req ram_lutcoef0_cen0, #pragma
     signal_name_suffix LUTCoef0:req "" )
     ...
```

Host Address Map

The Host Address Map section provides information about the host address map and the path to the `report_address_map.txt` file. For example,

```
Host Address Map:
```

```
  ( set_params -host_address_map <filename> )
      Copy imp_fir/Reports/report_address_map.txt to <filename>
    Symbol : PICO_PSW   Start Address : 0
    Symbol : cols   Start Address : 4
    Symbol : rows   Start Address : 8
```

Interface Timing

The Interface Timing section shows the timing delays at the interface of the design. For example,

```
Interface timing:
   Loop : PA0
         Source Name: invec.re
         RTL port: scalar_in_invec_re_data
            Input delay : 0.356729
         Source Name: invec.im
         RTL port: scalar_in_invec_im_data
            Input delay : 0.356729
         Source Name: DPD.re
         RTL port:
            Output delay : 0.623406
            ( set_params -scalar_output_delay
              DPD:0.623406)
         Source Name: DPD.im
         RTL port:
            Output delay : 0.616410
```

report_verify_rtl_performance.txt

The `verify -rtl` command provides a System Verilog simulation of the driver code using a Universal Verification Methodology (UVM) testbench, with the PPA running in RTL. This is a very accurate simulation of full hardware operation and execution with task overlap. The output shows a list of loop start/stop events:

```
Task 0
=======
Stall Count for task 0 : 310
Loop0, task 0 started at cycle 6
Loop0, task 0 stopped at cycle 328
Stall Count for Loop 0 : 0
Loop1, task 0 started at cycle 7
Loop1, task 0 stopped at cycle 329
Stall Count for Loop 1 : 0
Loop2, task 0 started at cycle 8
Loop2, task 0 stopped at cycle 343
Stall Count for Loop 2 : 310
State for FIFO = app_ppa. _serstrm:
```

```
full: 1; empty: 1; entries left: 0; full_stalls: 0; empty_stalls: 0
State for FIFO = app_ppa. _diffdata:
full: 1; empty: 1; entries left: 0; full_stalls: 0; empty_stalls: 310
```

Support for Performance Charting

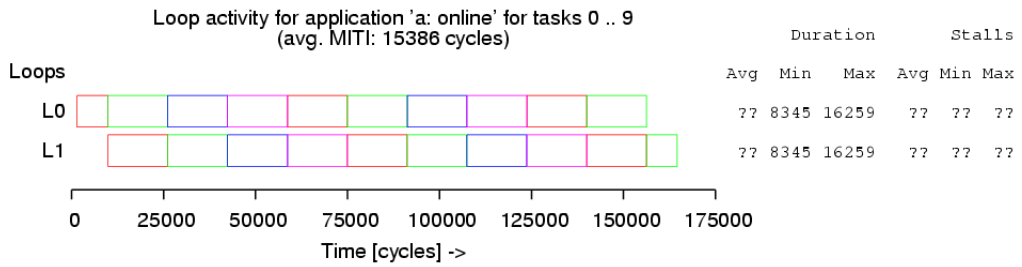
A Tcl command is available to visualize the output of the `verify` command and scthread performance reports. Use the following Tcl command to display the report:

```
report_implementation -performance
```

To see the report for a specific task or a range of tasks, you may use the tasks option, with a single task number (-tasks n) or with the desired range (-tasks m-n).

This command creates output in .PostScript or .PDF format showing the successive tasks running through the loop nests in the designs, as shown in the following figure.

See the following figure for an example of the performance output.



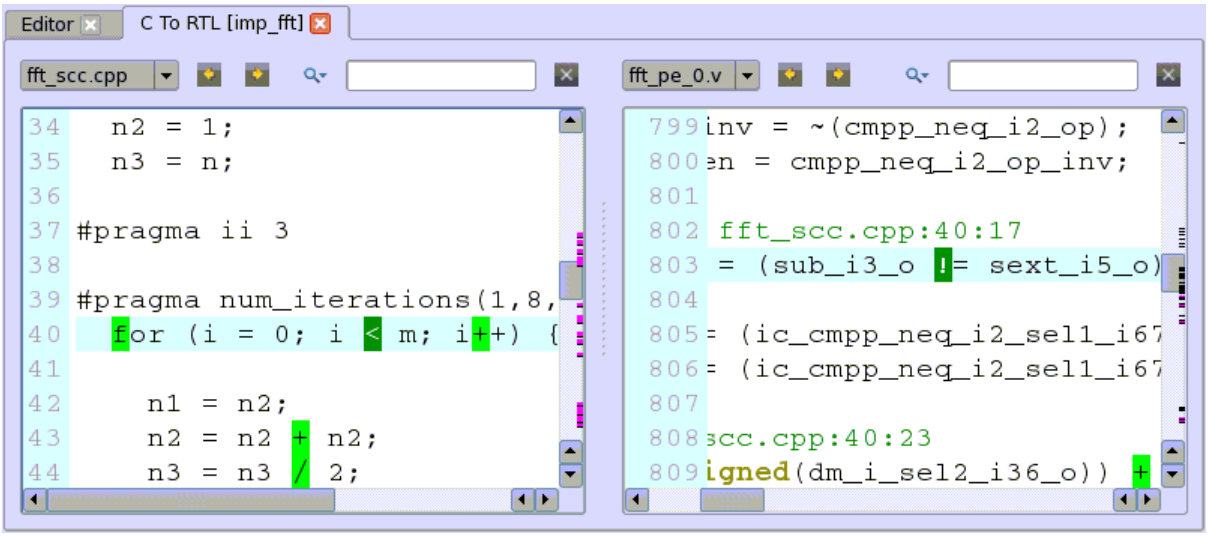
Each task is color-coded. The number of colors is chosen according to the maximum task overlap achieved in the run; so that successive tasks may be uniquely distinguished (no overlapping tasks will use the same colors).

If the system has a PDF viewer installed (such as xpdf or acroread), Synphony C Compiler tool generates PDF, otherwise it generates PostScript.

C to RTL Mapping

The C to RTL Mapping feature lets you compare the generated RTL code against the original C source code. Launch the window using Analysis > C to RTL Mapping. It displays a split window with the two files, where you can view the C to RTL mapping.

C File RTL File



You can navigate and cross check the C code and RTL description in several ways. For details, see “C to RTL Mapping” on page 2-67.

Power Estimation Reports

The following reports provide information on power (consumption):

- report_power_theater_offline.txt
- report_power_theater_sv.txt
- report_spyglass_offline.txt

report_power_theater_offline.txt

```
Power contribution                      Static  Dynamic    Total
----- ------------                      ------  -------    -----
Internal power
   Internal register power              75.7uW    744uW    820uW
   Internal latch power                     0W       0W       0W
   Internal memory power                    0W       0W       0W
   Other internal power                  516uW    983uW    1.5mW
   Total internal power                  592uW   1.73mW   2.32mW
IP Core power                               0W       0W       0W
Pad power                                   0W       0W       0W
Clock power                                 0W       0W       0W

Total power                              592uW   1.73mW   2.32mW
```

report_power_theater_sv.txt

```
                                              Power(Watts)
Power contribution                      Static  Dynamic    Total
----- ------------                      ------  -------    -----
Internal power
   Internal register power              75.6uW   1.38mW   1.45mW
   Internal latch power                     0W       0W       0W
   Internal memory power                    0W       0W       0W
   Other internal power                 524uW   4.81mW   5.34mW
   Total internal power                 599uW   6.19mW   6.79mW
IP Core power                               0W       0W       0W
Pad power                                   0W       0W       0W
Clock power                                 0W       0W       0W
Total power                              599uW   6.19mW   6.79mW
```

report_spyglass_offline.txt

```
report_spyglass_offline.txt
----------------------------------------------------------------------
```

```
Generated by rule           : PEPWR02
VCD file                    : /mnt/regression/px_8020/test/systolic/
                              verysimple/a2/rtl_package/dump_sg.vcd
VCD top name(inferred)      : verysimple_testbench.dut.ppa_0
Libraries                   : tcbn90ghpwc
Power Unit                  : Watts
Operating voltage           : 0.900000
Start Time                  : 0ps
End Time                    : 1420000ps
Simulation Time Window      : 1420000ps
Percentage of RTL nets
 not set from VCD/FSDB file : 0.42%
Percentage of design nets
with capacitance set using
  (a) wire load             : 0.00%
  (b) SPEF/set_load(SDC)    : 0.00%
-----------------------------------------------------------------------

-----------------------------------------------------------------------
Report Format               :
  module                            Leakage  Internal  Switching  Total
-------------------------------------------------------------------------
verysimple_ppa                      22.7uW    187uW    6.12uW     216uW

verysimple_ppa.paw_1
   (verysimple_paw_1)               8.61uW    80.4uW   2.67uW     91.7uW

verysimple_ppa.paw_1.pa
   (verysimple_pa_1)                8.38uW    79.5uW   2.65uW     90.5uW

verysimple_ppa.paw_1.pa.pe_0
   (verysimple_pe_1)                8.38uW    79.5uW   2.65uW     90.5uW

verysimple_ppa.paw_1.pa.pe_0.
tstrw_s2_noreset_0
(WORK_ststr_sx_noreset_32_32_2)     3.00uW    30.3uW   227nW      33.5uW

verysimple_ppa.paw_1.pa.pe_0.
ldstrw_s1_noreset_1
(WORK_ldstr_sx_noreset_32_32_1)     2.28uW    29.9uW   254nW      32.4uW

verysimple_ppa.reg_6_basic
 (sregn_noinit)                     103nW     888nW    7.09nW     998nW

verysimple_ppa.reg_5_basic
 (sregn_noinit)                     103nW     888nW    7.09nW     998nW

----------------------------- Clock Power --------------------------
Total Clock Power         Leakage    Internal   Switching  Total
                          0W         0W         0W         0W
-----------------------------------------------------------------------
Detailed Report :
Net       Frequency Registers Buff/Inv Leakage Internal Switching Total
```

```
            [MHz]    Count     Count    Power    Power    Power   Power
Verysimple_
ppa.clk      100.00   202        0        0W       0W       0W      0W
-----------------------------------------------------------------------

--------------------------- Memory Power ----------------------------
Total Memory Power          Leakage    Internal   Switching    Total
                             0W         0W         0W           0W
-----------------------------------------------------------------------
Total power: 216uW (Leakage=22.7uW, Internal=187uW, Switching=6.12uW)
```

Verilog Simulation Analysis Reports

The following report provides information on memory aliasing in the design:

- report_memory_access_trace.txt

report_memory_access_trace.txt

This report is used when analyzing any memory aliasing in the design. The Synphony C Compiler tool does memory access analysis but, in some cases, this may be too conservative. You can turn off memory access analysis for a particular array by using the "no_dependence Pragma" on page 7-37. If, during verification, it turns out that there is a failure, then the information in this report may be used to find out if there is, in fact, a memory access aliasing for an array for which the analysis had been disabled.

This report is produced during verify -rtl simulation phase, if the `-dump_memory_access_trace` option has been specified. After this, the Tcl command `report_implementation -memory_access_trace` displays this report. For more information on this command, see "report_implementation" on page 3-14.

A sample report is included. The first two columns show the memory access and the access which has been detected to alias with it. Each column gives the source file name and line number along with the time at which this operation got schedules and the hardware iteration count in which the memory access and the aliasing access occur.

The comments column indicates any memory aliasing access when the values in the accessed locations are the same.

```
-----------------------------------------------------------------------
Accesses Aliasing Accesses
( File:Line,unrolled_sched_time ( File:Line,unrolled_sched_time,
iteration count ) iteration count )
-----------------------------------------------------------------------
test.c:49 0 1 test.c:51 0 1
if(j==0) B[0]=10; output_scalar+=B[j]+B[j+1];
^ ^
test.c:50 3 1 test.c:51 0 2
B[j+1]= input1[j]+1; output_scalar+=B[j]+B[j+1];
```

14

System Integration and Verification

This chapter describes the various interfaces present in the Synphony C Compiler tool, their integration and verification. For more details, see the following:

- Overview of the Hardware Interface
- System Interface
- Handshake Interface
- Scalar Interface
- Stream Interfaces
- Local Memory Interfaces
- Direct Port Interface to Memories in PPA
- Memory-Mapped PPA Components
- Host Interface Bus
- Memory-Mapped PPA Control (PSW)
- Memory-Mapped Scalar Interface
- PPA Interface Port Naming and Polarity
- Interface Timing
- Deployment of Memory-Mapped PPA

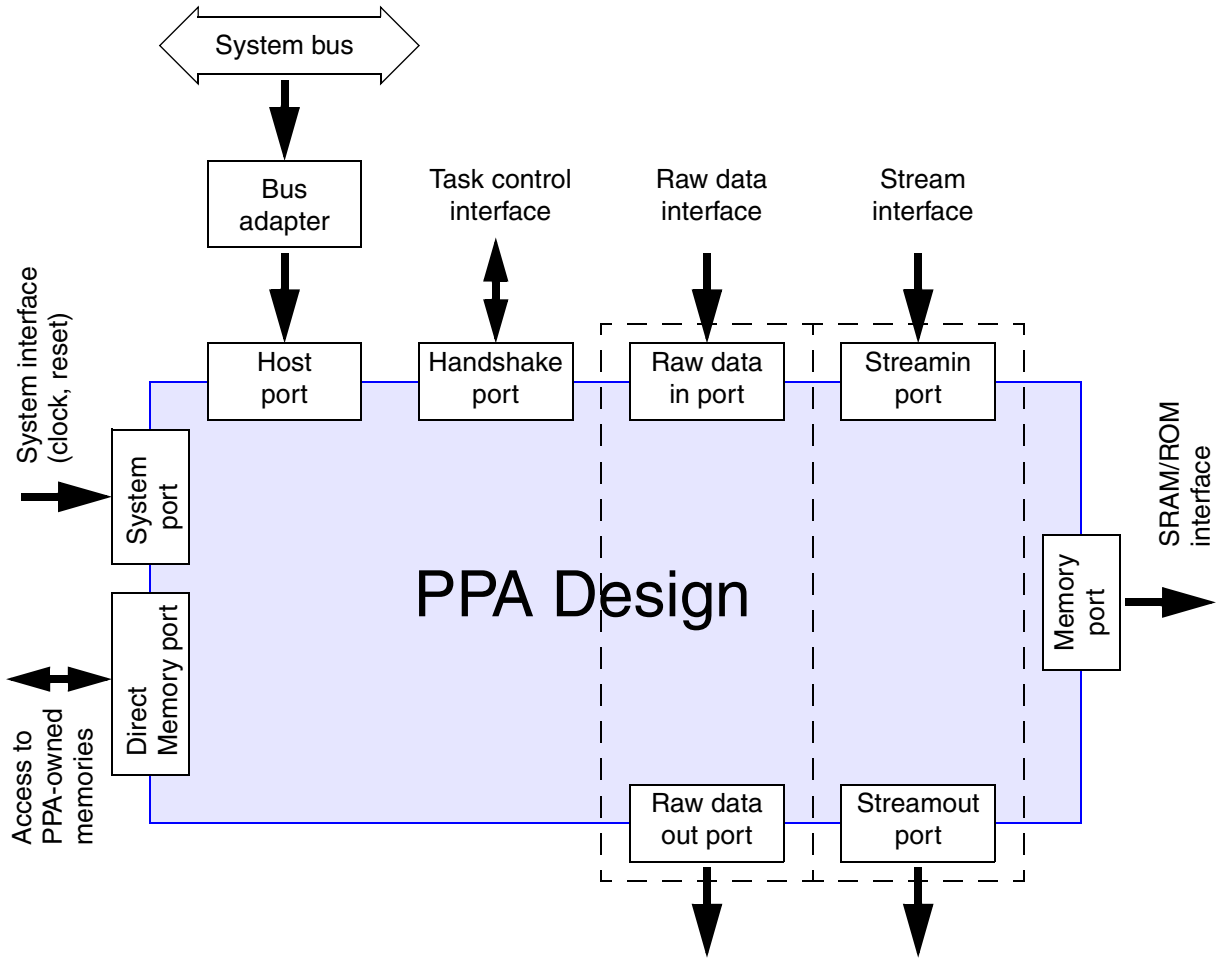
- Pragmas for EDK support (FPGA Only)
- TCAB Integration
- Verification
- SystemVerilog (SV) Universal Verification Methodology

Overview of the Hardware Interface

The Synphony C Compiler standard interfaces control the behavior of the synthesized hardware and for exchange of data across its boundaries. This chapter describes the different kinds of interfaces and their protocols that are generated at the boundary of the synthesized hardware.

The different hardware interfaces that are possible to be generated at the PPA interface are described in the following diagram. The Synphony C Compiler tool automatically generates only those interfaces that you specify. The following figure shows the PPA interface.

Figure 14-1 PPA Interface



This chapter summarizes the hardware interface for the PPA artifact. The categories of signals are:

- System Interface: This interface contains the basic signals like clock, clock enables, and reset.
- Task control interface: This is the handshake control interface for PPA task control.
- Local memory interfaces that connect to zero or more external SRAMs.
- Stream interfaces that connect to zero or more external input/output streams.
- Scalar Interface: These are ports for the external hardware to access the PPA's input and output scalars.
- Direct memory interface: These are ports for the external hardware to access the PPA's memories.
- Host Interface: This is the memory mapped port for the host to access PPA's control/memory/scalars in a memory mapped way.

The default interfaces that the Synphony C Compiler tool provides are the following:

- Direct ports for host-accessible memories, one per memory. A direct port is a combination of address bus, data buses, and enables.
- Raw ports for scalars.
- Handshake interface.

System Interface

This interface contains the clock and reset inputs in the hardware. In addition, the clock gating signals are also part of this interface. For more information on clock gating signals, see "Clock Gating (ASIC Only)" on page 15-19.

The polarity of reset can be controlled using `set_params -reset_level` and the polarity of clock signals can be controlled using `set_params -edge_clock`.

Clock and reset ports can be given specific names as described in "PPA Interface Port Naming and Polarity" on page 14-24.

Handshake Interface

The default interface of a PPA is a *handshake* interface for task start and done controls. In addition, the interface includes optional ports for enable, abort, and error. Per-task scalar inputs are synchronized with start handshake, and per-task scalar outputs are synchronized with done handshake, respectively.

This section explains the handshake interface with raw scalar inputs and outputs.

Advanced scalar interfaces such as `stable` scalars, hold registers, packed scalar interface, and memory-mapped scalars are explained the scalar interface section.

The following figure shows the handshake interface ports on the PPA.

Figure 14-2 PPA Handshake Interface Ports

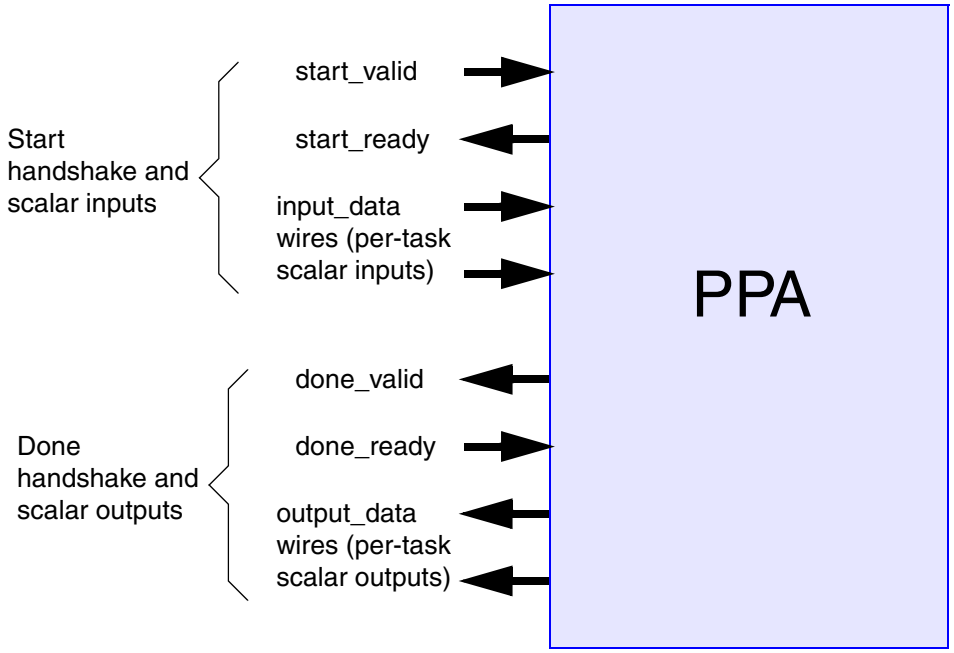


Table 14-1 describes the interface ports.

Table 14-1 Handshake Interface Ports

Interface	RTL Ports	Description
Start handshake	`start_valid`	`start_valid` is an active-high input signal
	`start_ready`	`start_ready` is an active-high output signal
	Scalar input data wires (optional)	The external hardware and the PPA communicate via a valid-ready protocol. Scalar inputs to a task are placed on the input data wires and `start_valid` is asserted by the external hardware. The PPA signals `start_ready` when it is ready to consume inputs and start a task. The cycle where `start_valid` and `start_ready` are high is the point at which the handshake is successful; the external hardware can change the data on the input wires on the next clock cycle.
Done handshake	`done_valid`	`done_valid` is an active-high output signal
	`done_ready`	`done_ready` is an active-high input signal
	Scalar output data wires (optional)	The PPA and the external hardware communicate via a valid-ready protocol. The scalar outputs from a task are placed on the output data wires and `done_valid` is asserted by the PPA. The external hardware asserts `done_ready` after it consumes the outputs. The cycle where `done_valid` and `done_ready` are high is the point at which the handshake is successful; the PPA can change the data on the output wires on the next clock cycle.
Other interface ports	enable	Input signal to enable or disable the PPA combinationally.
	abort	Active-high input pulse to abort the execution of the PPA. The abort takes effect combinationally in the cycle it is asserted and the PPA is ready to start a new task from the next cycle. This port exists only if you specified it using the Tcl command `set_params -generate_abort_logic yes`
	err	Active-high output to signal a PPA execution error. The external hardware must abort or reset the PPA to restart a fresh task. This port exists only if you specified it using the Tcl command `setvar -a synthesize_auxopts "-Fcreate_error_hw=yes"`

Polarities of any of the control ports can be specified explicitly by using the `set_params -system_port_polarity ...` command.

The PPA handshake interface can have specialized or simple integration to the external hardware in certain scenarios. If the external hardware needs to have the PPA start and drain tasks without its intervention, start_valid and done_ready inputs can be tied to 1. Such designs should not have per-task inputs and outputs.

The following diagrams show the timing for the handshake interface and for the enable, abort, and error events.

Figure 14-3 PPA Handshake Interface Timing

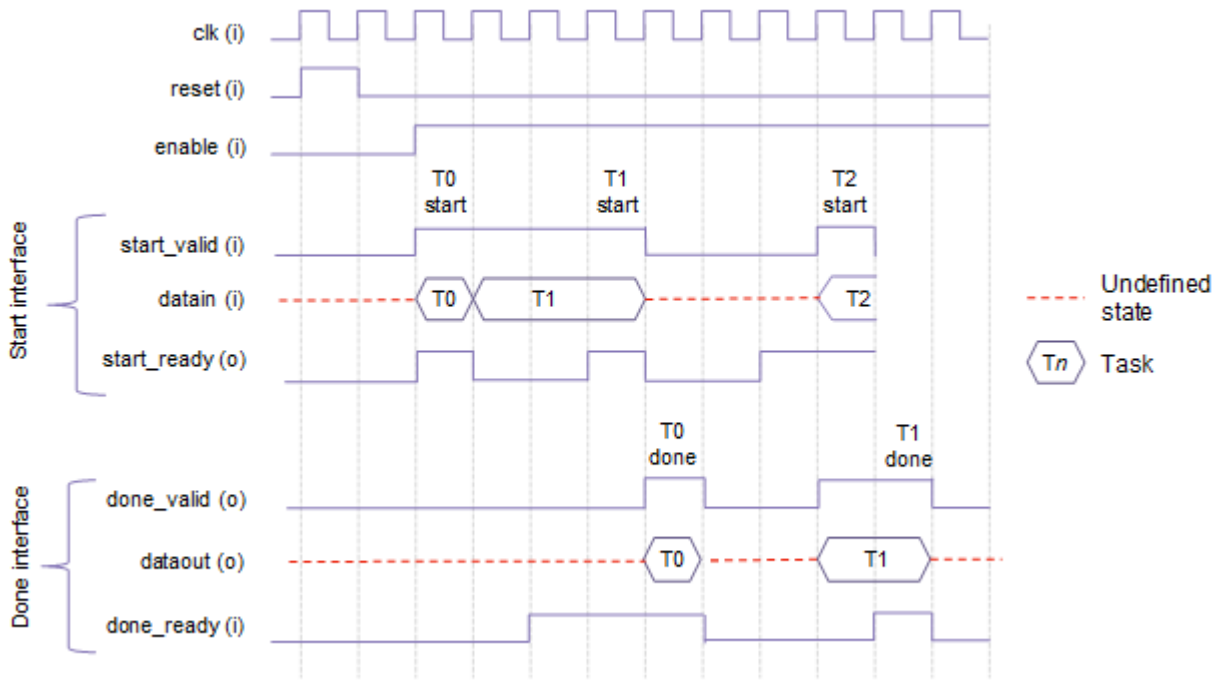
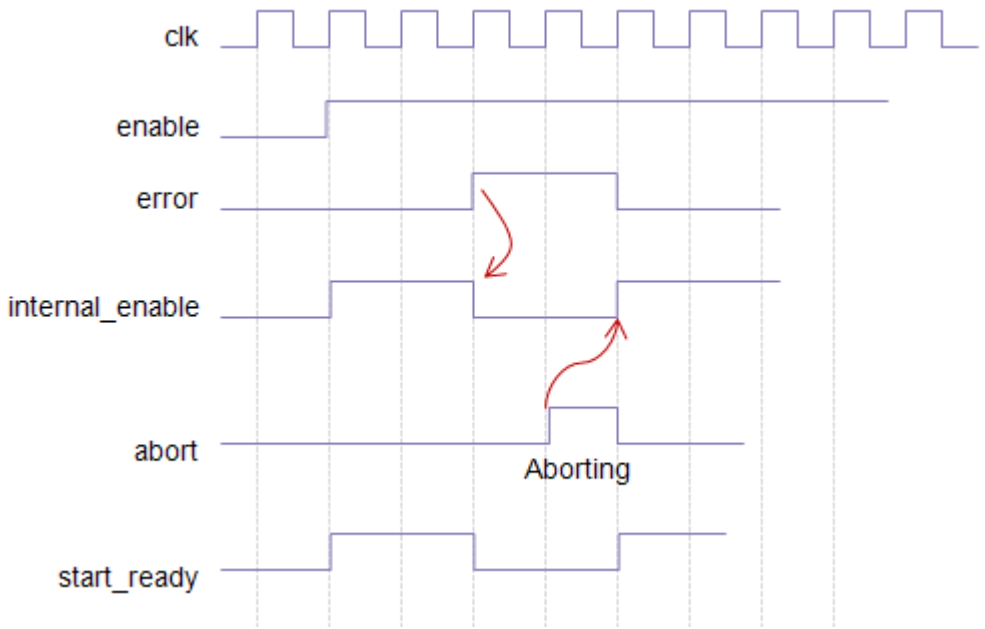


Figure 14-4 PPA Enable, Abort, and Error Interface Timing



Scalar Interface

Scalars in the application program are implemented as registers in the PPA. By default, Synphony C Compiler creates raw read and write interfaces for the external hardware to access the registers of externally visible scalars. For information on how to generate or restrict creation of these raw ports, see the Chapter 7, "Pragmas and Constraints."

There are two types of scalar inputs. Scalar inputs that are designated as constant by `#pragma stable ...` are assumed to be constant during entire execution of a pipelined set of tasks. Other per-task input scalars can change every task.

Both constant and per-task scalar inputs can be created with hold registers for data persistence. A scalar input with a hold register has a hold register enable for each data input. In addition, it allows the external hardware to compute next inputs while the PPA is still processing the previous inputs, thus allowing the external hardware and PPA overlap.

Scalar outputs operate per-task. Like scalar inputs, they can be created with hold registers. A hold register allows the PPA to compute next outputs when external hardware is still processing the previous outputs, thus allowing the external hardware and PPA overlap.

For a description of the scalar input and output interface behavior with and without hold registers, see Table 14-2.

Table 14-2 External Hardware and PPA Task Behavior, Raw Input/Output

Input/output type	Hold reg value	Behavior
stable scalar input	never (default), single_buffer	never: External hardware can't change input wire until pipelined execution completes. single_buffer: External hardware can change input wire after hold reg write, but cannot write to hold register until pipelined execution completes.
Per-task scalar input	never (default), single_buffer	never: On start handshake of Task(i), external hardware can change the input wire to Task(i+1). single_buffer: On start handshake of Task(i), external hardware can write Task(i+1) input to the hold reg and can change the input wire to Task(i+2).
Per-task scalar output	never (default), single_buffer	never: On done handshake of Task(i), PPA can compute the outputs of Task(i+1). single_buffer: On done handshake of Task(i), PPA can write Task(i+1) output to the hold reg and can compute the outputs of Task(i+2).

You can specify a packed scalar object of C struct type by using the following pragma:

```
#pragma packed struct_scalar_obj yes
```

In this case, scalar inputs and outputs that are the fields of the struct scalar object are created as a packed data ports containing the fields. For details, see "packed Pragma" on page 7-53.

Stream Interfaces

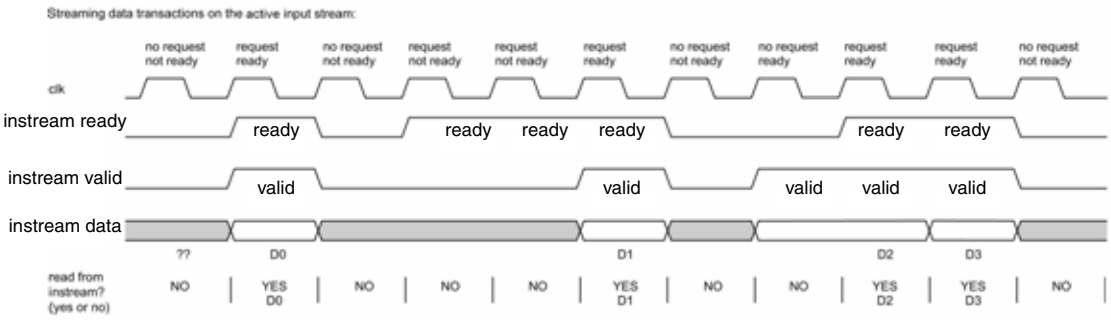
The PPA exports a three-signal handshake interface with each of the external input or output stream. The stream interface is explained in the following subsections.

Input Stream Interface

The following signals are used in the input stream interface:

- Instream ready – Request for data from PPA to stream. Active high.
- Instream valid – Data valid signal from stream to PPA. Active high.
- Instream data – Data input from stream to PPA.

The following figure shows the stream data transactions on an active input stream.

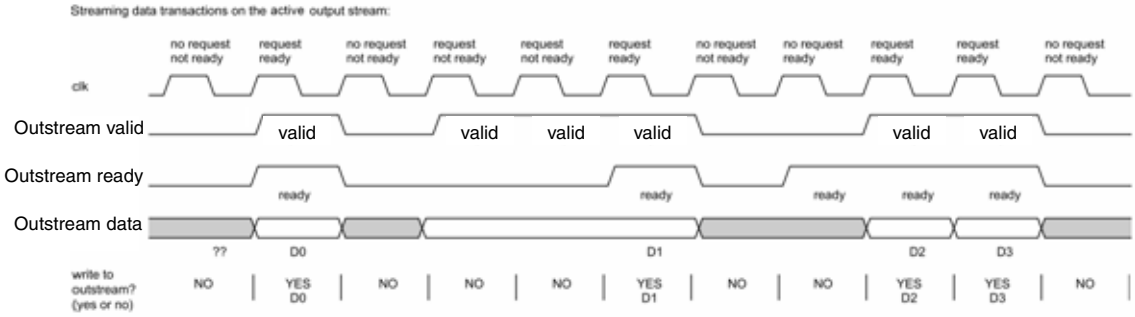


Output Stream Interface

The following signals are used in the input stream interface:

- Outstream valid– Data valid from PPA to stream. Active high.
- Outstream ready – Ready signal from stream to PPA. Active high.
- Outstream data – Data output to stream from PPA.

The following figure shows the stream data transactions on active output stream.



Stream Interface Customization

The valid, ready, and data ports can be eliminated on the PPA interface by using:

```
#pragma signal_production
```

For more information about this pragma, see "signal_production Pragma" on page 7-67.

The valid and ready signals are active-high by default. The polarity can be specified by using either of the following:

```
#pragma signal_polarity ...
set_params -stream_signal_polarity ...
```

By default, a stream created from C struct data type contains one packed data port. To create one data port per struct field, use the following pragma:

```
#pragma packed ...
```

Stream Interface ports can be assigned specific names. For details, see "PPA Interface Port Naming and Polarity" on page 14-24.

You can specify a passive or active stream interface on the PPA by using the following pragma:

```
#pragma control_mode ...
```

The stream data output is not guaranteed to be stable after the handshake. If such behavior is required on the PPA interface, use the following:

```
#pragma hold_register single_buffer
```

In such a case, the output stream data will be held stable by the PPA even after handshake success, but only until the next stream write by the PPA.

Stream interfaces can be specified to honor certain external delay constraints and to contain a boundary FIFO. Refer to the descriptions of pragmas and implementation parameters related to stream boundary registers and external delays in Chapter 7, “Pragmas and Constraints.”

Local Memory Interfaces

If the PPA addresses local memories (RAMs and ROMs) that are not internally synthesized (as flip-flops and appropriate logic), then the Synphony C Compiler tool exports a local memory signal interface for each such memory. These are usually implemented using RAM and ROM generators. For more information on SRAMs and ROMs control, see the Chapter 7, “Pragmas and Constraints.

For SRAMs as well as ROMs, the Synphony C Compiler tool emits a standard Artisan interface. The address and the data bus are parameterized by address and data widths respectively. The ports or signals may or may not be registered at the boundary of the PPA based on the assumptions or specifications of the boundary register requirements and the external delay constraints. Refer to the descriptions of pragmas and implementation parameters related to memory path boundary registers and external delays in Chapter 7, “Pragmas and Constraints.”

By default, the Synphony C Compiler tool assumes single cycle synchronous SRAM. You can specify memory access latency (intrinsic) using memory_access_latency implementation parameter or pragma. The actual latency will be (intrinsic_latency + 1) if the boundary registered on either forward or return path and not both, and will be (intrinsic_latency + 2) if boundary registered on both forward and return paths. The memory is assumed to be pipelined and intrinsic latency has to be >=1.

For SRAM timing, consult datasheets for Artisan™ or Virage™ synchronous SRAM generators.

Note:

For more information on packed memory interfaces, see “packed Pragma” on page 7-53.

Direct Port Interface to Memories in PPA

Synphony C Compiler creates some memories inside the PPA. Other memories are still owned by PPA, but must be created physically outside the PPA. For the external hardware, PPA provides direct access to such memories via specially created memory ports called *direct memory ports*.

These are the direct port interface signals for one memory:

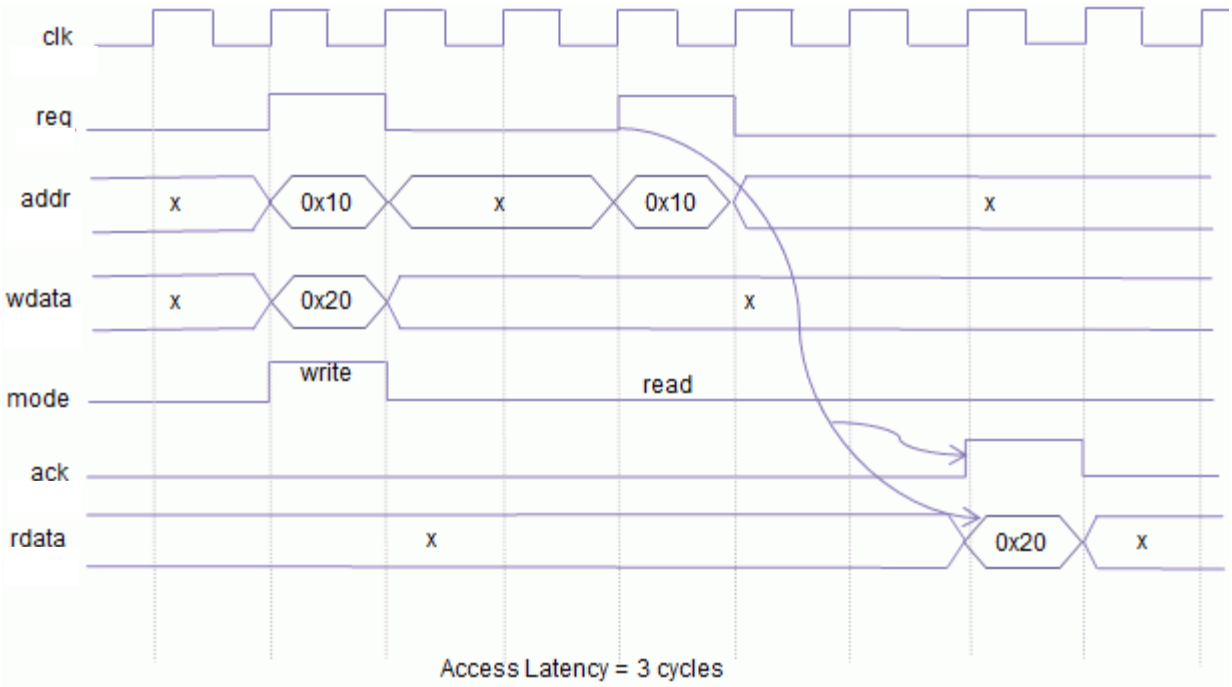
- `direct_memtype_cname_addr` – Memory address input to the PPA
- `direct_memtype_cname_wdata` – Memory data input to the PPA
- `direct_memtype_cname_rdata` – Memory data output the PPA
- `direct_memtype_cname_req` – Memory request into the PPA
- `direct_memtype_cname_ack` – Memory acknowledge for data read from PPA
- `direct_memtype_cname_mode` – Input to the PPA; 1=write, 0=read

If read-only or write-only accesses are specified, then only the ports needed for read-only or write-only access are created on the PPA.

The direct accesses can be pipelined. Latency of this direct access is same as the memory latency and can be obtained from the `report_implementation -summary` command.

Figure 14-5 shows the read and write timing using direct memory ports.

Figure 14-5 Memory Access Timing Using Direct Memory Ports



Memory-Mapped PPA Components

PPA has the following components that can be memory-mapped for access through host ports:

- Processor Status Word (PSW) – A 32-bit word reserved in PPA address space to memory-map the PPA control signals: handshake, abort, error, and enable signals. For details, see "Memory-Mapped PPA Control (PSW)" on page 14-19.
- Scalars – These are scalar inputs and outputs of the PPA. Memory-mapped scalar inputs and outputs are created with a hold register by default.
- `stable` scalars – These are input scalars that are constant across pipelined set of tasks. When memory-mapped, they always have hold registers.
- Memories – These are memories used for communication between the host and the PPA.

  Note:
  Memories arising out of C struct types and streams cannot be memory-mapped.

Specification to Memory-Map a PPA Component

This section gives the implementation parameters and pragmas that can be used for specification of memory-mapped access to the PPA components.

- Memories:
  - `set_params -host_memory_access`
  - `#pragma host_memory_access`

  For more information on this Tcl command and pragma, see "host_memory_access Pragma" on page 7-30, respectively.

  Example: set_params -host_memory_access lm
- Processor Status Word (PSW):
  - `set_params -interface_type`

  Example:

  ```
  set_params -interface_type memory_mapped_handshake
  ```
- Scalars:
  - `set_params -host_scalar_access`
  - `#pragma host_scalar_access`

For more information about pragma, see "host_scalar_access Pragma" on page 7-31.

Example: set_params -host_scalar_access lm

Address Map Input

You need not specify addresses to the components, in which case, the Synphony C Compiler tool assigns internally computed addresses. If you have specific address assignment requirement, the Synphony C Compiler tool allows you to specify the addresses using a Tcl-based address map file. The following Tcl command enables you to specify an address map file to the Synphony C Compiler tool.

```
set_params -host_address_map fileName
```

The specified address map is cached in cache sub directory. The file may contain addresses to all or some components. The Synphony C Compiler tool assigns internally computed addresses to the components that have no user specification. The Synphony C Compiler tool uses a variable-length Huffman coding scheme to minimize decoding logic cost; the ordering of these addresses is based on size and alphabetical name and invariant with respect to program declaration order.

Address Map File

You can use the following simple Tcl-based format to specify the address for various addressable components.

```
set HOST_ADDRESS_MAP {
{ {ADDR <addr>} {SYMBOL SCC_PSW} }
{ {ADDR <addr>} {SYMBOL
<scalar_input | scalar_output} }
{ {ADDR <base-addr>} {SYMBOL <array>} }
}
```

Example:

```
set HOST_ADDRESS_MAP {
{ {ADDR 0} {SYMBOL SCC_PSW} }
{ {ADDR 32} {SYMBOL input_1} }
{ {ADDR 36} {SYMBOL input_2} }
{ {ADDR 64} {SYMBOL arr_1} }
{ {ADDR 128} {SYMBOL arr_2} }
}
```

Note:

SCC_PSW needs four bytes.

Address Map Output

Reports/report_address_map.txt contains a text report of the final address map. In addition, it contains an executable Tcl code for the address map.

You can do valid modifications to this address map (in the Tcl section) if required and feedback it to Synphony C Compiler. The feedback process is as follows: Copy the address map file to `project` directory, edit it, specify this new file using `set_params -host_address_map` *`filename`* command, reset the implementation using reset_implementation command, and rerun the build flow.

Address Restrictions

Address decoding involves two steps. First, to decide which component is being addressed and then to send request to that component with correct address offset (in case of memories). Objective is to keep both decoding and address extraction logic simple and therefore, the Synphony C Compiler tool puts some restrictions on the addresses.

Consider an item j whose address ADDR being assigned falls between address of two already assigned items i and k.

ADDR is valid if both the following are true:

1. ADDR is exactly divisible by $2^{Bits(j)}$.

In other words, Bits[j] LSBs of the ADDR are all 0.

1. $addr(i) + 2^{Bits(i)} <= ADDR <= addr(k) - 2^{Bits(j)}$.

Condition (1) ensures that Bits[j] LSBs of the address can be directly used as the component address without requiring adders/subtractors for address decoding. Condition (2) ensures that prefix-MSBs of a component are not subset of another component's prefix-MSBs. This is a correctness requirement ensuring no overlap in addresses of different components.

Consider an example:

Item	Datatype*	Num_elements	Num_bits#
Input a	char(1)	1	0
Input b	long long(8)	1	3
Input c	int (4)	1	2
PSW	long long (8)	1	3

Item	Datatype*	Num_elements	Num_bits#
Mem x	short (2)	10	5
Mem y	char(1)	128	7

One valid assignment is: a = 0, b = 8, c = 16, PSW=24, x = 96 (32, 64 are also possible), y = 128.

Note:

If host_memory_access is always, then even the PPA local scoped memories get host access. However, the address specification is not allowed if there exist memories with same name in both local and global scopes.

Holes in the Address Map

A hole in the address map is an area that is either of the following:

- does not resolve to any addressable component
- resolves to an addressable component, but is out-of-bounds, for example, address >=10 supplied to a 10-element memory

If a host read address resolves to a hole in the address map, the host will still get the acknowledgement. The latency of the acknowledgment depends on hole position. It is recommended to use `set_params -hostlm_access_latency uniform` to make the latency predictable. Otherwise, host has to ensure valid pipelining of read requests. The data returned for a hole read is non-deterministic, although in certain special conditions, it is guaranteed to be 0.

A write to a hole can corrupt the PPA contents in certain cases and therefore is not recommended.

If the PPA is built for error hardware on (using `synthesize_auxopts -append "-Fcreate_error_hw=yes"`), then PPA signals an error whenever a read or write resolves to a hole in the address map.

Host Interface Bus

The Synphony C Compiler tool supports two types of interfaces:

- SCC native host interface
- OCP-compliant interface

The host interface data bus is always 32 bits wide. It can directly accommodate every C-datatype, except long long (64-bit) data, which can be handled through two separate transactions. The Synphony C Compiler tool provides APIs for host to transact with PPA. The host Interface address bus is as wide as required to address the PPA components.

Read and write transactions can be pipelined. If the PPA has been built with `hostlm_access_latency=minimal` (default), then read transactions to different latency components should be separated such that there are no return data collisions on the host interface bus.

Write transactions are fire-and-forget type, without any acknowledgement.

Read transactions return data after certain number of cycles. The `report_implementation -address_map` command shows the latencies of various memory-mapped components.

If PPA has been built with `hostlm_access_latency=minimal`, then scalar, PSW, internal_fast memory read responses are combinational. SRAM read response depends on the memory access latency.

If PPA has been built with `hostlm_access_latency=uniform`, then all latencies are equalized to the maximum latency component in the PPA.

SCC Native Host Interface

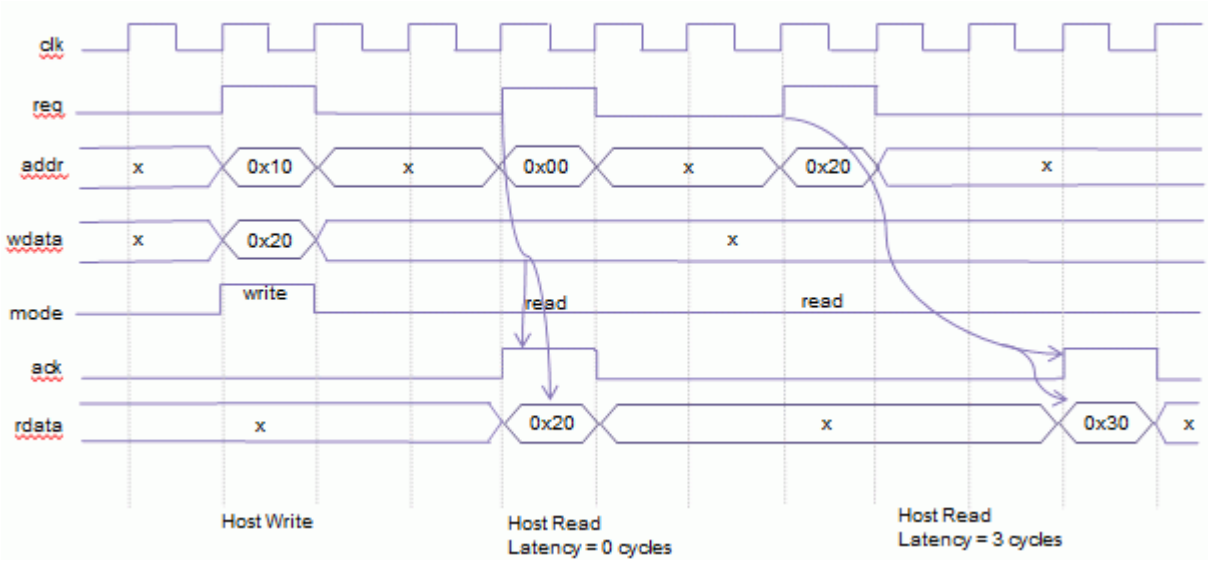
The following table shows the default host interface created by the Synphony C Compiler tool.

Table 14-3 SCC Native Host Interface Signals

Port name	Description	Property
host_req	Read or Write request	Input, Active high, pulse
host_addr	Address	Input
host_wdata	Write data	Input
host_rdata	Read data	Output
host_mode	Read or Write	Input, pulse, 1– Write, 0 – Read
host_ack	Read data valid	Output, pulse, Active high

Figure 14-6 shows the read and write access timing using the native host interface.

Figure 14-6 Memory Access Timing Using Native Host Interface



OCP Interface

Open Core Protocol (OCP) is a standard high-performance, memory-mapped interface that exists between IP core and on-chip communication subsystem. OCP supports high-performance data transfer models, and it is not dependent on the data transmission protocol that is connected to the CPU cores or buses, therefore it is widely used bus protocol in SoC design.

Memory mapped access to PPA commands and addressable components (like memory, registers) are provided through a host memory port. The Synphony C Compiler tool allows the host port to be OCP compliant. In this case, instead of the Synphony C Compiler native signals, PPA exposes the MAddr, MData, MCmd, SCmdAccept, SData, and SResp ports for the host access to the PPA. For the description of the protocol and the ports, see the OCP Reference Manual.

Memory-Mapped PPA Control (PSW)

The PPA Control interface can be memory-mapped by specifying the command:

```
set_params -interface_type memory_mapped_handshake
```

The Processor Status Word (PSW) is a 32-bit item reserved in the PPA address space for memory-mapping the handshake interface. The following figure shows the bit layout.

	31-8	7	6	5	4	3	2	1	0
Write	X	X	abort	DWE	WE	PDR	PSV	done_ready	start_valid
Read	X	error	X	RE	X	RPDR	RPSV	done_valid	start_ready

DWE: DataForWriteEnable, WE: WriteEnable, PDR: PersistentDoneReady, PSV: PersistentStartValid, RE: ReadEnable, RPDR: Read PersistenDoneReady, RPSV: Read PersistentStartValid
X: Don't care; writes have no effect and reads provide no information

The following table describes the behavior of each bit for writes and reads. See the handshake interface section for an explanation of the signals start_valid, done_ready, and so on. APIs are provided to do memory-mapped transactions from the host processor code.

Table 14-4 PSW Register Bit Uages

Bit	**Write**	**Read**	**Comments**
0 (start handshake)	1 – If bit 2 is 0, issue start_valid. If bit 2 is 1, issue persistent start_valid 0 – If bit 2 is 0, do nothing. Else, clear persistent start_valid.	1 – start_ready	Write – No effect for autostart designs Read – No information for autostart designs without status logic For PSV and PDR, see description later.
1 (done handshake)	1 – If bit 3 is 0, issue done_ready. If bit 3 is 1, issue persistent done_ready 0 – If bit 3 is 0, do nothing. Else, clear persistent done_ready	1 – done_valid	
2 (persistent start_valid or PSV)	1 – if bit 0 is 1, issue persistent start_valid. Else, clear persistent start_valid 0 – no effect	1 – PPA is in persistent start mode	
3 (persistent done_ready or PDR)	1 – if bit 1 is 1, issue persistent done_ready. Else, clear persistent done_ready 0 – no effect	1 – PPA is in persistent done mode	

Table 14-4 PSW Register Bit Uages

Bit	Write	Read	Comments
5-4 (enable)	11 – enable the PPA 01 – disable the PPA 00, 10 – do nothing	Bit 4 – No information. Bit 5 – 1 if PPA is enabled. 0 if PPA is disabled	Write – Ignored for always enabled PPA
6 (abort)	1 – abort the PPA	No information	Write – valid only if PPA is built with abort hardware
7 (error)	Ignored	1 – PPA has internal error	Read – usable only if PPA is built with error hardware
8-31 (unused)	Ignored	No information	

By default, the memory mapped PPA Control Interface creates raw output ports for `start_ready`, `done_valid`, and `err` (if the PPA is created with error hardware), which are the same as bit 0, 1, and 7 in the PSW, respectively. The host processor can query the `start_ready`, `done_valid`, and `err` bits via the PSW for busy-wait protocol or use these raw output ports as level-sensitive interrupts. You can use `set_params -signal_port_production` to remove these raw output ports if they are not needed.

Persistent start and done

For handshake interface, external hardware can wire start_valid to 1 to start a sequence of tasks. and done_ready to 1 to complete a sequence of tasks. For memory-mapped handshake interface, PSV and PDR bit help to achieve this purpose. PPA can be made to start a sequence of tasks without host interference by issuing PSV=1, start_valid=1; When host decides to stop the starts, it can issue PSV=1, start_valid=0; Similarly, PPA can be made to complete a sequence of tasks without host interference by issuing PDR=1, done_ready=1; When host decides to stop the completion, it can issue PDR=1, done_ready=0;

It is important to note that, tasks in the sequence should not have changing inputs or outputs during the sequence. Read of PSV and PDR is useful for the host to know if PPA is in persistent start and done mode respectively.

Memory-Mapped Scalar Interface

An alternative to raw scalar ports on the PPA is to memory-map them on the host bus. Memory-mapped `stable` scalar inputs always contain a hold register to hold the value throughout the execution of the pipelined set. Memory-mapped per-task scalar inputs and outputs always contain a hold register for each input and output. The hold register for per-task scalar inputs and outputs can be removed using the `hold_register` pragma.

For an input, a hold register provides persistence of the input value. If the next task input value is same as the previous, then host can avoid writing it. Without a hold register, the host should write each input for every task, even if the value has not changed. Similarly, for an output with a hold register, the host can avoid reading an output value (if don't-care). Without a hold register, the host should read each output for every task.

Hold registers allow overlap between the host and PPA. In other words, host can compute and write next-task inputs when the PPA is still using the current task inputs and the PPA can compute and write current task outputs when host is still reading the previous task outputs. Host-PPA overlap is critical with memory-mapped scalars because each scalar write/read via a single host bus consumes one dedicated clock cycle.

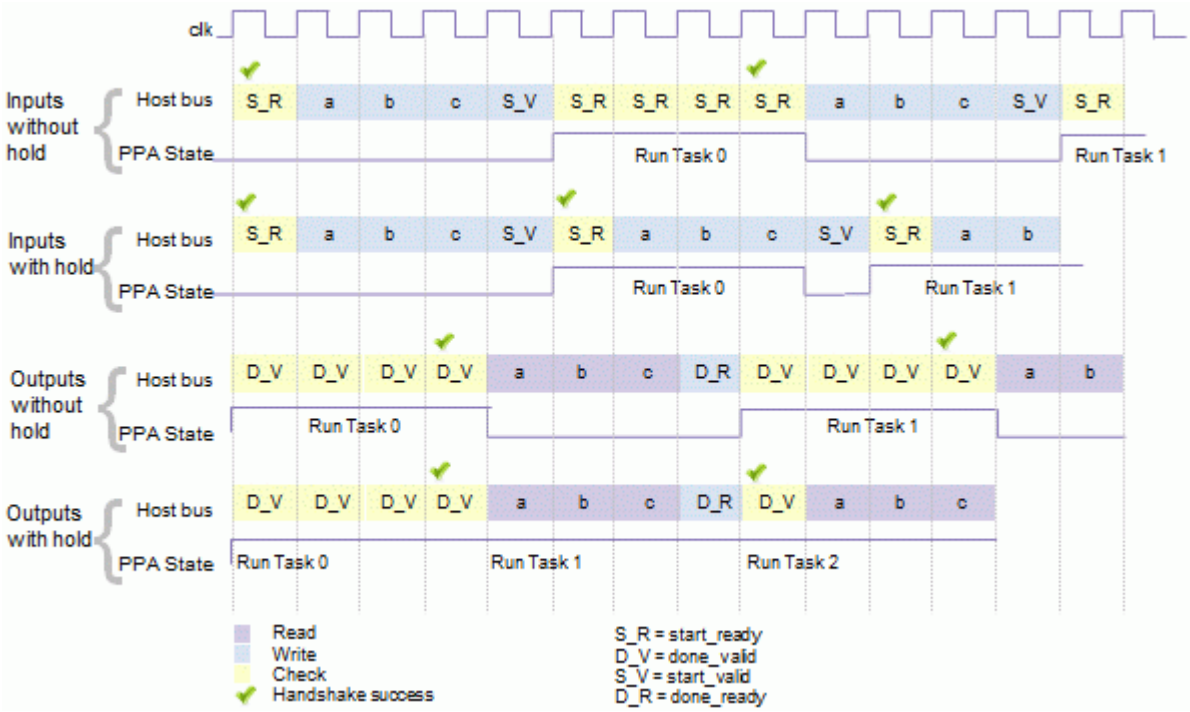
The following table shows the task behavior of the Host and the PPA for each combination of memory-mapped scalar input/output type and hold register type.

Table 14-5 Host and PPA Task Behavior, Memory-Mapped I/O

Input/output type	Hold reg value	Behavior
stable scalar input	single_buffer (default)	never: not allowed single_ buffer: External hardware can compute new value after hold reg write, but cannot write to hold register until pipelined execution completes.
Per-task scalar input	never, single_buffer (default)	never: External hardware can write Task(i) input only after PPA signals start_ready for Task(i). single_buffer: External hardware can write Task(i) input on start handshake of Task(i-1).
Per-task scalar output	never, single_buffer (default)	never: PPA can compute outputs of Task(i) only after done handshake of Task(i-1). single_buffer: PPA can compute outputs of Task(i) immediately after Task(i-1), but can write to hold reg only after done handshake of Task(i-1).

Figure 14-7 shows the host and PPA interface timing.

Figure 14-7 Host and PPA Interface Timing



The waveform takes an example with three inputs (a, b, and c) and three outputs (a, b, and c). The task runtime is four cycles. The waveforms describe four separate scenarios:

- Inputs without hold: There is no overlap between input writes by host and task run.
- Inputs with hold: Input writes by host, overlap with task run.
- Outputs without hold: There is no overlap between output reads by host and task run.
- Outputs with hold: Output reads by host overlap with task run.

PPA Interface Port Naming and Polarity

You can optionally control the names of the PPA interface ports. The port-naming method depends on the type of port:

- For non-programmatic ports (clock, reset, enable, abort, error, start handshake, done handshake, Host bus, scan/test ports), use the following command:

  ```
  set_params -system_port_name "key:signal_name ..."
  ```

- For ports resulting from programmatic entities, which are the memory, stream, scalar, and direct-access ports, use the following pragmas:

  ```
  #pragma signal_name_prefix variable prefix_string
  #pragma signal_name_identifier variable identifier_string
  #pragma signal_name_suffix variable suffix_string
  ```

  For details, see "signal_name_prefix, signal_name_identifier, and signal_name_suffix Pragmas" on page 7-60.

To specify the PPA top module name, use the following command:

```
set_params -top_module_name name
```

For example, `set_params -top_module_name fir_top` makes the top module name `fir_top` and the top module Verilog file name `fir_top.v`.

User-specified names must not start with the string "`scc_`". Note that gated clocks, clock gate enables, and OCP ports cannot be named by the user.

You can optionally control the polarity of enable, abort, error, start/done handshake control, and stream handshake control signals by using the following command:

```
set_params -system_port_polarity "key:signal_name ..."
```

The polarity of reset can be controlled using `set_params -reset_level` and the polarity of clock signals can be controlled using `set_params -edge_clock`.

Polarity for stream handshake signals can be specified on per-stream basis by using the following pragma:

```
#pragma signal_polarity stream_name:key high | low
```

For details, see "signal_polarity Pragma" on page 7-67.

To control the naming of system ports, use the following command:

```
set_params -system_port_name "key:signal_name ..."
```

This command specifies the signal name assigned to a system port function, with the function identified by a keyword: clock, reset, enable, abort, error, start_valid, start_ready,

done_valid, done-ready, host_req, host_ack, host_mode, host_wdata, host_rdata, or host_addr. For example,

```
set_params -system_port_name "clock:CLKA reset:RST"
```

Interface Timing

The Synphony C Compiler tool allows you to specify external delays on memories and streams. The boundary register and external delay specifications can be given using `set_params` command or pragmas. For more information on commands and pragmas, see the Chapter 3, “Synphony C Compiler Commands and Chapter 7, “Pragmas and Constraints respectively. The boundary registers might contain little combinational logic between the register and the PPA boundary.

The tool attempts to meet the delay specification by automatically inserting boundary registers in the synthesized hardware when needed.

- Even if boundary registers are present in the hardware, there exists some combinational logic between the register and the boundary. These are:
  - selector logic for arbitrated memory ports.
  - inversion logic for artisan memory ports.
  - prefetch logic for FIFOs on instream ports

    The Synphony C Compiler tool reserves a default delay of 5% of clock period for the post register logic including the clock -> Q delay for memories. This logic is present on the external path when the boundary register is present.
- The Synphony C Compiler tool reserves a default delay of 7% of clock period each on forward and return paths to accommodate some internal combinational logic for memories. However, this logic is present on the external path only if the boundary register is not placed.

The values influence the external delay that the tool can allow. These values may not suit all applications and you may increase the values in case of backend timing failures on this path. If the Synphony C Compiler tool errors out on not being able to meet the external delay, you may try to decrease this value. This can be done through synthesize_auxopts specification as shown (<number> should be a percentage of clock period):

```
"-Ftypical_post_register_delay=<number>"                    (for (1))
"-Ftypical_mem_forward_path_delay=<number>"                 (for (2))
"-Ftypical_mem_return_path_delay=<number>"                  (for (2))
```

These options should be specified at the beginning of the implementation. When changed, the implementation has to run at least from schedule onwards.

Deployment of Memory-Mapped PPA

This section describes how to integrate a PPA with a memory-mapped interface with C/C++ code running on a host processor. In this description, the term “driver code” to refers to the C/C++ code running on the host processor that interacts with the memory-mapped PPA.

API for Memory-Mapped PPA Deployment

Synphony C Compiler provides Application Program Interface (API) functions for the driver code to interact with the PPA. In the Synphony C Compiler installation, the directory `scc/simu/src/SimCode/` contains the `scc_mmap_depl_api.h` and `scc_mmap_depl_api.c` files, which contain the deployment APIs. The header file can be included in the driver code and the source file (.c) should be compiled on the host processor.

The following driver code and PPA code examples are used to demonstrate the procedure for deploying a memory-mapped PPA:

```
// DRIVER
#include <stdio.h>
void ppa(int ps1, // in scalar
         int& ps2, // inout scalar
         long long parr1[10] // inout array
         );
short gs1; // Global inout scalar
int garr1[10]; // Global inout array
int main()
{
  int i;
  // Initialize global array and the actual array passed to ppa
  long long aarr1[10];
  for (i = 0; i < 10; ++i) {
    garr1[i] = i; aarr1[i] = i + 1;
  }
  // Invoke 100 tasks on PPA
  for (i = 0; i < 100; ++i) {
    int inoutval;
    // Initialize per-task scalars
    gs1 = i * i;
    inoutval = i + 1;
    // Call PPA
    ppa(i, inoutval, aarr1);
    // Verify scalar output
    printf("\n inoutval = %d\n", inoutval);
  }
  // Verify memory output
  for (i = 0; i < 10; ++i) {
    printf("\n garr1[%d] = %d, aarr1[%d] = %d\n", i, i,
garr1[i],aarr1[i]);
  }
```

```
  return 0;
}


// PPA
extern short gs1;
extern int garr1[10];

void ppa(int ps1,
         int& ps2,
         long long parr1[10]
         )
{
  int i;
  for (i = 0; i < 10; ++i) {
    ps2 += parr1[i] + garr1[i];
    parr1[i] = ps1 + gs1;
    garr1[i] = ps1 + ps2;
  }
}
```

Procedure for Memory-Mapped PPA Deployment

Use the following procedure to deploy the memory-mapped PPA.

1. Generate the PPA by memory-mapping the control and other addressable components. For example, use:

   ```
   set_params -interface_type memory_mapped_handshake
   set_params -host_scalar_access lm
   set_params -host_memory_access infer,lm
   ```

2. If desired, assign specific addresses to various components using the following command:

   `set_params –host_address_map` *filename*

3. Verify the PPA by running simulation.

   The following steps might require some changes to the driver code. These changes must be done independently for deployment *outside* the Synphony C Compiler framework.

   The report file `Reports/report_address_map.txt` contains PPA Base Address and address offsets for each addressable components, namely:

   a. Scalar inputs and outputs

   b. Arrays implemented as memories and the base addresses for each array

   c. PSW for commands and status

This file has information you need about the hardware address or address assigned in the PPA for each component. This is the address map file for the example being considered:

```
# PPA BASE ADDRESS    0X1000
#
# Start      End      Num      Bitsize  Read/  Load     Store    Name
# Address    Address  Elements          Write  Latency  Latency
# (Byte)     (Byte)
# 0x0        0x0      1        32       R+W    0        0      Processor Status Word (PSW)
# 0x8        0x8      1        16       R+W    0        0      gs1 (Input variable)
# 0x10       0x10     1        32       R+W    0        0      ps1 (Input variable)
# 0x20       0x20     1        32       R      0        0      ps2[0] (Output variable)
# 0x30       0x30     1        32       R+W    0        0      ps2[0] (Input variable)
# 0x40       0x64     10       32       R+W    0        1      garr1 (Memory)
# 0x80       0xC8     10       64       R+W    0        1      parr1 (Memory)
}
```

4. Scalar inputs and outputs have both C and PPA copies. The driver code should write in the scalar inputs and read out the scalar outputs using `scc_write` and `scc_read` API calls. The hardware addresses of the scalar inputs and outputs can be obtained from the `Reports/report_address_map.txt` file.

5. Arrays declared in the PPA global scope have only the hardware or PPA copy. For this, change the array declaration to `extern` in the driver code and specify an absolute address for the array during C linking phase. The absolute address is the PPA base address + the array's base or start address as shown in the `Reports/report_address_map.txt` file. After you make this change, any read/write transaction to the array will automatically access the memory in the PPA.

6. Arrays declared as PPA parameters have both C and PPA copies. The driver code should copy in the memory values from the actual C array to the hardware addresses, one at a time, before starting tasks, and it should read out the memory values from the hardware address to the C array after completing tasks. Write and read can be done using `scc_write` and `scc_read` API calls, respectively.

7. Synphony C Compiler provides APIs for task start and done controls. These APIs provide command issue and status checking capabilities. Simple application without task overlap will follow these steps to execute a task on the PPA:

   a. Write memories and scalar inputs of a task

   b. Start a task

   c. Poll for completion or get interrupted by an interrupt from the PPA

   d. Read memories and scalar outputs of a task

   Basically, steps b and c replace the PPA procedure call in the driver code.

Figure 14-8 shows the Synphony C Compiler GUI with the original driver code on the left side and the modified driver code for deployment on the right side. Note that this driver code

modification is a specific example without task overlap and with polling APIs. If tasks must be executed in a pipelined manner, then more complex driver coding can be used.

Figure 14-8 Memory-Mapped PPA Driver Code

```
File  Edit  Tools  Syntax  Buffers  Window  Help

#include <stdio.h>

void ppa(int ps1 /* in scalar */, int& ps2 /* inout scalar */,
         long long parr1[10] /* inout array */);
short gs1; // Global inout scalar
int garr1[10]; // Global inout array

int main() {
  int i;
  long long aarr1[10];
  for (i = 0; i < 10; ++i) { // Init global and actual array
    garr1[i] = i;
    aarr1[i] = i + 1;

  }
  // Invoke 100 tasks on PPA
  for (i = 0; i < 100; ++i) {
    int inoutval;
    // Initialize per-task scalars
    gs1 = i * i;

    inoutval = i + 1;


    // Call PPA
    ppa(i, inoutval, aarr1);


    printf("\n inoutval = %d\n", inoutval); // Verify scalarout
  }
  for (i = 0; i < 10; ++i) { // Verify memory output

    printf("\n garr1[%d] = %d, aarr1[%d] = %d\n",
            i, i, garr1[i], aarr1[i]);
```

```
#include <stdio.h>
#include "scc_mmap_depl_api.h"
void ppa(int ps1 /* in scalar */, int& ps2 /* inout scala
         long long parr1[10] /* inout array */);
short gs1; // Global inout scalar
extern int garr1[10]; // Global inout array. C Linker me
const int PPA_BASE=0x1000;
int main() {
  int i;
  long long aarr1[10];
  for (i = 0; i < 10; ++i) { // Init global and actual ar
    garr1[i] = i;
    aarr1[i] = i + 1;
    scc_write((void*)(PPA_BASE + 0x80 + i * sizeof(long l
              &aarr1[i], SCC_LONGLONG);
  }
  // Invoke 100 tasks on PPA
  for (i = 0; i < 100; ++i) {
    int inoutval;
    // Initialize per-task scalars
    gs1 = i * i;
    scc_write((void*)(PPA_BASE + 0x8), &gs1, SCC_SHORT);
    inoutval = i + 1;
    scc_write((void*)(PPA_BASE + 0x30), &inoutval, SCC_I
                                                  // write i
    // Start PPA
    scc_start_handshake((void*)(PPA_BASE + 0x0));
    while (!scc_is_done_valid((void*)(PPA_BASE + 0x0)))
          ; // Busy Wait for task completion
    scc_read(&inoutval, (void*)(PPA_BASE + 0x20), SCC_INT
                                                  // read inou
    scc_issue_done_ready((void*)(PPA_BASE + 0x0));
    printf("\n inoutval = %d\n", inoutval); // Verify sca
  }
  for (i = 0; i < 10; ++i) { // Verify memory output
    scc_read(&aarr1[i], (void*)(PPA_BASE + 0x80
             + i * sizeof(long long)), SCC_LONGLONG);
    printf("\n garr1[%d] = %d, aarr1[%d] = %d\n",
            i, i, garr1[i], aarr1[i]);
```

Pragmas for EDK support (FPGA Only)

The following pragmas are used to indicate the bus standard and the type used by the streaming interface. Note that most of the interface changes specified through these EDK pragmas are applied on a wrapper layer around PPA.

Bus Name

Some bus standards are best modeled as a combination of multiple streams, giving all the related streams the same bus name allows for connection in edk. You can specify the name of the bus that is used for communication using the following pragma:

```
#pragma bus_scc_streamname streamName busName
```

If you do not specify a name for the bus, the default name would be the name of the stream without the `scc_stream_input` or `scc_stream_output` prefix.

Bus Standard

You can specify the bus standard that the stream uses for communication using the following pragma:

```
#pragma bus_standard streamName busStandardName
```

Example:

```
#pragma bus_standard scc_stream_output_A STANDARDX
```

This means `scc_stream_output_A` is available for integration using a bus standard called `STANDARDX`.

The bus standard can also be specified as none. The stream will not be bound to any bus protocol and the signals of this stream appear as separate ports visible as independent entities to a third party integration tool. You can manually hook the port connections for this particular stream as required. This is especially useful in cases where some behavior of an interface can be exploited to reduce the amount of hardware and the number of interconnects.

For example, if the data is always going to be available, the ready signal can always be tied to its active level. This could not be done when the stream is associated with a bus standard (where all the ports were required to be connected to a particular signal of the other protocol).

If you do not specify a bus standard, and have used a struct stream, then the name of the standard is the name of the struct. It must be noted that you must name the struct. If you use a typedef, it is required to name the struct as well.

Example:

```
typedef struct XIL_LL_STREAM {
unsigned int LL_Data;
unsigned int LL_SOF_n;
unsigned int LL_EOF_n;
} XYZ;
```

The struct name `XIL_LL_STREAM` is used as the bus standard name for any struct stream of this type.

If you do not specify a bus standard, and have a stream of a base C datatype, then the name of the standard will be `SCC_STREAM`*`datawidth`*, where *`datawidth`* is the width of the data port of the stream (1..64).

Bus type

The bus type can be specified using the following pragma.

```
#pragma bus_type streamName busTypeName
```

Example:

```
#pragma bus_type scc_stream_output_sample SLAVE
```

If you do not specify a bus type, then by default, the `bus_type` is `TARGET` for input streams and `INITIATOR` for output streams.

Naming the pcore Ports of Synphony C Compiler stream

There are three cases to be considered here:

- the desired streaming interface is three signals,
- the desired streaming interface is greater than three signals, and
- the default behavior.

1. Three signal interfaces

   You can specify the Synphony C Compiler streaming interface ports that are to be bounded to the port or field of the BUS using the following pragmas:

   ```
   #pragma data_name streamName busPortName1
   #pragma request_name streamName busPortName2
   #pragma ready_name streamName busPortName3
   ```

   Example:

   ```
   #pragma data_name scc_stream_output_sample DATAX
   #pragma request_name scc_stream_output_sample SENDX
   #pragma ready_name scc_stream_output_sample ACKX
   ```

   In the example, data port of output stream sample is bound to `DATAX`, request port of output stream sample is bound to `SENDX` and ready port of output stream sample is bound to `ACKX`.

2. Greater than three signal interfaces

   The additional ports have to be specified using a struct

- For example: Xilinx local link has the following required ports
  - `LL_Data`: data
  - `LL_SOF_n`: start of frame
  - `LL_EOF_n`: end of frame
  - `LL_SrcRdy`: source ready
  - `LL_DstRdy`: destination ready
- Here `LL_SrcRdy` and `LL_DstRdy` map to the Synphony C Compiler stream control signals, Request and Ready. You can bind these ports to request and indicate ready signals using the following pragmas:

  ```
  #pragma request_name scc_stream_output_sample LL_SrcRdy

  #pragma ready_name scc_stream_output_sample LL_DstRdy
  ```

- The other three ports are supported using a struct whose fields have the same name as the ports of the bus protocol.

  ```
  struct XIL_LL_STREAM {
       unsigned int LL_Data;
       unsigned int LL_SOF_n;
       unsigned int LL_EOF_n;
  };
  ```

  If using the struct inside a struct, the name of the data port is:

  `instream/outstream_`*`name_of_embedded_struct_instance`*`_di/do_0_`*`field_name`*

  For example,

  ```
  struct coordinate {
      int x;
      int y;
  };
  struct LL {
      struct coordinate data;
      unsigned int sof;
  };
  struct LL scc_stream_input_sample (void);
  ```

The wrapper port names are `instream_sample_di_0_data_x`, `instream_sample_di_0_data_y`, `instream_sample_di_0_sof`.

You can change the name of the BUS signal to which the port (field of the structs) is bound to, using the following pragma:

```
#pragma data_name streamName .fieldName busPortName1
```

Example:

```
#pragma my_data_name scc_stream_input_sample.data.x DATAX
```

If you do not specify bindings for stream ports with the given pragmas, then the following bindings are used by default.

- Input Stream
  - data port is bound to DATA
  - request port is bound to ACK
  - ready port is bound to SEND
- Output Stream
  - data port is bound to DATA
  - request port is bound to SEND
  - ready port is bound to ACK

The Synphony C Compiler's stream can be active or passive. By default, the stream is active. When the stream is active, the request signal from the Synphony C Compiler's stream is sent irrespective of the state of the ready signal from the host. When the stream is made passive, the request signal from the Synphony C Compiler's stream is sent only after receiving the ready signal from the host. The following pragma can be used for changing the control mode of the streams:

```
#pragma control_mode streamName active | passive
```

Some protocols do not allow the request signal to be sent unless the host sends a ready signal. After receiving the ready signal from the host, the stream would send the request signal. You can make use of the pragma for making the stream passive for this purpose.

The PPA name is visible to you in the EDK as the module to be connected. This module uses a certain bus for communication and is given. name is the name of the bus that is visible to you.

- `scc_stream_`*`input_name`*
- `scc_stream_`*`output_name`*

The bus can be connected to another bus of compatible standard and type.

TCAB Integration

The previous sections describe the way a PPA can be integrated into user environment. This section describes how a TCAB can be integrated into user environment.

TCABs are by default built with simple interfaces so that they can be easily deployed into a higher level design. The Synphony C Compiler tool simplifies the TCAB interfaces by first building an *importable* TCAB layer and then building a *deployable* TCAB layer around it. The deployable TCAB interfaces are simpler, and customizable to aid manual integration into a higher level design. All the reports and views in TCAB implementations are for the deployable TCAB. When you import the TCAB into a higher level design using the Synphony C Compiler tool, the importable layer is instantiated at the higher level. So, the cost of imported TCABs in top-level implementations may be lower than the cost of the TCAB as shown in the TCAB implementation.

TCAB Interface and Integration Guidelines

This section describes the ports on the deployable TCAB layer and guidelines to integrate the TCAB into a higher level. The following table lists the deployable TCAB interface.

Signal	Input / Output	Signal protocol	Comments
clk	Input	clock signal	If the TCAB is purely combinational, this port is not generated.
reset	Input	Level signal	System reset
start	Input	Pulse, active high	Refer integration description for details
enable	Input	Level, active high	Chip enable for the TCAB
stallbarout	Output	Level, Active Low	Container should be disabled whenever low.
Memory and stream ports	Input and Output	Standard memory/stream interface protocols	Memory/Stream is used by the TCAB in the closed interval [resource_first_use_time, resource_last_use_time] with respect to start.

Signal	Input / Output	Signal protocol	Comments
Scalar data input ports	Input and output		Input data is consumed by the TCAB at input_sample_time (w.r.t. start). A stable scalar is not registered inside the TCAB and therefore the value should be present from input_sample_time of the first task till the end of the last task in a group of tasks which execute in pipelined fashion.
Scalar data output ports	Output		Output data is produced by the TCAB at output_production_time (w.r.t. start). The data is available only in that cycle.
Externalized TCAB ports	Input and Output	Same as externalized TCAB	For each TCAB that physically lies outside the current TCAB, a set of interface ports is created.
Clock gate enable	output	Level, active high	Present only when clock gating is enabled for this TCAB.
Abort	input	Level, active high	Present only if TCAB is built with abort logic enabled

Consider the case of a TCAB with the following interfaces.

Scalar Inputs: data_in0, data_in1

Scalar Outputs: data_out

Memories: memA

Streams: strA_In

Consider the following properties (all architectural) of the interfaces mentioned in the *TCAB_Implementation*/Reports/report_summary.txt.

Architectural Pipelinability = 9

Input sample time of data_in0 = 0

Input sample time of data_in1 = 2

Output production time of scalar data_out = 4

Resource first use time of memory memA = 2

Resource last use time of memory memA = 5

Resource first use time of stream strA_In = 1

Resource last use time of stream strA_In = 4

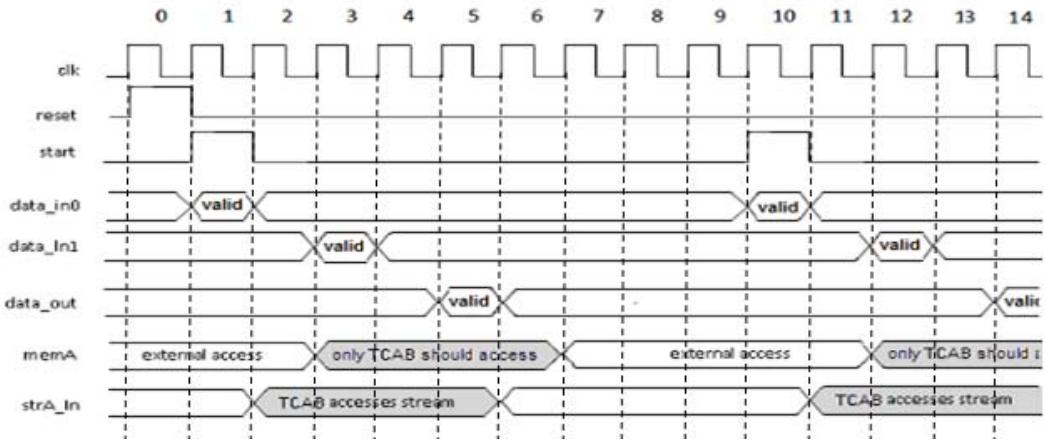
This TCAB can be integrated into higher level designs using the following guidelines.

- The minimum number of cycles between start pulses for different tasks cannot be less than the architectural pipelinability of the TCAB (9 in this example).
- The stallbarout signal indicates if the TCAB is stalling due to any reason. Any cycle count for input/output needs to discount the cycles where the stallbarout is low (stallbarout is active low signal).
- The scalar inputs do not have enable signals by default for the TCAB. Inputs need to be given to the RTL on the cycle specified by the input_sample_time value (The cycle with start pulse would need to be counted as cycle 0). In this example, the inputs would be given on the following cycles:

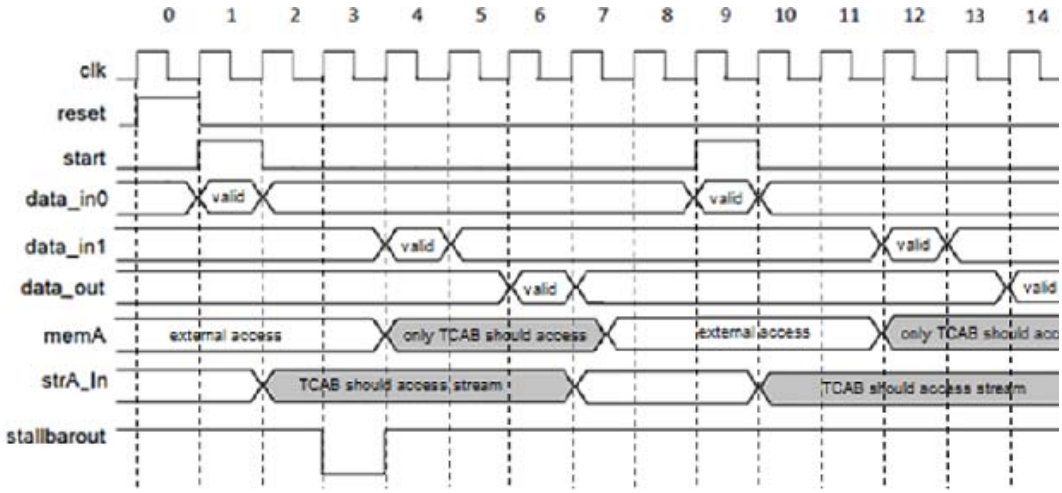
  data_in0: 0 (same as start)

  data_in1: 2
- The scalar outputs do not have enable signals by default for the TCAB. Outputs will be produced on the cycle specified by the output_production_time value. (cycle 4 in this example).
- MemA is reserved by the TCAB in the closed time interval [2,5] and Str_AIn is reserved by the TCAB in the closed time interval [1,4].

The timing diagrams are as shown.



If the stream strA_In does not have data in cycle 1 when it makes its request, this will cause the TCAB to stall and the stallbarout would be asserted. The revised timing diagram in that case would be as shown in the following figure.



For more information on building and using TCABs, see the Chapter 10, “Hierarchical Synthesis.

Customization of TCAB Interface Ports

You can customize the control ports on the deployable TCAB boundary using the following implementation parameter.

```
set_params always_enabled:[yes|no]
 conditional_start:[yes|no] conditional_outputs: [yes|no]
```

`always_enabled:yes | no`	If set to yes, the enable port is suppressed at the deployable TCAB interface. The default is yes.
`conditional_start:yes | no`	The no option removes the start port on the deployable TCAB. This is similar to the auto_start option for the PPA where the TCAB does not require any explicit start signals. Default is yes.
`conditional_outputs:yes | no`	The yes option qualifies the scalar outputs of the TCAB with an output enable signal. These signals are removed at the TCAB interface if no is selected. Default is no.

Verification

The Synphony C Compiler tool provides a framework for you to perform simulation-based verification at several stages during the process of transforming the original application C source code to hardware. The framework also automatically carries forward the functional tests provided by your driver code such that simulation at each stage may use these test vectors. The following types of simulations are supported:

- C-Simulation
- Verilog simulation
- Gate-level simulation

C-Simulation

C-simulation is a C code level simulation that uses a software model of the PPA and driver code to produce output. A single command can initiate the C simulation process.

```
verify -golden | -lint | -rtl | -clean [-options ]
```

-status	show current settings
-ccompiler_args “<options>...”	specify C compiler options
-cppcompiler_args “<options>...”	specify C++ compiler options
clinker_args “<options>...”	specify C linker options
-compile_only	do not run, just compile
-run_only	skip compile, run only
-nostdout	do not compare the results produced on standard output
-exec_args “<args>...”	specify run option for executable
-gcov	specify simulation level after synthesis: 1-6(default 6)
-clear_gcov_history	enable code coverage (golden only)
	clear code coverage data (golden only)

First, your application C source code is simulated along with your driver code. This is termed the golden simulation because the results (the output scalar and array variable values, and the output stream values) of this simulation serve as reference for all other simulations. The files used in this simulation are (<ext> is the file extension depending on whether the source is C or C++):

- <appname>.<ext>: Your application C/C++ source code.

- <drivername>.<ext>: Your driver code.
- golden.o: Contains the high-level API function definitions that must be linked-in for this simulation. Available from your (site-wide) Synphony C Compiler installation directory: <installation_directory>/simu/src/SimCode/.
- libpdextn.a: This library contains the tool specific function definitions for supporting streams, saturating arithmetic, and sliding windows. This file is available from your (site-wide) Synphony C Compiler installation directory <installation_directory>/simu/src/SimCode/.

In the lint phase, your application C source code is checked for syntax and other forms of compliance. The resulting preprocessed application code may be simulated along with your driver code, and its results verified against the golden simulation. The files used in this simulation are:

- <appname>_lint.<ext>: This file, which includes scc_remap.h and scc_streams.h from <installation_directory>/simu/include, is the preprocessed C code for the application.
- <drivername>.<ext>: Your driver code.
- golden.o: Contains the high-level API function definitions that must be linked in for this simulation. This file is available from your (site-wide) Synphony C Compiler installation directory: <installation_directory>/simu/src/SimCode/.
- libpdextn.a: This library contains the tool specific function definitions for supporting streams, saturating arithmetic, shifts. This file is available from your (site-wide) Synphony C Compiler installation directory: <installation_directory>/simu/src/SimCode/.

After the schedule phase, you can verify the correctness of loop-iteration scheduling and other frontend compiler transformations by performing C code level simulation. For this simulation, an option is provided to create register, memory and stream dumps which can be used for offline Verilog verification. The files used in this simulation are:

- <appname>_systolic.<ext>: The file is the transformed C code for the application, and represents an intermediate C level software model of the PPA. It has a single function that is called by the driver code.
- <appname>_entrystub.<ext>: This file, which includes <appname>_init.h and npa_info.h, both generated by the schedule phase, contains the generated C level driver code for the PPA. It is the transformed code for the user-supplied function and includes calls to the PPA API functions to initialize the PPA, start the PPA, write register input values, and read back the register output values.
- <appname>_init.<ext>: C simulation-specific initialization code for the PPA resides in this file. It has variable declarations for the pointer variables referred to in the PPA and the function definitions for the routines that will help form a part of the C simulation environment.

- <drivername>.<ext>: Your driver code.
- libpdextn.a: This library contains the tool specific function definitions for supporting streams, saturating arithmetic, shifts. This file is available from your (site-wide) Synphony C Compiler installation directory: <installation_directory>/simu/src/SimCode/.
- libsimdump.a: This library contains function definitions, which are used for dumping register and memory input/output values into files for each task and also stream input and output values. This is required if you want to perform offline Verilog simulation later using the test data specified by the C driver. Both libsimdump.a and libnosimdump.a are available from your (site-wide) Synphony C Compiler installation directory: <installation_directory>/simu/src/SimCode/.

You also have the option to perform C-simulation at several levels during the synthesis phase. You can use this simulation to verify the correctness of instruction scheduling and other backend compiler transformations. The files used in this simulation are:

- <appname>_rsim.c: These files, which include <appname>_rsim.c.inc, <appname>_rsim.c.tbls and benchmark_data_init.simu.c are the transformed C code for the application, and represent an intermediate C level software model of the PPA.
- <appname>_entrystub.<ext>: This file, which include <appname>_init.h and npa_info.h, both generated by the schedule phase, contains the generated C level driver code for the PPA. It is the transformed code for the user-supplied function and includes calls to the PPA API functions to initialize the PPA, start the PPA, write register input values, and read back the register output values.
- <appname>_init.<ext>: C simulation-specific initialization code for the PPA resides in this file. It has variable declarations for the pointer variables referred to in the PPA and the function definitions for the routines that will help form a part of the C simulation environment.
- driver.<ext>: Your driver code.
- libpdextn.a: This library contains the Synphony C Compiler specific function definitions. This file is available from your (site-wide) Synphony C Compiler installation directory: <installation_directory>/simu/src/SimCode/.
- libsimdump.a This library contains function definitions, which are used for dumping register and memory input/output values into files for each task and also stream input and output values. Both libsimdump.a and libnosimdump.a are available from your (site-wide) Synphony C Compiler installation directory: <installation_directory>/simu/src/SimCode/.

  Note:

  Synphony C Compiler passes a compiler directive to all the stages except the golden simulation. Therefore, to print debug information in the code during the golden simulations, you can use the following mechanism to gate that segment of code.

```
... // Synphony C Compiler ready code (Stmt block 1)
#ifndef SCC
//Any debug code. (Stmt block 2)
#endif
```

This way the code in (Stmt block 2) would be valid only during golden simulations and would not be visible to the successive stages.

Post-Netlist-Synthesis Simulation Support

Gate-level simulation support is available in the Synphony C Compiler tool. The Synphony C Compiler Simulation Framework includes scripts to support Gate-level Verilog simulation. This facility allows an alternative Verilog model with equivalent ports, (e.g. gate-level model or FPGA Verilog model) to be substituted as the Design-Under-Test in the Synphony C Compiler testbench.

To substitute an alternative module or gate-level Verilog model, use the verify command with the -alt_ppa option and provide a full-path or filename or rtl_package - relative filename for the gate-level Verilog model:

```
-alt_ppa <path/alt_topvlog_filename>
```

If a technology library is referenced by this model, you may need to supply a path to the library's Verilog simulation models or the library Verilog simulation model file using the `verify` option:

```
 -vcompiler_args "-y libfile_path"
```

or:

```
 -vcompiler_args "-v libfile_name"
```

Example:

```
verify -rtl -simulator vcs -alt_ppa synth/run/gateppa.vg \
 -vcompiler_args "-y libfile_path"
```

Note that only functional simulation is supported, since the simulator option `+delay_mode_zero` may need to be imposed.

That is, all the delay specifications and all module path delays should be set to zero, so Verilog timing simulation is not supported with the Synphony C Compiler testbenches.

SystemVerilog (SV) Universal Verification Methodology

The Universal Verification Methodology (UVM) standard for simulation testbenches, which is built upon the SystemVerilog (SV) language, is being widely adopted in the hardware verification community. Synphony C Compiler can automatically generate a simulation testbench based on this standard and run a Verilog simulation of the RTL operating with this testbench. This feature is called SV-UVM simulation or just SV simulation.

- It follows a UVM standard for testbenches, so it is more understandable and reusable.
- It runs the C/C++ application and driver code as a reference, captures the data inputs and outputs in the testbench infrastructure, and provides it to the device under test (DUT), the RTL for the device. It is still recommended that you run `verify -golden` and `verify -lint` command to find any application errors.
- It compares the DUT results against the reference results using scoreboards.

After you run the `build` command, to run the SystemVerilog (SV) simulation with a UVM testbench, run the `verify -rtl` command

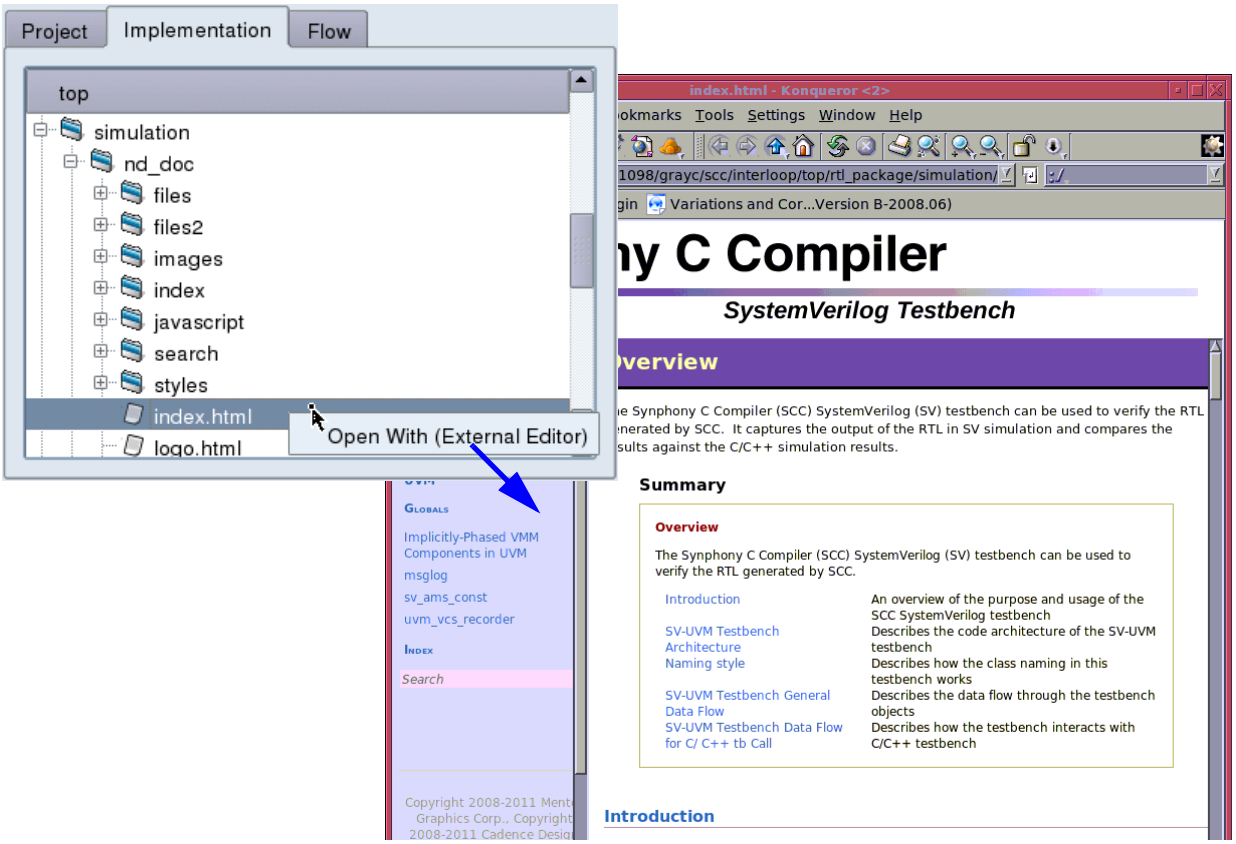
Online Documentation for Testbench

After you run the `build` command, you can view the online documentation for the SV-UVM testbench in HTML format. In the Synphony C Compiler GUI, go to the Implementation pane, and navigate to the following file:

```
rtl_package/simulation/nd_doc/index.html
```

Right-click the index.html file and choose “Open With External Editor,” as shown in Figure 14-9. This opens the documentation in your default HTML browser. In the browser window, navigate to the desired topic.

Figure 14-9 SV-UVM Testbench Documentation



Simulation Reporting

During an SV simulation, the shell console window and the report_summary.txt file report the task II, latency, and throughput numbers.

The SV simulation generates the following report files:

```
implementation-name_performance_sv.pdf
report_verify_rtl_performance_summary.txt
```

Application Coding Guidelines

Observe the following application coding guidelines:

- In a C++ file, use `extern "C"` in front of `int main(int argc, char* argv[])`. Omitting this statement leads to the following linking error in VCS:

  ```
  make: *** [compile] Error 2
  …/implementation-name/rtl_package/simulation/uvm/.../scc/src/
  ```

```
golden.c:20: undefined reference to `pico_main'
verify -rtl compilation failed.
```

- Use `return` instead of `exit` in the driver code. Exiting from the application or driver code results in exiting the entire SV simulation, which prevents the simulation from running to completion and showing status for debugging.
- Include the `scc.h` file at your top-level driver code.

SV Simulation Features

SV simulation supports all Synphony C Compiler features, except for the following issues:

- The `-alt_ppa` option is not supported.
- In some cases, instances of `scc_dsp()` output `X` during simulation, but this does not happen if `verify -rtl` command iis run with the `-debug_all` option.

Limitations

The following limitations apply to the SV simulations feature:

- If a design has multiple objects of a channel, SV simulation is not supported in the channel implementation.
- Gate-level simulations are not supported.

Compiler and Execution Options of verify -rtl Command

The following table lists the compiler and execution options of the verify -rtl command and how they are used in SV simulation.

Table 14-6 SystemVerilog Simulator, Compiler and Execution Options

verify option	SV simulation usage
-ccompiler_args	Passed to all gcc/g++ and systemC compiles
-clinker_args	Passed on during link phase calls to gcc/g++ & simulator link.
-cexec_args	Passed to SV executable

Table 14-6 SystemVerilog Simulator, Compiler and Execution Options

-cppcompiler_args	Passed to all gcc/g++ and systemC compiles
-vcompiler_args	Passed to Verilog/SystemC compile command line
-vexec_args	Passed to Verilog/SystemC Simulator execute command line

Interfacing to SCC SystemC Transaction Level Models

One can interface to Synphony C Compiler generated models via transaction-level model (TLM) methods used by PPA interfaces. The Synphony C Compiler generated TLMs use the classes defined in the file scc_tlm.h, found in the file:

```
<installation_directory>/simu/include/scc_tlm.h
```

or in any rtl_package:

```
<implementation>/rtl_package/scc/include/scc_tlm.h
```

The following classes are defined and the methods may be used by PPA interfaces:

```
scc_stream_in_if
scc_stream_out_if
scc_fifo_if
scc_direct_read_mem_if
scc_direct_write_mem_if
scc_direct_mem_if
scc_tlm_mem_if
scc_mem_if
scc_tlm_raw_if
scc_tlm_interrupt_if
scc_reg_if
scc_tlm_read_mem
scc_tlm_readwrite_mem
```

The following interface ports are defined, using the `sc_port` datatype:

```
scc_reg_port
scc_stream_in_port
scc_stream_out_port
scc_mem_read_port
scc_mem_write_port
scc_mem_port
scc_tlm_mem_port
scc_fifo_port
```

```
scc_tlm_interrupt_port
scc_tlm_raw_port
```

The following channels are defined, using the `sc_channel` base class:

```
scc_fifo_channel
scc_stream_input_channel:
scc_stream_output_channel:
scc_tlm_channel
```

Refer the prototypes in the file scc_tlm.h for details.

Comparison Strategy Overview

In its various levels of SV simulation, Synphony C Compiler supports a uniform strategy to compare results of the device under test. This strategy has some minor prerequisites that will be described under the User Guidelines section, in terms of specifying output result filenames to the Synphony C Compiler to assure that the output results are properly associated and compared.

There are three aspects of the strategy that will be noted here:

1. Handling of writes that you would normally send to stdout.
2. Handling of result files and stream output data files to enable simulation compares
3. Directory structure of the standalone package to support this functionality in Synphony C Compiler.

SystemVerilog Verification Guidelines

Follow these guidelines to correctly verify and compare the expected outputs of your application:

- Result/output data and stream outputs:

  You are advised to write any result/output data to result files so that they may be compared by the Synphony C Compiler tool. The result/output filenames to be produced by the application (data files and stream output files) should be specified to the tool via the Tcl command:

```
set_params -results "<results_filelist>"
```

The Command-Line Interface will copy the output files for compare into the directory rtl_package/output in the RTL package.

The SV simulation and compare process will use the user-specified files as the references for compare.

You are advised to use PPA-specific file descriptors to identify PPA-specific results/output to files. These result files will then be compared, on a PPA basis when declared via `set_params -results`.

When output streams are instantiated, you must output your data to separate output files, on a per-stream basis, via fprintf(). This assures that task-overlap can occur without mixing/interleaving the data in a single file. You can name stream data output filenames anything you would like, provided that the filenames be specified as results/output files in the Command-Line interface:

```
set_params -results "<results_filelist>"
```

- Writing results to stdout or cout:

  The strategy dictates that if you opt to rely upon simply using stdout or cout for your output, then you should also recognize that effects such as task overlap could cause some differences in the order of data between the reference output and the generated output from simulations. (e.g. if there are any overlapped tasks or if there are extraneous writes to stdout from other sources, such as the Verilog simulator).

Streamproc and Driver Creation Guidelines

When your application uses streams external to the PPA, you should consider the following when writing the driver for the stream procedure (streamproc.c)

- If an instream interface contains a prefetch fifo, then streamproc should allow for pre-fetching past the end of task for `verify -rtl` command to work correctly.
- In the functions stubs for external streams, you should not call other streams, which are part of the PPA. In effect, the stream functions in the driver are not similar to interrupt handlers.

Other SystemVerilog Guidelines

The following SystemVerilog guidelines also apply:

- Include scc.h to the driver file, which contains the definition of main( ).
- main( ) should have the standard C style declaration int main (int argc, char** argv).

15

Synphony C Compiler Output

This chapter describes the Synphony C Compiler Outputs. This chapter is organized as follows:

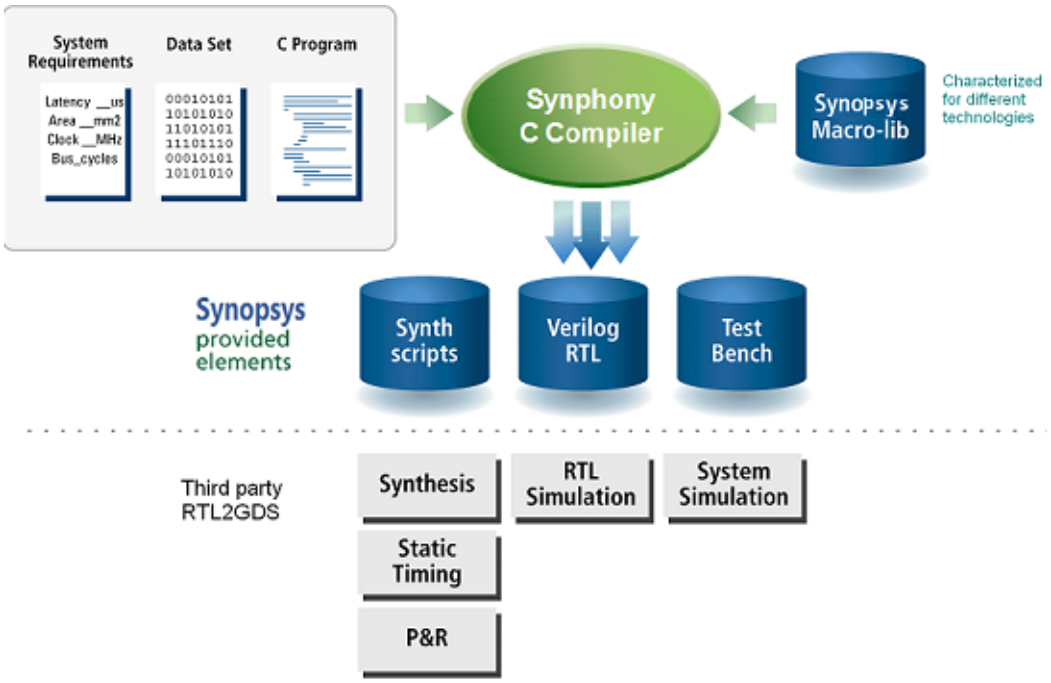
- Overview of Synphony C Compiler Output
- Hierarchy of Generated RTL
- Package Directory Structure
- Directory Structure to Support SV Simulation
- Corner Case Simulation Support
- Assertions
- RTL Features
- STARC Rule Support

Overview of Synphony C Compiler Output

The Synphony C Compiler tool generates a directory <implementation>/rtl_package when you use the create_rtl_package command. This rtl_package directory contains the generated RTL and several scripts and utilities that you can use to verify the intended functionality of the RTL and also to synthesize the RTL into netlist using other third party tools.

The create_rtl_package command initiates the packaging process. It creates a standalone directory of all the files needed to compile and run Verilog simulation of the RTL or to compile and run RTL synthesis. For more information on the directory structures, see "Package Directory Structure" on page 15-4.The advantages gained within the Synphony C Compiler verification and synthesis flow are maintained through creating the RTL artifact, by assuring synthesis, timing, simulation compatibility, and routability.

The following figure shows a reference flow starting with the input algorithmic C application, through the Synphony C Compiler tool, and to the industry standard tools used to transform Synphony C Compiler RTL to hardware.



The Synphony C Compiler RTL is produced in Verilog™. By default, the v2k constructs are used to produce RTL. You can use the following synthesize_auxopts to make the Synphony C Compiler tool produce v95 conforming RTL.

- force_v2k_syntax

  Specify a “:” separate list of macrocell names for which v2k cells should be used. This override is provided in case using v95 macrocells in your design causes any QoR degradations on your application, and you want to selectively allow v2k macrocells to be used in your design. You must specify the macrocell name that needs to be overridden. However, you can specify sramff_raw to override all the sramff_raw_<n> cells in the design.

  Example usage from Tcl command and GUI are:

  UI:

  setvar synthesize_auxopts "-Fforce_v2k_syntax=minmaxw:sramff_raw_2"

  GUI:

  On Implementation > Parameters > Synthesize > Additional Options Specify -Fforce_v2k_syntax=minmaxw:sramff_raw_2

Hierarchy of Generated RTL

The following describes the generated RTL files.

File Name	Module Name	Instance Name
simu_stubs/<procname>_dut.v	<procname>_dut	dut
rtl/<procname>_ppa_wrapper	<procname>_ppa_wrapper	ppa_wrapper_0
rtl/<procname>_ppa.v	<procname>_ppa	ppa_<ppa_id>
rtl/<procname>_paw_<pa_id>.v	<procname>_paw_<pa_id>	paw_<pa_id>
rtl/<procname>_pa_<pa_id>.v	<procname>_pa_<pa_id>	pa
rtl/<procname>_pe_<pa_id>.v	<procname>_pe_<pa_id>	pe_<pa_id>

Package Directory Structure

A result of the process initiated by the create_rtl_package command is the packaging of the synthesizable RTL, simulation support files, synthesis support files, and other associated files into a directory.

This RTL package provides a standalone directory containing all the files necessary to support simulation and synthesis of the RTL. The package is kept in a directory named rtl_package, and has appropriately named subdirectories for the various target processes, sources, and files.

This package can be used by a designer or engineering team to easily export their resultant RTL, simulation support files, and RTL synthesis scripts in a concise form for delivery to other receivers, e.g. verification engineer, design engineer, physical design engineer.

The directory is structured as follows:

<implementation>/rtl_package/packaged directory

Directory	Contents
rtl/	Contains the RTL Verilog files representing the Synphony C Compiler generated artifact. The RTL files end with the extension .v, indicating Verilog. The top-level file is named <APP_NAME>_ppa.v. The <APP_NAME> is all in lower-case even if the original APP_NAME had uppercase letters. The files contained within this directory should not be modified.
macrocells/	Contains the underlying macrocell library RTL files. These files end with a .v extension indicating a Verilog library file. The files contained within this directory should not be modified.
scripts/	Contains the scripts to compile and run the Verilog simulation.
cache/	Contains your driver files and data files to test the application. Used for Verilog SV simulation.

Directory	Contents
synth/	Contains a README file and directories for different synthesis tools. Results of synthesis are placed in subdirectories of this directory. The synth/README and synth/toolname/README files provide information about the do_synth executable wrapper script, which can be used to perform synthesis, and example constraint files generated for use with synthesis. In some cases, project (.prj) files are also created for use with synthesis.
pcores/	Contains the netlist for EDK integration, produced during logic synthesis. This directory is created only in the FPGA product.
scc/	Contains specific Synphony C compiler support files (C code wrappers and stubs) for running the C/C++ driver code during RTL transaction-level SV simulation with the source files in the cache/ directory. Makefiles for running SV simulation in the rtl_package are located in the rtl_package/simulation/uvm directory.
scc/scc_dumps/	Repository for all input and output dumps for all tiles as well as transcript files for each tile run (Refer the following definition of transcript file). These per-tile transcript files, name transcript.<NUM1>-<NUM2> will also contain a SUCCESS/FAILED line at the end to indicate success/failure of the associated tile's simulation run.
scc/scc_dumps/transcript.tile.<NUM1>-<NUM2>.	File containing runtime info for tiles in the range NUM1 to NUM2. If a single tile is run, NUM1=NUM2.
simu_stubs/vsim/pli/	Contains supporting C files needed for simulation. These c-files contain function calls and routines used by the simulation stubs, corner-case generation tools, and simulation testbench.
simu_stubs/uvm	Contains all non-synthesizable testbench environments and SystemC files for Verilog SV simulation.
Logs/	The Logs directory contains various logs, status/statistics and flags generated during packaging and simulation compile and run phases.

Directory	Contents
urgReport/	Contains coverage data for VCS code coverage. The dashboard.html file in this directory contains the coverage data for RTL simulation using the VCS tool.
output/	Contains result or output files for golden simulation and the output file for SV simulation. For more information, see "Directory Structure to Support SV Simulation" on page 15-7.
simulation/vsim/	Work directory for Verilog simulation executables.
simulation/uvm	Contains Makefiles and temporary files used for SV simulation from the rtl_package.
simulation/sc_bit	Contains files for systemC bit accurate simulation.
simulation/sc_thread	Contains files for systemC thread accurate simulation.
Reports/	Copy of the <implementation>/Reports. <implementation> is the one from which the current rtl_package is created.
vsim_info	File that contains local environment variables preserving package info; used to convey package-specific environment info for simulation scripts.
VERSION	The Synphony C Compiler tool version used to create the current rtl_package.
transcript_file	File containing various initialization and finalization info, simulation start, stop cycle info in respect to the design-under-test. This file is created only for RTL simulation.
dump_offline.vpd	It is the waveform dump of the last task execution during RTL simulation with the VCS tool.
dump_offline.vcd	It is the waveform dump of the last task execution during RTL simulation
rtl_package/ dump_sv.vpd	It is the waveform dump of the complete SV simulation in VPD format (VCS tool only).
rtl_package/ dump_sv.vcd	It is the waveform dump of the complete SV simulation in VCD format.

Directory	Contents
<appname_PPA>_intf_strl.strl	The Synphony C Compiler tool produces an Esterel interface file for its generated RTL. The app-specific Esterel interface file name is: <appname_PPA>_intf_strl.strl. This file is an interface port-list for the PPA in Esterel format, the following is an example of the file:

```
/** Generated by the Synphony C Compiler tool **/
/** ========================================
Esterel Port interface file
Interface for ppaname PPA block
========================================== **/
interface ppaname_strl_intf:
  input input1;
  output output1;
end interface
interface ppaname_lm_strl_intf:
  input lmem_q0;
  output lmem_do;
end interface
```

Note:

The vsim_info file saves the following app/package characteristics in the variable names: `ASYNC_RESET`, `APPL_NAME`, `CLOCK_PERIOD`, `CFILE_NAME`, `COMBIN_TCAB`, `COMBIN_IMPORTED_TCAB_NAMES`, `DRIVER_FILES`, `FILE_NAME`, `HEADER_FILES`, `IMPORTED_TCAB_NAMES`, `INPUT_FILES`, `OUTPUT_FILES`, `SCC_CLOCK_EDGE`, `SCC_MCELL_EXT`, `SCC_MCELL_PREFIX`, `SCC_RESET_LEVEL`, `SCC_TCAB_MEM_FILES`, `SCC_TCAB_FLAG`, `SCC_TECHLIB_PATH`, `PPA_FILE_NAME`, `SCC_VFILE_EXT`, `TCAB_NAME` and (if specified during the create_rtl_package step), `CCOMPILER_ARGS`, and `SCEXEC_ARGS`. The values of these variables are saved and used by subsequent scripts. Thus, the package is, in a sense, locked to these variables.

To change these settings for the package, (other than `CCOMPILER_ARGS` and `SCEXEC_ARGS`, which can be overridden during compile/run from the package), it is recommended that you re-package the implementation with the new settings and then run `verify -rtl` command.

Directory Structure to Support SV Simulation

The following section describes the Synphony C Compiler RTL Package directory and file structure under the implementation directory to support SV simulation from the RTL package. This will allow the reuse of the rtl_package as the repository for result scc_dumps/tiles/output files and even allows the Synphony C Compiler tool to collapse the number of files that are currently in the implementation directory.

Note that Output files generated for a particular level of simulation will appear in the simulation/<sim_level>/ directory and will be copied or linked to the appropriate output directory for the result compare process.

The file names and directories for outputs generated by SV simulation are given:

```
Directories:
<imp_dir>/rtl_package/rtl_package/output/golden/cache/output/
Files:
rtl_package/output/golden/golden.output
             cache/
                         output/sc_bit/sc_bit_exec.output
                         output/sc_thread/sc_thread_exec.output
                         output/uvm/sv_ref.stdout
rtl_package/simulation/
simulation/                  // work directories for simulation
simulation/uvm/          // rtl-systemc work dir - result files & obj files
simulation/uvm/<resultfiles>
```

For more information on scripts, see the Chapter 17, “Scripts and Utilities.”

Corner Case Simulation Support

The Synphony C Compiler SV simulation supports corner-case verification using perturbation of stream interfaces. Perturbation involves delaying or modifying the values used by the testbench to drive the input signals of the DUT (device under test).

The System Verilog perturbation is specified in a config file. A default config file is generated in the implementation with the testbench. You can find the config file at, rtl_package/simulation/uvm/sv_perturbation.config.

The format of the config file is simple. Every line of the file specifies the perturbation of 1 stream DUT interface driver. Each line is a sequence of values in the following order, separated by whitespace:

driver_class type hold_min_value hold_max_value burst_min_value burst_max_value

Type	What it Implies
driver_class	UVM driver class that drives the stream
type	Keyword. Keyword must be one of, PERTURB_NO or RANDOM.
hold_min_value	Minimum number of cycles to delay the next stream handshake

Type	What it Implies
hold_max_value	Maximum number of cycles to delay the next stream handshake
burst_min_value	Minimum number of cycles to temporarily stop perturbing the stream handshake
burst_max_value	Maximum number of cycles to temporarily stop perturbing the stream handshake

Assertions

Synphony C Compiler-created RTL includes several types of assertions to help verify correct functionality. The Synphony C Compiler tool automatically creates these assertions.and inserts them in appropriate locations
throughout the RTL. You cannot create or modify assertions.

An assertion example is shown.

```
`timescale 1 ns / 10 ps
module vadd_assertions ();
always @(manytoone_memport_select_2_1_w14_16.enable) begin
      if ((manytoone_memport_select_2_1_w14_16.enable != 0) &&
         (^manytoone_memport_select_2_1_w14_16.enable !=1)) begin
          $display ("ERROR at Simtime = %f : ", $time);
          $display ("manytoone_memport_select_2_1_w14_16.enable =

        0x%h", manytoone_memport_select_2_1_w14_16.enable);

           $display ("Please contact your Synopsys Application

                                                Engineer\n\n");

          $finish;

       end

    end

endmodule
```

Monitors and Checkers

A Performance monitor Verilog module is included in the Verilog testbench within the Synphony C Compiler Verilog simulation framework. The monitor assists you with timeout and deadlock detections.

When these conditions are detected, Verilog simulation exits with an appropriate message.

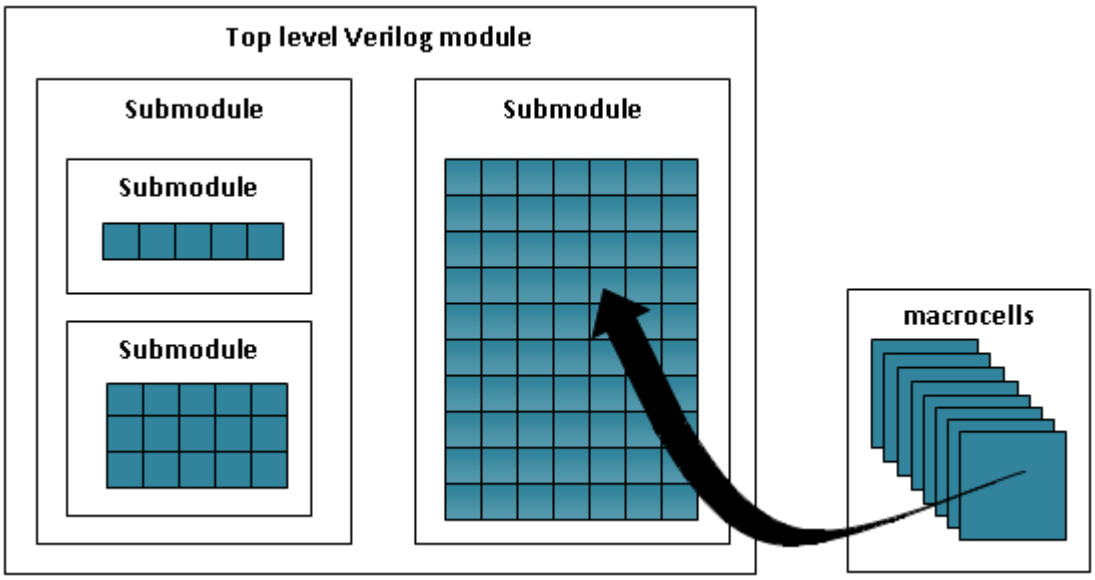
RTL Features

This section describes the following RTL features:

- Overall Hardware Strategy
- Registers
- Clocks
- Resets
- RTL Timing Closure
- Timing Closure and False Timing Paths
- Synthesis Assumptions
- Design-for-Test
- Abort
- Error Signaling
- Autostart
- Infinite Run
- FIFO Hierarchy Priority
- Feature Interactions
- Clock Gating (ASIC Only)
- Memory Arbitration
- FIFO Implementation (FPGA Only)

Overall Hardware Strategy

A Synphony C Compiler generated RTL consists of a small set of Verilog files, which reference a collection of Verilog library files. These library files, called macrocells, implement structured Verilog functionality, and are combined by the Synphony C Compiler tool in such a way as to realize customer functionality. A common hierarchy is shown in the following figure.



Registers

A Synphony C Compiler generated RTL is designed to be synthesized into a fully latchless, flip-flop based design. Currently, the rising edge of the signal clk triggers all flip-flops. Gated clocks can be achieved using appropriate tools such as Synopsys Power Compiler.

Clocks

The generated synthesis scripts treat the clock signal clk as an ideal network. They are not intended to generate a physical clock tree. Clock related variables used in synthesis include period, half-period, uncertainty and transition delay.

Resets

The Synphony C Compiler tool can be configured to generate RTL, which supports either a synchronous or asynchronous reset methodology, using a positively or negatively asserted reset signal. This needs to be selected before the RTL is generated, and cannot be changed afterwards. It is not necessary to enable the PPA to perform a reset operation.

Using a synchronous methodology, reset is entered on the first rising clock edge following the assertion of *reset*. An asynchronous methodology results in reset being entered at the assertion of *reset*, regardless of *clock*. To avoid metastable conditions, you are responsible for ensuring that reset is always a de-asserted synchronous to *clock*.

No guarantees are made as to the state or output of the RTL before the assertion of reset.

Note:
Flip-flops may be reset to either logic “0” or “1”.

RTL Timing Closure

The Synphony C Compiler generated RTL is designed to provide timing closure in the first pass. In certain circumstances, the Synphony C Compiler timing estimates are found too conservative and might lead to higher register cost.

To get the best cost/performance tradeoff, Synphony C Compiler makes more aggressive timing estimates by default. However, this has the risk of not reaching timing closure during logic synthesis in the first pass.

If you encounter such issues in your design, you can do one of the following:

- Re-target the Synphony C Compiler tool to a slightly lower target frequency than you would synthesize the RTL at. You can still synthesize the RTL at the desired frequency by specifying a higher frequency during packaging as follows:

  ```
  create_rtl_package - desiredFrequency
  ```

- You can make Synphony C Compiler timing estimates more conservative by turning off macrocell delay extrapolation. Note that this may lead to slightly higher register cost because the tool is not allowed to chain operators as much as it may otherwise do:

  ```
  setvar -append synthesize_auxopts
   "-Fextrapolate_for_faster_implementations=no"
  ```

- The Synphony C Compiler tool cannot arbitrate a memory without forward path boundary register. It creates more ports and proceeds with a user warning.

Timing Closure and False Timing Paths

Under a certain situation, false timing paths may be created in the Synphony C Compiler synthesizable RTL, which may prevent timing closure by backend synthesis tools. This is when two loops in a multi-loop design access the same internal fast memory.

In this case, the following timing path would be created:

one loop's memory address_out --> addrin of memory --> dataout of memory --> the second loop's memory datain -->

This is a false timing path because the execution of two loops accessing a common memory is sequentialized by the tool. If such a false path is seen, then a false path constraint may be added manually before running the backend synthesis tool.

Synthesis Assumptions

Physical Floorplan

For synthesis using Synopsys Physical Compiler, we assume, but do not require, a square unbounded and unobstructed core area, along with random perimeter I/O. These settings should be modified to meet customer constraints.

Area estimates and other physical parameters are included in the synthesis generated reports.

Power Grid

No power grid assumptions are made. No power grid is used for synthesis.

Design-for-Test

The Synphony C Compiler tool generates no test-specific RTL items and you are free to implement test using your favored methodology.

Several test-related items are discussed, including:

- Asynchronous or synchronous reset methodology
- Flip-flops may be reset to “0” or to “1”
- Not all flip-flops receive “reset”

In ASIC, few of you would have option to place all the registers.

Abort

This feature aborts the execution of all tasks in the PPA. This feature is not present in Synphony C Compiler generated RTL by default. The generation of abort hardware or logic can be turned on through the Command-Line Interface by setting the following implementation parameter:

```
set_params -generate_abort_logic yes
```

Aborting a PPA combinationally stops the currently executing tasks and brings the PPA to a ready state for start of a new task from the next cycle. It is not necessary to enable the PPA to perform an abort operation. If the PPA is built with “auto-start-restart hardware”, then the PPA will automatically start the next task after abort and enable.

Aborting a PPA

The PPA may be aborted by the host in one of the following two ways:

- If the RTL is generated with all control signals exposed as part of the raw hardware interface, then the host aborts the PPA by asserting the abort input port high for 1 cycle.
- If the control signals are mapped to the memory-mapped host interface, then the host aborts the PPA by writing a 1 to the abort PSW bit (PSW bit 6) through the memory-mapped host interface.

You can verify the abort RTL feature using the Synphony C Compiler Verilog simulation and perturbation capabilities. For information about the timing of abort, enable, and error conditions, see Chapter 14, “System Integration and Verification.”

Reset and Abort

Reset	`set_params -reset_data_registers no`	Does not reset multibit (>1-bit) data registers in the RTL and does not reset any data registers in macrocells. However, resets control registers and all 1-bit registers, some of which could be data registers.
Reset	`set_params -reset_data_registers yes`	Resets *all* registers (multibit, 1-bit, data, and control registers)

Abort	`set_params -init_data_registers no`	Does not initialize multibit (>1-bit) data registers in the RTL and does not initialize any data registers in macrocells. However, resets control registers and all 1-bit registers, some of which could be data registers.
Abort	`set_params -init_data_registers yes`	Initializes *all* registers (multibit, 1-bit, data, and control registers) except certain types of data registers in macrocells, such as delays and multicycle multipliers/dividers.

As indicated in the table, reset with `-reset_data_registers no` results in the same behavior as abort with `-init_data_registers no`. However, reset with `-reset_data_registers yes` is different from abort with `-init_data_registers yes`.

Error Signaling

This feature is used to detect and signal errors encountered during hardware execution.

This feature is not present in Synphony C Compiler generated RTL by default. The generation of error signaling logic can be turned on through the Command-Line Interface by issuing the following command prior to issuing the build or synthesize command:

```
setvar synthesize_auxopts "-Fcreate_error_hw=yes"
```

The PPA signals an error by asserting high the PSW error bit (PSW bit 7). This may be observed by the host in one of the following two ways:

- If the RTL has been generated with the err port exposed as part of the raw interface, then the error may be observed through the err output port.
- If the error bit has been mapped to the memory-mapped host interface, then the host might read the PSW error bit through the memory-mapped host interface.

The PSW error bit combines error states from external buses, SRAMs, internal functional units generating error (if any), arbitrators, FIFOs etc. Currently, the following interface protocol errors and memory errors are flagged:

- The host tries to perform a read from or write to transaction to an incorrect address (due to holes in the PPA memory address map). The possible causes for this are:
  - The address does not correspond to any local memory or addressable PPA register.
  - The host addresses a memory with an invalid address value.

  - The host addresses the PSW using the memory-mapped interface while the RTL is generated with all control signals exposed as part of the raw hardware interface.

- More than one request is presented to a memory port in a clock cycle. This may happen if the host tries to access a local memory while the PPA is running.
- A local memory (external SRAM, internal SRAMFF) is addressed outside its valid address range.
- For a given task, a producer loop writes more data into an Inter-Loop Stream FIFO (ILS FIFO) than read by a consumer loop. Similarly, a producer loop writing less data into an ILS FIFO for a task than read by a consumer loop is an error condition.

When an error occurs, all components internal to the PPA hardware are disabled. Error recovery is not supported currently. The hardware state is unpredictable after an error is raised, and the host can read the state elements while the PPA is in this error state. The host cannot force the PPA enable high by writing to the PSW enable bits 4 and 5 (or by raising the enable input port high), it should instead issue a reset or an abort to make the PPA come out of the error state. The PSW error bit returns to 0 when the reset or abort is issued. Upon completion of the reset or abort, the PSW error bit stays at 0 (unless the error generating situation is encountered again), and the PPA is ready to start task execution.

There is no direct way to test the error signaling RTL feature automatically. However, you can manually introduce error conditions during the RTL execution by instrumenting the Verilog testbench or the application.

For information about the timing of abort, enable, and error conditions, see Chapter 14, “System Integration and Verification.”

Autostart

The autostart feature builds the PPA to start and restart without any external control. The PPA starts the first task on reset or abort. If there is no task overlap, then the PAs start the next task after the current task is completed. If task overlap is allowed, then each PA automatically restarts the next task when it is ready to do so.

Autostarting designs simplify integration. More importantly, such designs with task overlap help reduce hardware cost by eliminating start and done controls, and also let the independent PAs run freely, apart from each other by any number of tasks.

Autostarting designs should not have any per-task scalar inputs and outputs or output memories. If there are any `stable` inputs (without hold only) or memories to be initialized in the beginning, then the PPA should be reset in the disabled state, and all initializations should be done and enabled only after the memory and the input are initialized.

Autostart designs are difficult to verify automatically within the Synphony C Compiler standard testbench framework. Typically the standard testbench times out due to lack of input data (when using streams) and matches partial output data.

Infinite Run

This feature makes the PPA hardware run indefinitely after a task is started on it. This is useful when the hardware needs to continuously process incoming data.

For this feature to be present in the Synphony C Compiler generated RTL, the `build` command must be run with the `-infinite_run_ppa` option or the `set_params` command must be run with the `-infinite_run_ppa yes` option. When set, every PA is built to run infinitely.

Abort and error signaling exhibit the expected behavior, that is, the PPA will stop further execution until a new task is started by the host.

The verification recommendations for auto-start designs are applicable to infinite run designs too.

Localization

Synphony C Compiler builds hierarchical designs in a bottom-up fashion. From the current design, it externalizes memories, FIFOs, registers, and TCABs to enable them to be shared across other blocks at higher levels. However, when the sharing opportunity ceases to exist, the externalized component might remain at a hierarchy needlessly far from the blocks that access it.

Localization is a step in synthesis that moves memories, FIFOs, registers, and TCABs into the appropriate physical hierarchy closer to the blocks that access them. This process improves the RTL readability. More importantly, it improves area and timing of “no ungroup” backend flows and enhances the success rate of formal verification.

The process of localization can change the interface and internals of the imported TCABs in their container. This is because components that were externalized out of a TCAB at its build time could be relocated back into the TCAB in the container implementation.

For example, consider a TCAB “B” externalized out of TCAB “A.” Consider an instance of TCAB “A” in a PPA. If there is no sharing opportunity of TCAB B in PPA, then the localization process relocates TCAB “B” back into the TCAB “A” instance. This process changes the internals and interface of TCAB “A.”

To preserve the internals and interface of a TCAB, you can use the following command at TCAB build time:

```
set_params -dont_touch yes
```

This treats the TCAB being built as `dont_touch` wherever it is imported. Neither the internals nor the interface of a `dont_touch` TCAB can be modified after the build. Marking TCAB “T” as `dont_touch` would only import its interface into the container and therefore no internal information of TCAB “T” would be available in the container’s reports.

By default, localization of internal components of a design happens wherever possible. It happens by default in Channels, PPA, and `dont_touch` TCAB implementations.

FIFO Hierarchy Priority

The PPA hardware contains stream-buffer FIFOs for flow-controlled communications, including user-defined FIFOs from the program and other FIFOs used for control and data handshakes. You can use the `-fifo_hierarchy` implementation parameter to specify the order of the physical locations of the FIFOs in the design, as follows:

```
set_params -fifo_hierarchy top:src:dest (default)
set_params -fifo_hierarchy top:dest:src
set_params -fifo_hierarchy src:top:dest
set_params -fifo_hierarchy src:dest:top
set_params -fifo_hierarchy dest:top:src
set_params -fifo_hierarchy dest:src:top
```

For example, `src:dest:top` means that the tool first tries to place the FIFO in the source (writer) entity; or if there are multiple writers, it tries to place the FIFO in the destination (reader) entity; or if there are multiple readers, it places the FIFO in the top (PPA) entity.

Feature Interactions

Autostart, infinite run, and loop early exits may be combined in several ways to create hardware that behaves in a manner that is consistent with the required mode of operation for the application. The combinational behavior of these features is described. In each of these cases, abort and error signaling exhibit the expected behavior.

Autostart and Infinite run

Upon reset, the PPA self-starts when enabled, and after it starts, it runs forever.

Infinite Run and Loop Early Exit

Every time a task is started, the PPA runs indefinitely until the early exit condition is triggered for each of the loops in the PPA.

Autostart and Loop Early Exit

Upon reset, the PPA self-starts when enabled, and after it is started, runs until the task is completed (either normal completion or due to early exit), and restarts automatically.

Autostart, Infinite Run and Loop Early Exit

Upon reset, the PPA self-starts when enabled, and after it is started, runs indefinitely until the early exit condition is triggered for each of the loops in the PPA, and restarts automatically upon termination.

Clock Gating (ASIC Only)

Clock gating is a technique for power reduction. Two types of clock gating are possible.

- PA_level clock gating
- TCAB_level clock gating

A coarse-grained PA_level clock gating shuts off entire processing blocks to reduce power consumption in the design. TCAB-level clock gating provides fine-grained control at the TCAB level for additional power savings. No manual analysis is needed to decide “when” a block can be turned off - the tool does this automatically. No manual work is needed to insert the clock enable generation logic into the RTL - the tool does this automatically as well. The clock enable output ports can be used to generate the clocks for the gated blocks using the clock gating cells outside the PPA.

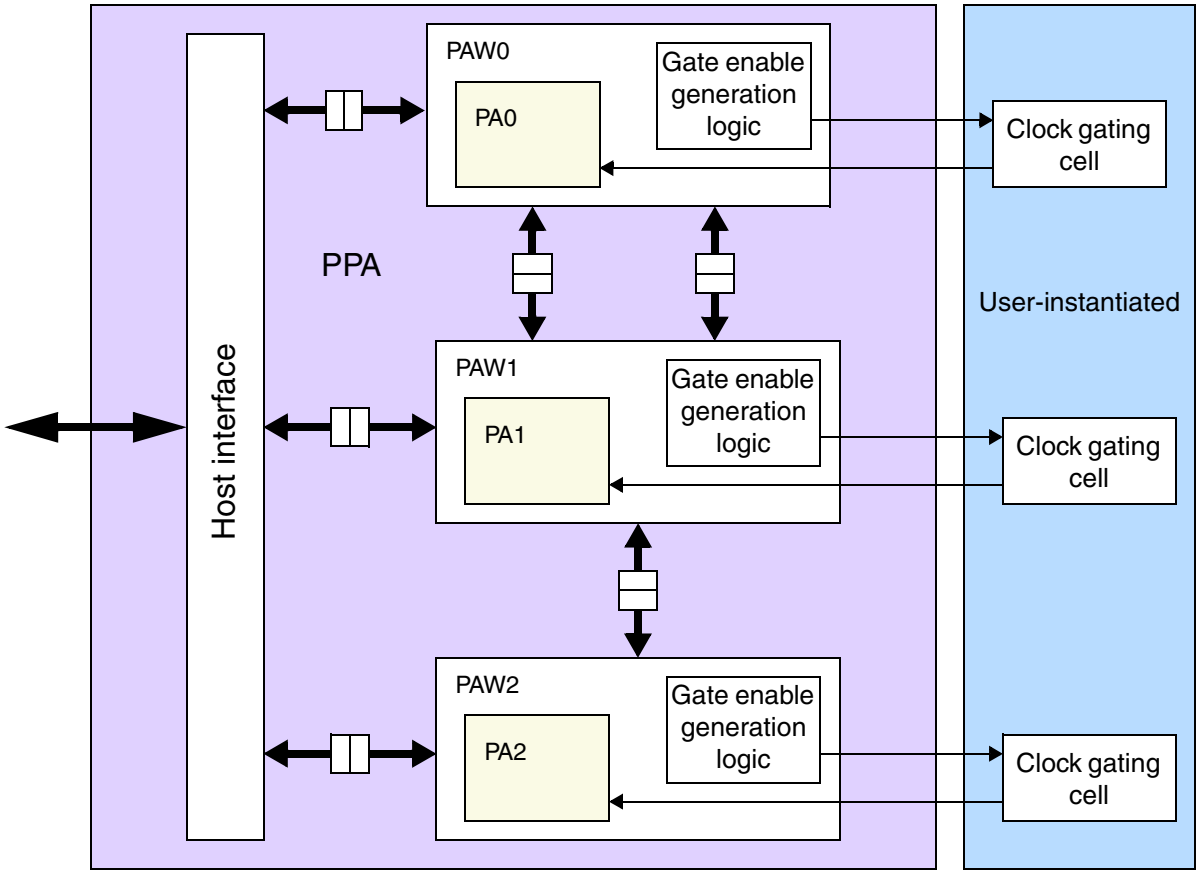
A PA or TCAB is considered “active” when it has a task running on it. PA is additionally considered “active” when it has to register a scalar input. Clock gate enables are asserted whenever the gated block is “active”. Note that, a block is considered “active” even when it is stalling during task execution. Therefore, clock gate enables are not deasserted when the block stalls during a task.

Synchronously reset blocks need the clock to be ON during reset. However, this is not enforced by clock gate enable generated from PPA. In other words, clock gate enable is not asserted for the sake of synchronous reset. You have to explicitly turn on the clock during synchronous reset.

Clock gating ports on the PPA is a pair and are: *<clk_to_block_i, and clk_gate_enable_of_block_i>*.

Clock gate enable is active high. The blocks may include PAs and TCABs under the PA hierarchy. Output port *clk_gate_enable_of_block_i* should be used outside the PPA to gate the clock for the *block_i*. The gated clock is fed as *clk_to_block_i*.

The following figure depicts the clock gating feature within a PPA with 3 PAs.



The Synphony C Compiler function verification methodology is extended to automatically check the sequence of clock gating. This is supported in offline as well as SV Verilog simulations, and works with both coarse-grained as well as fine-grained clock gating. In addition, gated clock waveforms are generated automatically in the VCD and FSDB output files, enabling automatic measurement of the power savings in downstream power analysis tools.

The following table shows different ports generated for different combinations of TCAB imports, where TCABs or containers are differently clock gated. It is for an example of a PPA importing TCAB A, which in turn imports TCAB B.

Clock gating			Notes
PPA	**TCAB A**	**TCAB B**	
Off	Off	On	The (clock enable, clock) pair of TCAB B is ported out of the PPA interface
Off	On	Off	The (clock enable, clock) pair of TCAB A is ported out of the PPA interface
Off	On	On	The (clock enable, clock) pair of TCAB A is ported out of the PPA interface and the (clock enable, clock) pair of TCAB B is also ported out of the PPA interface
On	Off	Off	The (clock enable, clock) pair of PAs is ported out of the PPA interface
On	Off	On	The (clock enable, clock) pair of PAs is ported out of PPA interface and the (clock enable, clock) pair of TCAB B is also ported out of the PPA interface
On	On	Off	The (clock enable, clock) pair of PAs is ported out of the PPA interface and the (clock enable, clock) pair of TCAB A is ported out of the PPA interface
On	On	On	The (clock enable, clock) pair of PAs is ported out of the PPA interface, the (clock enable, clock) pair of TCAB A is ported out of the PPA interface and the (clock enable, clock) pair of TCAB B is ported out of the PPA interface.

Memory Arbitration

The Synphony C Compiler tool can arbitrate SRAM memory ports. You can specify the maximum number of ports to a memory with the `read_write_ports` pragma, and the tool tries to meet these requirements. If it cannot meet the limit using analysis and scheduling heuristics, the resulting memory requestors are arbitrated.

The Synphony C Compiler tool implements multi-stall domain arbitration. Stall domains are PAs and independent stall domain TCABs inside the PAs. Consider a m:n arbitrator where m > n (m requests and n ports for a memory). Whenever the number of requests is greater than n, the arbitrator grants a maximum of n requests per cycle. Requestors that are not granted access to memory are stalled appropriately to maintain the latency requirements.

The arbitrator is called multi-stall domain, because it only stalls the stall-domains of the requestors that were not granted access to memory.

The SCC tool uses a fixed priority arbitrator. Priority{arbitrator_port_0} > Priority{arbitrator_port_1}. It assumes a default priority of stall domains (PAs and independent stall domain TCABs) for wiring to the arbitrator.

Multi-Stall Domain Arbitration

Event simulation models multi-stall domain arbitration. You can use it to specify porting constraints as well as arbitration priorities on memories and determine the porting that gives acceptable performance. After finding the acceptable porting, build the design by specifying the porting constraint with the `read_write_ports` pragma and the priority with `synthesize_auxopts`. For more information, see "read_write_ports Pragma" on page 7-58 and "Synthesize Auxiliary Options" on page 3-31.

Priority Order

The Synphony C Compiler tool uses the order in which the PAs and independent stall domain TCABs appear in the program as the priority order.

The following design has Loops L0, L1, L2, L3 and independent stall domain TCABs T1, T2, T3, T4 and T5, with a call graph as shown:

```
L0
     T1
L1
     T2
          T4
     T1
     T3
L2
     T5 (shared with L3.T5)
L3

     T5 (shared with L2.T5)
```

The default priority order is as shown:

L0 > L0.T1 > L1 > L1.T2 > L1.T2.T4 > L1.T1 > L1.T3 > L2 > L2.T5 > L3 > L3.T5

You can change the priority order using `synthesize_auxopts`. For more information, see "Synthesize Auxiliary Options" on page 3-31.

FIFO Implementation (FPGA Only)

You can map stream buffer (FIFO) storage to registers or block RAMs. For more informations, refer to the `set_params -fifo_synthesis_threshold` command and the "internal_blockram Pragma" on page 7-32.

STARC Rule Support

The Semiconductor Technology Academic Research Center (STARC) is an organization that develops SoC design technologies. It establishes various rules for chip design in its style guides. The SCC ASIC product supports the following STARC rules for the Synopsys Leda RTL checking tool:

Leda Rule	Description
VER_1_1_1_2A	Use only alphanumeric characters and underscores (_).
VER_1_1_1_2B	The first character must be a letter of the alphabet.
VER_1_1_1_3A	Do not use Verilog or VHDL keywords.
VER_1_1_1_6A	The top-level module name must not end with an underscore (_).
VER_1_1_1_6B	The top-level module name must not use consecutive underscores (_).
VER_1_1_1_6C	The top-level port name must not end with an underscore (_).
VER_1_1_1_6D	The top-level port name must not use consecutive underscores (_).
VER_1_1_1_9B	The top-level module name must not mix upper case and lowercase characters.
VER_1_1_1_9D	The top-level port name must not mix upper-case and lower-case characters. The only case where the SCC tool does not support this rule is for the OCP hostLM interface (set_params -interface_protocol ocp).
VER_1_1_2_1A	The module name must be at least 2 characters long.
VER_1_1_2_1B	The module name must not exceed 32 characters (Note 1).
VER_1_1_2_1C	The instance name must be at least 2 characters long.
VER_1_1_2_1D	The instance name must not exceed 32 characters (Note 2).

Leda Rule	Description
VER_1_1_3_3A	The signal names, port names, parameter names, define names and function names must be at least 2 characters long.
VER_1_1_3_3B	The signal names, port names, parameter names, define names and function names must not exceed 40 characters in length (Note 3).
VER_1_1_4_6	Do not propagate parameters through ports.
VER_1_3_1_3	Do not use asynchronous set or reset pins for anything other than initial reset.
VER_1_3_1_5	Do not use the //synopsys_sync_set_reset Design Compiler directive.
VER_1_6_4_1	Do not include glue logic at top level.
VER_2_2_3_1	Do not mix blocking assignments (=) and non-blocking assignments (<=) in combinational always blocks.
VER_2_2_3_2	Do not assign over the same signal using a non-blocking assignment for combinational circuits.
VER_2_2_3_3	Do not assign over the same signal in an always construct for sequential circuits.
VER_2_3_6_1	Do not mix flip-flops with and without asynchronous resets/sets/ loads in the same always construct.
VER_2_8_1_5	Do not use full_case directives in case statements.
VER_2_8_5_1	Do not use parallel_case directives in case statements.

- Note 1:

  To produce RTL that obeys this rule, use the following setting:

  ```
  setvar -append synthesize_auxopts "-Fmax_module_length=32"
  ```

- Note 2:

  To produce RTL that obeys this rule, use the following setting:

  ```
  setvar -append synthesize_auxopts "-Fmax_instance_length=32"
  ```

- Note 3:

  To produce RTL that obeys this rule, use the following setting:

  ```
  setvar -append synthesize_auxopts "-Fmax_signal_length=40
     -Fmax_port_length=40"
  ```

  You can use the settings described with any other length (>= 32) according to your design requirements. For example:

```
setvar -append synthesize_auxopts "-Fmax_signal_length=64
   -Fmax_port_length=64 -Fmax_module_length=48
   -Fmax_instance_length=54"
```

16

Hardware Architecture

This chapter describes the Synphony C Compiler tool generated hardware elements. For more information, see the following:

- Introduction
- Pipeline of Processing Arrays
- Tightly Coupled Accelerator Block
- Processing Array (PA)
- Core Processor
- Interconnect
- Selector Wiring at the Input of the Functional Unit
- Registers
- Merge Registers
- Predicate Merger

Introduction

This chapter explains the hardware schema, which is the underlying template that the Synphony C Compiler tool uses for generating RTL from C/C++ code.

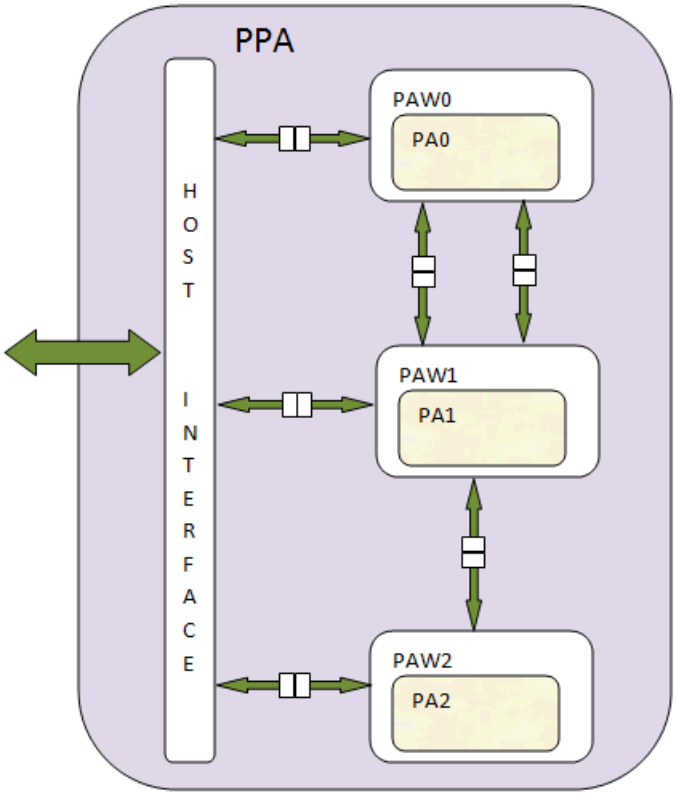
The core components of the schema are described in this chapter. Note that, as many of the schema diagrams are only conceptual, the component name or port names seen in the actual RTL for a specific example may be different from what is shown in this chapter. Further, the schema components may not appear as separate modules; instead, they may appear flattened inside the parent entity.

Pipeline of Processing Arrays

The Synphony C Compiler tool creates the top-level module called the Pipeline of Processing Arrays (PPA). The PPA consists of hardware blocks called PAs (processing arrays) which correspond to a loop nest or code block in the original application, and other hardware components, such as FIFOs, arbiters, etc. to control their interaction.

All hardware elements inside a PA operate in lock-step (except for Independent stall domain TCABs) and thus are said to be in the same stall domain. Each PA is in a different stall domain, so they can get ahead or fall behind one another to the extent allowed by data dependencies and flow control. In a nutshell, a PPA is an elastic pipeline of PAs.

The following figure shows the components that can be present in the PPA layer.



- PA: This is the hierarchy representing a loop nest. The PA is implemented as a pa-pe hierarchy in RTL. pe is the processing element. pa is an array of processing elements (currently the tool synthesizes only one processor).

  Each PA is a pipelined iterative compute engine. Most of the cost of a PA is associated with the actual computation it performs. The code mapped to each PA pipeline operates in lockstep. When a PA stalls (due to a flow control or arbitration event), its entire pipeline stalls together (except for independent stall domain TCABs). Note that at the PPA level, the operation is different - separate PAs are free to advance and stall independently.

  A PA can communicate data with other PAs or external world using internal memories, external memories, scalars, or FIFOs. Internal memories are flip-flop based synthesizable SRAMs in the design. External Memories could be SRAMs or BRAMs that the user supplies for connecting to the PPA. FIFOs are inter-loop streams to exchange stream data between PAs. Scalars are wires or registers.

- PAW: This is a wrapper around the PA and contains handshake logic to communicate with other PAs or the host. The handshake logic involves data for scalar communications

and controls for shared stream and memory communications. The handshake between any two entities can be optionally intercepted by stream buffers (FIFOs).

- The Synphony C compiler tool generates memory path logic and stream path logic for memories and streams respectively:
  - Memory path logic: This is the path from PAs or host to the internal memories or memory ports on the PPA. This involves multiplexing (for mutex accessors), arbitration (for potentially concurrent accessors) and multi-buffering control logic.
  - Stream path logic: This is the path from PAs to the inter PA stream FIFOs or stream ports on the PPA.
- The scalars that are used between the PPA and host processor have at least one register associated with it unless marked as `stable` by the user (explained).
- Input scalars that are only read and not written by the PPA can be marked `stable` if the input value is the same for each set of pipelined tasks. `stable` scalars have no registers inside the PAs, avoiding redundancy. An optional hold register can be present at the PPA layer so that the PPA can hold the constant value across tasks.
- Host Interface: The Synphony C Compiler tool makes several state elements visible through the host interface. The Synphony C Compiler tool generates a *Hostlm address decoder* which is a variable length decoder used to decode the host processor specified address into the PPA commands.
- Glue logic: There is also some glue logic for various PPA operations such as exchanging shared scalars, PPA status bits, and abort or error (if specified). The Synphony C Compiler PPA template includes the ability for the PPA to detect errors (such as out-of-bounds access to a memory) and signal it through the PSW. It also includes a feature to abort the operation of a PPA task in the middle of the run. By default, these features are disabled. Typically, the additional cost of abort or error circuitry is a few hundred gates. For more information on options to turn on these features, see "Abort" on page 15-14 and "Error Signaling" on page 15-15.

Each PA in a PPA exists in its own stall domain. A stall domain represents a block that may advance or stall independently of other blocks in the design. Separating stall domains enables the PPA to maximize throughput in the presence of flow control, arbitration and other factors. To support stalling, load units for external memories include a FIFO or *replay buffer*. The depth of the replay buffer is related to the latency of the load operation including its boundary registers.

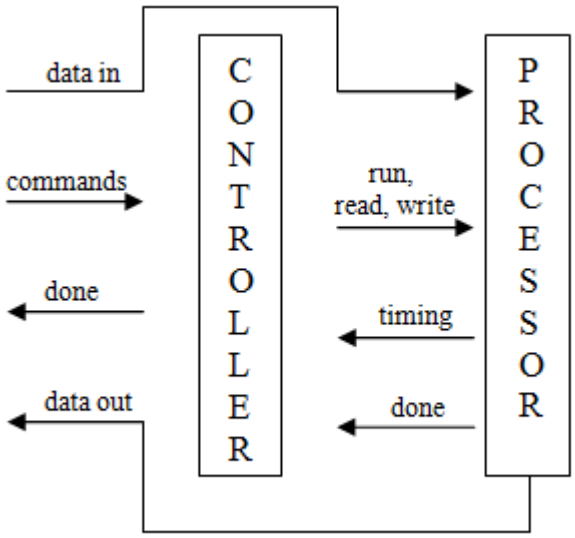
Tightly Coupled Accelerator Block

Tightly Coupled Accelerator Block (TCAB) is a Synphony C Compiler tool-created, top-level RTL module (named tcab). This always contains one paw-pa-pe representing a PA.

The PA inside the TCAB is similar to the PA described for PPAs. An independent stall domain TCAB can also have a latency synchronizer to synchronize the operation of the TCAB and its parent.

Processing Array (PA)

The processing array (PA) is the RTL module that represents a loop nest. The following figure shows a schematic view of a processing array.



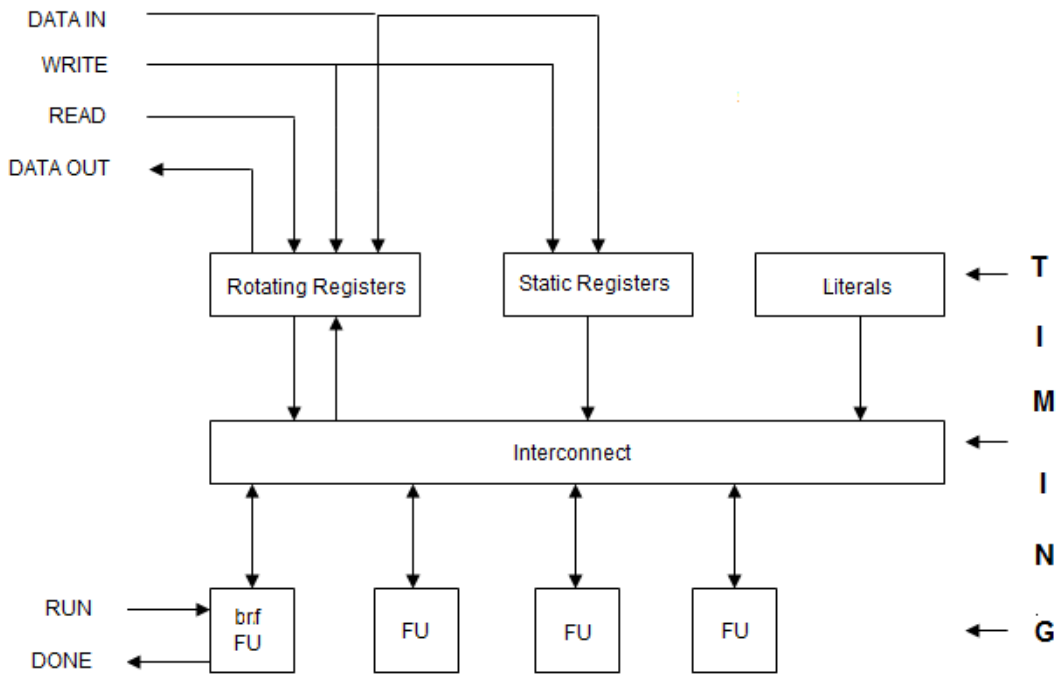
The PA has two components:

- Controller: It converts the commands to control signals. Outputs II number of phase buses one corresponding to each phase. A design with II = n has 'n' phases or timeslots ranging from 0 to n-1. Also, we have, current execution phase = current_cycle_number / II. It also generates first and last iteration time signals for input or output reasons.
- Processor: This is the core processor representing the PU.

We describe the schema of the core processor in the following sections.

Core Processor

The following figure shows the schematic for the core processor.



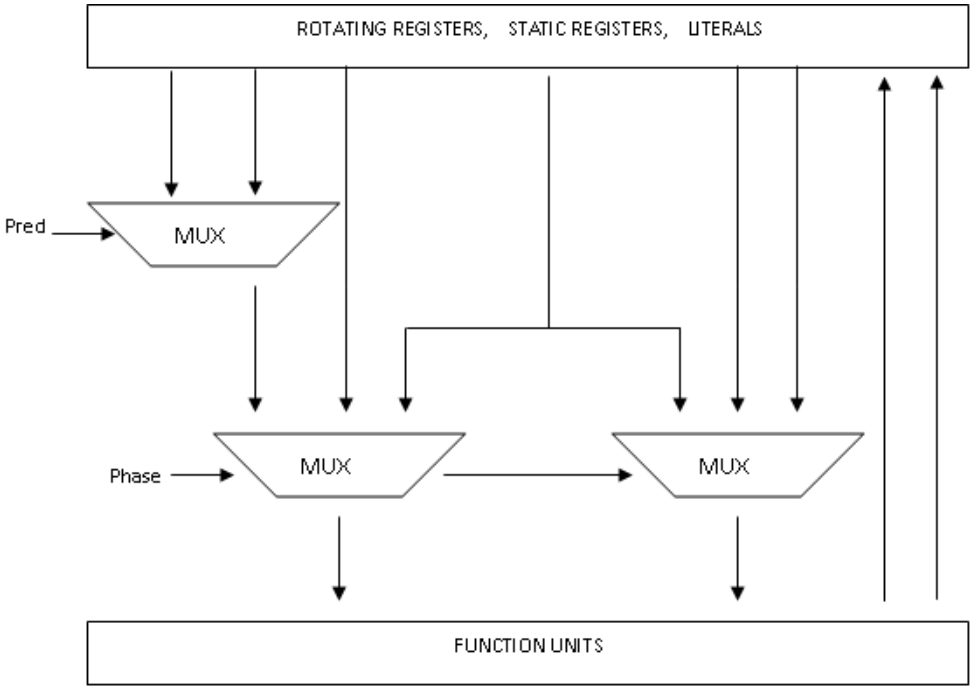
The core processor consists of three components:

- Registers and Literals: Registers are the storage elements. Literals in the program are represented as Verilog constants. Registers are of two types: static and rotating (explained later in the chapter).
- Functional Units (FU): These are the components executing the operations.
- Interconnect: These contain selectors, sign extenders etc. to interface registers to functional units.

We continue further discussion starting with a description of the interconnect component.

Interconnect

The following figure shows the interconnect schema.



The selectors (the term multiplexer (MUX) is also used interchangeably with the term selector) at the functional unit (FU) inputs select one of many variables depending on the predicate and timeslot (phase). The variables which are input to the functional units may be from the registers or from other functional units bypassing the registers in case, the output of the previous FU has been chained to the input of the current FU.

As mentioned earlier, a design with initiation interval II repeats 'II' phases (phase 0 to II-1). The modulo scheduler does the resource and time bindings for each operation. The result is that, for a given operation, the modulo schedule maps it to the FU on which it executes and the timeslot (or phase) i.e. the modulo scheduler produces the following map:
predicate: operation --> (fu, phase).

Any number of operations can be assigned to an FU. However, at each phase or timeslot, only one operation can actually execute on the FU. It is also possible that at some phase, no operation is scheduled on an FU.

The interconnect schema is designed to select the right input to the FU based on the phase and the predicate. We will describe the selector connections at the input of an FU in more detail in the following section.

Selector Wiring at the Input of the Functional Unit

As far as the selector wiring in the datapath from one FU to another FU is concerned, there are a few scenarios to consider:

- FU chaining: In this case the output of an FU producing a data value is used at the input of the FU consuming the data value without an intervening register in the datapath between the two. This is the case when the operation producing the data value and the corresponding operation consuming the data value are scheduled in the same cycle in the unrolled schedule.
- Non-chaining case: In this case, there would be one or more registers in the datapath from the output of the data-producing FU to the input of the data consuming FU. This is the case when the operation producing the data value and the corresponding operation consuming the data value are scheduled in different cycles in the unrolled schedule.
- II sharing: As mentioned in the previous section, an FU may be shared by operations across II phases, with one operation executing in each phase.
- Predicated FU sharing: Two or more operations scheduled in the same cycle of the unrolled modulo schedule but under mutually exclusive predicates may all execute on the same FU, in the same phase.
- Predicated data merge: Two or more operations executing under mutually exclusive predicates may write into the same data variable which is read as source input by the data-consuming FU.

It may be noted that the selectors in the RTL are one-hot selectors which makes it easy for verification. An n: 1 selector has n pairs of inputs with each pair being a (data, enable) pair.

As the name suggests, the data input is the connection for passing the data value to the input of the FU. When there are more than one data inputs, the input for which its corresponding enable input to the selector is high is the one that is passed to the input of the data-consuming FU.

Each enable input to a selector may be viewed to be generated as follows:

enable = phasebus & pred.

The phasebus is the input that selects one phase from among the II phases. For example, phasebus0 selects the zeroth phase from the II phases. The pred input is present if one or more of the following conditions is true:

- The variable being read by the FU comes from a chained source operation.
- The operation for which the variable is being read by the FU is involved in predicated FU sharing with a mutually exclusive operation scheduled in the same cycle of the unrolled schedule.

Registers

We now turn our attention to the second constituent of the core processor, i.e. the register. There are different kinds of registers:

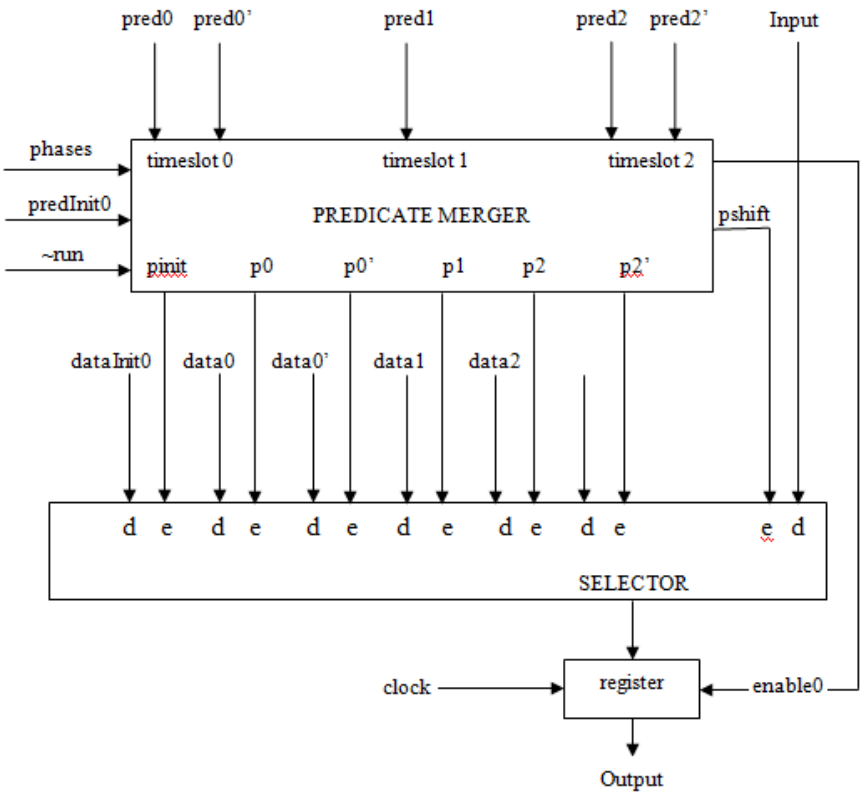
- Static Register: This is a single merge register. (The merge register is explained later). Static variables are stored in static registers.
- Rotating Register: This is a chain of merge registers which store multiple instances (or values) of the same variable originating from different iterations. Rotating registers are implemented in two different ways in Synphony C Compiler tool:
  - Single Variable Rotating register: One rotating register chain per variable. Used for designs with initiation interval (II) of one. The input values are stored in the steady state portions and wrapped around.
  - Shiftq: One rotating register chain for one or more variables. Used for designs with initiation interval (II) greater than one to share the registers among variables. The input values are stored in a separate prefix queue ahead of the steady state shiftq.

Merge Registers

The merge register is a special register used in Synphony C Compiler to handle multiple writes to the register. The merge register contains a component called predicate merger followed by a selector and the actual register. Predicate merger accepts various predicates for the writes and produces the select signals to be used for selecting the data input to the actual register and to enable the signal to be wired to the enable input of the actual register.

All the defining operations have write data buses to the merge register's selector. These data buses are selected based on the predicate merger's output select signals.

The merge register schema is explained next for a merge register in II = 3 design. In time slot 0, there are two predicated writes under mutually exclusive predicates pred0 and pred0'. In timeslot 1, there is one predicated write under predicate pred1. In timeslot 2, there are two predicated writes under mutually exclusive predicates pred2 and pred2'.



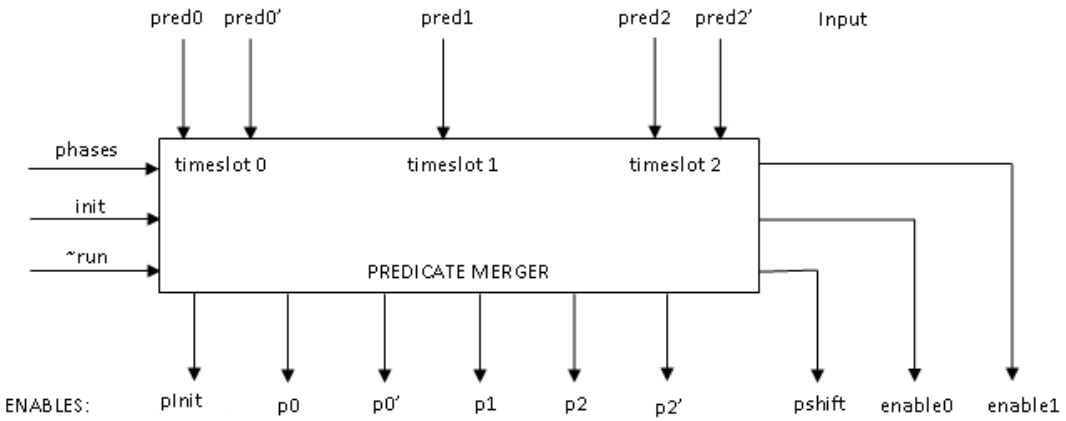
merge register

The types of predicates output by the predicate merger are - init predicate (pinit) used to initialize with the input value (dataInit0), steady state predicates (p0, p1 etc.) used to select the steady state writes and the shift (pshift) predicate used to select the default input. It also outputs an enable signal to be used for enabling the actual register.

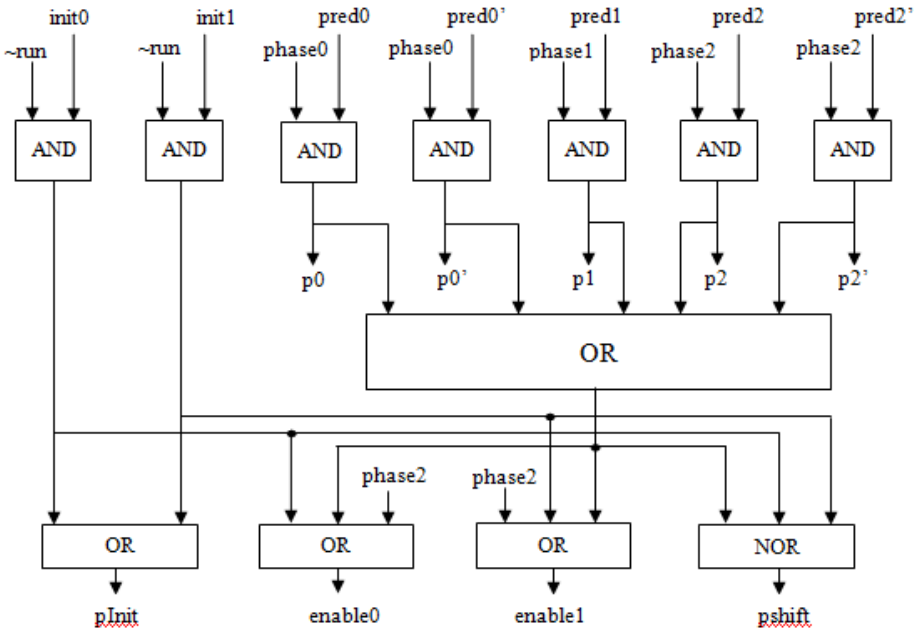
Predicate Merger

The predicate merger is used to merge the predicates at a given timeslot and at different timeslots. Under init predicate (predInit) and not running condition (~run), it sets the pInit high and this is used for input initialization. For register width > 32-bits, 2 registers are created in the merge register schema. reg [31:0] is initialized using init0 and reg [width-1:32] is initialized based on init1. p0, p0', p1, p2, p2' are set based on phase and the input predicates. These are used to decide the data to be selected when ~run=0. pshift is set high if none of the other predicates is high and the it is timeslot 2 (II-1).

Enable0 is high if any of the non-init predicates is high or the init0 is high during notrun or if it is time slot 2 (II-1). enable1 is high if any of the non-init predicates is high or the init1 is high during notrun or it is timeslot 2 (II-1).



The predicate merger schema block diagram



Implementation of the predicate merger schema

17

Scripts and Utilities

This chapter describes the following:

- Introduction
- Script for Debussy and nLint Tools
- Script for Verplex Conformal LEC Tool
- Script for Synopsys Virsim Tool

Introduction

A number of useful utilities are provided in the implementation/rtl_package/scripts directory. These scripts should be run in the Linux shell from the implementation/rtl_package/ directory. You may have to customize these scripts to fit your site or personal PATHs and tools environment.

Script:	Description:
run_debussy.sh run_verdi.sh	Shell scripts or templates to run the Novas Debussy™, Verdi, and nLint™ tools. Optionally creates a Debussy or Verdi-compatible `fsdb` file from the dump.vcd file, (which is created if verify was run with -dump_vcd). Requires `$NOVAS_HOME` environment variable.
run_lec.sh	Shell script that creates a dofile and runs the Cadence Verplex or Conformal LEC equivalence checking tool. Default paths are set to compare a Synphony C Compiler-generated RTL versus synthesized gate-level Verilog model. Requires `$VERPLEX_HOME` environment variable.
run_virsim.sh	Shell script or template to run the Synopsys Virsim™ interactive simulation debug and waveform viewing tool. Requires `$VCS_HOME` environment variable.

Script for Debussy and nLint Tools

```
$> scripts/run_debussy.sh -h

Usage: run_debussy.sh [ -d | -n ] [ -g ] [ -f ]
Runs Debussy debug and waveform tool or nLint tool: -d (debussy) | -n
(nlint)
use -h for help and usage info.

Feature                          Option
--------------                   -------------------
debussy                          -d (Default = debussy)
nlint                            -n nlint
gui(nlint)                       -g -n  (Default = nogui)
convert vcd2fsbd                 -f (Assumes a dump.vcd file exists
                                 in current-dir)
                                 creates file = "dump.fsdb"
```

```
Top=${APPL_NAME}_testbench.v     -t (Default Top =
                                 ${APPL_NAME}_testbench.v)
Top=${APPL_NAME}_ppa.v           -m (Default Top =
                                 ${APPL_NAME}_testbench.v)
```

```
$> scripts/run_verdi.sh -h

Usage: run_verdi.sh [ -v (verdi) -d (debussy) -n (nlint) ] [ -g ] [ -f ]
[ -command_line_ options ]
use -h for help and usage info.


Feature                          Option
--------------                   -------------------
debussy                          -d (Default = debussy)
nlint                            -n nlint
verdi                            -v verdi
gui (nlint)                      -g -n  (Default = nogui)
convert vcd2fsbd                 -f (Assumes a dump.vcd file exists
                                 in current-dir)
                                 creates file = "dump.fsdb"
sv dumpfile                     -o Use the file=dump_sv.vcd-->
                                 dump_sv.fsdb
dumpfile name                    -N <filename>: Use <filename> as
                                 the dumpfile name
Top=${PPA_FILE_NAME}_ppa.v       -m (Default Top =
                                 ${PPA_FILE_NAME}_testbench.v)
Top =${FILE_NAME}_testbench.v)   -t (Default Top  =
                                 ${FILE_NAME}_testbench.v)
Cmd-line options                 -c <Novas tool cmd-line options>


Optionally creates a Debussy/Verdi-compatible "fsdb" file from the
"dump.vcd" file, (which is created if verify was run with "-dump_vcd").
```

Script for Verplex Conformal LEC Tool

```
$> scripts/run_lec.sh -h
Usage: run_lec.sh [-g <path_to/gate_verilog_file>]
                  [-l <path_to/gate_library_file>]
                  [-y <path_to/library_dir>]
```

Abort Points

If Conformal LEC hits abort points, it may be necessary to use some additional analysis and compare options to allow LEC to achieve a comparison.

After running the script run_lec.sh, you can edit the application-specific LEC dofile, rtl_package/lec/runlec.do and add the following commands:

```
> analyze datapath -merge -verbose -share  // NOTE:(requires lec
                                           // -ultra)
> analyze abort -verbose
> analyze abort -compare
> set compare effort complete
> compare -single
```

Note that the analyze datapath option will require the Ultra option for Conformal LEC.

If you have access to the Conformal LEC Ultra option, then you can invoke Conformal LEC with the command line:

```
> lec -ultra <other opts>
```

Also, one could alternatively use the -qual30 option for analyze datapath as shown:

```
> analyze datapath -merge -verbose -share -qual30
```

The option flag `-qual 30` causes all abort points with success rate over 30 to use the analyze datapath algorithm (default is 50%).

The command `compare -single` causes points to be compared individually rather than in groups.

The compare effort could alternatively be set to super, but complete is higher.

Script for Synopsys Virsim Tool

```
$> scripts/run_virsim.sh -h

 Usage: run_virsim.sh[ -t ] [ -h ]  [ -c ]
use -h for help and usage info.

Feature                     Option
--------------            -------------------
convert vcd2vpd             -c
Top=${APP_NAME}_ppa.v       -t (Default Top =
                            ${APPL_NAME}_testbench.v)
```

18

ASIC Macrocell Characterization

This chapter describes the commands and files used to run ASIC characterization with the Novo tool, which is included as part of the Synphony C Compiler ASIC product.

- Overview of Synphony C Compiler Novo
- Novo Command Syntax
- Characterization Configuration File
- Novo Input and Output Files

For information about using the tool, see “Characterizing ASIC Macrocells” in the *Synphony C Compiler User Guide*.

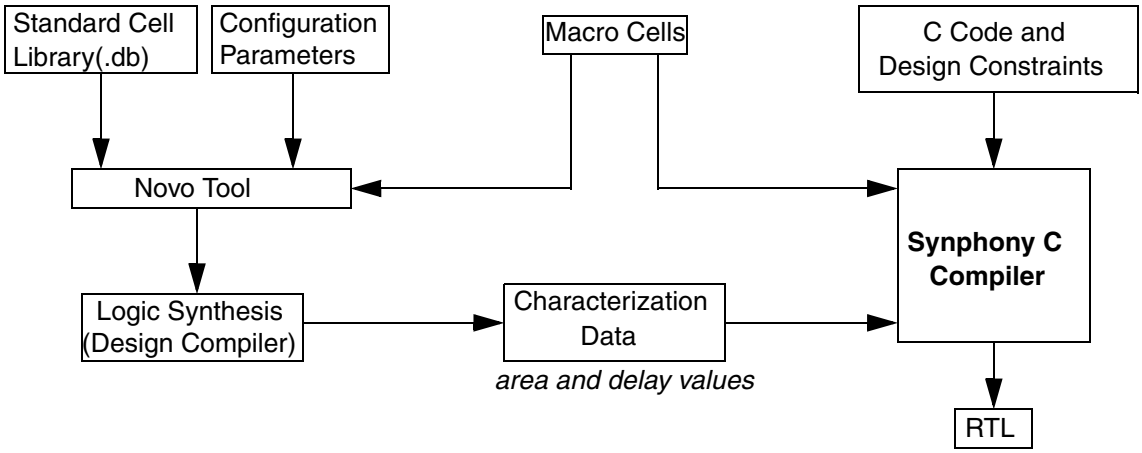
Overview of Synphony C Compiler Novo

The ASIC version of the Synphony C Compiler product includes the Novo tool for characterizing macrocells. Macrocells are basic SCC building blocks, like adders and multipliers. The Novo tool is fully integrated into the SCC ASIC software, but has a separate executable. It characterizes macrocells for a specified standard cell library, and produces a set of area-delay trade-off points that the SCC tool uses for timing-driven optimal-area synthesis.

The characterization process uses the Synopsys Design Compiler logic synthesis tool to profile the timing and area characteristics of the macrocells. The profile is propagated to the Synphony C Compiler tool, so that it produces area-gate estimates that correlate with the logic synthesis results, and accurate RTL that meets timing with minimal area.

The Tool Flow

The following diagram provides a graphic overview of the interactions between the Synphony C Compiler and Novo tools during the characterization process:



Novo Macrocell Characterization

For subset of parameter values for each macrocell, the Novo tool first generates the infrastructure scripts for characterizing each data point to be characterized. The data points are area, delay, and clock frequency. It then launches Design Compiler jobs on LSF queues. It assesses the number of licenses and available resources, and manages the jobs, taking care to load the LSF queues optimally. It generates logs and raw area delay data files for each macrocells.

The tool then post-processes the data. It parses and verifies all raw data files for correctness, and then generates datasheets and the final machine-readable data that is used by the Synphony C Compiler tool.

Novo Command Syntax

This is the syntax to run the Novo tool and characterize ASIC timing data:

```
synphonycc_novo
  -config-file fileName
  -remote-method method
  -synthesis-licenses number
  [-characterization]
  [-o remote_options="moreLSF/SGEoptions"]
  [-macrocell-list-file fileName]
  [-verbose level] [-help][-version] [-check_env] [-install-test]
```

-config-file *fileName*	Specifies the configuration file to be used for characterization. The configuration file must be created by modifying the template configuration file provided in docs/template, because the template file is revised in each release. See "Characterization Configuration File" on page 18-4 for details about the parameters.
-remote-method *method*	Specifies the method of parallelization: `lsf`, `sge`, or `local`.
-synthesis-licenses *number*	Specifies the number of Design Compiler licenses to use in parallel.
-characterization	Runs characterization.
-o remote-options= "*moreLSF/SGEoptions*"	Specifies any additional arguments for the LSF/SGE queue to launch jobs, in accordance with the local user environment. For example -o bsub_additional_options="-R rhe32" specifies that the jobs should be only run on 32-bit Linux machines.
-macrocell-list-file *fileName*	Optional argument that specifies a file with a list of macrocells to be characterized. The default file (characterization_macrocell.list) is located in *installDir*/scc/macrocells, and includes all the macrocells available in the current release. To specify a subset of the cells, copy the characterization_macrocell.list file to your working directory and edit it to contain just the cells you need.

-verbose *level*	Sets the level of reporting. The level can range from 1 to 10, with 10 producing the most detailed output logs. Levels 1-3 are usually sufficient for most users.
-help	Displays the command line usage syntax.
-version	Displays the version of the tool.
-check_env	Prints the environments settings and license settings before running characterization.
-install-test	Verifies the availability of the logic synthesis tool and LSF/SGE installations. This is an optional argument to be used with the `-config-file` option to run the installation test.

Characterization Configuration File

The configuration file defines various parameters required for ASIC characterization. Copy and modify the template file provided in *SCCInstallDir*/docs/ config.template. Then specify the file with the `-config` option to the `synphonycc_novo` command when you run characterization. The following sections provide additional details about the configuration file:

- Sample Configuration File
- Configuration File Parameters

Sample Configuration File

The tool includes a sample configuration file (*SCCinstallDir*/docs/config.template) that you can modify for your design. The following is an example of a configuration file:

```
##########################################################
pico_macrocell_directory /tools/<scc_installDir>/pico/macrocells
pico_characterization_directory
pico_characterization_preserve_scratchdirs 1
pico_techlib_name tcbn90ghpwc
pico_techlib_db /tools/synth_libs/synopsys_tsmc/tcbn90ghp/tcbn90ghpwc.db
pico_techlib_lib /tools/synth_libs/synopsys_tsmc/tcbn90ghp/
tcbn90ghpwc.lib
pico_techlib_verilog /tools/synth_libs/synopsys_tsmc/tcbn90ghp/
tcbn90ghp.v
pico_clk_transition               0.2
pico_clk_uncertainty              0.2
pico_driver_cell                  ND2D2
pico_driver_pin                   ZN
pico_default_load                 0.06
pico_techlib_gate_scaler          5.1
pico_synthesis_tool               synopsys_asic
pico_compile_type                 compile_ultra
##########################################################
```

Configuration File Parameters

The following table contains an alphabetical list of the parameters in the configuration file. All parameters are mandatory, unless noted otherwise.

Parameters	Description
pico_characterization_directory	Required parameter. Top-level directory in which the Novo tool runs. The tool creates sub-directories here to store intermediate data and final results.
pico_characterization_preserve_scratchdirs 1 | 0	Determines whether the tool saves intermediate data and log files. 1 preserve the intermediate data directories and log files. This setting is helpful for debugging characterization failures. 0 removes temporary directories and log files.

Parameters	Description
pico_clk_transition pico_clk_uncertainty	Required parameters. Define the clock transition time and clock uncertainty, respectively. These parameters are used to characterize the macrocells. The values are expressed in the technology library units, and must be based on what you see as the typical clock transition time and uncertainty in their designs for that technology library. For example: pico_clk_transition 0.2 pico_clk_uncertainty 0.2 This parameter is written in the SYNTHESIS_PARAMS file, and passed to the Synphony C Compiler tool.
pico_compile_type compile_ultra | compile	Optional parameter that specifies the compile type for the Design Compiler tool to use for characterization. Set the compile type as follows: compile_ultra - DC Ultra compile - DC Expert The default setting is compile_ultra.
pico_default_load	Required parameter. Defines the capacitive load on macrocell outputs during characterization. The value is in technology library units. Set it based on what you see as the typical capacitive load on logic gates in designs for that technology library. We recommend setting it to 10 medium-sized NAND gate input capacitances. For example: pico_default_load 0.05 This parameter is written in the TECHLIB_CORRELATION and SYNTHESIS_PARAMS files, and passed to the Synphony C Compiler tool.
pico_dont_use_standard_cell	Specifies standard cells that should be avoided for characterization. The settings are propagated through the SYNTHESIS_PARAMS file, so RTL synthesis does not use them either. pico_dont_use_standard_cell artisan_library/*DFFS* artisan_library/PREICG*

Parameters	Description
pico_driver_cell pico_driver_pin	Optional parameters that specify the driver cell and pin for macrocell input ports during characterization. Set the values based on what you see as typical drive strengths for that technology library. We recommend picking a medium-sized NAND gate as the driver cell. For example: pico_driver_cell ND2D2 pico_driver_pin ZN
pico_macrocell_directory	Required parameter. Directory location of the macrocells to be characterized. This is usually inside the Synphony C Compiler installation directory. For example: pico_macrocell_directory /tools/*sccInstallDir*/pico/macrocells
pico_synthesis_tool synopsys_asic	Required parameter. Specifies the logic synthesis tool to be used for characterization. Currently the only supported tool is Synopsys Design Compiler (synopsys_asic), and the parameter is set to it by default.
pico_techlib_cap_unit	Specifies the capacitance unit for the technology library when you specify pico_techlib_lib. If it is not specified, the characterization tool determines the capacitance unit from the technology library .lib file. This is an example of this parameter: pico_techlib_cap_unit pF This parameter is written in the SYNTHESIS_PARAMS file, and passed to the Synphony C Compiler tool.
pico_techlib_db	Required parameter. Specifies the technology library .db, files to be used during macrocell characterization. For example: pico_techlib_db /tools/synth_libs/synopsys_tsmc/tcbn90ghp/tcbn90ghpwc.db After characterization, the parameter is passed to the Synphony C Compiler tool in the SYNTHESIS_PARAMS file.

Parameters	Description
pico_techlib_lib pico_techlib_name pico_techlib_pdb pico_techlib_verilog	Specify the technology library, and the fully-qualified path name to the relevant .lib, .pdb and .v files to be used during macrocell characterization. Specify multiple parameters on the same line, separated by spaces. Here are examples: pico_techlib_name tcbn90ghpwc pico_techlib_lib /tools/synth_libs/synopsys_tsmc/tcbn90ghp/tcbn90ghpwc.lib pico_techlib_verilog /tools/synth_libs/synopsys_tsmc/tcbn90ghp/tcbn90ghp.v After characterization, the parameters are passed to the Synphony C Compiler tool in the SYNTHESIS_PARAMS file.
pico_techlib_gate_scaler	Required parameter. Specifies the um^2 per gate for the technology library. Typically, you set this to the um^2 area of a minimally sized NAND gate. For example: pico_techlib_gate_scaler 2.8 This parameter is passed to the Synphony C Compiler tool, which uses the value to report design cost in gates.
pico_techlib_opcond	Optional parameter that describes the operating condition of the technology library. For example: pico_techlib_opcond WCCOM
pico_techlib_time_unit	Specifies the time unit for the technology library when you specify pico_techlib_lib. If it is not specified, the Novo characterization tool determines the time unit from the technology library .lib file. For example: pico_techlib_time_unit ns This parameter is written in the SYNTHESIS_PARAMS file, and passed to the Synphony C Compiler tool.
pico_wireload_model pico_wireload_mode pico_wireload_selection_group	Optional parameters that specify the wireload model, wireload mode, and the wireload selection group respectively.If you do not define them, the logic synthesis tool uses the default values from the technology library. : pico_wireload_model ZeroWireload This parameter is written in the SYNTHESIS_PARAMS file, and passed to the Synphony C Compiler tool.

Novo Input and Output Files

The following describe the input files needed to characterize a library, and the output files generated by the Novo tool with the characterized data.

Input Files

This is a list of the input files required for Novo ASIC characterization:

Macrocells	*installDir*/scc/macrocells Contains the behavioral Verilog macrocell implementations for the cells to be characterized. The characterization_parameter_configs file in this directory contains data for the various configurations that were characterized.
Configuration template file	*SCCinstallDir*/docs/config.template Defines key parameters for the characterization process, like the Synphony C Compiler path and technology library path. The file is a template, so copy this file to your characterization area, and modify the data as needed for your design. See "Characterization Configuration File" on page 18-4 for a description of the parameters and an example of the file.
Standard cell library	This is the standard cell library to be used for the design.

Output Files

During the characterization process, the Novo tool generates these output files:

- AreaDelay.pdf

  This datasheet is located in the characterization working directory, and provides an easy way to evaluate the characterization results.

- Raw data files

- Picodata files

  The picodata files are generated from the raw data files for the cells that were successfully characterized. These machine-readable files are used as input for application synthesis in the Synphony C Compiler tool. The Synphony C Compiler software uses the timing and area information from the picodata files to produce timing and area estimates.

It is a good practice to copy the picodata directory to a central location, that is regularly backed up, to avoid losing data.

The following table describes the picodata files:

*.lmdes2	*characterization_dir*/picodata/ Machine-readable files for input to the SCC tool.
SYNTHESIS_ PARMS	*characterization_dir*/picodata/ Preserves all technology library and synthesis environment settings that were used for characterization. The Synphony C Compiler tool uses it in the generation of RTL synthesis scripts with the create_rtl_package command. This file ensures perfect synchronization between the characterized library and later logic synthesis runs.
TECHLIB_CORR ELATION	*characterization_dir*/picodata/ Produced for internal use. Do not modify this file. The Novo tool creates a temporary scratch directory for each configuration that is characterized. If you want them to be automatically deleted when characterization is complete, set characterization_preserve_scratchdirs to 0 in the configuration file. See "Intermediate Files and Directories" on page 18-10 for descriptions of the directories.

- Log files

  There are individual log files for the characterization of each macrocell. The logs track progress and record errors for failed characterizations. The progress logs for LSF runs and synthesis tool runs are in separate logs, and will be explained in a later section.

The following table lists the output files and their locations:

Intermediate Files and Directories

For each configuration that is characterized, the tool creates a temporary directory named <characterization>.<LSFquename>. This directory contains files that are deleted when characterization is complete, if `characterization_preserve_scratchdirs` is set to 0 in the configuration file.

In addition, the tool generates raw data files and log files, the majority of which you do not have to use unless you are debugging. The following table lists the raw data files and the most important log files:

Raw data	*characterization_dir*/rawdata/ Contains area delay information for the different clock frequencies that were used during the macrocell characterization.
CHARACTERIZATION_TRACKER (Log file)	*characterization_dir*/logs/ Generated after the characterization process starts, and is continuously updated with details like running host, execution host, total running time, number of successful configurations, number of failures, the running time for each configuration, and so on. Periodically check these details to monitor the progress, and take corrective actions if there is a failure.
CHARACTERIZATION_TRACKER_SUMMARY (Log file)	*characterization_dir/* This file contains few important details obtained from 'CHARACTERIZATION_TRACKER' such as running host, total running time, total pass and total failures. This file helps to get a quick look at the characterization status without too many details and it can be found in <pico_characterization_directory>/.
logs/characterization.log	Lists problems found during the characterization stages
logs/*cellname.config*	Lists problems related to macrocell characterization with the synthesis tool.
characterization_dir/cellName.config/directories	Contains snapshots of the synthesis runs and intermediate characterization raw data.

Characterization Directory Structure

The following directory structure shows where the main directories and files are located:

```
characterization_working_directory
   AreaDelay.pdf
   CHARACTER_TRACKER_SUMMARY
   picodata
      *.lmdes2
      SYNTH_PARMS
      TECHLIB_CORRELATION
rawdata
logs
   CHARACTER_TRACKER
macrocells
   <macrocellFiles>
```

19

DSP Mapping for FPGAs

Digital Signal Processing (DSP) units are custom blocks embedded in the FPGA fabric for performing DSP-style functions such as multiply-accumulate and complex multiply. When you use a set of DSP API functions provided with Synphony C Compiler, the tool maps your C++ code into the DSP blocks of the Altera Stratix V and Arria V FPGA family or Xilinx Virtex-4, Virtex-5, Virtex-6, Virtex-7, Atrix-7, Kintex-7, Spartan6-l, Spartan-6, Spartan-3a DSP, Spartan-3e, and Zynq-7000 FPGA families, thereby implementing DSP functions with high efficiency and low cost.

This chapter covers the following topics:

- Overview of DSP Mapping
- DSP Resource Sharing
- Altera DSP Mapping
- Xilinx DSP Mapping
- APIs for Commonly Used Functions

Overview of DSP Mapping

Digital Signal Processing (DSP) units are custom blocks embedded in the FPGA fabric for performing DSP-style functions such as multiply-accumulate and complex multiply. When you use a set of DSP API functions provided with Synphony C Compiler, the tool maps your code into the DSP blocks of the target FPGA device. The efficient mapping of DSP functions can help you meet your area and timing requirements.

The Synphony C Compiler tool provides a class and a set of DSP functions for the target FPGA device. A macrocell implements the DSP functions of a particular DSP unit. The term “DSP block” in this document refers to a DSP unit embedded in the FPGA fabric of the device.

The DSP mapping feature of Synphony C Compiler offers the following major features:

- Supports the Altera V Series FPGAs, Stratix V and Arria V.
- Supports the Xilinx DSP48E primitive used in the Xilinx Virtex-5 family and the DSP48E1 primitive used in the Virtex-6, Artix-7, Kintex-7, Virtex-7, Spartan and Zynq-7000 FPGA families.
- Provides a user Application Programming Interface (API), a set of C++ functions to control how functions are mapped into DSP blocks.

Why Use the DSP Mapping API?

Efficient DSP mapping is critical to meet the your area and timing requirements. In the absence of the DSP API for the target FPGA device, the mapping of complex functions to the DSP blocks can be unpredictable. Minor changes in code in and around multipliers can generate less than optimal results for the following reasons:

- To get the optimum DSP inference, the RTL must be written in a particular style. Minor changes in schedule of operations due to changes in either the code or constraints can cause RTL synthesis tools to use less than optimal DSP mapping.
- The RTL generated by Synphony C Compiler is largely structural. RTL synthesis tools sometimes are unable to infer the optimum DSP mapping, so that only the multipliers, and not the surrounding logic, get mapped into the DSP block.

As a result, the RTL synthesis tools might implement much of logic outside the DSP block, leading to the waste of DSP resources. The logic not absorbed into the DSP fabric consumes additional resources outside the FPGA DSP block and also contributes to the delay of timing-critical paths.

The FPGA vendors provide certain primitives (Altera megafunctions and Xilinx primitives) that give greater control over the amount of logic that can be mapped into a DSP block. This

capability takes better advantage of the DSP block resources, reduces the logic utilization of the FPGA, and improves the critical path timing around the DSP blocks.

What Does the DSP Mapping API Do?

Synphony C Compiler provides a DSP mapping API in the form of a scc_dsp class and a set of member functions. Using the API, Synphony C Compiler translates your code into the right primitives in Verilog to result in better DSP mapping. The API does the following:

- Allows Synphony C Compiler to create FPGA-specific instantiations of DSP-units, thereby preventing or reducing back-end tool inference
- Provides a mechanism for you to directly control how your algorithms are mapped to DSP units

In other words, the DSP mapping API gives you finer control over the configuration of the DSP blocks and the operations that are performed on those blocks at different times from C++ source code itself.

If you use this API with the supported technology libraries, the mapping to DSP blocks is optimal. On the other hand, when you use this API with an ASIC product or an unsupported FPGA device, the member function are inlined and structural RTL for that logic is generated in place of DSP primitives. DSP mapping is left to the RTL synthesis tool's inference capabilities.

For example, consider the following code segment, which performs a complex multiplication:

```
void complexMultiply( int *p_real, int *p_imag, int a_real,
  int a_imag, int b_real, int b_imag) {
      *p_imag = (a_real * b_imag) + (a_imag * b_real) ;
      *p_real = (a_real * b_real) - (a_imag * b_imag) ;
}
```

Without the API, the Synphony C Compiler results after RTL synthesis often tend to map only a single multiplier per DSP block and also keep all the adder logic outside of the DSP block. On the other hand, by using the API, all of the multipliers can be mapped to a single DSP block, along with the supported addition/subtraction operations.

To use the API on supported FPGA devices, you can rewrite this design as follows.

```
#include "scc.h"
using namespace scc::altera;
...
void complexMultiply( int *p_real, int *p_imag, int a_real,
  int a_imag, int b_real, int b_imag) {
    scc_dsp dsp_a;
    scc_dsp dsp_c;
```

```
    *p_real = dsp_a.calc(MULT_SUB,a_real,b_imag,0,a_imag,b_real,0) ;
    *p_imag = dsp_c.calc(MULT_ADD,a_imag,b_imag,0,a_real,b_real,0) ;
}
```

Synphony C Compiler then generates RTL with the right primitives so that the RTL synthesis tools can map the two multipliers and the addition/subtraction operations into one DSP.

Synphony C Compiler maps the complete data path into one DSP block, so any sharing of this datapath leads to even better QoR.

DSP Mapping Usage Overview

1. You must include the following statement in the files where either the DSP blocks are instantiated or where the wrapper functions are used:

   ```
   #include "scc.h"
   ```

2. Use one of the following namespaces, depending on the target FPGA device:

   ```
   using namespace scc::altera;
   using namespace scc::xilinx;
   ```

3. There are a few different ways in which DSP inference can be used. This can be explained with the following example.

   ```
   e = (a + b) * c +d;
   ```

   - Declare an instance of the DSP class and call the calc method on the object to perform a function:

     ```
     scc_dsp<PREADDER_INPUT_ADD, REG_INPUT_OUTPUT>dsp_a;
     e = dsp_a.calc(MULT_ADD,a,b,c,d,0,1) ;
     ```

   - You can also use the scc_dsp class operator to accomplish the same functionality:

     ```
     scc_dsp<PREADDER_INPUT_ADD, REG_INPUT_OUTPUT>dsp_a;
     e = dsp_a(MULT_ADD,a,b,c,d,0,1);
     ```

   - Some wrapper functions have been provided along with the class; each such function hides the complexity of the DSP mapping class.

     ```
     e = add_mult_add(a,d,c,b);
     ```

For detailed usage instructions, see “Altera DSP Mapping” on page 19-8 or “Xilinx DSP Mapping” on page 19-19.

Verilog Simulation

The RTL generated by Synphony C Compiler contains DSP megafunctions or primitives to represent the functionality of the Altera or Xilinx DSP block.

To perform Verilog simulation with Altera FPGAs, the Altera Quartus synthesis tool must be available and the QUARTUS_PATH environment variable must point to the Quartus installation directory.

To perform Verilog simulation with Xilinx FPGAs, the Xilinx ISE tool must be available and the XILINX environment variable must point to the Xilinx installation directory.

Limitations

The DSP mapping feature of Synphony C Compiler has the following limitations:

- Automatic generation of DSP primitives from generic C/C++ code is not supported. You must use the DSP functions of the API.
- This feature does not provide support for ANSI C. Since the API is based on C++, you must at least rename your files to have C++ extensions.
- Sharing of DSP blocks happens only when the template parameters of individual API instances are same. The only exception is when accumulation mode is used. In that case, the hardware generated has a persistent state across multiple calls to the hardware. So different DSP instances are not shared on the same DSP block.

  For example, the following API instances cannot be shared on the same DSP block due to different modes of pipelined registers, which would change the latency of the individual instances:

  ```
  scc_dsp<PREADDER_SIMPLE, REG_INFER>dsp_a;
  scc_dsp<PREADDER_SIMPLE, REG_INPUT_OUTPUT>dsp_c;
  ```

- Only the modes described in this document are supported in the Altera DSP primitive.
- The DSP primitives have the capability to sample different inputs on different time slots. This is similar to TCABs with input_sample_time or resource_first_use_time set as infer. So, similar to the TCABs, simulation using verify -rtl is not supported in designs with the DSP mapping API.
- Width analysis does not happen across the DSP operations. Therefore, it is recommended that you explicitly specify the width of the inputs and outputs of the different DSP blocks.

Recommendations

The DSP mapping feature of Synphony C Compiler produces better results when you follow these recommendations:

- Explicitly specify the bit sizes of the inputs and outputs of a DSP block to accommodate overflow and precision loss.
- By default, the REG_INFER mode might not successfully infer the correct I/O register implementation. To ensure proper register usage, explicitly specify the RegMode as REG_INPUT, REG_OUTPUT, or REG_INPUT_OUTPUT.

DSP Resource Sharing

Synphony C Compiler shares DSP operations on the same DSP block where the template parameters of the DSP object are identical (with the exception of the accumulation modes). The opcodes passed to each call do not impact the sharing.

For example, consider the following example of a complex multiplication.

```
typedef struct {
int i;
#pragma bitsize i 16
int q;
#pragma bitsize q 16

} Complex;
Complex res, a,b;
#pragma bitsize res.i 18
#pragma bitsize res.q 18
...
      res.i = (a.i * b.q) + (a.q * b.i) ;
      res.q = (a.i * b.i) - (a.q * b.q) ;
...
```

This design can be coded using the DSP mapping API in the following two forms.

Design 1: Only REG_INFER register mode is used in both the instances

```
scc_dsp<PREADDER_SIMPLE > dsp_a;
scc_dsp<PREADDER_SIMPLE > dsp_c;
res.r = dsp_a(MULT_SUB,a.r,b.i,0,a.i,b.r,0) ;
res.i = dsp_c(MULT_ADD,a.i,b.i,0,a.r,b.r,0) ;
```

Design 2: Mix of REG_INPUT and REG_OUTPUT modes used

```
scc_dsp<PREADDER_SIMPLE, REG_INPUT> dsp_a;
scc_dsp<PREADDER_SIMPLE, REG_OUTPUT> dsp_c;
res.r = dsp_a(MULT_SUB,a.r,b.i,0,a.i,b.r,0) ;
res.i = dsp_c(MULT_ADD,a.i,b.i,0,a.r,b .r,0) ;
```

If both the designs are run at II=2, then Synphony C Compiler can share the DSP blocks in Design 1, even though the DSP blocks have MULT_ADD and MULT_SUB functionality, but it cannot do so in Design 2 because the template parameters differ in Design 2.

The only exception to this rule is that sharing cannot be done when accumulation is enabled. In that case, each DSP object contains an accumulation register, and sharing two such instances on a single DSP block would be invalid, so each DSP object is mapped to a separate DSP block. The following example demonstrates this principle.

```
#define SIZE 10
typedef {
    long long i; long long q;
#pragma bitsize i 34
#pragma bitsize q 34
} cmplx;

cmplx complex_multiply_accumulate(cmplx x[SIZE], cmplx window[SIZE]) {

    cmplx result;
#pragma bitsize window.i 18
#pragma bitsize window.q 18
#pragma bitsize x.i 16
#pragma bitsize x.q 16

// Initialize result to 0.
result.i = 0;
result.q = 0;

// SCC DSP instances which accumulate the i and q fields separately.
scc_dsp<PREADDER_SIMPLE, REG_INPUT_OUTPUT, SYSTOLIC_NO, ACCUM_SINGLE >
dsp_a;
scc_dsp<PREADDER_SIMPLE, REG_INPUT_OUTPUT, SYSTOLIC_NO, ACCUM_SINGLE >
dsp_b;

#pragma unroll
  for (int i=0; i<SIZE; i++){ // Unrolled loop
      cmplx c= x[i];
      if (i==0) {
       result.i = dsp_a(MULT_SUB_ACCUM_CONST,c.i,window[i].i,0,
                  c.q,window[i].q);
       result.q = dsp_b(MULT_ADD_ACCUM_CONST,c.i,window[i].q,0,
                  c.q,window[i].i);
      } else {
       result.i = dsp_a(MULT_SUB_ACCUM,c.i,window[i].i,0,
```

```
                    c.q,window[i].q);
        result.q = dsp_b(MULT_ADD_ACCUM,c.i,window[i].q,0,
                    c.q,window[i].i);
      }
  }
return result;
}
```

The foregoing example has two DSP blocks with the same template parameters. But these DSP instances cannot be shared in the DSP API at II=2 because that would corrupt the state of the accumulation on each of the {i,q} datapaths. When targeting the Arria V, you can use the double accumulation mode instead to get a single DSP block that automatically accumulates both the {i,q} components internally, as shown in the following example.

```
// SCC DSP instances which accumulate the i and q fields separately.
scc_dsp<PREADDER_SIMPLE, REG_INPUT_OUTPUT, SYSTOLIC_NO, ACCUM_DOUBLE >
dsp_a;

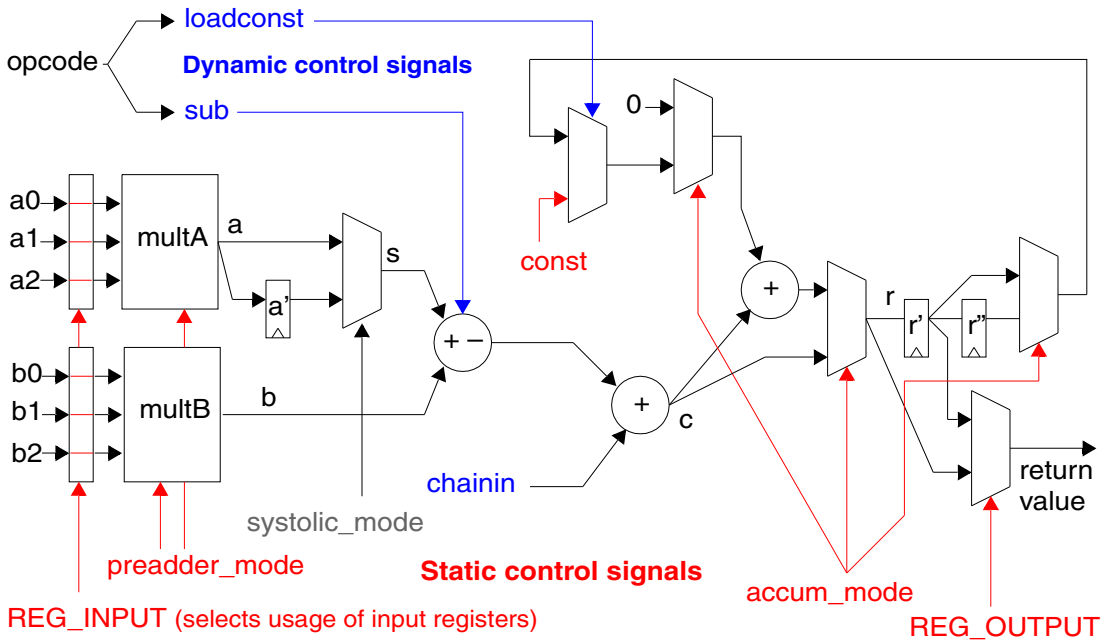
#pragma unroll
  for (int i=0; i<SIZE; i++){ // Unrolled loop
      cmplx c= x[i];
      if (i==0) {
       result.i = dsp_a(MULT_SUB_ACCUM_CONST,c.i,window[i].i,0,
                  c.q,window[i].q);
       result.q = dsp_a(MULT_ADD_ACCUM_CONST,c.i,window[i].q,0,
                  c.q,window[i].i);
      } else {
       result.i = dsp_a(MULT_SUB_ACCUM,c.i,window[i].i,0,
                  c.q,window[i].q);
       result.q = dsp_a(MULT_ADD_ACCUM,c.i,window[i].q,0,
                  c.q,window[i].i);
      }
return result;
  }
```

Altera DSP Mapping

Synphony C Compiler provides a class which can be used to instantiate DSP objects corresponding to DSP blocks in Altera V Series FPGAs, Straitix V and Arria V.

The following block diagram shows the supported features in the Altera DSP block architecture.

Figure 19-1 Altera DSP Block Diagram



Note:

In the current release of Synphony C Compiler, the `systolic_mode` signal is not used and the systolic mode is not available.

The dynamic control signals are shown in blue and the static control signals are shown in red. The static properties are set only once and represent the properties of the inferred DSP block. The dynamic controls are for controlling the operation that are performed by the individual blocks at different call points.

You can have different objects created with different static controls in the same code. Only DSP APIs with same static controls can be shared on the same DSP block.

Static Control

The preadder consists of blocks multA and multB, each of which can be used to perform addition, subtraction, and multiplication with a value stored in ROM. The multA block has three inputs, designated a0, a1, and a2, and the multB block similarly has three inputs.

The preadder has the following features:

- ROM that can be programmed using the API, described later.
- Preadder configuration – The multA and multB blocks can be configured to perform the operations indicated in the following table.

Table 19-1 Preadder Configuration

Configuration setting	MultA (a)	MultB (b)
PREADDER_SIMPLE	a = a0 * a1	b = b0 * b1
PREADDER_COEF_ADD	a = (a0 + a1) * ROMA[a2]	b = (b0 + b1) * ROMB[b2]
PREADDER_COEF_SUB	a = (a0 - a1) * ROMA[a2]	b = (b0 – b1) * ROMB[b2]
PREADDER_INPUT_ADD	a = (a0 + a1) * a2	b = (b0 + b1) * b2
PREADDER_INPUT_SUB	a = (a0 - a1) * a2	b = (b0 - b1) * b2
PREADDER_SQUARE_ADD	a = (a0 + a1) * (a0 + a1)	b = (b0 + b1) * (b0 + b1)
PREADDER_SQUARE_SUB	a = (a0 - a1) * (a0 - a1)	b = (b0 - b1) * (b0 - b1)
PREADDER_CONSTANT	a = a0 * ROMA[a2]	b = b0 * ROMB[b2]

- Post multiplier add/subtract operation – You can dynamically add or subtract the results of the multiplier blocks by using the opcode, which is a dynamic setting, unlike the preadder configuration.
- Chainin adder – You can cascade the result of one DSP block into another by using the chainout adder. This chainin can be used only to cascade the result of a DSP operation and not any other variable. It is an error condition otherwise.
- Accumulation logic – The DSP block is capable of accumulating results across multiple invocations.

The following table describes the accumulator mode configuration settings.

Table 19-2 AccumMode Settings

AccumMode	**Functionality**
ACCUM_NONE	The DSP block does not accumulate the results across different operations.

Table 19-2 AccumMode Settings (Continued)

AccumMode	Functionality
ACCUM_SINGLE	The DSP block accumulates results from every cycle.
ACCUM_DOUBLE	The DSP block accumulates two results from operations on every alternate cycle.

Consider the following accumulation logic example.

```
scc_dsp<PREADDER_SIMPLE, REG_INPUT_OUTPUT, SYSTOLIC_NO,
 ACCUM_SINGLE, 10 > dsp_a;
int result;
for (int i=0; i<SIZE; i++){
  if (i==0)
       result = dsp_a(MULT_SUB_ACCUM_CONST,a[i],b[i],0,
        c[i],d[i],0);  // Initialization
else
       result = dsp_a(MULT_SUB_ACCUM,a[i],b[i],0,
        c[i],d[i],0);  // Running accumulation
}
```

In the foregoing example, the first iteration initializes the accumulator register to the following value.

```
Accum_reg = (1 << 10 ) + a[0]*b[0] - c[0] * d[0] ;
```

The CONST template parameter is used to help implement rounding operation. The default value of the CONST template parameter is 64, which disables rounding.

The dsp_a() call in the else portion accumulates the result of the computation into the accumulation register.

```
Accum_reg = Accum_reg + a[i]*b[i] - c[i] * d[i] ;
```

In Arria V devices, you can perform two parallel accumulations on the same DSP by using the double accumulation mode. Consider the following example.

```
scc_dsp<PREADDER_SIMPLE, REG_INPUT_OUTPUT, SYSTOLIC_NO,
 ACCUM_DOUBLE, 10 > dsp_a;
int result[2];
for (int i=0; i<SIZE; i++){
   for (int k=0; k<2; k++){
      if (i==0)
         result[k] =
   dsp_a(MULT_SUB_ACCUM_CONST,a[i][k],b[i][k],0,
    c[i][k],d[i][k],0);  // Initialization
      else
```

```
        result[k] =
   dsp_a(MULT_SUB_ACCUM,a[i][k],b[i][k],0,
    c[i][k],d[i][k],0);  // Running accumulation
   }
}
```

Here the same DSP block is used to accumulate the results of two parallel accumulation operations, for k = 0 and k =1. Note that this works only if the accumulation operation alternates in every call.

The following table describes the register mode configuration settings.

Table 19-3 RegMode Settings

RegMode	Functionality
REG_INFER	Infer the registers at the inputs and outputs based on timing.
REG_INPUT	Register all the inputs of the DSP block.
REG_OUTPUT	Register all the outputs of the DSP block.
REG_INPUT_OUTPUT	Register all the intputs and outputs of the DSP block.

Dynamic Control

The opcode setting lets you change the functionality at runtime. The following modes are available.

Table 19-4 Opcodes for Dynamic Control

Opcode	Configuration
MULT_ADD	Configures the post-adder to add the result of the multiplier blocks
MULT_SUB	Configures the post-adder to subtract the result of the multiplier blocks
MULT_ADD_ACCUM	Configures the post-adder to add the result of the multiplier blocks and accumulate it into the result of previous DSP operation
MULT_SUB_ACCUM	Configures the post-adder to subtract the result of the multiplier blocks and accumulate it into the result of previous DSP operation

Table 19-4 Opcodes for Dynamic Control (Continued)

MULT_ADD_ACCUM_CONST	Configures the post-adder to add the result of the multiplier blocks and add it to a constant initialization value
MULT_SUB_ACCUM_CONST	Configures the post-adder to subtract the result of the multiplier blocks and add it to a constant initialization value

Synphony C Compiler Class for Altera DSP

Synphony C Compiler provides an API in the form of an scc_dsp class and its methods that can be used to achieve the desired functionality using Altera DSP blocks. Synphony C Compiler has the flexibility to map method calls of different objects to the same physical DSP block or method calls of the same object to different DSP blocks.

The basic structure of the Altera V Series API is shown. The API designed to capture the semantics, not the hardware, of the DSP functions, with the exception of the `reg_mode`, which only affects the boundary registers.

```
namespace scc {
  namespace altera {
   template <
      PreadderMode preadder_mode=PREADDER_SIMPLE,
      RegMode reg_mode=REG_INFER,
      SystolicMode systolic_mode=SYSTOLIC_NO,
      AccumMode accum_mode=ACCUM_NONE,
      long long CONST=64,
      int coefA0=0, int coefA1=0, int coefA2=0, int coefA3=0,
      int coefA4=0, int coefA5=0, int coefA6=0, int coefA7=0,
      int coefB0=0, int coefB1=0, int coefB2=0, int coefB3=0,
      int coefB4=0, int coefB5=0, int coefB6=0, int coefB7=0
   >
   class scc_dsp {
     public:
     long long calc(Opcode opcode,
                    long long a0, long long a1, long long a2=0,
                    long long b0=0, long long b1=0, long long b2=0,
                    long long chainin=0);
     long long operator()(Opcode opcode,
                    long long a0, long long a1, long long a2=0,
                    long long b0=0, long long b1=0, long long b2=0,
                    long long chainin=0){
          return calc(opcode, a0, a1, a2, b0, b1, b2, chainin);
     }
   }; // class scc_dsp
  }; // namespace altera
}; // namespace scc
```

```
typedef enum {
 PREADDER_SIMPLE,
 PREADDER_COEF_ADD,
 PREADDER_COEF_SUB,
 PREADDER_INPUT_ADD,
 PREADDER_INPUT_SUB,
 PREADDER_SQUARE_ADD,
 PREADDER_SQUARE_SUB,
 PREADDER_CONSTANT
} PreadderMode;


typedef enum {
  SYSTOLIC_NO,
} SystolicMode


typedef enum {
  ACCUM_NONE,
  ACCUM_SINGLE,
  ACCUM_DOUBLE
} AccumMode;


typedef enum {
  REG_INFER,
  REG_INPUT,
  REG_OUTPUT,
  REG_INPUT_OUTPUT
} RegMode;


typedef enum {
  MULT_ADD,
  MULT_SUB,

  MULT_ADD_ACCUM,
  MULT_SUB_ACCUM,

  MULT_ADD_ACCUM_CONST,
  MULT_SUB_ACCUM_CONST,
} Opcode;
```

Not all possible configurations of the DSP block are supported. For example, the RegMode parameter settings `REG_INPUT` and REG_INPUT_OUTPUT force registers on all inputs, including control inputs; there is no separate control for registering individual inputs.

There are also restrictions on how multiple calls can be cascaded together. The chainin input port must be a dedicated connection from the return value of a previous call, and

Synphony C Compiler must be able to chain these signals to use the DSP block. Otherwise, it uses inlining to implement the function.

Using DSP Mapping on Altera

You can instantiate a DSP object using template parameters in the scc_dsp class. The template parameters indicate the static configuration parameter of the DSP block. The API takes in dynamic parameters that could be used to control the operations in the DSP such as multiply-accumulate or pre-add with post multiply-accumulate.

These are the steps for using the Synphony C Compiler DSP mapping on Altera:

1. Include header file scc.h:

```
#include "scc.h"
```

2. Define the namespace for Altera family for coding convenience:

```
using namespace scc::altera;
```

3. Create the scc_dsp object with appropriate static parameters, which are template parameters:

```
scc_dsp<REG_INFER> dsp1, dsp2;
```

4. Use the calc method or the operator “()” on the class object for creating the desired functionality. Suppose we are using the DSP to implement the quadratic function “a*x*x + b *x + c” as “(a*x +b)*x +c”:

```
int quadratic (int a, int b, int c, int x)
{
    scc_dsp <PREADDER_SIMPLE,REG_INFER> dsp1, dsp2;
    int t1 = dsp1.calc(MULT_ADD, a, x, b);
    int t2 = dsp2(MULT_ADD, t1, x, c);
    return t2;
}
```

5. For the Altera families, Synphony C Compiler expects the ALTERA environment variable to be set to point to the Altera installation directory and the QUARTUS_PATH environment variable to be set to point to the Quartus binary executable directory.

6. Choose a supported device using the appropriate implementation parameter. For example, you can use the following Tcl command to set Stratix V family as the target FPGA:

```
set_params -techlib altera-stratix5
```

7. Run the flow in Synphony C Compiler to generate RTL. Check the Resource browser in the GUI or the report_cost.txt file for the number of DSPs instantiated in your design.

Check report_pseudo_code.txt or operation schedule viewer in the GUI to see the scheduling of the DSPs.

Altera Coding Examples

The following example shows how to configure the Altera DSP block and perform DSP functions in C++ code.

Example 1

This example performs a complex multiply-and-accumulate operation.

```
#include "scc.h"
using namespace scc::altera;
#define SIZE 10
typedef struct {
    long long i; long long q;
#pragma bitsize i 34
#pragma bitsize q 34
} cmplx;

cmplx complex_multiply_accumulate(cmplx x[SIZE], cmplx w[SIZE])
{
  #pragma bitsize w.i 18
  #pragma bitsize w.q 18
  #pragma bitsize x.i 16
  #pragma bitsize x.q 16

  // Initialize result to 0
  cmplx result={0,0};

  // SCC DSP instances which accumulate the i and q fields separately.
  // SINGLE_ACCUM mode

scc_dsp<PREADDER_SIMPLE,REG_INPUT_OUTPUT,SYSTOLIC_NO,ACCUM_SINGLE>dsp_a;

scc_dsp<PREADDER_SIMPLE,REG_INPUT_OUTPUT,SYSTOLIC_NO,ACCUM_SINGLE>dsp_b;
  #pragma unroll
  for (int i=0; i<SIZE; i++){ // Unrolled loop
      cmplx c= x[i];
      if (i==0) {
       result.i = dsp_a(MULT_SUB_ACCUM_CONST,c.i,w[i].i,0,c.q,w[i].q);
       result.q = dsp_b(MULT_ADD_ACCUM_CONST,c.i,w[i].q,0,c.q,w[i].i);
      } else {
       result.i = dsp_a(MULT_SUB_ACCUM,c.i,w[i].i,0,c.q,w[i].q);
       result.q = dsp_b(MULT_ADD_ACCUM,c.i,w[i].q,0,c.q,w[i].i);
      }
  // Same result as:
  // result.i += (c.i*w[i].i - c.q*w[i].q);
  // result.q += (c.i*w[i].q + c.q*w[i].i);
```

```
  }
  return result;
}
```

There are two separate accumulation channels (one for i and another for q) in this design. Each DSP block is configured in the ACCUM_SINGLE mode. On the first iteration (i==0), each of the units was initialized with the MULT_SUB_ACCUM_CONST and MULT_SUB_ACCUM_CONST opcodes, based on the requirement of the way in which the inputs were processed before the accumulation. This mode loads the constant to be accumulated, 0 in this case, to the result of the first iteration operation. On subsequent iterations, each of the DSP blocks accumulates the result of the current operation to the stored value. The return value of each DSP operation is the current state of the accumulator inside the DSP block.

Example 2

This example is the same as Example 1 except that it uses the double rather than single accumulator mode.

```
#include "scc.h"
using namespace scc::altera;
...
  // Initialize result to 0
  cmplx result={0,0};

  // SCC DSP instances which accumulate the i and q fields separately.
  // DOUBLE_ACCUM mode

scc_dsp<PREADDER_SIMPLE,REG_INPUT_OUTPUT,SYSTOLIC_NO,ACCUM_DOUBLE>dsp_a;
  #pragma unroll
  for (int i=0; i<SIZE; i++){ // Unrolled loop
      cmplx c= x[i];
      if (i==0) {
       result.i = dsp_a(MULT_SUB_ACCUM_CONST,c.i,w[i].i,0,c.q,w[i].q);
       result.q = dsp_a(MULT_ADD_ACCUM_CONST,c.i,w[i].q,0,c.q,w[i].i);
      } else {
       result.i = dsp_a(MULT_SUB_ACCUM,c.i,w[i].i,0,c.q,w[i].q);
       result.q = dsp_a(MULT_ADD_ACCUM,c.i,w[i].q,0,c.q,w[i].i);
      }
  // Same result as:
  // result.i += (c.i*w[i].i - c.q*w[i].q);
  // result.q += (c.i*w[i].q + c.q*w[i].i);
  }
  return result;
}
```

This mode uses the two different accumulation channels available in each DSP block. Therefore, the API uses only a single DSP instance. The rest of the operation is similar to the single accumulation mode. The return value of each DSP operation is the current state of the accumulator inside the DSP block.

Example 3

Suppose you have a filtering algorithm can be implemented by the following sum-of-products code:

```
const int c[5]  = { -30, 75, 166, 75, -30 };
// filtering
output =  c[4] *  f[0] +  c[3] * f[1] +
          c[2] *  f[2] +  c[1] * f[3] +
          c[0] *  f[4];
```

This same function can be efficiently implemented in an Altera DSP block by using the following C++ code:

```
#include "scc.h"
using namespace scc::altera;
 ...
 scc_dsp<PREADDER_COEF_ADD,REG_INPUT_OUTPUT,SYSTOLIC_NO,ACCUM_NONE,0,
  -30, 75, 166, 0, 0, 0, 0, 0,
  -30, 75, 166, 0, 0, 0, 0, 0> dsp_a;
 int out_i = dsp_a.calc(MULT_ADD,f[1],f[3],1,f[0],f[4],0);
 #pragma bitsize out_i 16

 // filtering
 output = c[2] * (int) f[2] + out_i;
```

The `scc_dsp` statement configures the functional behavior of DSP block “a”. The PREADDER_COEF_ADD keyword configures the DSP block to perform the addition followed by multiplication with ROM-based coefficients:

```
a = (a0 + a1) * ROMA[a2]
b = (b0 + b1) * ROMB[b2]
```

The REG_INPUT_OUTPUT keyword specifies that the inputs and outputs are to be registered. The SYSTOLIC_NO and ACCUM_NONE keywords specify that the systolic mode is disabled and the output is not accumulated. The sequence of numbers specifies the c[n] constant values mapped into ROM of the DSP block and used in the algorithm.

This algorithm takes advantage of the fact that the coefficients are symmetrical, with c[0] = c[4] and c[1] = c[3]. In the preadder-coefficient configuration, the DSP block performs addition of two data values first, then multiplication by the shared coefficient stored in ROM.

In the next line of code, the MULT_ADD function of `dsp_a.calc` performs the operation by mapping the array variables to the inputs of the DSP block. It stores the result in variable out_i:

```
a = (f[1] + f[3]) * C[1]
b = (f[0] + f[4]) * C[0]
out_i = a + b
```

Finally, the last line sets the output variable to the sum of out_i and the product of the remaining coefficient and data value:

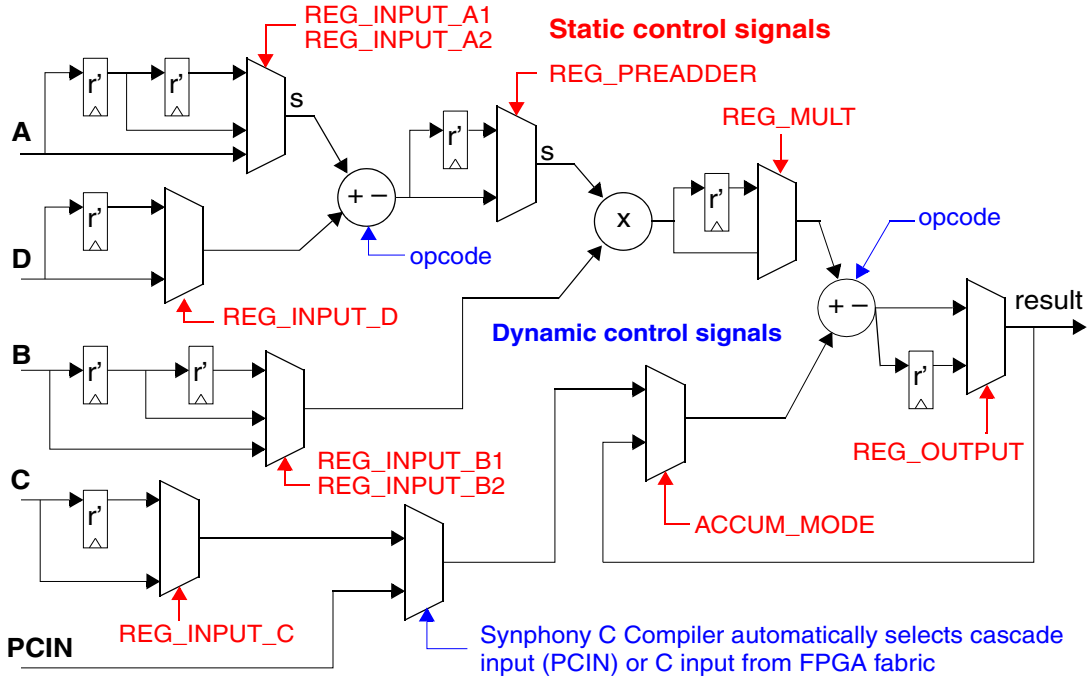
```
output = c[2] * (int) f[2] + out_i;
```

Xilinx DSP Mapping

Synphony C Compiler provides a class which can be used to instantiate DSP objects corresponding to DSP slices in the Xilinx Virtex-5, Virtex-6, and 7-series FPGA families.

The following block diagram shows the supported features in the Xilinx DSP block architecture.

Figure 19-2 Xilinx DSP Block Diagram



The static control signals are shown in red and the dynamic control signals are shown in blue. The static properties are set only once and represent the properties of the inferred DSP block. The dynamic controls are for controlling the operation that are performed by the individual blocks at different call points.

You can have different objects created with different static controls in the same code. Only DSP APIs with same static controls can be shared on the same DSP block.

Static Control

You configure the Xilinx DSP block by setting its accumulator mode and register mode.

The accumulator mode can be set to either ACCUM_NO to disable usage of the accumulator or ACCUM_YES to accumulate the results from every cycle. When using the accumulator, you must initialize it. Generally this is done by using the `MULT` opcode. When doing accumulation, the post-adder C input in the hardware is not used.

Note:

If you are using pre-adder mode and targeting a Virtex-5 device, the pre-adder is mapped to logic in the FPGA and the REG_PREADDER and REG_INPUT_D specifications of the API are ignored.

You can use the RegMode template parameters of the DSP class to specify the required registers in the DSP. You can use the “|” operator to specify different registers required for the particular DSP object. To enable a selection of the nine available registers, use syntax similar to the following:

```
scc::xilinx::scc_dsp<REG_INPUT_A2|REG_INPUT_B2|REG_OUTPUT> dsp_object;
```

The different RegMode settings are given in the following table.

Table 19-5 RegMode Settings

RegMode	Functionality
REG_INFER	Automatically infer appropriate registers.
REG_INPUT_A1	Use one register at A input.
REG_INPUT_A2	Use two registers at A input.
REG_INPUT_B1	Use one register at B input.
REG_INPUT_B2	Use two registers at B input.
REG_INPUT_C	Use one register at C input.
REG_INPUT_D	Use one register at D input.
REG_PREADDER	Use one register after preadder. This is also known as the AD register.
REG_MULT	Use one register at multiplier output
REG_OUTPUT	Use one register at output. This register is required when accumulator is used.

Dynamic Control

The opcode setting lets you change the functionality at runtime as described in the following tables. Table 19-6 shows the opcodes and functionality without the accumulator and Table 19-7 shows the same information with the accumulator.

Table 19-6 Xilinx DSP Functions, AccumMode = ACCUM_NO

Opcode	Functionality	Operator on “dsp” object
MULT	result = a * b	result =dsp (MULT, a, b)
MULT_ADD	result = c + (a * b)	result =dsp (MULT_ADD, a, b, c)
MULT_SUB	result = c - (a * b)	result =dsp (MULT_SUB, a, b, c)
ADD_MULT_ADD	result = c + ((d + a) * b)	result =dsp (ADD_MULT_ADD, a, b, c, d)
ADD_MULT_SUB	result = c - ((d + a) * b)	result =dsp (ADD_MULT_SUB, a, b, c, d)
SUB_MULT_ADD	result = c + ((d - a) * b)	result =dsp (SUB_MULT_ADD, a, b, c, d)
SUB_MULT_SUB	result = c - ((d - a) * b)	result =dsp (SUB_MULT_SUB, a, b, c, d)

Table 19-7 Xilinx DSP Functions, AccumMode = ACCUM_YES

Opcode	Functionality	Operator on “dsp” object
MULT	result = a * b	result = dsp (MULT, a, b)
MULT_ADD	result = result_prev + (a * b)	result =dsp (MULT_ADD, a, b)
MULT_SUB	result = result_prev - (a * b)	result =dsp (MULT_SUB, a, b)
ADD_MULT_ADD	result = result_prev + ((d + a) * b)	result =dsp (ADD_MULT_ADD, a, b, 0, d)
ADD_MULT_SUB	result = result_prev - ((d + a) * b)	result =dsp (ADD_MULT_SUB, a, b, 0, d)
SUB_MULT_ADD	result = result_prev + ((d - a) * b)	result =dsp (SUB_MULT_ADD, a, b, 0, d)
SUB_MULT_SUB	result = result_prev - ((d - a) * b)	result =dsp (SUB_MULT_SUB, a, b, 0, d)

When the accumulator mode is disabled (ACCUM_NO), the C input can be used as a post-adder input and the final result is not accumulated. When the accumulator mode is enabled (ACCUM_YES) the C input is not used and each final result is accumulated by

adding it to the previous result. In Table 19-7, the notation result_prev means the result of the previous computation.

Synphony C Compiler Class for Xilinx DSP

Synphony C Compiler provides an API in the form of an scc_dsp class and its methods that can be used to achieve the desired functionality using Xilinx DSP slices. Synphony C Compiler has the flexibility to map method calls of different objects to the same physical DSP block or method calls of the same object to different DSP blocks.

The basic structure of the Synphony C Compiler class is shown. The class designed to capture the semantics, not the hardware, of the DSP functions, with the exception of the RegMode, which only affects the registers inside the DSP.

```
namespace scc {
  namespace xilinx {
  template <RegMode regmode=REG_INFER,
            AccumMode accum_mode=ACCUM_NO>
  class scc_dsp {

    public:
      long long calc(Opcode opcode, int a, int b,
                     long long c=0, int d=0);
      long long operator()(Opcode opc, int a, int b,
                       long long c=0, int d=0) {
          return calc(opc, a, b, c, d);
      }
   }; // class scc_dsp
  }; // namespace xilinx
}; // namespace scc

typedef enum {
  REG_INFER,
  REG_INPUT_A1,
  REG_INPUT_A2,
  REG_INPUT_B1,
  REG_INPUT_B2,
  REG_INPUT_C,
  REG_INPUT_D,
  REG_PREADDER,
  REG_MULT,
  REG_OUTPUT
} RegMode;

typedef enum {
  ACCUM_NO,
  ACCUM_YES,
} AccumMode;
```

```
typedef enum {
  MULT,
  MULT_ADD,
  MULT_SUB,
  ADD_MULT_ADD,
  ADD_MULT_SUB,
  SUB_MULT_ADD,
  SUB_MULT_SUB,
} Opcode;
```

Using DSP Mapping on Xilinx

You can instantiate a DSP object using template parameters in the scc_dsp class. The template parameters indicate the static configuration parameter of the DSP block. The API takes in dynamic parameters that could be used to control the operations in the DSP such as multiply-accumulate or pre-add with post multiply-accumulate.

These are the steps for using the Synphony C Compiler DSP mapping on Xilinx:

1. Include header file scc.h:

   ```
   #include "scc.h"
   ```

2. Define the namespace for Xilinx family for coding convenience:

   ```
   using namespace scc::xilinx;
   ```

3. Create the scc_dsp object with appropriate static parameters, which are template parameters:

   ```
   scc_dsp<REG_INFER> dsp1, dsp2;
   ```

4. Use the calc method or the operator “()” on the class object for creating the desired functionality. Suppose we are using the DSP to implement the quadratic function “a*x*x + b *x + c” as “(a*x +b)*x +c”:

   ```
   int quadratic (int a, int b, int c, int x)
   {
       scc_dsp <REG_INFER> dsp1, dsp2;
       int t1 = dsp1.calc(MULT_ADD, a, x, b);
       int t2 = dsp2(MULT_ADD, t1, x, c);
       return t2;
   }
   ```

5. For the Xilinx families, Xilinx ISE needs to be available and setup. SCC expects the XILINX environment variable to be set to point to the Xilinx installation directory.

6. Choose a supported device using the appropriate implementation parameter. For example, you can use the following Tcl command to set xc6vlx240t-ff784-1 as target device:

```
set_params -techlib xilinx-virtex6
  -device xc6vlx240t-ff784-1
```

7. Run the flow in Synphony C Compiler to generate RTL. Check the Resource browser in the GUI or the report_cost.txt file for the number of DSPs instantiated in your design. Check report_pseudo_code.txt or operation schedule viewer in the GUI to see the scheduling of the DSPs.

Xilinx Coding Examples

The following examples show how to use DSP mapping on Xilinx devices.

Example 1

Complex multiply using two DSPs and ACCUM_YES mode

```
// re += x*c  -  y*s
// im += x*s  +  y*c
void cmplx_mpy(short c, short s, short x, short y, int &re, int &im) {
  static scc_dsp<REG_INFER,ACCUM_YES> dsp1, dsp2;
  for (int iac = 0; iac < 2; iac++) {
    if (iac == 0){
      // initializing the accumulator
      re = dsp1(MULT, c, x);
 im = dsp2(MULT, s, x);
    } else if (iac == 1) {
      re = dsp1(MULT_SUB, s, y, re);
      im = dsp2(MULT_ADD, c, y, im);
    }
}
```

This example shows a complex multiply using two multipliers and two DSP blocks. It does a multiply operation, which is stored in accumulator and then does a multiply-add or multiply-subtract based on the opcode.

Example 2

FIR filter

```
//sum = taps[0] * window[0] +
// taps[1] * window[1] +…
// … + taps[NTAPS-1] * window[NTAPS-1]
#define NTAPS 15
long filter_core(const short taps[NTAPS], short in)
{
```

```
  static short window[NTAPS] = {0};
#pragma unroll
  for(int i = NTAPS-1; i > 0; i--) {
    window[i] = window[i-1];
  }
  window[0] = in;

long sum = 0;
scc_dsp<REG_INFER, ACCUM_NO> dsp;
#pragma unroll
  for (int i = 0; i < NTAPS; i++) {
    sum = dsp(MULT_ADD, window[i],taps[i],sum);
  }
  return sum;
}
```

This example shows a direct-form FIR filter implementation using an scc_dsp object. The output is cascaded to the input and the loop is unrolled.

Example 3

Transpose form of the FIR filter in Example 2

```
// window[NTAPS-1] = taps[0] * in + window[NTAPS-2]
// window[NTAPS-2] = taps[1] *in + window[NTAPS-3] ….
// window[0] = taps[NTAPS-1] * in
#define NTAPS 15
long filter_core(const short taps[NTAPS], short in)
{
  static long window[NTAPS] ={0};

  long sum = 0;
  scc_dsp<REG_INFER , ACCUM_NO> dsps;
  #pragma unroll
  for (int i = 0; i < NTAPS-1; i++) {
window[NTAPS-1-i] = dsps(MULT_ADD, in, taps[i],
                                 window[NTAPS-2 -i]);

  }
     window[0] = dsps(MULT, taps[NTAPS-1], in);

  return window[14];
}
```

Example 4:

Symmetric FIR filter with odd number of taps

```
//sum = taps[0] *(window[0]+window[NTAPS-1] ) +
// taps[1] *( window[1] +window[NTAPS-2])  +…
// taps[NTAPS/2]* window[NTAPS/2]
#define NTAPS 15
```

```
#define NTAPS2 8
long filter_core(const short taps[NTAPS2], short in)
{
  static short window[NTAPS] = {0};
  #pragma unroll
  for(int i = NTAPS-1; i > 0; i--) {
    window[i] = window[i-1];
  }
  window[0] = in;

  long sum = 0;
  scc_dsp<REG_INFER, ACCUM_NO> dsps;
  #pragma unroll
  for (int i = 0; i < NTAPS/2; i++) {
    sum = dsps(ADD_MULT_ADD, window[i], taps[i],sum,window[NTAPS-1-i]);
  }
  sum = dsps(MULT_ADD, window[NTAPS/2], taps[NTAPS/2],sum);
  return sum;
}
```

The foregoing example shows the coding for Symmetric filter using pre-adder of the DSP.

APIs for Commonly Used Functions

A higher-level, function-based API is provided for easy usage of commonly used stateless functions in both Altera and Xilinx DSP blocks. The functions give you simplified use models to perform certain operations. The exact modes required to perform these operations are automatically handled by the wrapper functions internally.

The following table lists and describes the functions.

Table 19-8 APIs for Common Functions

Function being implemented	Synphony C Compiler API
result = (c + (a * b))	template <RegMode r> long long mult_add(long long a, long long b, long long c);
result = (c - (a * b))	template <RegMode r> long long mult_sub(long long a, long long b, long long c);

Table 19-8 APIs for Common Functions (Continued)

Function being implemented	Synphony C Compiler API
result = ((a + b) * c)	template <RegMode r> long long add_mult(long long a, long long b, long long c);
result = ((a - b) * c)	template <RegMode r> long long sub_mult(long long a, long long b, long long c);
result = (c + ((d+a) * b))	template <RegMode r> long long add_mult_add(long long a, long long b, long long c, long long d);
result = (c – ((d+a) * b))	template <RegMode r> long long add_mult_sub(long long a, long long b, long long c, long long d);
result = (c + ((d-a) * b))	template <RegMode r> long long sub_mult_add(long long a, long long b, long long c, long long d);
result = (c – ((d-a) * b))	template <RegMode r> long long sub_mult_sub(long long a, long long b, long long c, long long d);

Each of these functions is a template function, with the register-mode as the template parameter.

Index

Symbols

$SCC labeling 7-3

A

addresses 14-15
  restrictions 14-16
affine
  defined 5-34
Architecture Analyzer
  Code Tree View 2-22
array access checks 12-6
array dependency pragma 7-37
array pragmas 7-15
arrays 5-35
  pragmas 7-17
ASIC characterization
  datasheet 18-9
  input files 18-9, 19-11
  output files 18-9
  picodata files 18-9
assertopms 15-9
autostart 15-16

B

banking 8-15
bitsize
  checks 12-3
bitsize pragma 7-20
boundary registers 7-20, 14-25
boundary_register pragma 7-20
build command 3-6
bus_name 7-24
bus_name pragma 7-24
bus_standard pragma 7-24
bus_type pragma 7-25

C

C simulation 14-38
C to RTL Mapping 2-67, 13-74
channel synthesis 11-1
  build flow 11-10
  examples 11-4
  implementation parameters 11-12
  limitations 11-13
  predefined classes 11-2
  simulation 11-9
clock gating 15-19
clocks 15-11
coding style
  bitsize 12-3
coding styles 5-62

configuration file
  Novo 18-4
console window 2-14
constant_across_pipelined_calls pragma 7-25
constraints 7-2
  global 7-2
  Tcl 7-6
  tool 7-2
continuous processing 5-8
conventions for documentation xxi
core processor, PA 16-6
create_implementation command 3-13, 3-14
customer support xxii
cycle-level scheduling 7-10

D

data_name pragma 7-26
definition 4-3
dependency pragma 7-37
deployable TCABs 14-34
deployment of memory-mapped PPA 14-26
design summaries 13-4
desogm for test 15-13
direct port interface to memories in PPA 14-12
directory structure 4-4
document view window 2-15
dont_care_code pragma 7-27
driver creation guidelines 14-47
DSP mapping 19-1
  Altera 19-8
  common function APIs 19-26
  overview 19-2
  resource sharing 19-6
  Verilog Simulation 19-5
  Xilinx 19-19
Dynamic Performance Graph 2-65

E

e timing closure 15-12
error signaling 15-15
estimate_power command 3-8
export_implementation command 3-9
expression labeling 7-3
external streams 8-25

F

false dependencies 7-40
feedback path viewer 2-51
feedback paths 5-33
fifo_length pragma 7-27
FIFOs 8-24, 15-23
  passthru and non-passthru 8-31
FIFOs.*See also* streams
File menu 2-4
flush pragma 7-28
forward_boundary_register pragma 7-20
forward_path_external_delay array pragma 7-29
function declaration labeling 7-3
function list for pragmas 7-14
functional units 16-8

G

gate-level simulation 14-41
global arrays
  scalarization 8-24
GUI 2-2

H

handshake interface 14-5
hardware allocation 5-15
hardware architecture 16-1
hardware interface 14-3

hardware schema 16-2
hardware sharing 5-48
Hierarchy Graph 2-38
host access
  pragmas 7-14
host interface bus 14-17
host_memory_access pragma 7-30
host_scalar_access pragma 7-31
hostlm 14-17

I

II 5-26
ii pragma 7-32
implementations 4-3
infinite run 15-17
interconnect, PA 16-7
interface timing
  boundary registers 14-25
interface to memories in PPA 14-12
interface, local memory 14-12
internal streams 8-25
  specification 8-31
internal_blockram pragma 7-32
internal_fast pragma 7-33

L

labeling 7-3
lint checks 12-2
local memories 14-12
local memory interface 14-12
localize_access pragma 7-33
loop distribution 5-21
loop jamming 5-23
loop sequentialization 5-41
loop unrolling 5-14

M

max_schedule_length pragma 7-35
memories
  banking 8-15
  boundary registers 7-23
  mapping components 14-14
  mapping to addresses 14-15
memory arbitration 15-21
memory interface, local 14-12
memory mapping 5-41
memory porting 5-38
memory_access_latency pragma 7-36
memory_structure pragma 7-36
memory-mapped PPA control, PPA control, memory-mapped 14-19
memory-mapped PPA, deployment 14-26
memory-mapped scalar interface 14-22
menus 2-4
multi_buffer pragma 7-37
multi-buffering 8-15
multiplier pipeline stages 3-31

N

name pragmas 7-61
native host interface, host interface, SCC native 14-18
no_dependence pragma 7-37
no_inter_loop_memory_analysis pragma 7-42
no_inter_loop_stream_analysis pragma 7-42, 7-43
no_inter_task_memory_analysis pragma 7-45
no_inter_task_stream_analysis pragma 7-46
no_inter_tcab_subtask_memory_analysis pragma 7-49
Novo
  characterization process 18-2
  configuration file 18-4
  configuration file parameters 18-5
  input files 18-9, 19-11

output files 18-9
overview 18-2
syntax 18-3
num_iterations pragma 7-50

O

OCP interface 14-19
opencore protocol 14-19
Operation Schedule Viewer 2-41
out-of-bounds access 12-6
output 15-2
overflow. *See* bitsize

P

package directory 15-4
packed pragma 7-53
parallelism 5-55
PAs
description 16-5
perfectization 5-16
picodata files 18-9
polarity pragma 7-67
port interface to memories in PPA 14-12
power
clock gating 15-19
PPA Graph 2-31
PPAs
description 16-2
pragmas 7-2
alphabetical list 7-18
and set_implementation command 7-13
conflicting values 7-13
EDK support 14-29
host access 7-14
miscellaneous 7-18
proc_list 7-14
quick reference for syntax 7-14
scalarization 8-23
specification 7-13
syntax checks 12-2
predicate merger 16-10
predicates 16-10
preserve_array pragma 7-57
production pragma 7-67
programming model 5-3
projects 4-2
PSW (processor status word) 14-19

R

RAM interfaces 14-12
rate matching 5-25
read_write_ports pragma 7-58
ready_name pragma 7-26
registers 16-9
merge registers 16-9
Relative Cost Viewer 2-63
report_constraints.txt 13-4
report_implementation command 3-14
report_summary.txt 13-6
request_name pragma 7-26
resets 15-12
Resource Browser 2-57
return_boundary_register pragma 7-20
return_path_external_delay array pragma 7-29
ROM interfaces 14-12
rom pragma 7-60
RTL
features 15-10
rtl_synthesize command (ASIC) 3-18
rtl_synthesize command (FPGA) 3-19
Run menu 2-6

S

scalar interface 14-8
scalar interface, memory-mapped 14-22

scalarization 8-21
  hardware implementations 8-24
  pragmas 8-23
scalars 5-34
SCC
  ASIC 1-4
  FPGA 1-4
  introduction 1-2
  output 15-2
  related tools 1-7
scc_stream() class 8-26
  simulation methods 8-28
  streaming methods 8-27
scheduling, cycle-level 7-10
scripts 17-2
semantic checks 12-3
Sequentialization Graph 2-26
set_params command
  and pragma specification 7-13
shell window 2-14
signal name pragmas 7-61
signal polarity pragma 7-67
signal production pragma 7-67
simulation comparisons 14-46
SolvNet
  accessing xxii
  documentation xx
STARC support 15-23
statement labeling 7-3
stream interface 14-9
  customization 14-11
stream procedure guidelines 14-47
streams 8-24
  blocking 8-30
  boundary registers 7-22
  class-based definition 8-25
  function-based definition 8-29
  non-blocking external 8-30
  pragmas 7-16, 7-17
SV simulation 14-42, 14-44
Synphony C Compiler
  introduction 1-2
Synphony C Compiler EV 3-38
synphonycc command 3-2
synphonycc_novo command 18-3
syntax checks 12-2
synthesize_auxopts 5-9
system interface 14-4
System Verilog 14-42
SystemC TLM interface 14-45

T

task
  description 5-4
task processing in PPAs 5-4
  continuous processing 5-8
  PPA hardware 5-6
  shared resource handling 5-5
  synthesize_auxopts 5-9
Task_II 5-27
TCABs
  design summary report 13-8
  integration 14-34
testbench, UVM 14-42
TLM 14-45
transaction-level models 14-45

U

uninitialized variables 12-5
Universal Verification Methodology 14-42
unroll pragma 7-70
user_supplied pragma 7-72
user_supplied_fpga pragma 7-72
utilities 17-2
UVM testbench 14-42

V

View menu 2-9
verify command 3-30