Synphony C Compiler User Guide

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 User Guide xviii

Customer Support. xx

1. Introduction to Synphony C Compiler Tool

About High-Level Synthesis 1-2

Overview of Synphony C Compiler Tool 1-2

Synphony C Compiler Executables. 1-2

Synphony C Compiler ASIC 1-3

Synphony C Compiler FPGA 1-3

User Interfaces of the Synphony C Compiler Tool 1-3

Graphical User Interface (GUI) 1-3

Command Line Interface 1-4

Technical Support 1-5

Related Synopsys Tools 1-5

Design Compiler Tool for ASIC Synthesis 1-5

Synplify Products for FPGA Synthesis. 1-5

VCS Tool for Simulation 1-6

2. Understanding the Graphical User Interface

Understanding the Graphical User Interface 2-2

Menus 2-4

File Menu 2-4

Run Menu 2-6

Analysis Menu 2-7

View Menu 2-9
Help Menu 2-9
The Tool Windows 2-9
The Dashboard Window 2-10
The Console Window 2-14
The Document View Window 2-15
Analysis Views 2-17
Report View 2-17
Diagnostics 2-20
Code Tree View 2-21
Sequentialization Graph 2-25
PPA Graph 2-30
Hierarchy Graph 2-37
Operation Schedule Viewer 2-40
Feedback Path Viewer 2-50
Context-Sensitive Menu for an Instruction 2-55
Resource Browser 2-56
Relative Cost Viewer. 2-62
Dynamic Performance Graph 2-64
C to RTL Mapping. 2-66

3. Setting Up Your Design For Synthesis

Getting Started 3-2
Starting the Tool. 3-2
Getting Help. 3-2
Terminologies Used 3-3
Setting Up Your Design for Synthesis 3-4
Working with Projects 3-4
Creating a New Project. 3-4
Opening an Existing Project 3-5
Specifying Parameters. 3-6
Working With Implementations 3-7
Creating a New Implementation 3-7
Creating a New Implementation From an Existing Implementation 3-8
Configuring Projects and Implementations 3-9
Setting Parameters 3-10
Locking and Unlocking an Implementation 3-12
Linking an Implementation. 3-13

Exporting an Implementation 3-15

4. Building and Verifying the Design

Overview of Build and Verification 4-2
Build Steps 4-2
Simulation Steps 4-3

Generating the RTL 4-3
Analyze 4-3
Build 4-4

Running the Build Step 4-4

Running C Simulations 4-5
Running Golden Simulation 4-6
Running Lint Simulation 4-7

RTL Simulations 4-9

Running RTL Simulation 4-10
Running RTL Simulation 4-10
Affect of Source File Modifications on the Project 4-12
Running SV-UVM Simulation 4-12
Running RTL Simulation on the RTL Package 4-13
Running Equivalence Checking on the RTL Package 4-15

Running Functional Gate-Level Simulation 4-16

Setting Build and Simulation Parameters 4-16
Setting Parameters for C Simulation 4-17
Setting Parameters for RTL Packaging 4-19
Specifying RTL Synthesis and Simulation Tools 4-19
Setting Parameters for RTL Simulation 4-20

Debugging Simulation Issues 4-22
Debugging Golden Simulation Issues 4-22
Compilation Failure 4-22
Non-Zero Return Value 4-23
Segfault 4-23
Mismatch Between Golden and Reference Results 4-23
Error Message: “Found no data in stream" 4-24
Warning Message: “Fifo not empty at start of task” 4-24

Verilog RTL Simulation Issues 4-24
RTL Compilation Failure 4-24
Error: Deadlock Detected 4-24
Simulation Timed Out 4-25
Mismatches 4-25
Scoreboard File Dumps in RTL Simulation 4-25

Verifying Corner Cases 4-26
Running Verification for Corner Cases 4-26
Creating a Config File 4-27

Checking Code Coverage 4-27
Checking C Code Coverage with Golden Simulation. 4-28
Checking Verilog Code Coverage 4-29

The Simulation Regression Flow 4-29
Running Simulation Regressions from a Script 4-30
Running Simulation Regressions from the GUI 4-31
Running Synphony C Compiler EV Regressions. 4-32

5. Analyzing the Design

Analyzing After Different Build Phases 5-2
Checking the Design Summary 5-2
Analyzing After RTL Simulation 5-3
Analyzing After RTL Simulation 5-3
Viewing RTL Waveforms 5-3

Checking the Source Code 5-5

Analyzing Timing and Performance 5-5
Analyzing Schedules and Timing Paths. 5-6
Analyzing Feedback Paths. 5-6
Analyzing False Paths 5-7
Checking Performance 5-7

Analyzing Resources and Costs 5-8
Analyzing Cost 5-8
Checking Hierarchy 5-8

Checking Power Consumption 5-9

Analyzing System Integration 5-9

Checking System Integration Reports 5-9
Keeping Address Maps 5-9

Working in Graphical Analysis Views 5-10
Performing Common Analysis View Operations 5-10
Crossprobing to C Code and RTL 5-12
Using the PPA Graph 5-14
Using the Sequentialization Graph 5-15
Example of Interpreting Dependencies 5-16
Using the Feedback Path Viewer 5-17
Using the Operation Schedule Viewer 5-19
Using the Resource Browser 5-21
Using the Relative Cost Viewer 5-23
Using the Dynamic Performance Graph 5-25
Using the Hierarchy Graph 5-26
Using Synphony C Compiler EV 5-27
Running Regressions in the Synphony C Compiler EV 5-28
Synphony C Compiler Driver Coding Styles 5-29

6. Synphony C Compiler Output

Overview of Synphony C Compiler Output 6-2

Hierarchy of Generated RTL 6-3

Package Directory Structure 6-4

Directory Structure to Support RTL Simulation 6-7

Corner Case Simulation Support 6-8

Assertions 6-9
Monitors and Checkers 6-9

RTL Features 6-10
Overall Hardware Strategy 6-10
Registers 6-11
Clocks 6-11
Resets 6-11
RTL Timing Closure 6-12
Timing Closure and False Timing Paths 6-12
Synthesis Assumptions 6-13
Physical Floorplan 6-13

Power Grid 6-13
Design-for-Test 6-13
Abort 6-13
Aborting a PPA 6-14
Reset and Abort 6-14
Error Signaling 6-15
Autostart 6-16
Infinite Run 6-16
Localization 6-17
FIFO Hierarchy Priority 6-17
Feature Interactions 6-18
Clock Gating (ASIC Only) 6-18
Memory Arbitration 6-21
Multi-Stall Domain Arbitration 6-22
Priority Order 6-22
FIFO Implementation (FPGA Only) 6-22

STARC Rule Support 6-23

7. Building Hierarchical Designs

Building Hierarchical Designs 7-2
Using Hierarchical Blocks 7-2
Considerations for Hierarchical Design 7-3
Benefits of Using Hierarchy 7-3
Limitations of Using Hierarchy 7-4
Using Deployable TCABs or PPAs for the Top Level 7-5

Hierarchical Design Flow 7-5
Synphony C Compiler Hierarchical Design Flow 7-5
Hierarchical Design Flow With No Iterations 7-6
Hierarchical Design Flow With Iterations 7-6
Running the Hierarchical Design Flow 7-8
Example Script for Hierarchical Design 7-8

Building Hierarchical Blocks 7-9
Creating and Configuring a Hierarchical Block 7-9
Guidelines for Coding Hierarchical Blocks 7-11

Hierarchical Synthesis Using Importable PPA 7-12
Building a PPA 7-12

Importing a PPA 7-13
Specifying Performance Constraints for Importable PPA 7-13
Advantages of Using Importable PPAs 7-14
Composition 7-15
Parallel PPAs 7-16
PPAs With Task Overlap 7-17
Views and Reports 7-19
Features of Importable PPA 7-19
Example of an Importable PPA 7-21
Limitations of an Importable PPA 7-24

Setting Parameters for TCABs 7-25
Specifying Blocks to be Built as TCABs 7-26
Specifying Combinational TCABs 7-28
Setting Pipelining Parameters for TCABs 7-29
Specifying Stall Domains for Hierarchical Blocks 7-31
Specifying Independent Stall Domain Blocks 7-31
Specifying Shared Stall Domain Blocks 7-32
Specifying External Delays for Hierarchical Blocks 7-34
Controlling the Inference of Boundary Registers 7-34
Specifying Latency for Hierarchical Blocks 7-35
Gating Clocks on Hierarchical Blocks 7-38
Sharing Hierarchical Blocks 7-39

TCAB Sharing and Inlining 7-41

Building Hierarchy and Importing Blocks 7-43
Building Hierarchy 7-43
Importing Blocks 7-45

Analyzing Hierarchical Blocks 7-45
Reports for the Hierarchical Blocks 7-45

8. Designing Efficient Hardware In C/C++

Coding Guidelines in Synphony C Compiler Tool 8-3
General Coding Restrictions 8-3
Supported C/C++ Syntax 8-3
Data Types 8-4
Modifiers 8-4
Enum and Typedef 8-5

Parameterized Data Type 8-5
Arrays 8-6
Pointers. 8-6
Functions and Methods 8-7
Class and Structure 8-7
Unsupported C/C++ Syntax. 8-7
Interfaces 8-8
System and Control Interface 8-8
Scalar Variables and Parameters 8-9
Arrays. 8-11
Streams 8-13
Synthesizable C/C++ Code 8-14
Coding Requirements. 8-15
Example Source File. 8-15
Hierarchical Synthesis 8-17
Building Memories 8-18
Coding Arrays and Mapping Memories 8-19
Mapping C Arrays to Memories in ASIC Designs 8-19
Mapping C Arrays to Memories in FPGA Designs. 8-21
Specifying Memory Properties. 8-23
Building FIFOs 8-25
Using FIFOs Instead of Memories. 8-25
Building External FIFOs. 8-26
Building External Streams in C++ Designs. 8-27
Building External Streams in C Designs. 8-28
Building Internal FIFOs 8-28
Building Internal Streams in C++ Designs 8-29
Building Internal Streams in C Designs 8-29
Building Shift Registers. 8-30
Optimizing the Build Steps 8-31
Optimizing the Build Run 8-31
Optimizing the Schedule Run 8-32
Optimizing the Synthesize Run 8-33
Influencing Perfectization 8-33
Manually Perfectizing Code 8-34
Code Sinking Examples. 8-35
Example of Effective Code Sinking. 8-35

Example of Ineffective Code Sinking 8-36
Loop Unrolling Example. 8-36
Loop Distribution Example. 8-37
Iteration Space Extension Example. 8-38

Flushing the Pipeline 8-38
Flushing the Pipeline to Avoid Stalls 8-39
Flushing the Pipeline to Relax Feedback Constraints 8-40
Flushing the Pipeline to Enable Resource Sharing 8-40

Optimizing Memories for Rate Matching. 8-41
Banking Memories. 8-41
Multibuffering Memories 8-42

Optimizing for Area 8-43
Optimizing Bit-Widths 8-44
Examples of Bit-Width Propagation 8-46
Inferring Registers 8-46
Specifying Constants. 8-47
Optimizing Local Memories 8-48
Comparing Scalarization and internal_fast Memories 8-49
Minimizing Memory Ports 8-50
Exploiting Mutual Exclusion. 8-51
Jamming Sequential Loops 8-52
Merging Parallel Loops 8-53

Optimizing Performance 8-53
Meeting Throughput Goals 8-54
Resolving Memory Bottlenecks 8-54
Analyzing Feedback Paths. 8-55

Coding to Maximize Parallel Optimizations. 8-56
Loop Index Expressions. 8-57

Task Processing 8-58
Constraints for Task Processing 8-58
Synthesize Reports for Task Processing 8-59
RTL Simulation Reports. 8-62
RTL Simulation Problems 8-64
Resizing FIFOs for Better Performance. 8-64
Example Design. 8-65

No Task Overlap 8-67
Inter-PA Task Overlap 8-68
Inter-PA and Intra-PA task overlap 8-69
Resizing FIFOs for Performance 8-70
Using Templates 8-72
Working with Templates 8-73
Creating a Template 8-73
Correlating Timing With ASIC Synthesis 8-74
Working With a Representative Design 8-74
Working Without a Representative Design 8-75

9. Characterizing ASIC Macro Cells

ASIC Characterization Flow 9-2
Setting Up the Environment for Characterization 9-3
Performing Characterization 9-4
Debugging the Characterization Process 9-5
Performing Incremental Characterization 9-6
Using Characterization Data 9-7
Validating Characterization Data 9-7
Validating Timing Closure 9-7
Validating Area Optimization 9-8

Appendix A. Synphony C Compiler Commands

About the Command Line Interface A-2
Command-Line Entry A-2
Tool Startup Commands A-2
Commands by Type A-4
Alphabetical List of Commands A-5
Command Descriptions A-6
analyze A-6
build A-6
Syntax A-6
Description A-7
Example A-7
close_project A-7

Syntax A-7
Example A-7
delete_implementation A-7
Syntax A-7
Description A-7
Example A-8
delete_project A-8
Syntax A-8
Example A-8
estimate_power A-8
Syntax A-8
Description A-9
export_implementation A-9
Syntax A-9
Description A-10
Example A-10
get_defaults A-10
Syntax A-10
get_implementations A-11
Syntax A-11
get_params A-11
Syntax A-11
Description A-11
help A-11
Syntax A-12
Description A-12
link_implementation A-12
Syntax A-12
Description A-12
lock_implementation A-12
Syntax A-13
new_implementation A-13
Syntax A-13
Description A-13
Example A-13
new_project A-13
Syntax A-13
open_project A-14
Syntax A-14
Example A-14

quit A-14
Syntax A-14
Description A-14
report_implementation A-14
Syntax A-14
Description A-17
Example A-17
reset_defaults A-17
Syntax A-17
reset_rtl_package A-18
Syntax A-18
Description A-18
Example A-18
rtl_synthesize (ASIC) A-18
Syntax A-18
Description A-19
Example A-19
rtl_synthesize (FPGA) A-19
Syntax A-19
Example A-19
select_implementation A-20
Syntax A-20
Description A-20
Example A-20
set_params A-20
Syntax A-20
Description A-20
Example A-28
show A-28
Syntax A-29
unlock_implementation A-29
Syntax A-29
verify A-30
Syntax A-30
Description A-30
Example A-30
view_waveform A-30
Syntax A-30
Switching Activity Interchange Format (SAIF) File A-31

Additional SCC Options A-31
Schedule Auxiliary Option A-31
Synthesize Auxiliary Options A-31
Multiplexing Non-Mutex Memory Ports A-32
Arbitration Priority A-33
TCAB Sharing Controls A-33
Threshold for Cost Based TCAB Sharing A-34
Interactions Between Sharing and Timing A-34
Data Width Threshold for Predicated Multiplier Sharing A-35
Minimum width for sharing functional units A-36
II-based Opportunistic Sharing A-36
Sharing Versus Increased Schedule Length A-36
Characterization Data Scaling A-37
Control for Interconnect Delays A-37
Interconnect Delay Estimation A-37
Latency Selection A-38
Default Case Assignment to 0 . . . . . A-38
Synphony C Compiler EV Commands A-38
Synphony C Compiler EV Launch Command A-39
Synphony C Compiler EV Supported Commands A-39

Index

Preface

This preface includes the following sections:

- About This User Guide
- Customer Support

About This User Guide

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/PagesK-2015.06

1

Introduction to Synphony C Compiler Tool

This chapter provides an overview of high level synthesis and the Synphony C Compiler tool.

This chapter contains the following sections:

- About High-Level Synthesis
- Overview of Synphony C Compiler Tool
- Synphony C Compiler Executables
- Technical Support
- Related Synopsys Tools

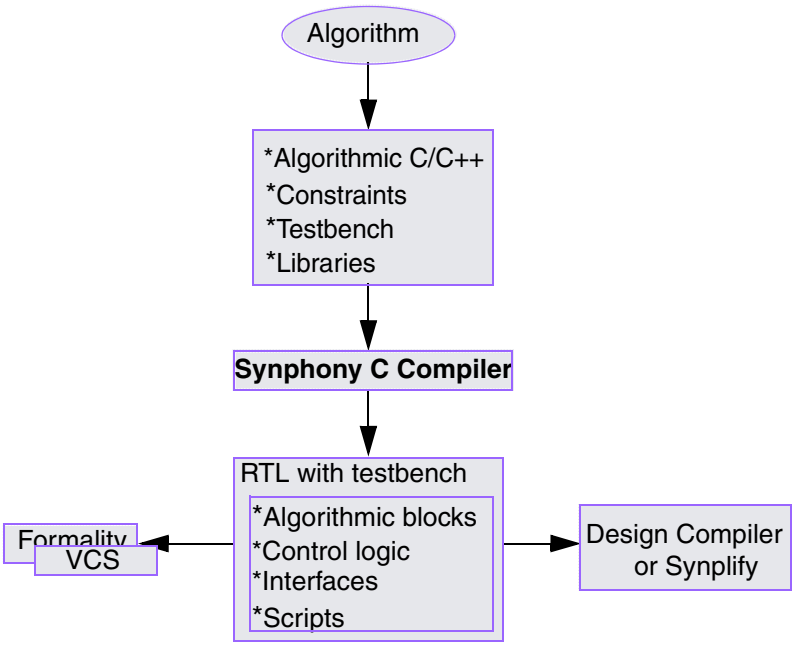
About High-Level Synthesis

High-level synthesis is an automated design process that interprets an algorithmic description of a desired behavior and creates the digital hardware that implements the behavior. The description is generally decoupled from the minute details such as clock-level timing. The code is analyzed, architecturally constrained, and scheduled to create a register-transfer level hardware design language (HDL), which is synthesized to a gate-level design by using a logic synthesis tool.

Overview of Synphony C Compiler Tool

High-level synthesis tools such as Synphony C Compiler, reduce the development cycle time and cost by shifting the focus of hardware implementation from designing at the RTL level to designing at the algorithmic level. As shown in Figure 1-1, you can describe the hardware in algorithmic C/C++ language, use Synphony C Compiler tool to fine-tune the architecture for tradeoffs in performance, power, and area, and generate highly optimized RTL implementations for ASIC or FPGA.

Figure 1-1 Synphony C Compiler and High-Level Synthesis



Synphony C Compiler Executables

You can use Synphony C Compiler for both ASIC and FPGA designs.

You start Synphony C Compiler by entering `synphonycc` command in a UNIX shell. Specify the absolute path to indicate the Synopsys root that contains the Synphony C Compiler installation:

```
% synphonycc
```

This invokes Synphony C Compiler for ASIC. To invoke Synphony C Compiler for FPGA, specify,

```
synphonycc_fpga
```

Synphony C Compiler ASIC

The Synphony C Compiler executable for ASIC generates RTL optimized for ASICs. In ASIC executable, the defaults for certain parameters are optimized for ASIC designs.

Synphony C Compiler FPGA

The Synphony C Compiler FPGA executable generates device-specific RTL for Altera or Xilinx FPGAs. The default settings are optimized for FPGA designs.

For more details about the supported devices and speed grades in each family, see the *Synphony C Compiler Reference Manual*

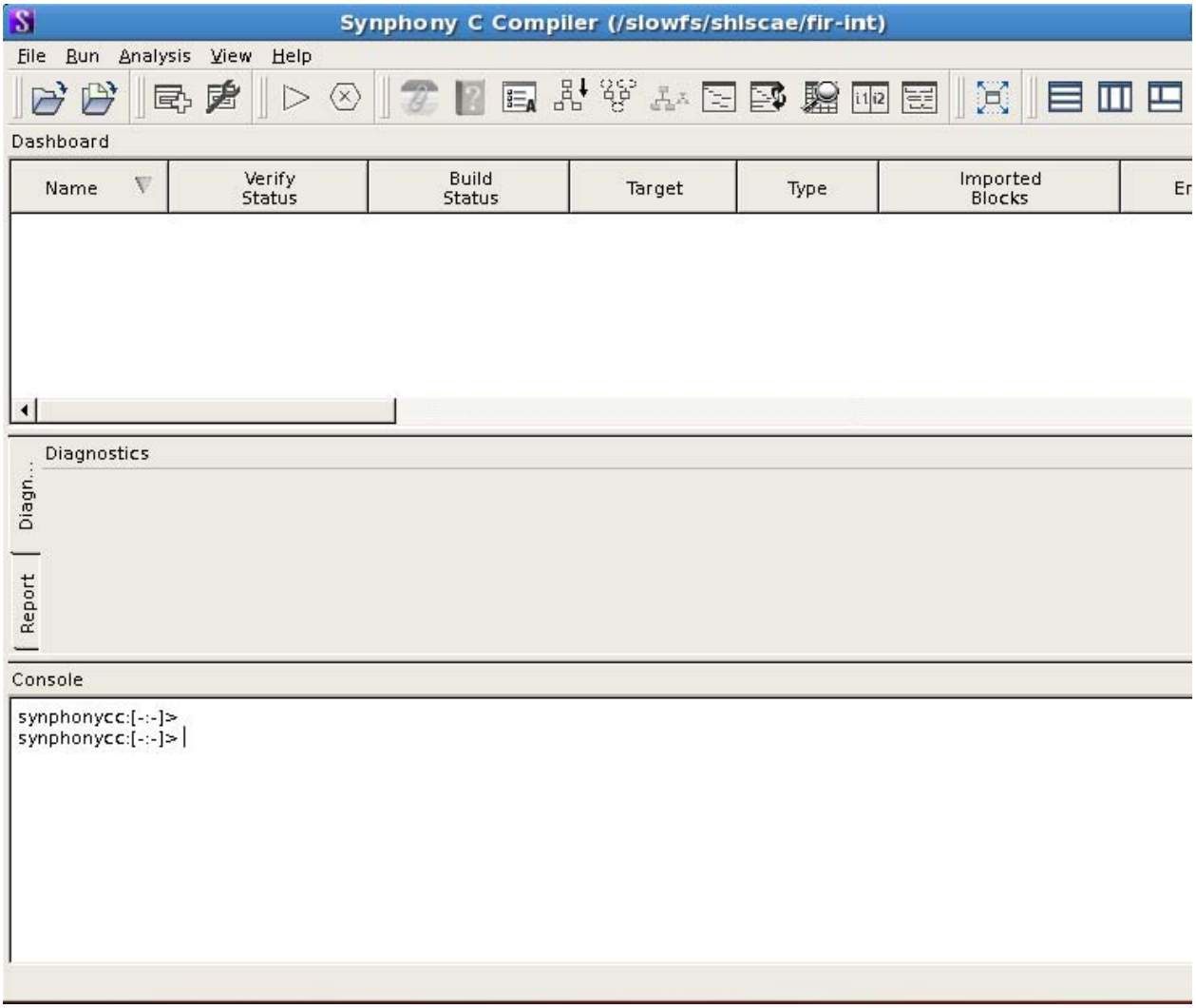
User Interfaces of the Synphony C Compiler Tool

Synphony C Compiler supports the following user interfaces:

Graphical User Interface (GUI)

Launch the Synphony C Compiler GUI by running the `synphonycc` executable. Figure 1-2 shows the Synphony C Compiler GUI.

Figure 1-2 Synphony C Compiler GUI



For more details about the GUI, see Chapter 3, “Setting Up Your Design For Synthesis and the *Synphony C Compiler Reference Manual*.

Command Line Interface

Launch the Synphony C Compiler command-line interface by running the executable as follows:

```
synphonycc -c
```

You can also specify a Tcl script when you run the executable, as shown in the following example:

```
synphonycc my_script.tcl
```

The Tcl script is a text file containing a sequence of Synphony C Compiler commands.

You can also run shell scripts in the command line or in the console window as shown in the following example:

```
source my_script.sh
```

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 might 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 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

Understanding the Graphical User Interface

This chapter introduces you to the Graphical User Interface (GUI) Synphony C Compiler (SCC) tool.

- Understanding the Graphical User Interface

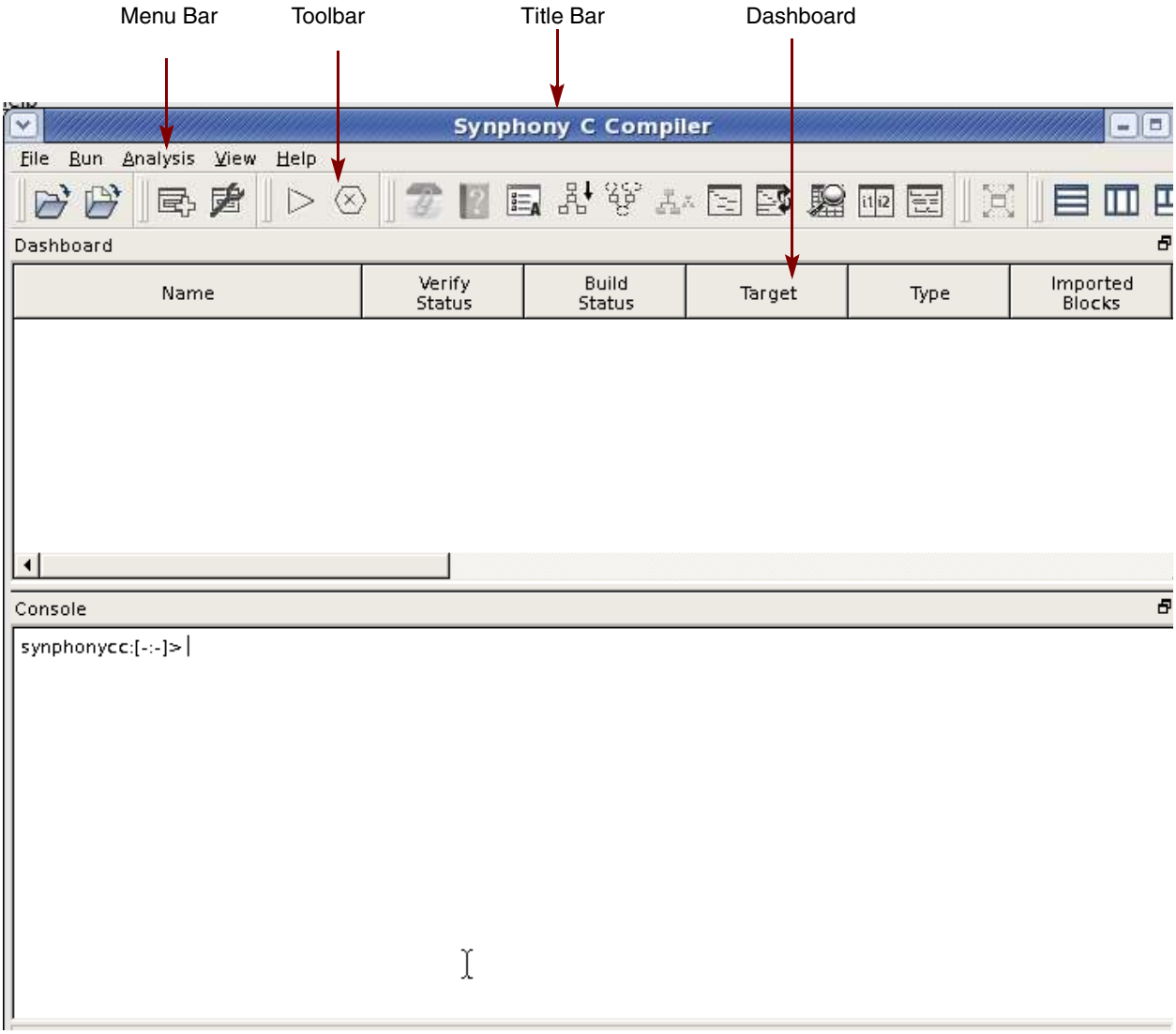
Understanding the Graphical User Interface

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 Appendix A, “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.



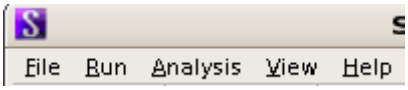
The GUI consists of the following:

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.
Toolbar	Displays various icons that are shortcuts to commands.

Item	Description
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.
Open Project	Opens a project. The Analysis view opens when you open a project.

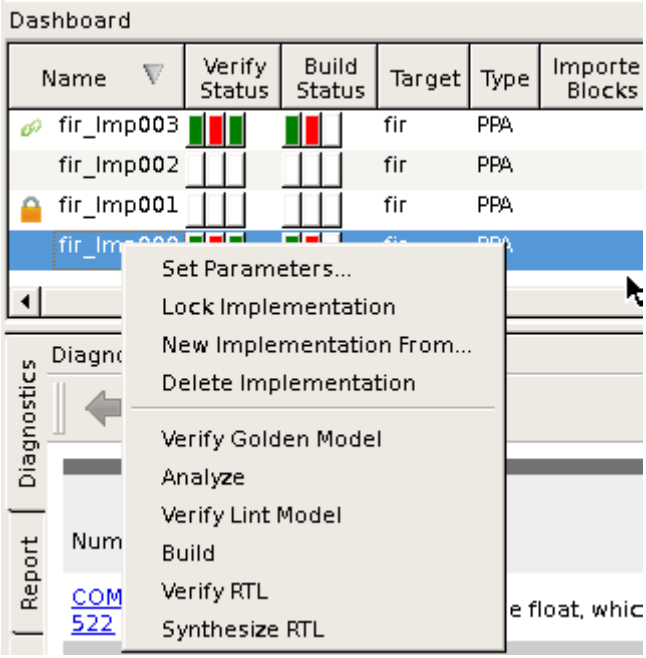
Command	Description
Close Project	Closes a project. When you close a project, the Analysis view gets hidden.
Delete Project	Deletes a project from the Dashboard window.
Refresh Projects	Refreshes the Dashboard window, and any opened project for the active implementation.
Recent Projects	Displays a list 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.
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 within the Synphony C Compiler tool and generates a 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 from 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 on 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 Analysis view opens only when you open a project.

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, 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

Command	Description	Prerequisite
PPA Graph	Helps you understand the resource usage in your design.	Successful build
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 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.	

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:

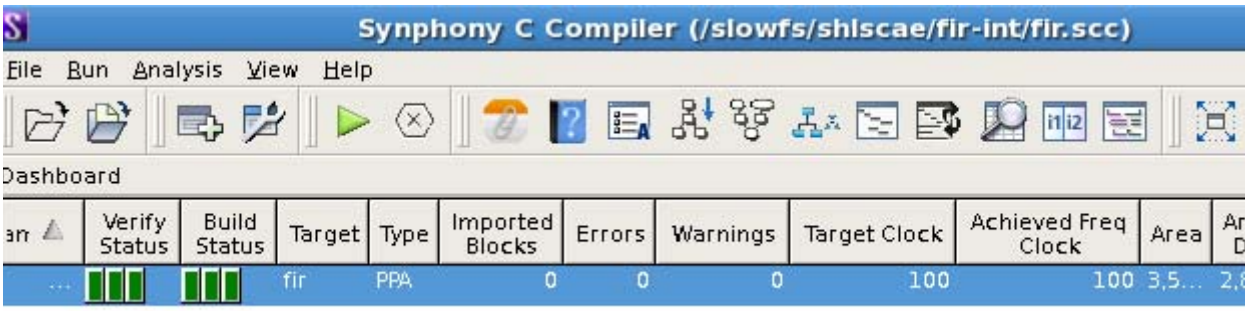
- The Dashboard Window
- The Console Window

- 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 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:

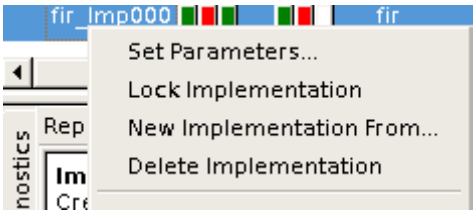


The window shows various implementation properties, which you can use for analysis. Note that the Achieved Clock Frequency differs from the Input Clock Frequency value if the tool cannot schedule operations at the desired input frequency.

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.

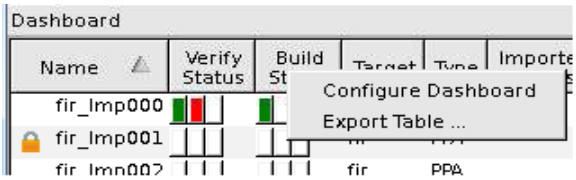


You can do the following operations in this window to manipulate the information and the design:

- 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

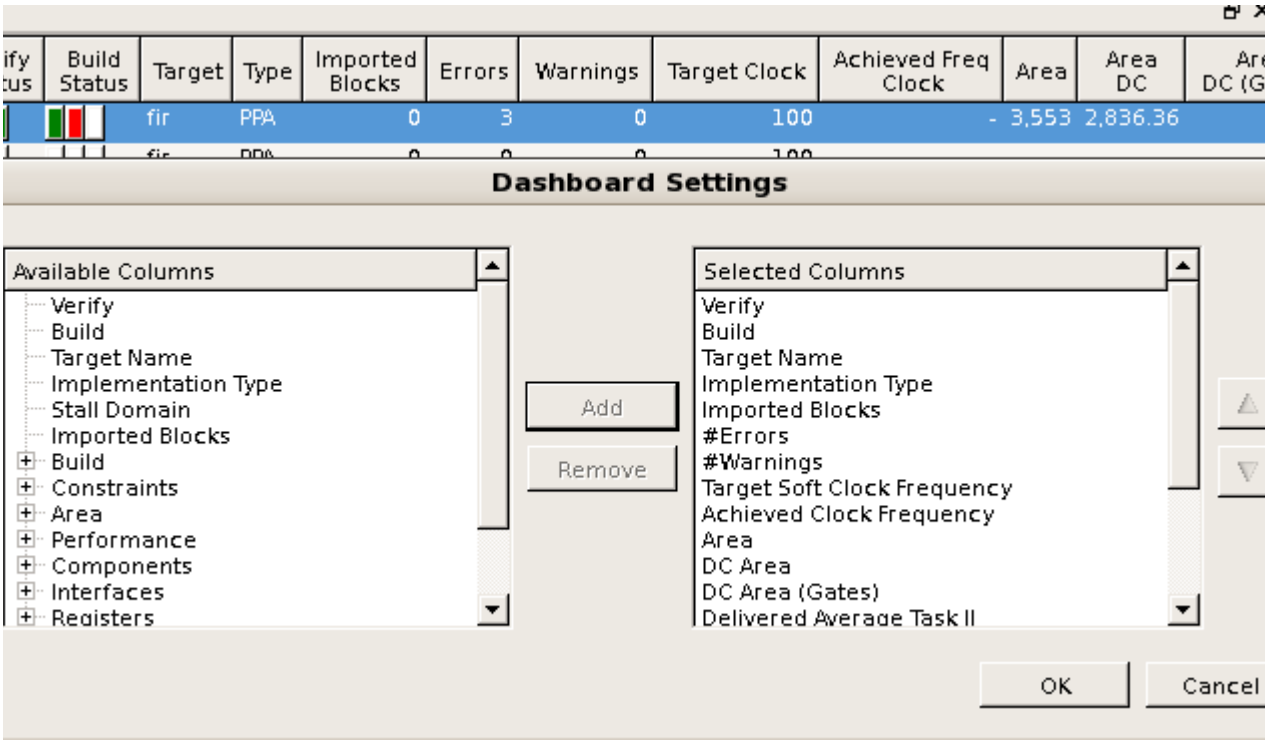
Right-click on the header in the Dashboard window, and select Configure Dashboard option.



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.

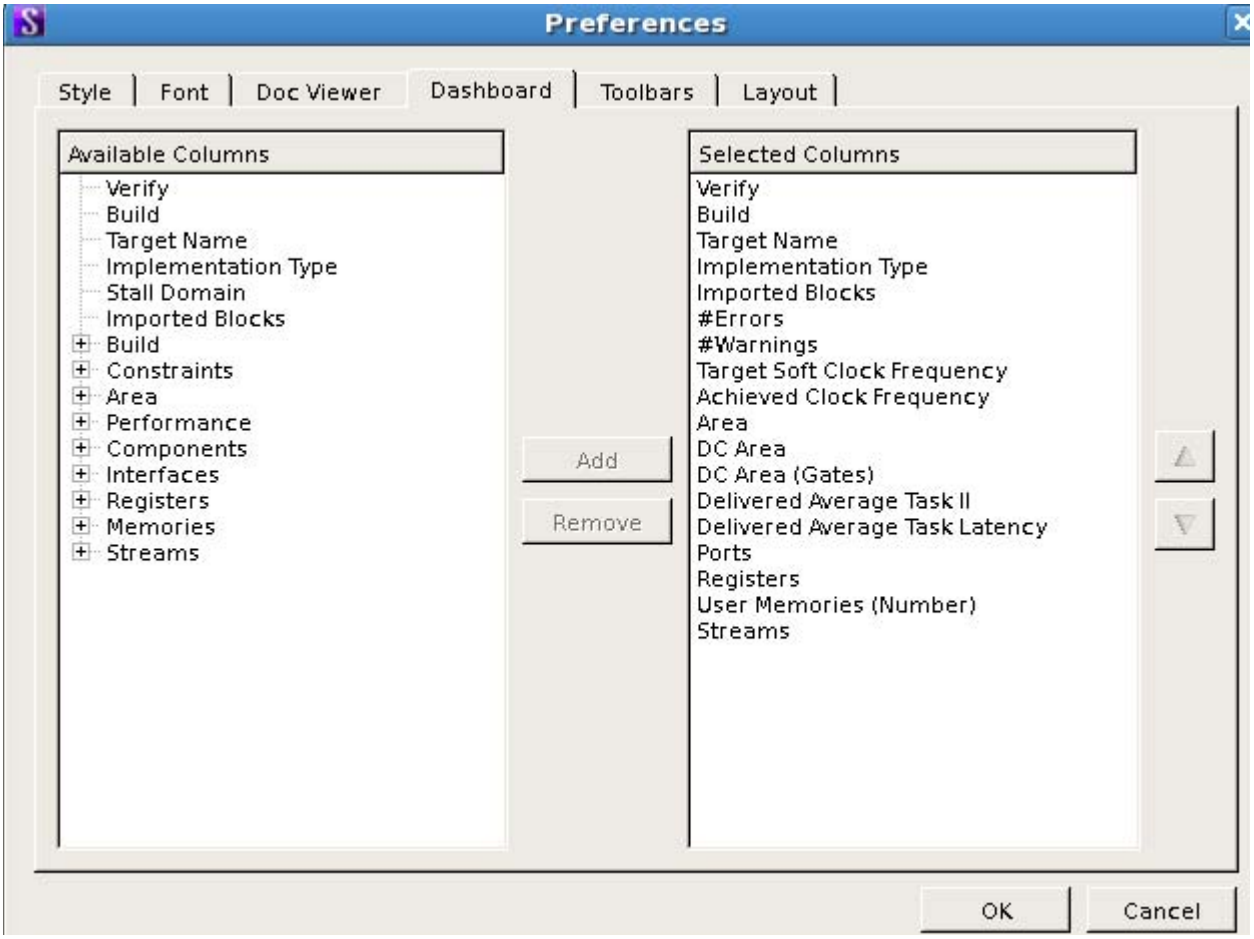
In the Dashboard, you can resize the column width. When you open a new session of the GUI, the column width setting is retained.

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

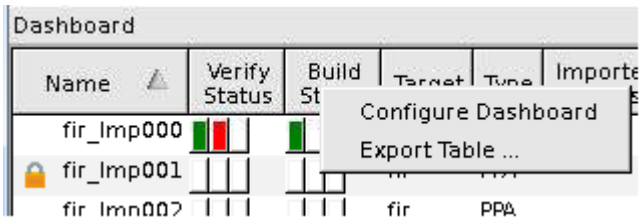
Dashboard settings.

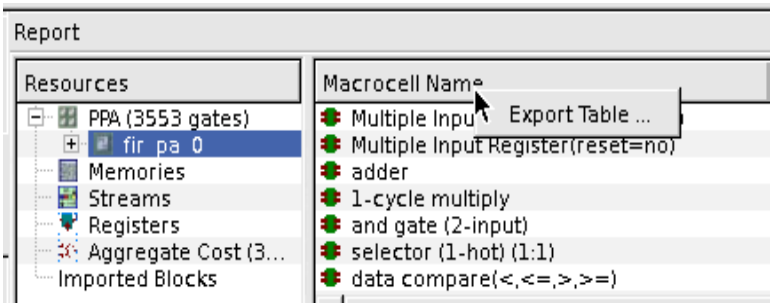


Exporting Implementation Data to CSV Format in GUI

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 .csv format from the GUI.

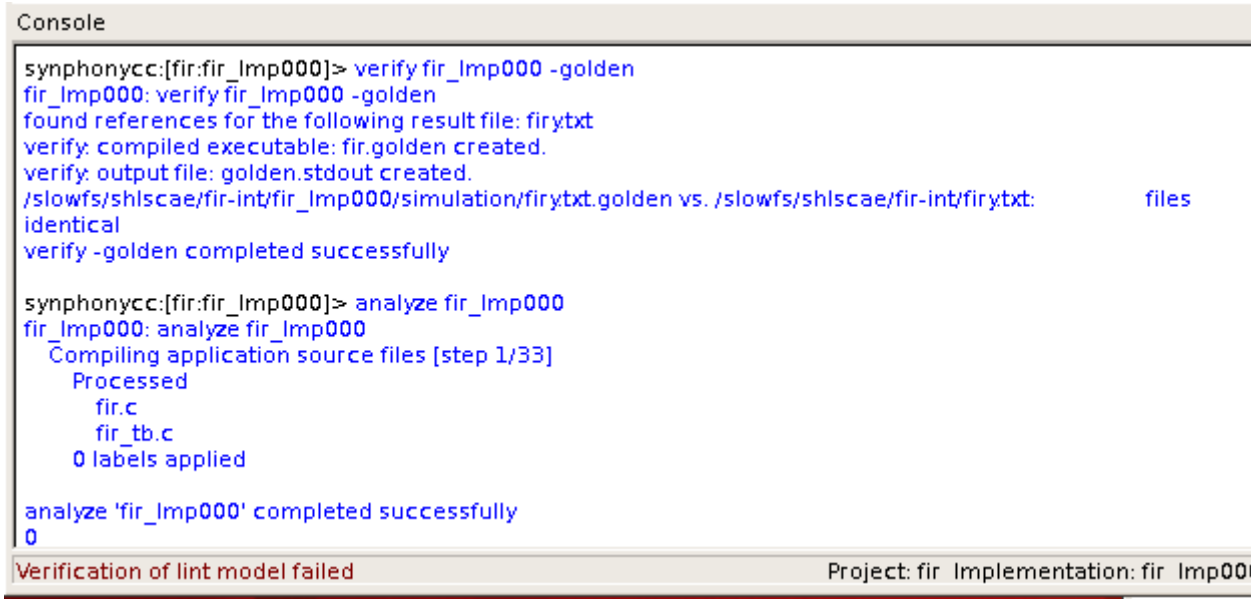
You can also export the hardware resources into a file in .csv format, when you right-click on the Resources tab in the GUI, as shown in the following figures:





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.



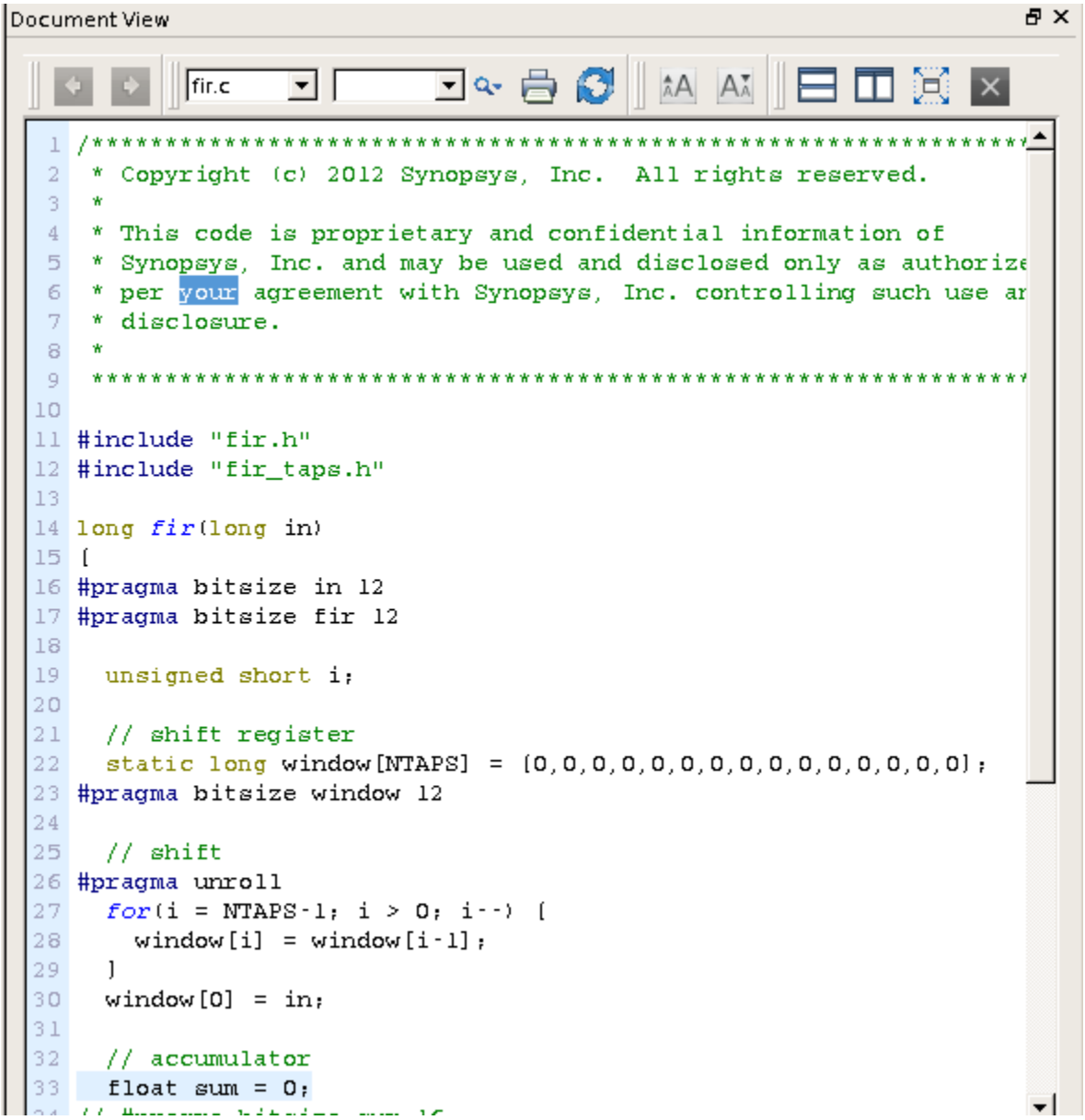
The console window lets you do the following to navigate and manipulate text in the window:

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.

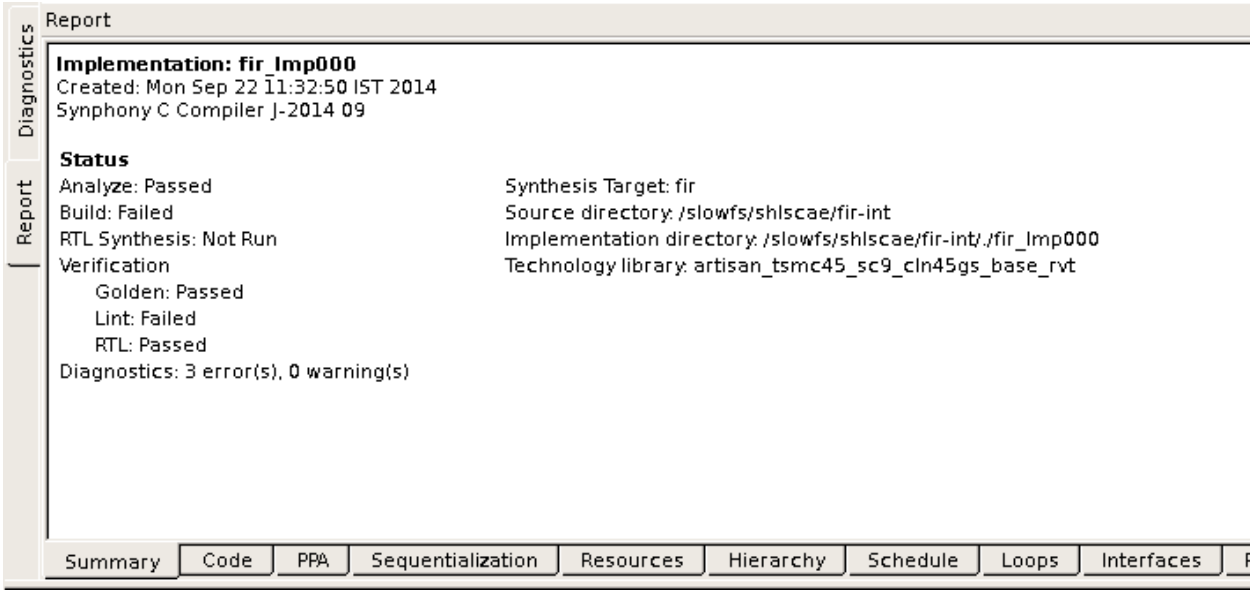
This section describes the GUI analysis views.

- 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:

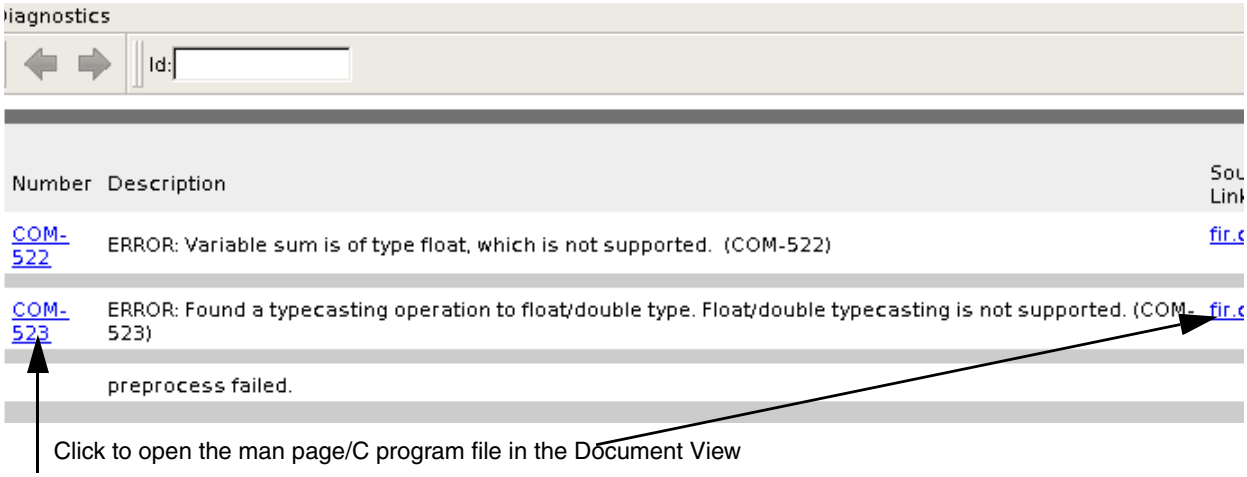
Report View Tab	Description
Summary Tab	Shows the results of the build and verification flow.
Code Tab	Contains a tree view of the C/C++ structure and hierarchy of functions, conditions, loops, and function calls in your code.
PPA Tab	Shows the PPA graph. Helps you understand the resource usage in your design.
Sequentialization Tab	Shows the Sequentialization graph. Helps you understand whether 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 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 tool and the rest of your RTL design.
Performance 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 in both the 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

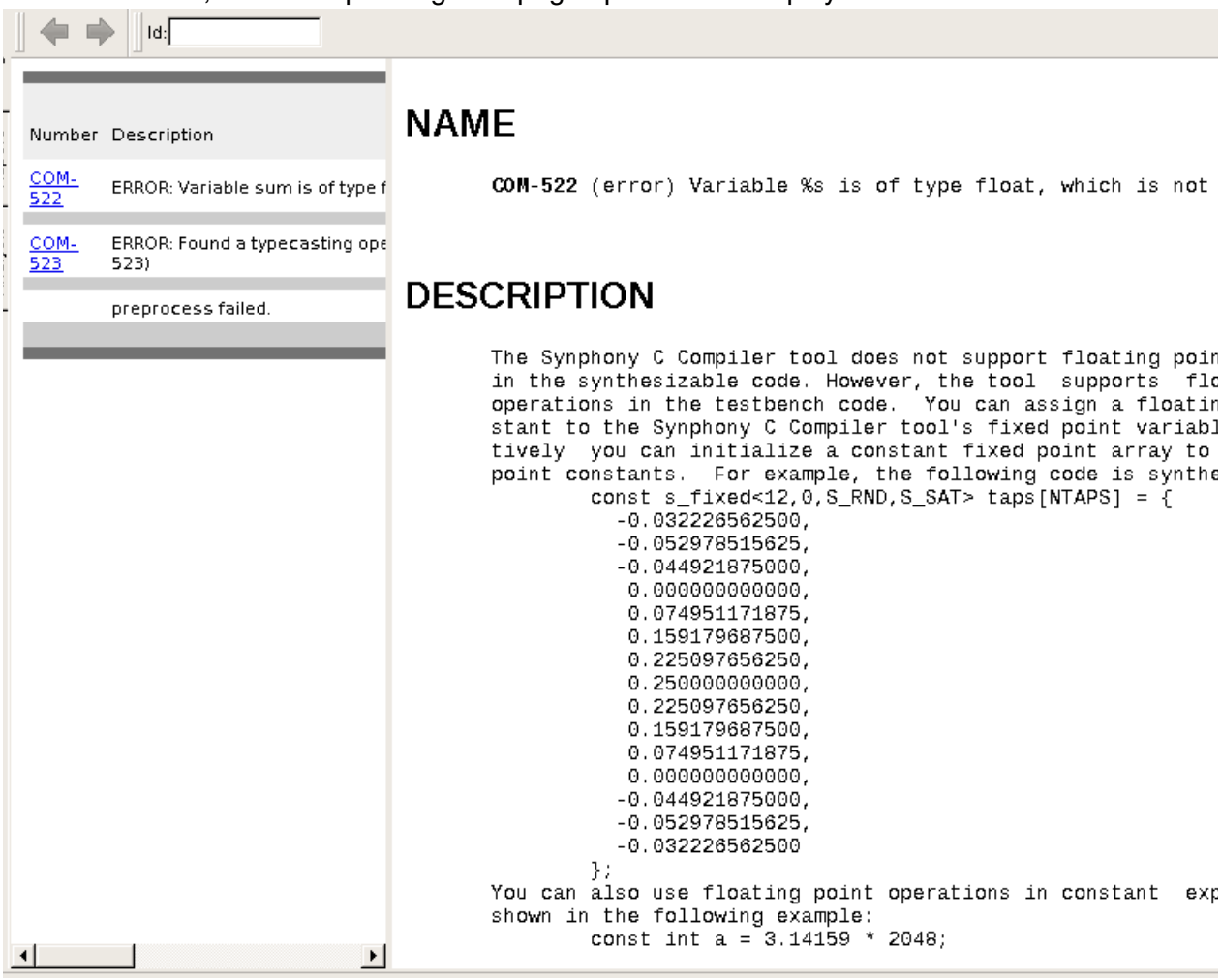
The Diagnostics View presents the errors and warning messages of the most recent run in an organized format. Multiple errors of the same type are reported together.

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.



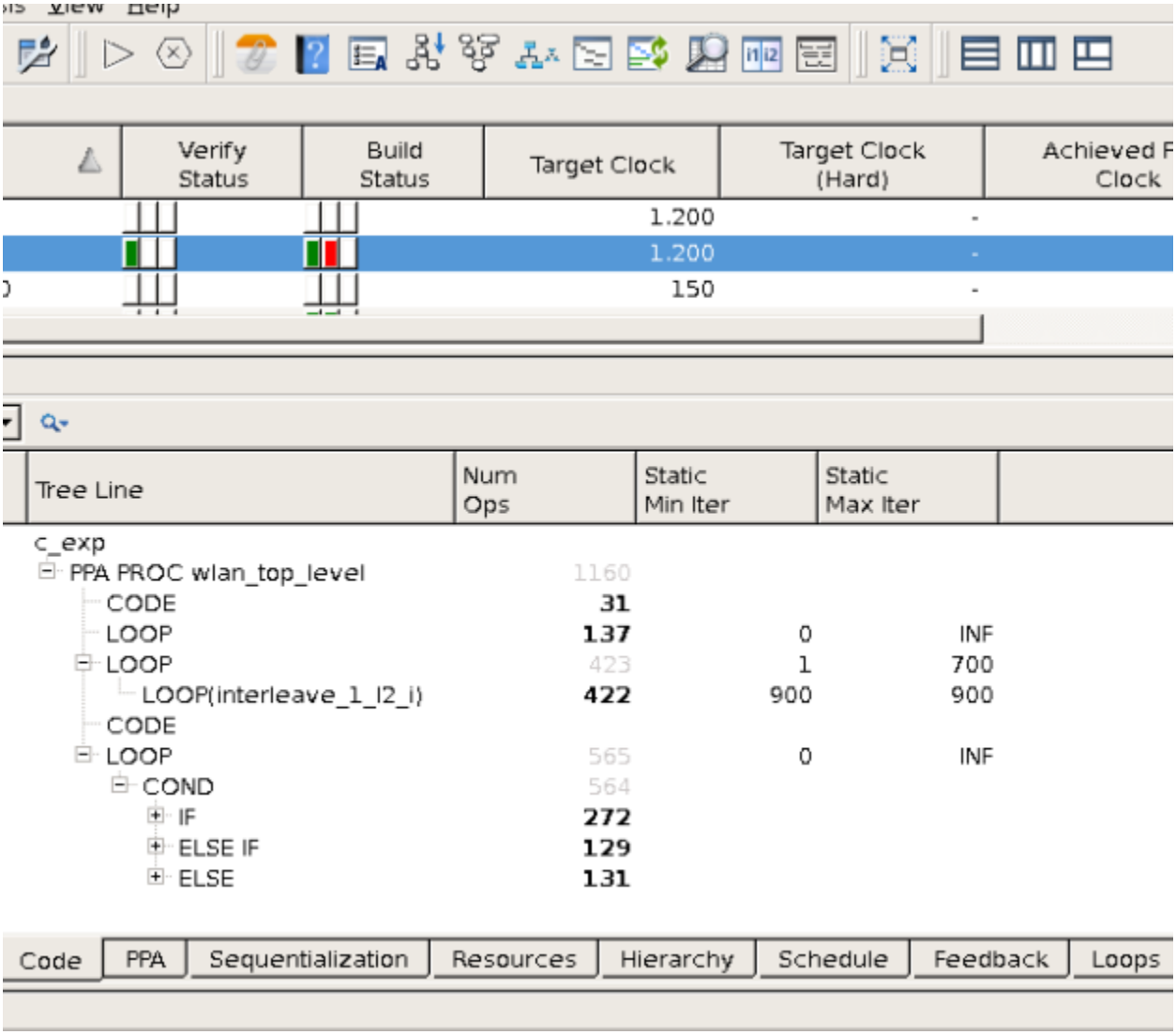
You can click the blue error number to display the man page of the error, or click a blue file name to open the file in a Document View window. The triangle buttons at the top let you navigate forward and backward through various views, as in a web browser. To show or hide errors, warnings, and notes,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



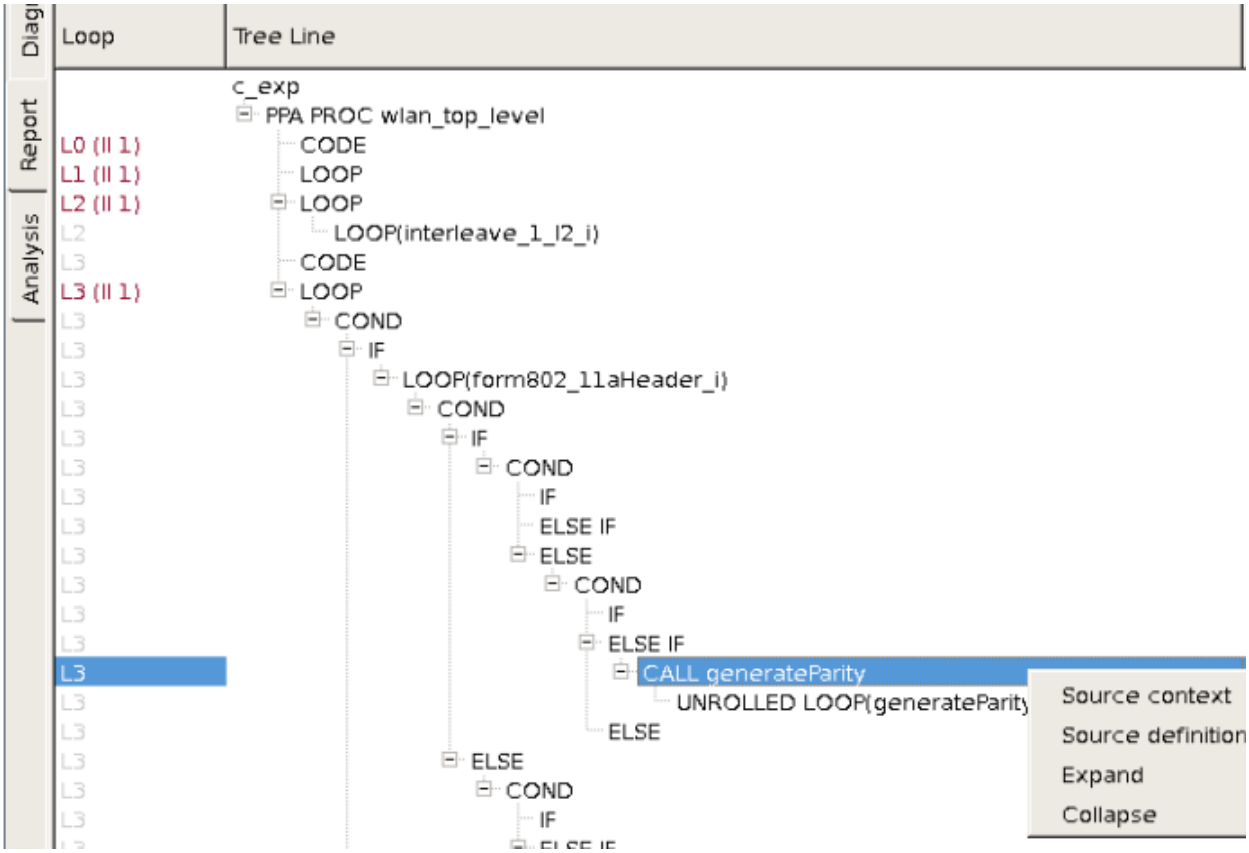
This view consists of the following columns:

Header	Description
Tree Line	Displays the procedure, TCAB or PPA and condition branches within each loop in a tree lined fashion.
Loop	Displays the loop numbers for each item and the II of the loop nest shown in the Tree view.

Header	Description
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.

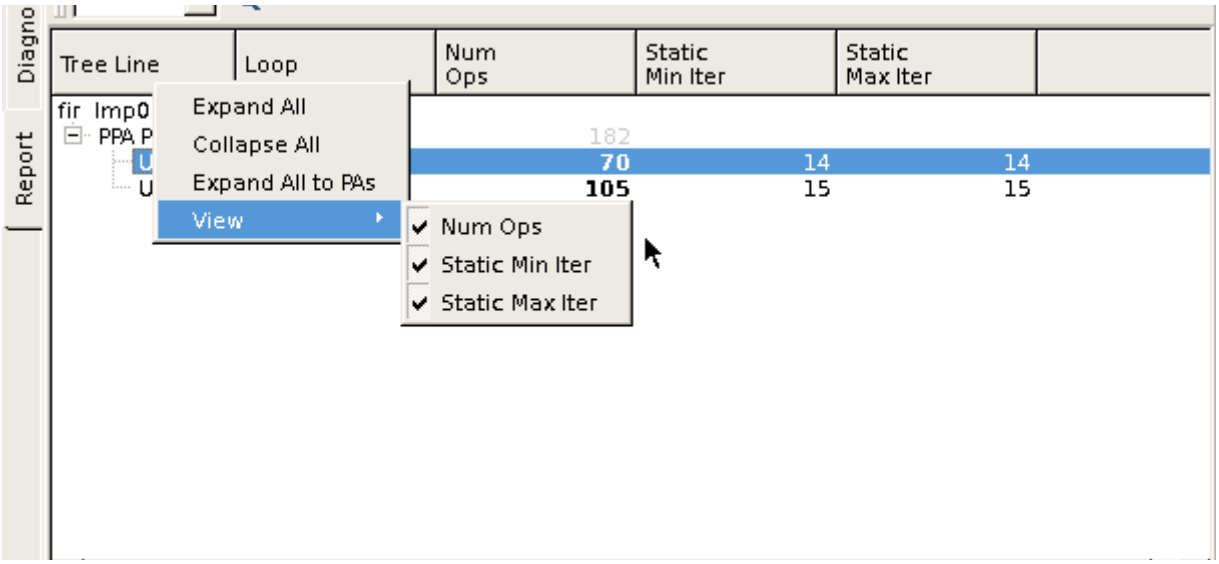


The Node menu commands are:.

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.



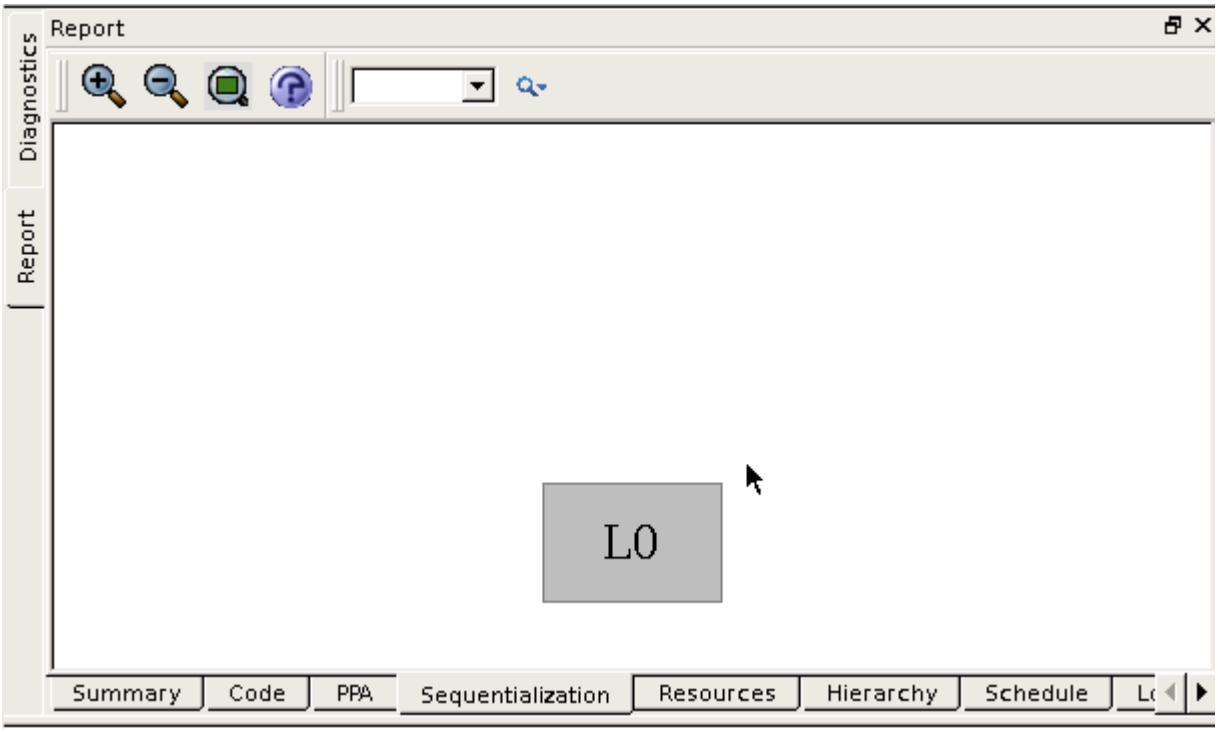
The Line Header menu commands are:

Command	Description
Expand All	Expands the entire tree view.
Collapse All	Collapses the entire tree view to display only the major nodes (Loops).

Command	Description
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.

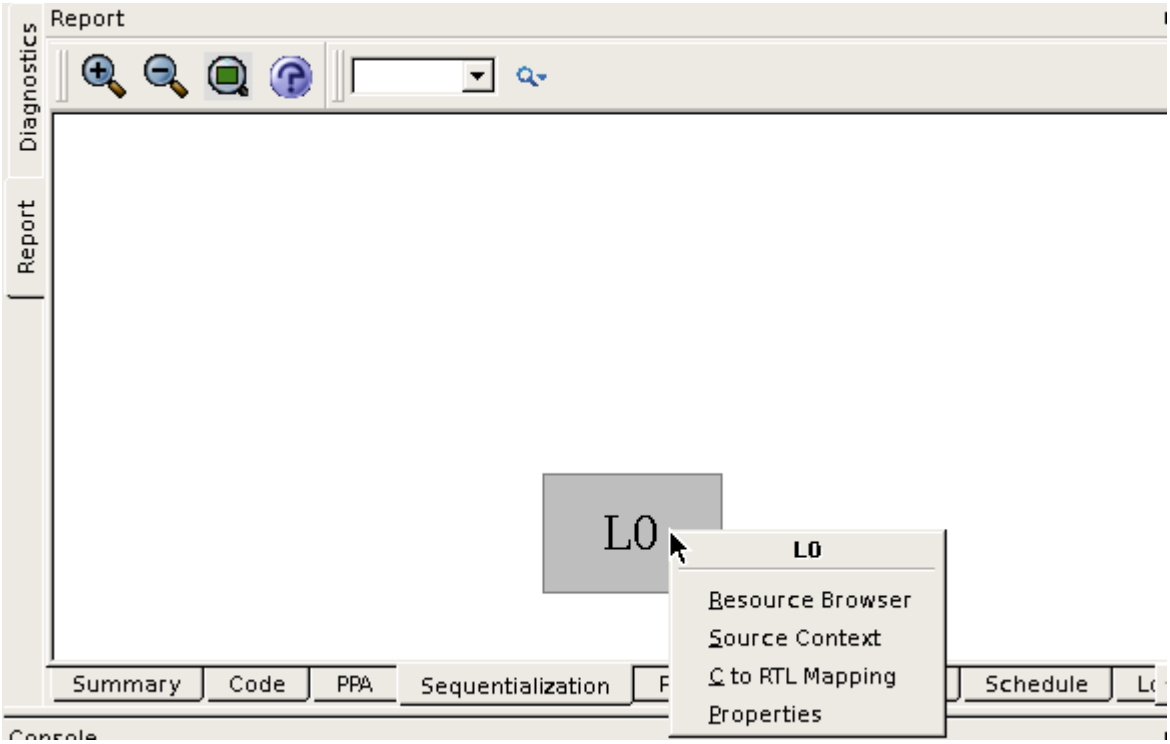
Toolbar Icons

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

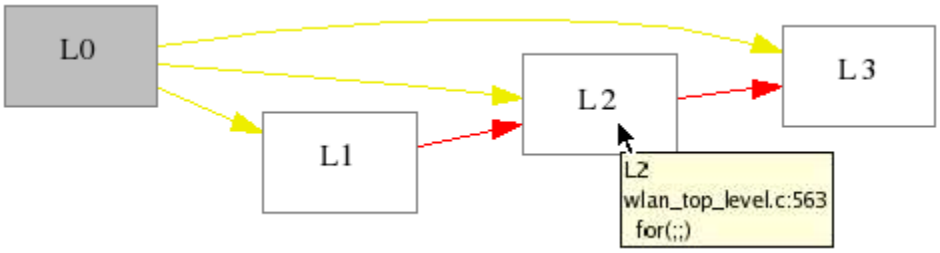
Right-click an object in the graph to see the context-sensitive menu for the object. The following figure shows the menu for loops.



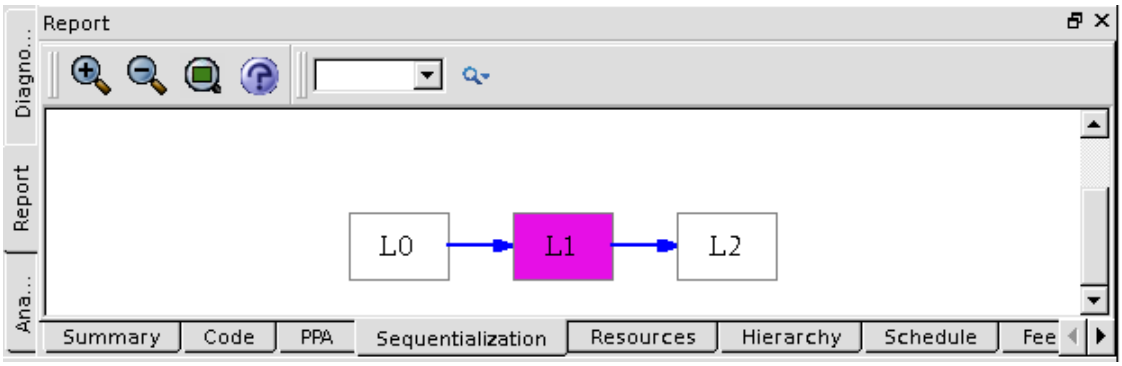
This table describes the commands:

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 2-56.
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 2-66.
Properties	Displays loop information for the selected element. See "Object Properties" on page 2-28.

Hovering over an object shows the ToolTip with the label.

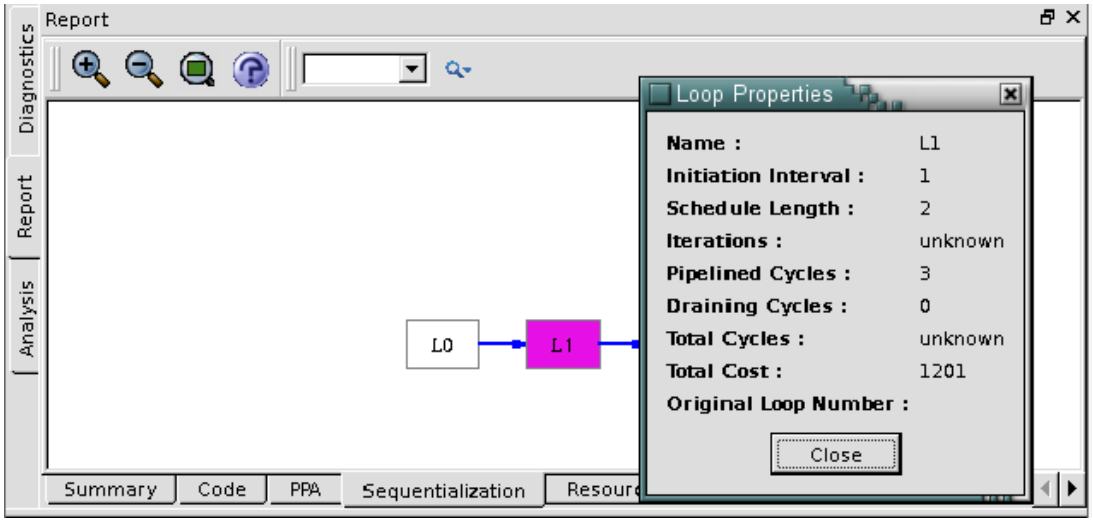


Clicking an object highlights it and makes all its incoming and outgoing edges bold. Clicking on the text highlights the text and opens the source context.



Object Properties

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

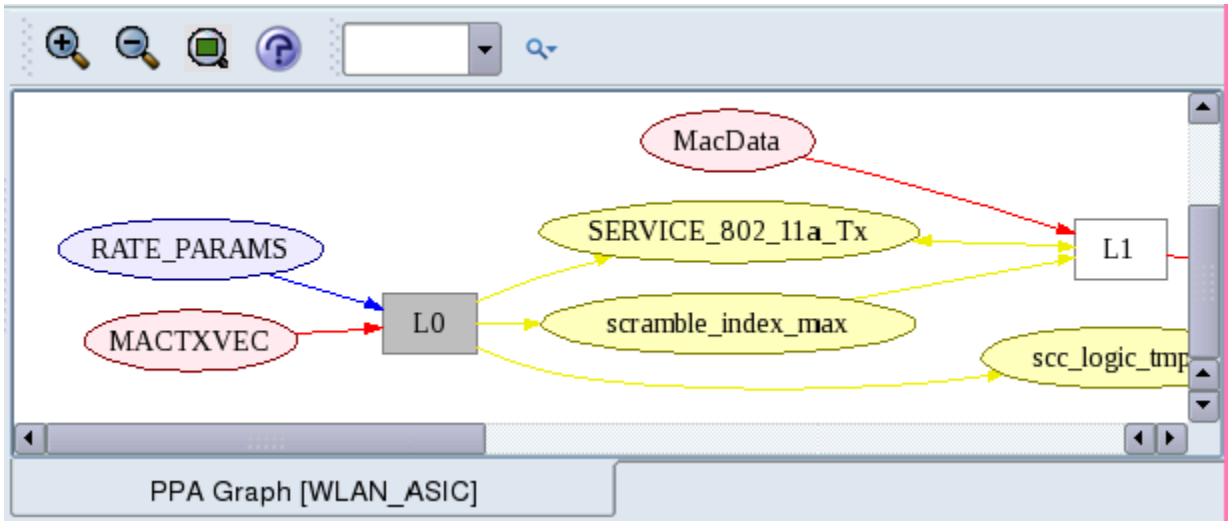


The Loop Properties window displays the following information of the selected element:

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.

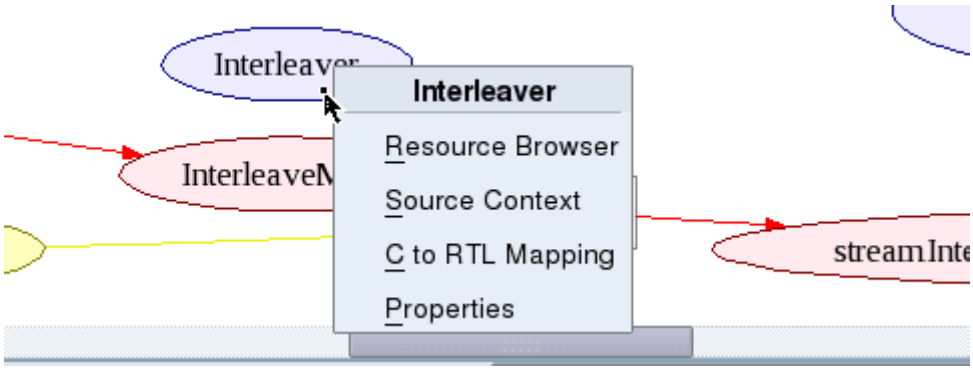
Red arrows	FIFO communication.
Blue arrows	Stream communication.
Yellow arrows	Scalar communication.

See the following for more information about the PPA Graph:

- "Toolbar Icons" on page 2-26
- "Context-Sensitive Information" on page 2-31
- "Loop, Stream, Memory, and Scalar Properties" on page 2-33

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.

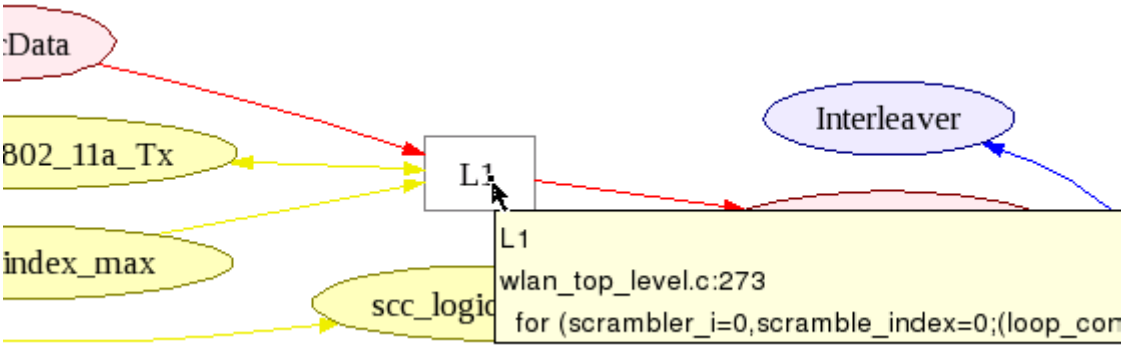


The following table describes the commands in the menu:

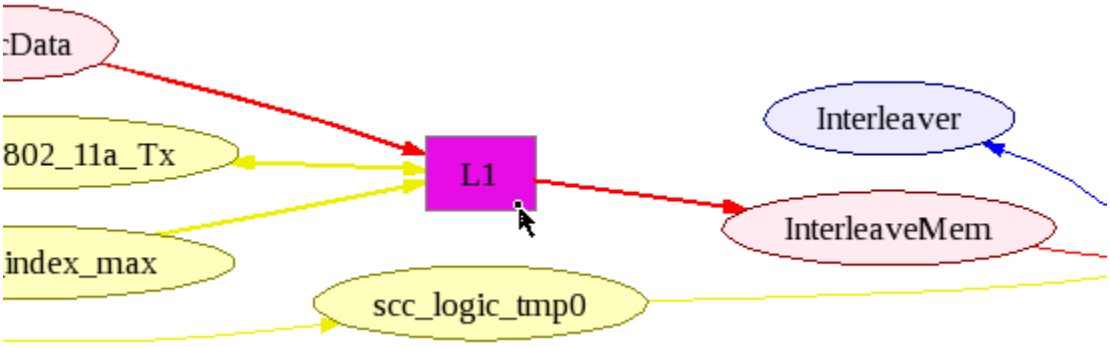
Commands	Description
Resource Browser	Displays the hardware resource corresponding to the selected element in the Resource Browser. See "Resource Browser for PPA Graph" on page 2-59.
Source context	Opens a split text editor window with C or C++ source code. The cursor is placed on corresponding source 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 2-33 for details.

Hovering over an object shows the ToolTip with the label:

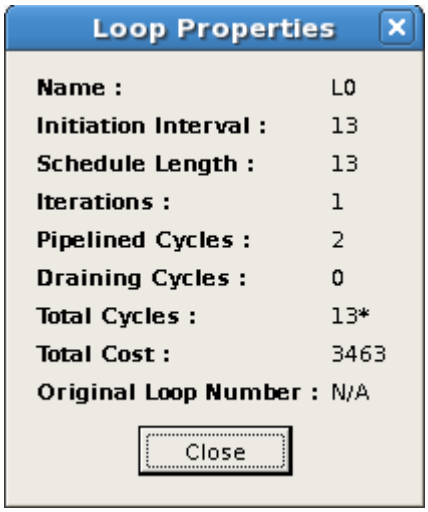


Clicking on an object highlights it and all its incoming and outgoing edges become bold. Clicking on the text highlights the text boundary and opens the source context.



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.

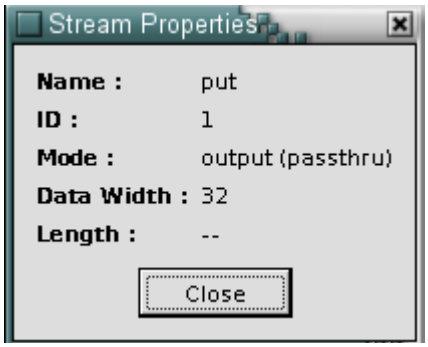


The Loop Properties window displays the following information of the selected element:

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



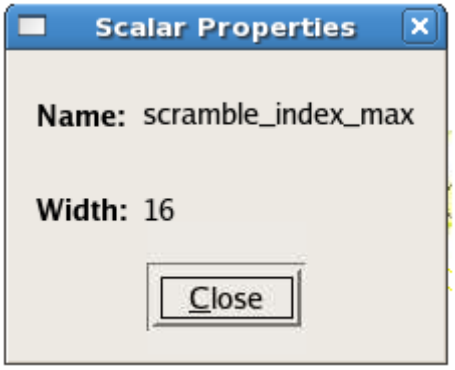
Properties	Description
Name	Specifies the name of the loop.
Identification Number	Specifies the time gap between two successive runs of the loop.
Mode	Specifies the length of the instruction schedule in clock cycles.
Data Width	Specifies the number of iterations in the loop nest.
Length	Specifies the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

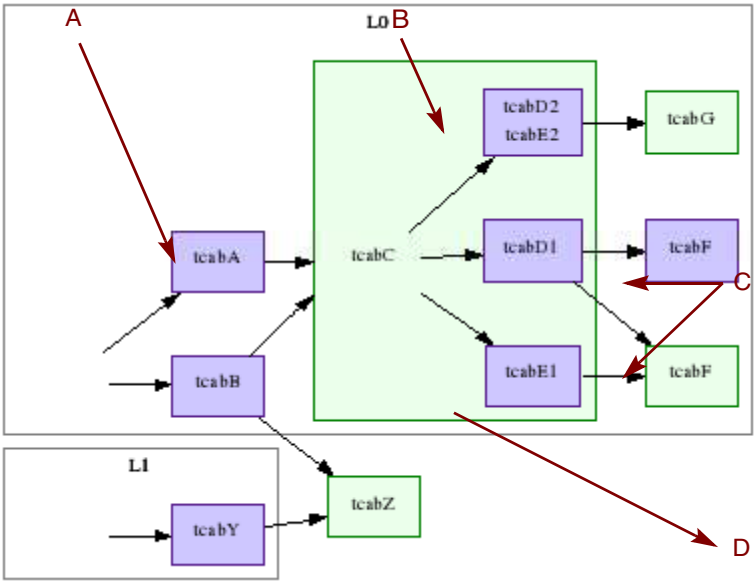


Property	Description
Name	Specifies the name of the scalar.
Width	Specifies the width of the scalar.

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.

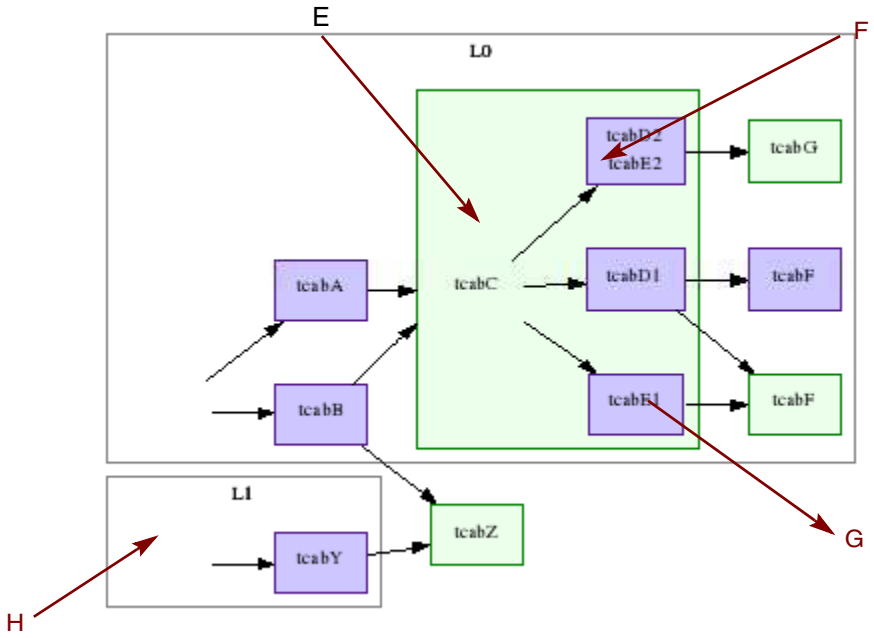


A A string in a box
The string is the name of the instantiated TCAB function or loop.

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.

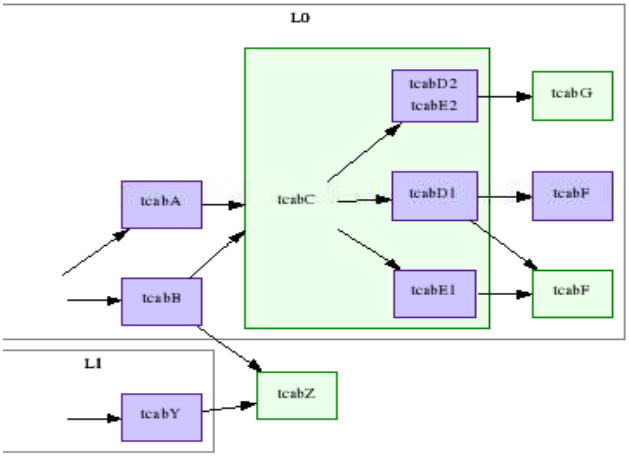
The following figure and table describes the arrows in the Hierarchy Graph.



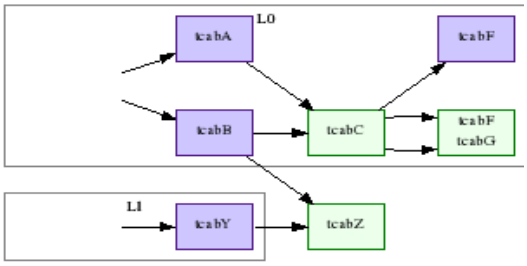
E	Arrow The start of the arrow indicates the caller and its end indicates what is being called.
F	Start of an arrow connected to a box The callers are all the TCAB instances or loops associated with that box.
G	End of an arrow connected to a box The instances being called are the TCAB instances associated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



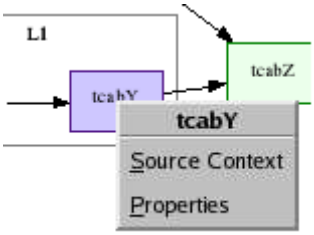
Detailed View



Top-level View

Context-Sensitive Menu

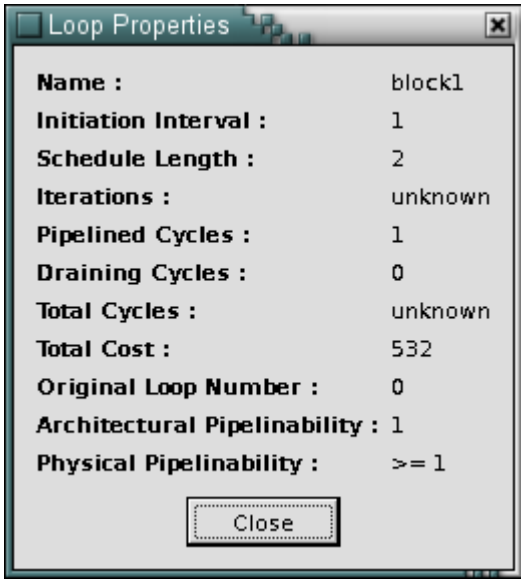
Right-click text to see the properties and source for that particular TCAB instance.



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.

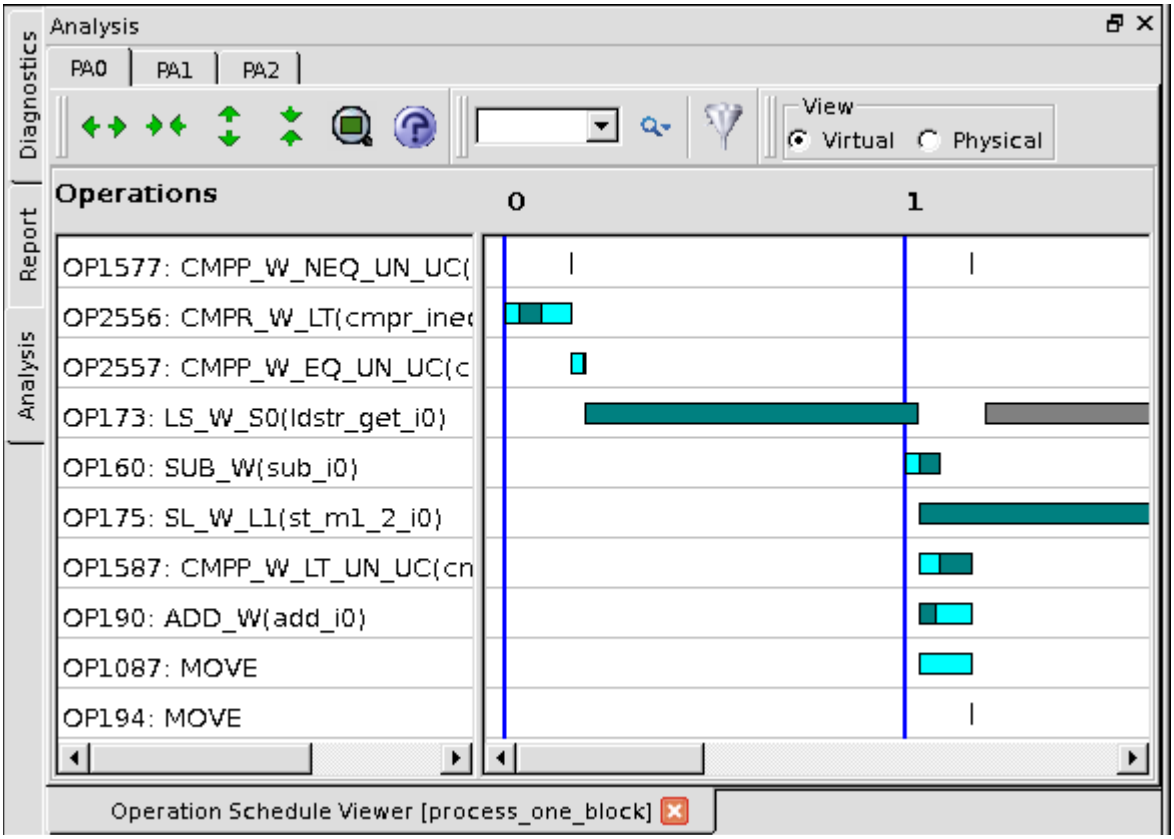


Operation Schedule Viewer

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.

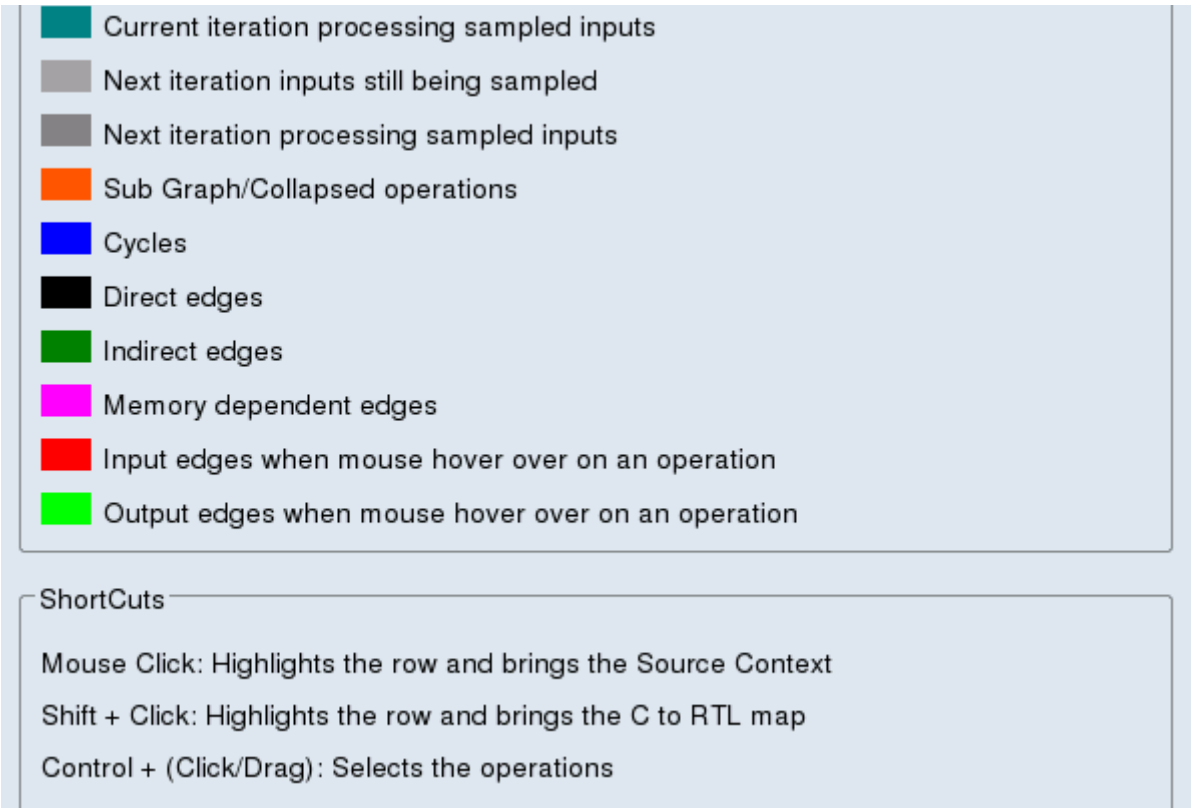
The Operations list box on the left shows a list of compiler op numbers, the compiler opcode, and the functional unit name for operations other than MOVE.

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.

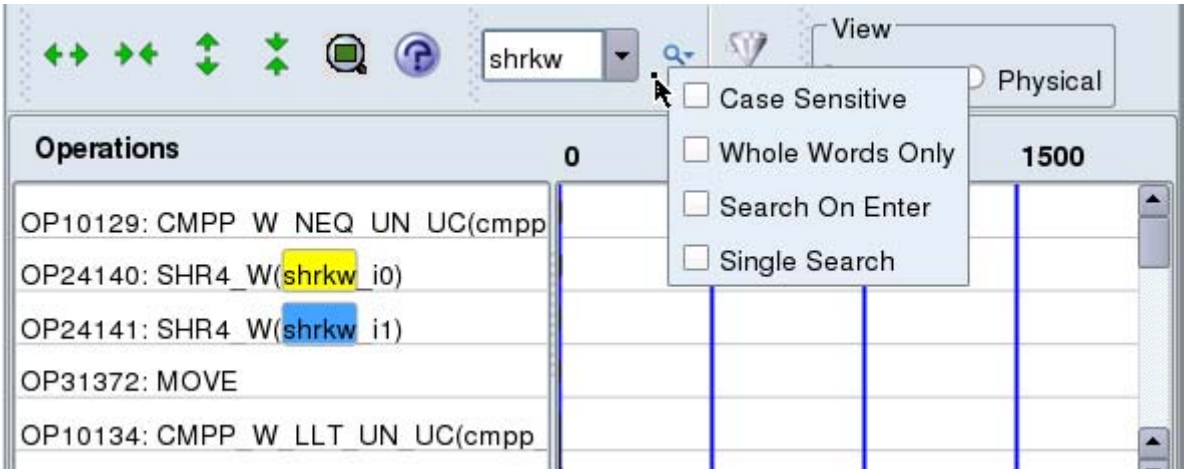
Operations that participate in feedback paths show a second, gray-colored bar. This bar shows the next execution of that operation in a subsequent loop iteration.

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:



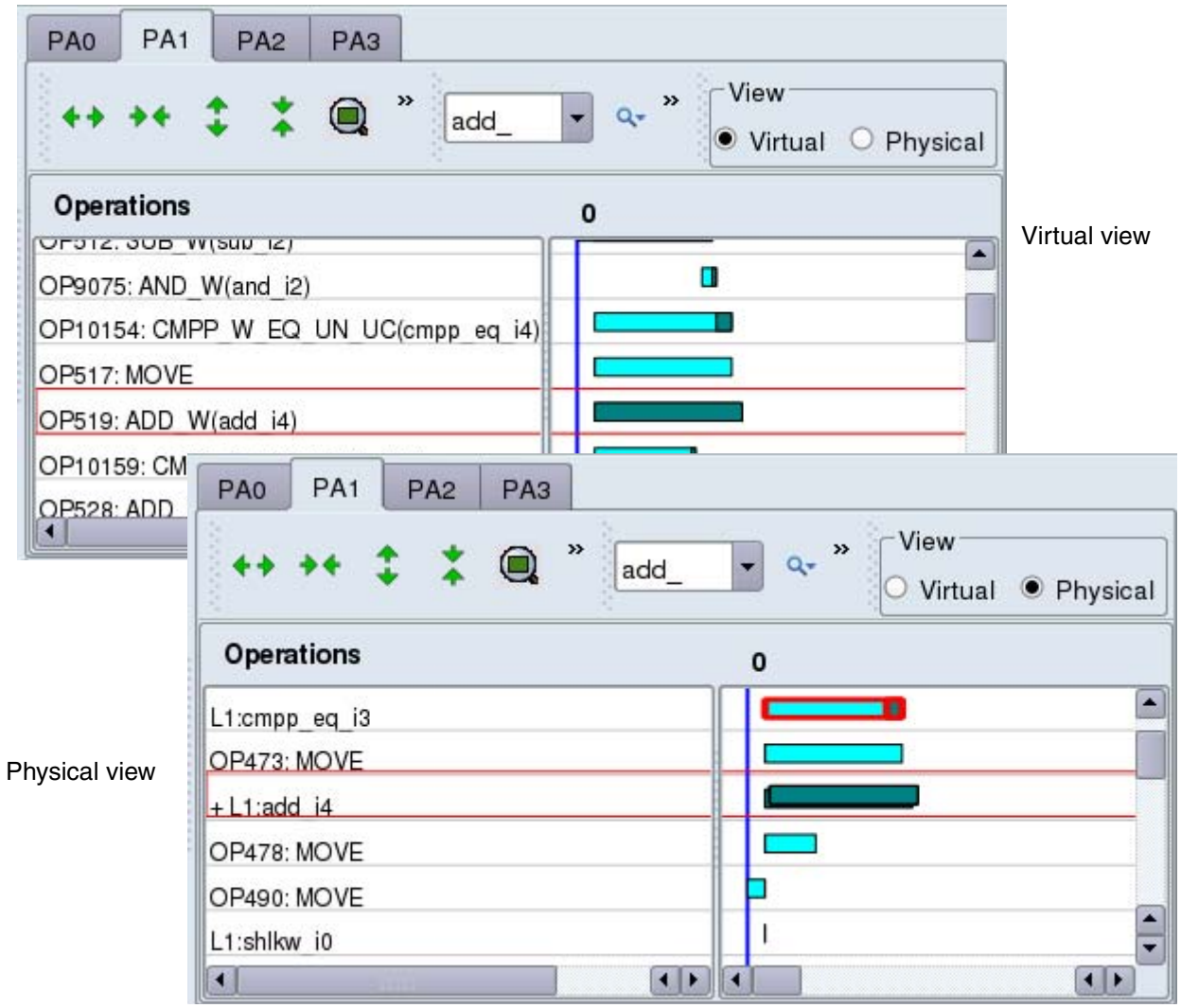
The Search text box lets you search for occurrences of a string within the Operations list. To change the search options, use the magnifying glass button to the right of the search field:



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:

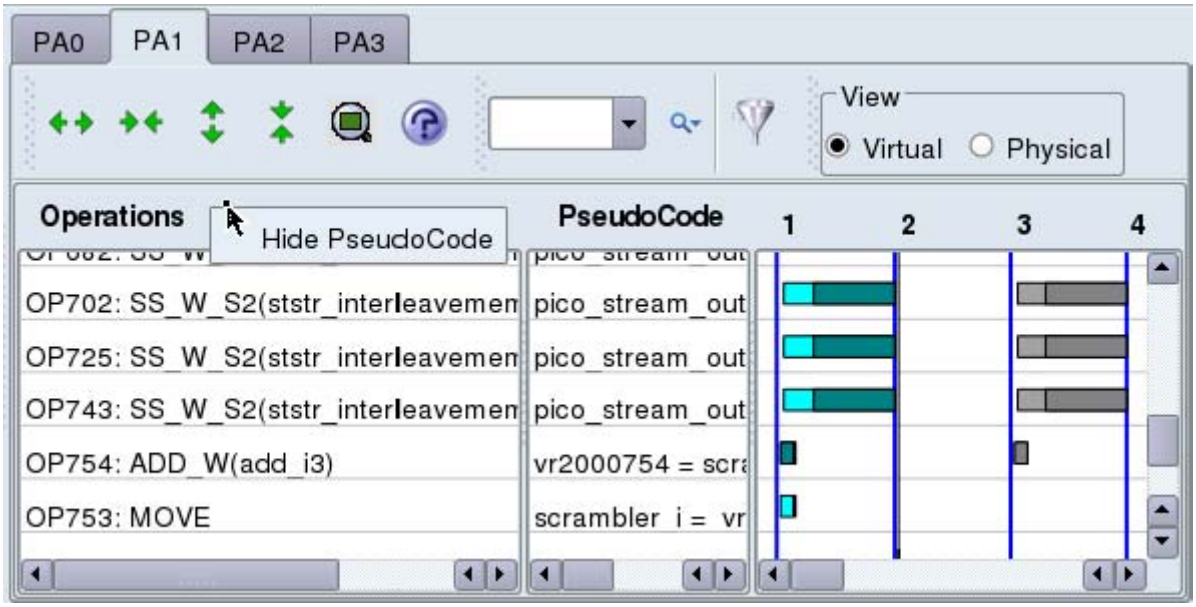
- Virtual – The program view. Each row shows an individual operation from your code.
- Physical – The hardware view. Each row shows a functional unit from the hardware. If multiple operations are mapped to one unit, the row shows the operations overlapping. Note that only two overlapped rectangles are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



Virtual view

Physical view

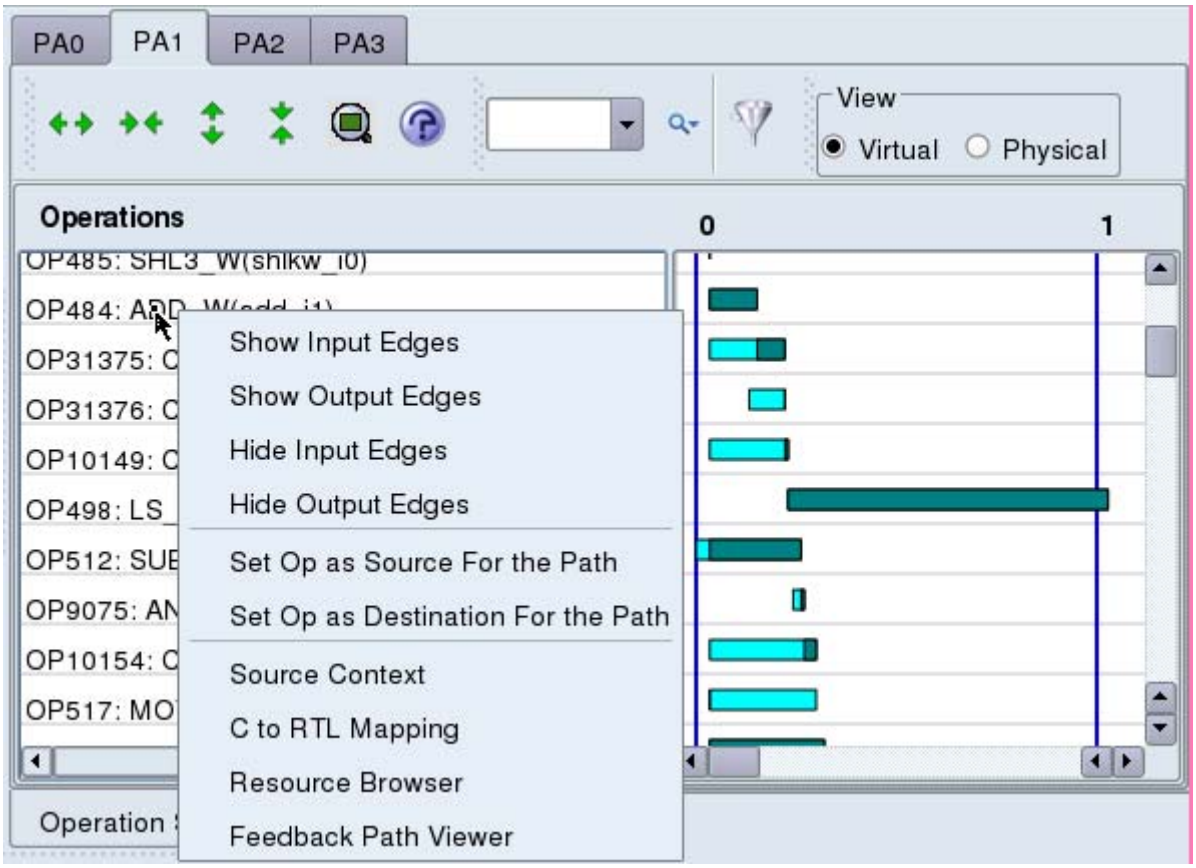
Right-click the header at the top of the Operations list or cycle graph to show or hide the PseudoCode view. Pseudo code is a detailed description of each operation showing the input and output variables. The strings are available for searching.



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

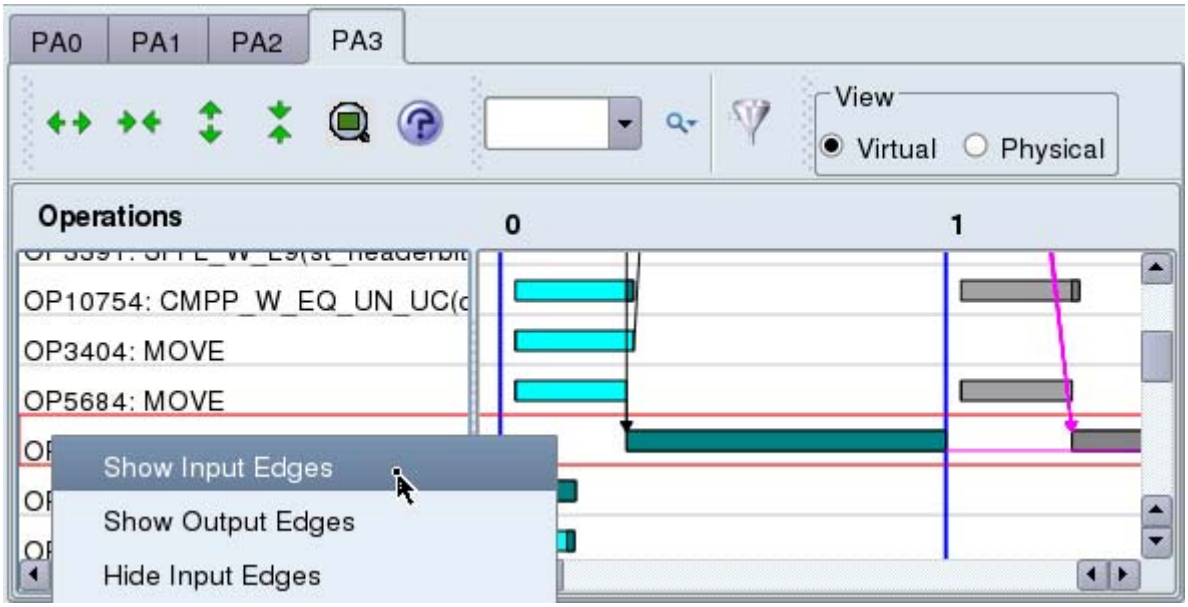
The following figure shows the menu displayed by right-clicking an operation.



Edges and Paths

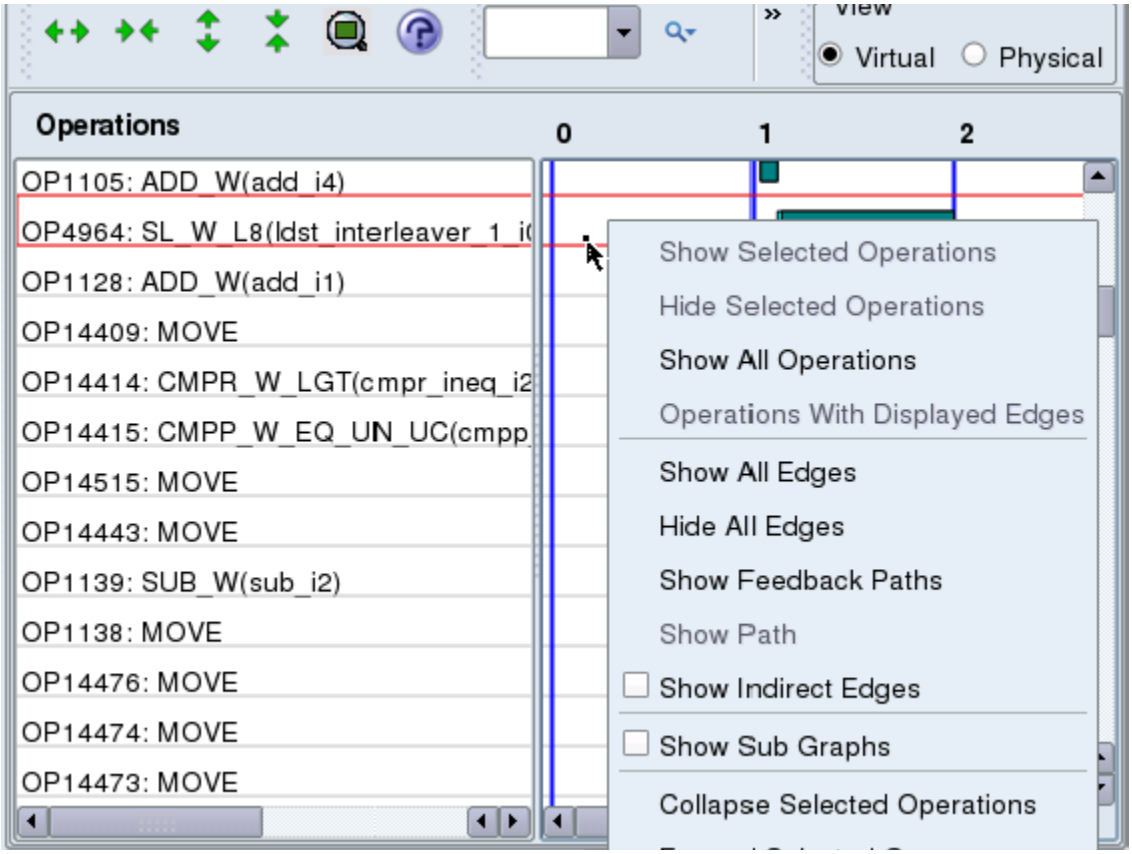
A path is a sequence of edges starting from a source operation and ending at a destination operation. You can show or hide edges, and view paths . To show or hide edges, right-click the operation, and choose the appropriate options such as Show Input Edges and Show Output Edges.

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.

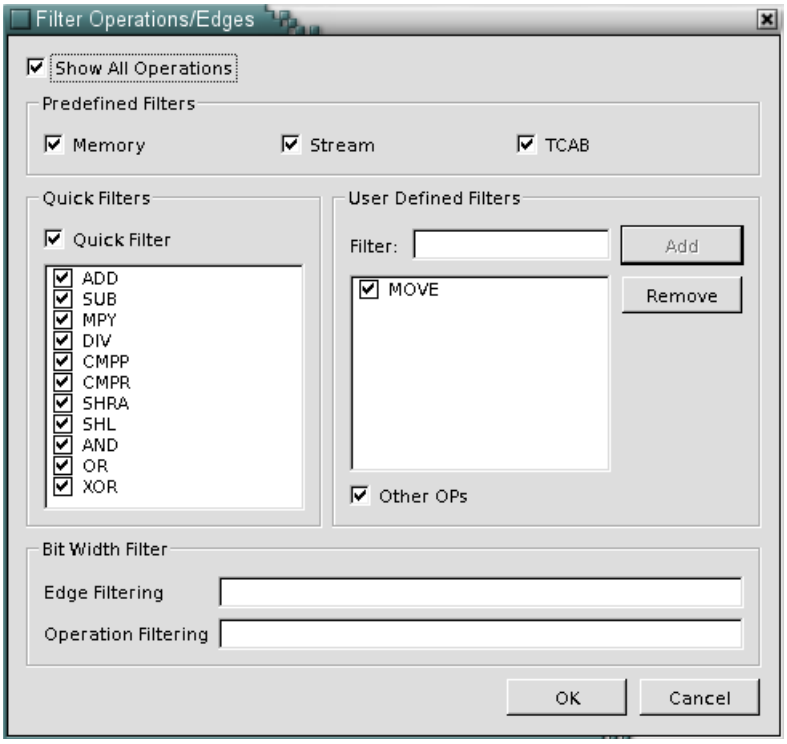
Menu Command	Description
Show Path	When one operation is selected as the source and another as the destination, this option shows the path between them. If operations are hidden between this path, it brings them into view.
Show Indirect Edges	Shows the indirect edges. If any operation is not visible, 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

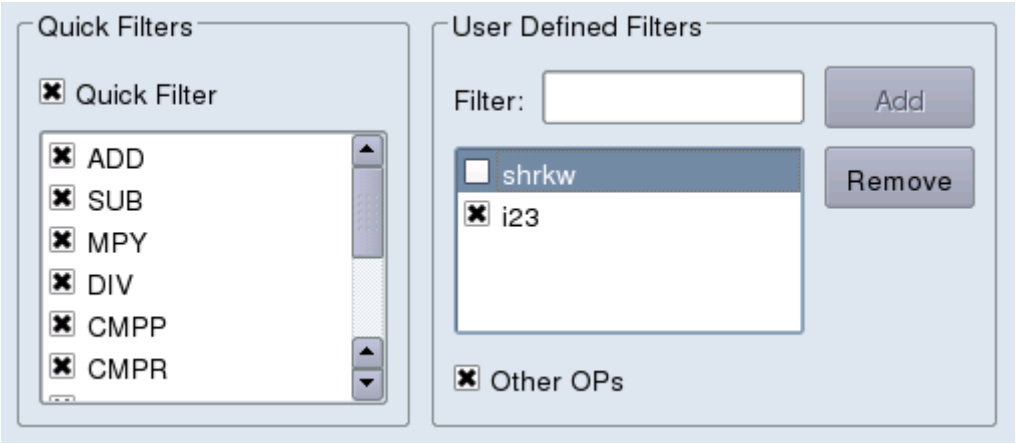
To restrict the list of operations to only those that meet the specified criteria, click the Filter Funnel button. This opens the Filter Operations/Edges dialog box.

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.

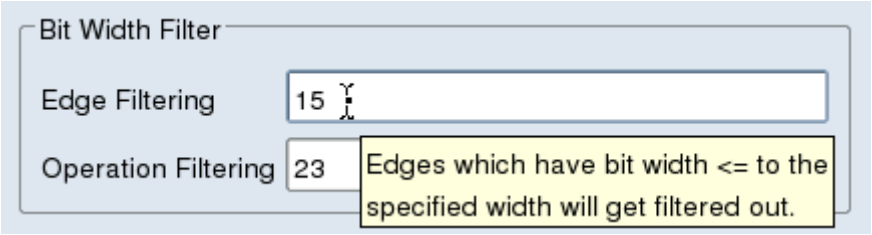
Use the User Defined Filters section of the dialog box to enable or disable the display of operations containing a specified string. Enter the string in the Filter field, then click Add to add the string as a filter. Then, you can enable or disable the display of operations containing that string.



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.

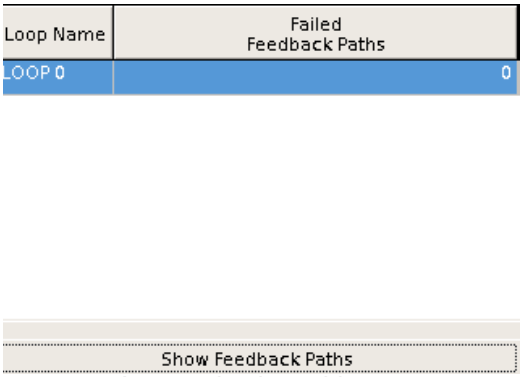


Feedback Path Viewer

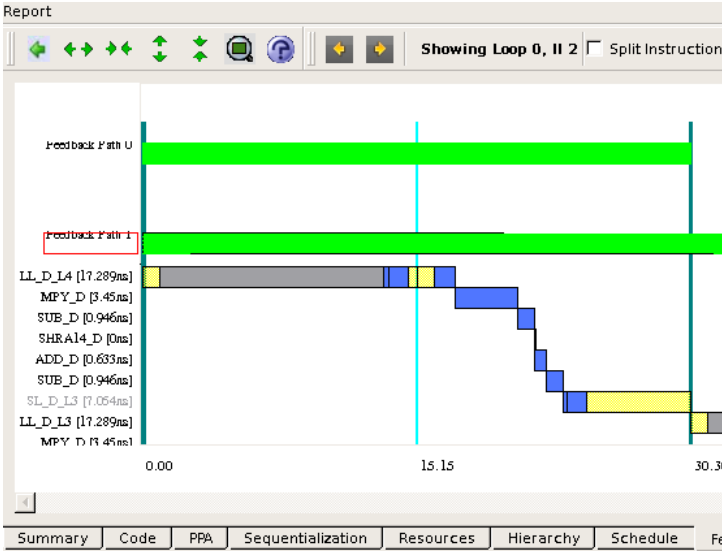
The feedback path viewer shows the critical feedback paths in the design on a per-loop basis. Each feedback path contains details about the operations involved in the feedback, their corresponding source positions (where available), and many other properties. You can analyze the feedback paths and take 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.

Button



Loop selection table



Feedback paths in a loop

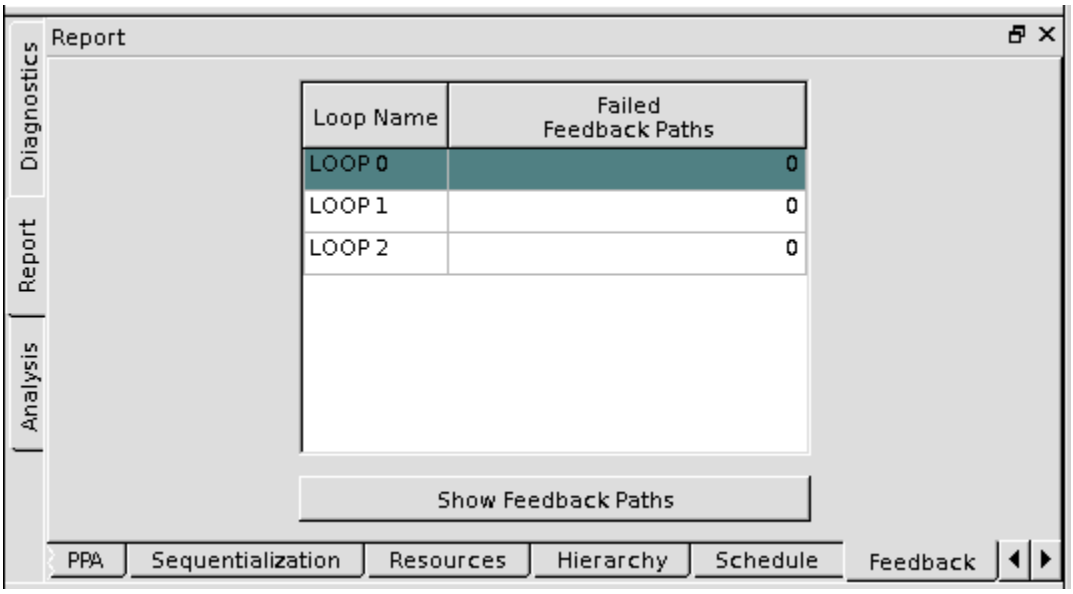
The Feedback Path Viewer displays information about the feedback paths:

- 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.

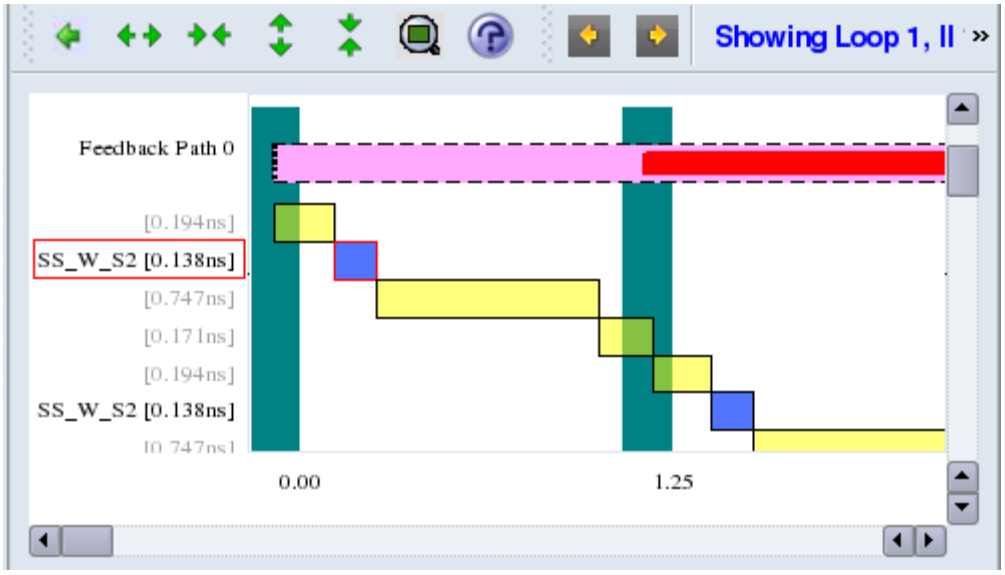
Loop Selection Table

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

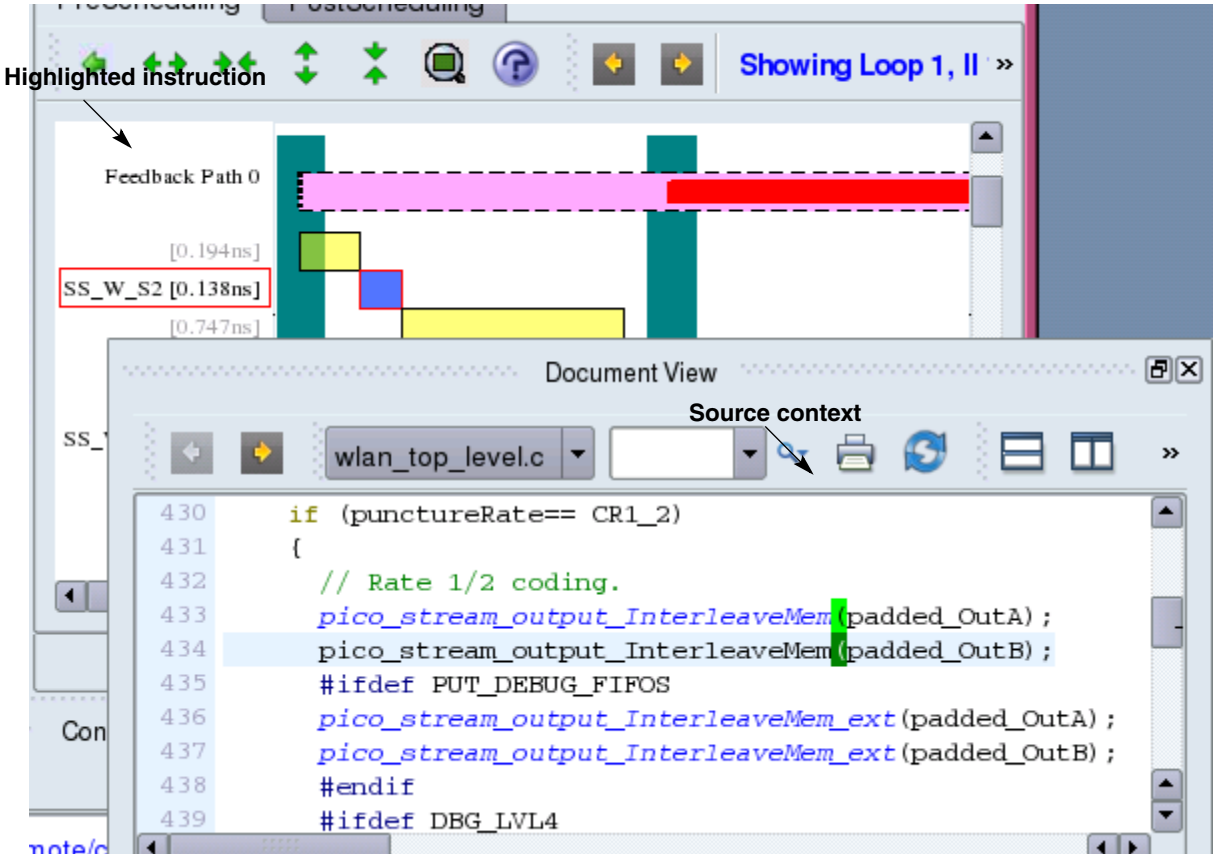
Time in nanoseconds is used as the scale for the x-axis. The initiation interval is marked with a different color to clearly distinguish it.

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.

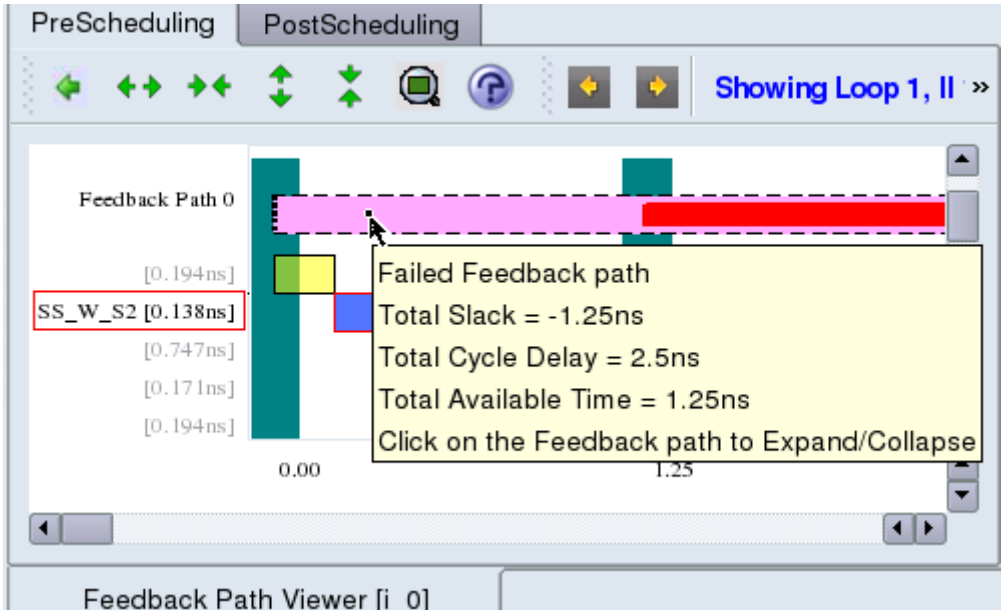


When multiple instructions point to the same source code, you can hover the cursor over the highlight in the source code to view the list of opcodes in a ToolTip window.

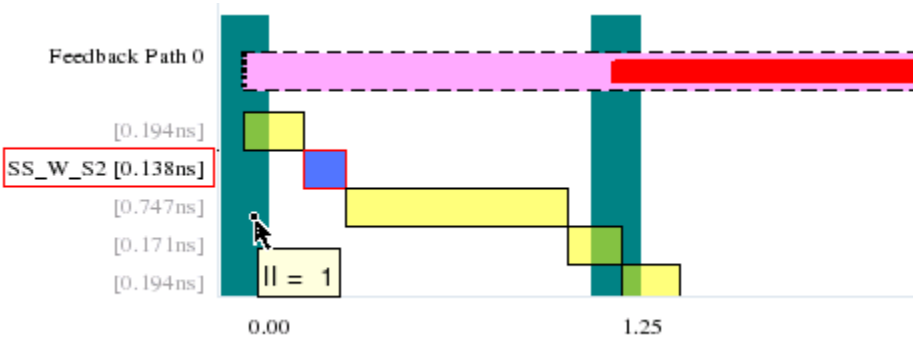
Feedback Path ToolTip

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.

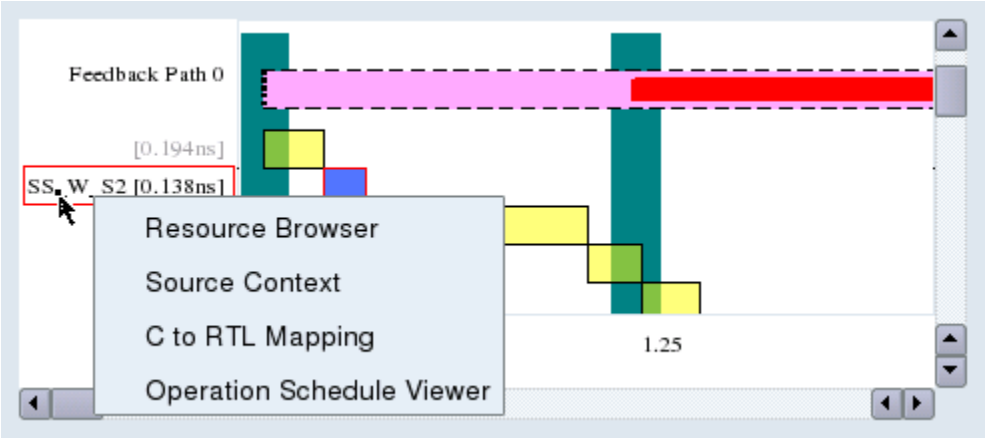


- Initiation Interval
  Displays initiation interval information for that loop:



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 2-61 or "C to RTL Mapping" on page 2-66 or "Operation Schedule Viewer" on page 2-40.

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.

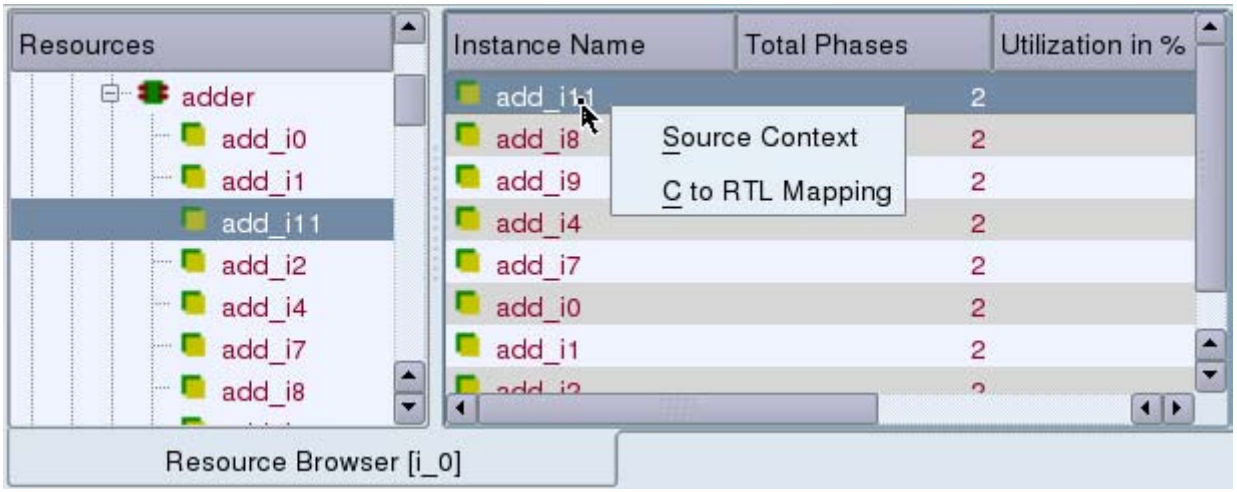
Element Properties in Tree View

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.

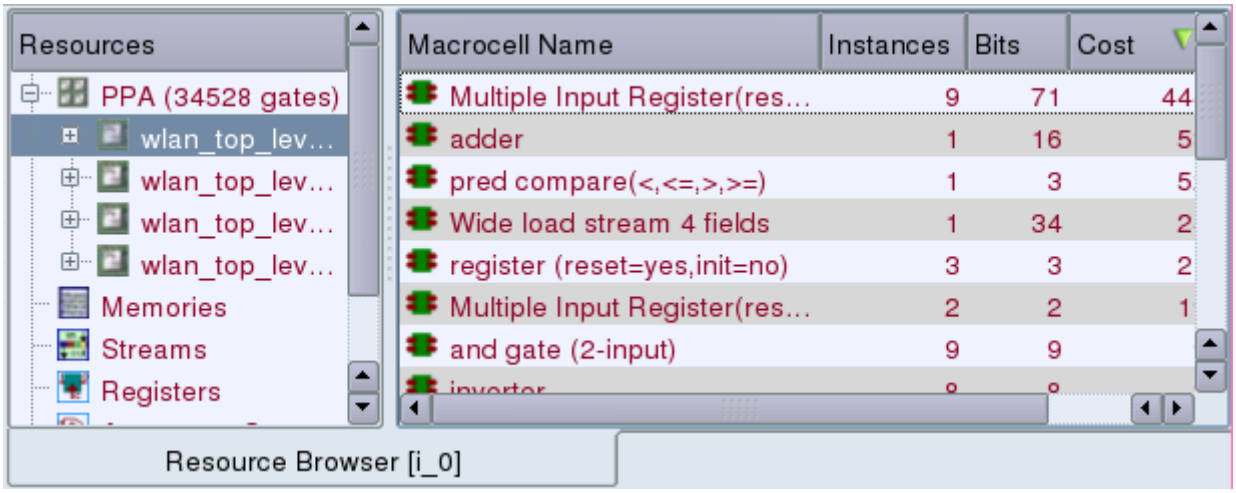
Resource Browser for Sequentialization Graph

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.

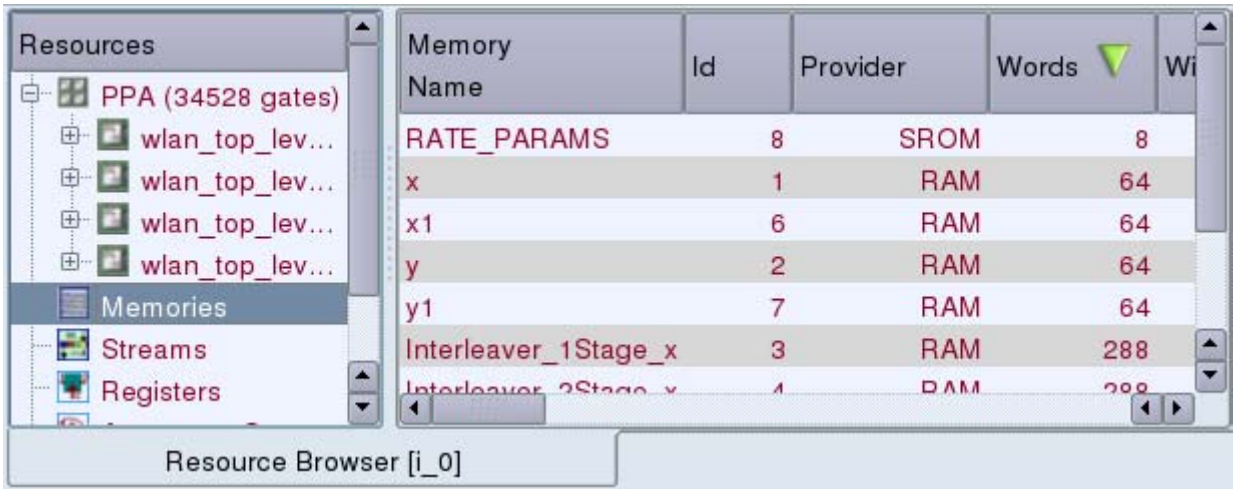
Resource Browser for PPA Graph

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.

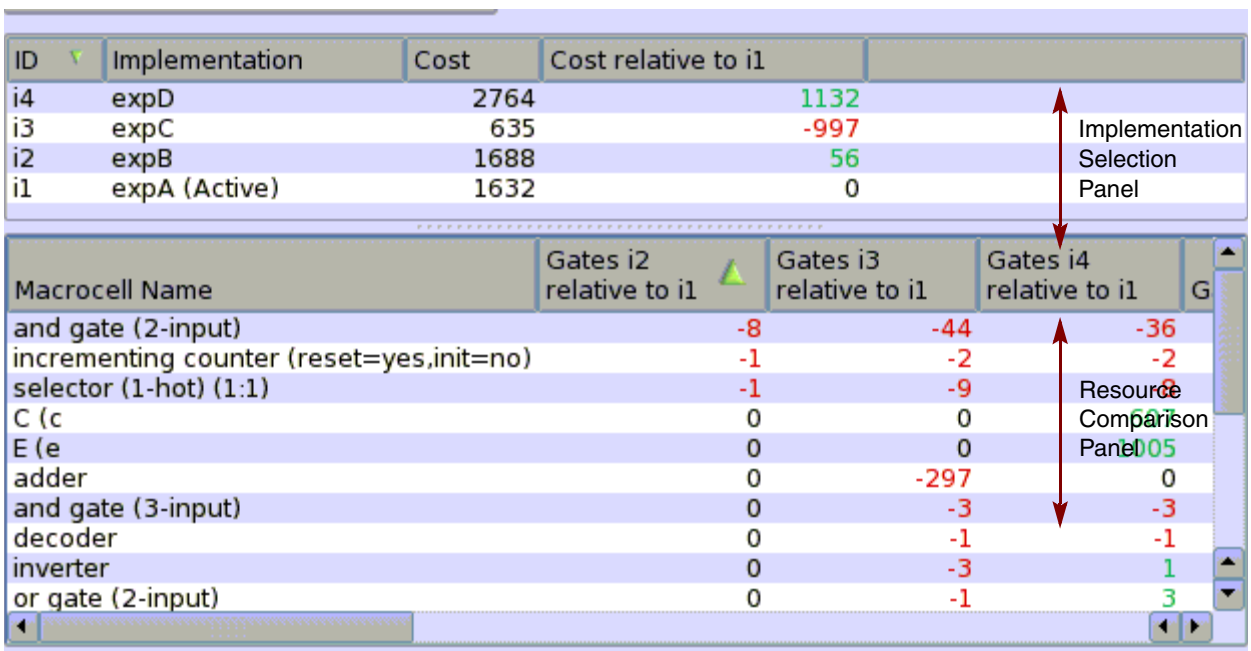
Resource Browser for Feedback Path View

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:

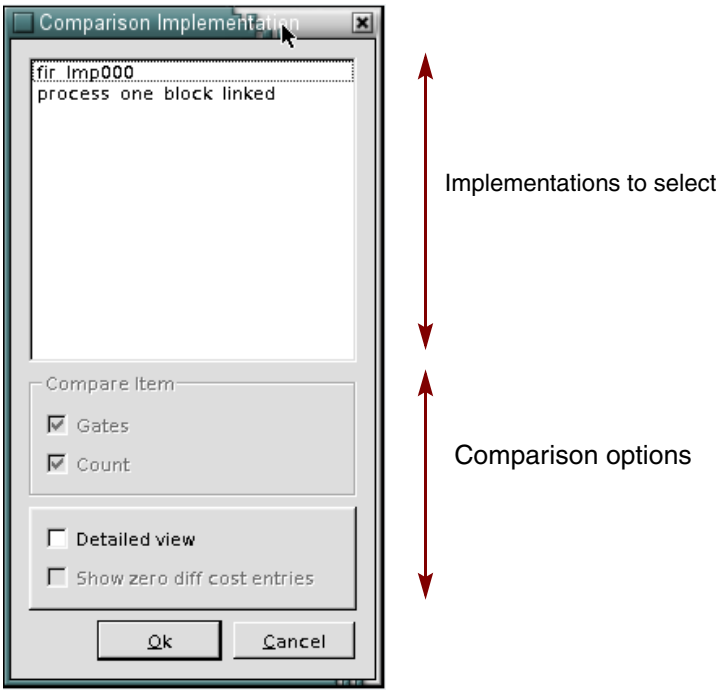
Select Comparison Implementation	Selects the implementations to be compared against the active implementation. Opens the Comparison Implementation dialog box when you click Select comparison implementations. For more information, see "Implementation Comparison" on page 2-62.
Resource Comparison Panel	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.



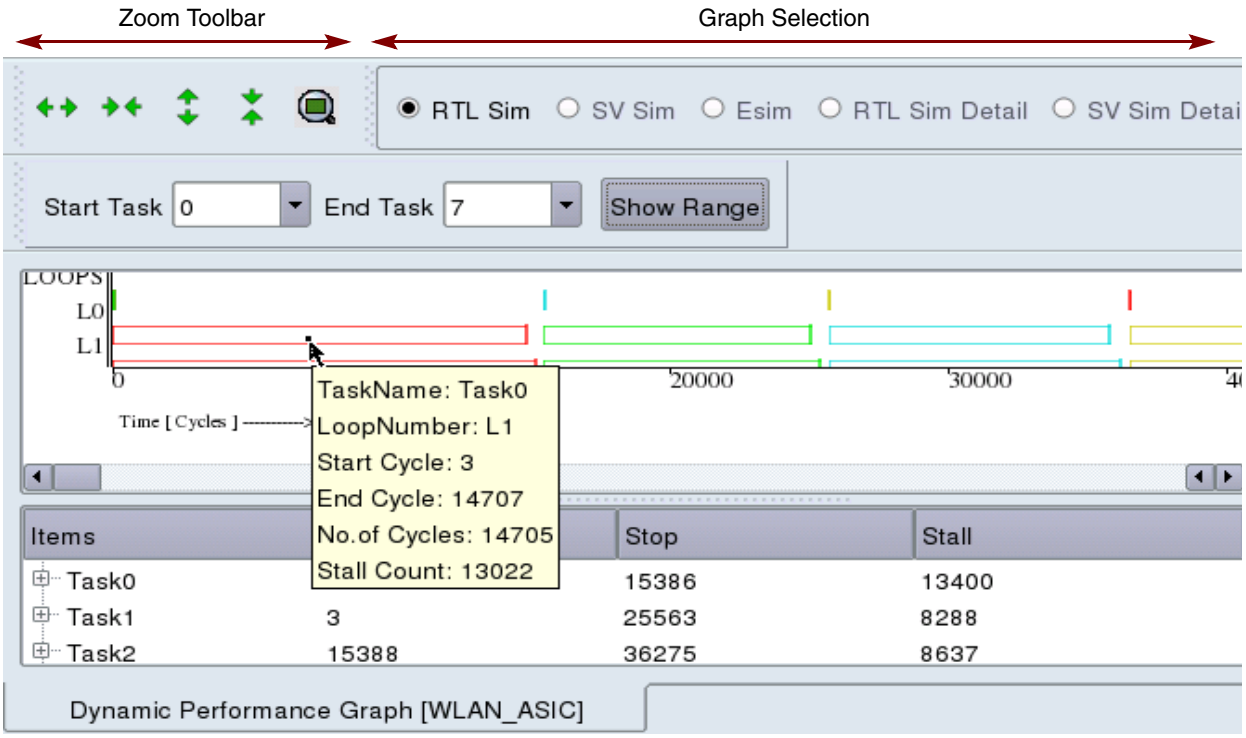
Implementation list	Lists the implementations available for comparison. The implementations share the same technology library. Select implementations to compare against the current implementation. Click implementations in succession to select multiple implementations.
Gates	Enable this to compare the 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.

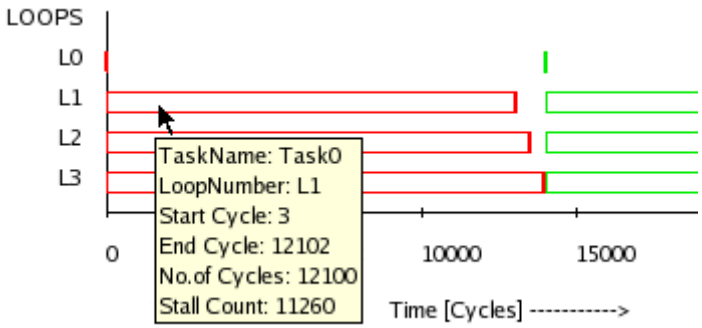
Items	Start	Stop	Stall
Task0	2	12732	18730
Task1	12733	21131	11806
Task2	21132	28600	8924
Task3	28601	33645	5546
Task4	33646	39142	6181
Task5	39143	42911	3808
Task6	42912	47544	5609
Task7	47545	51313	4338

Tooltips for Loops

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

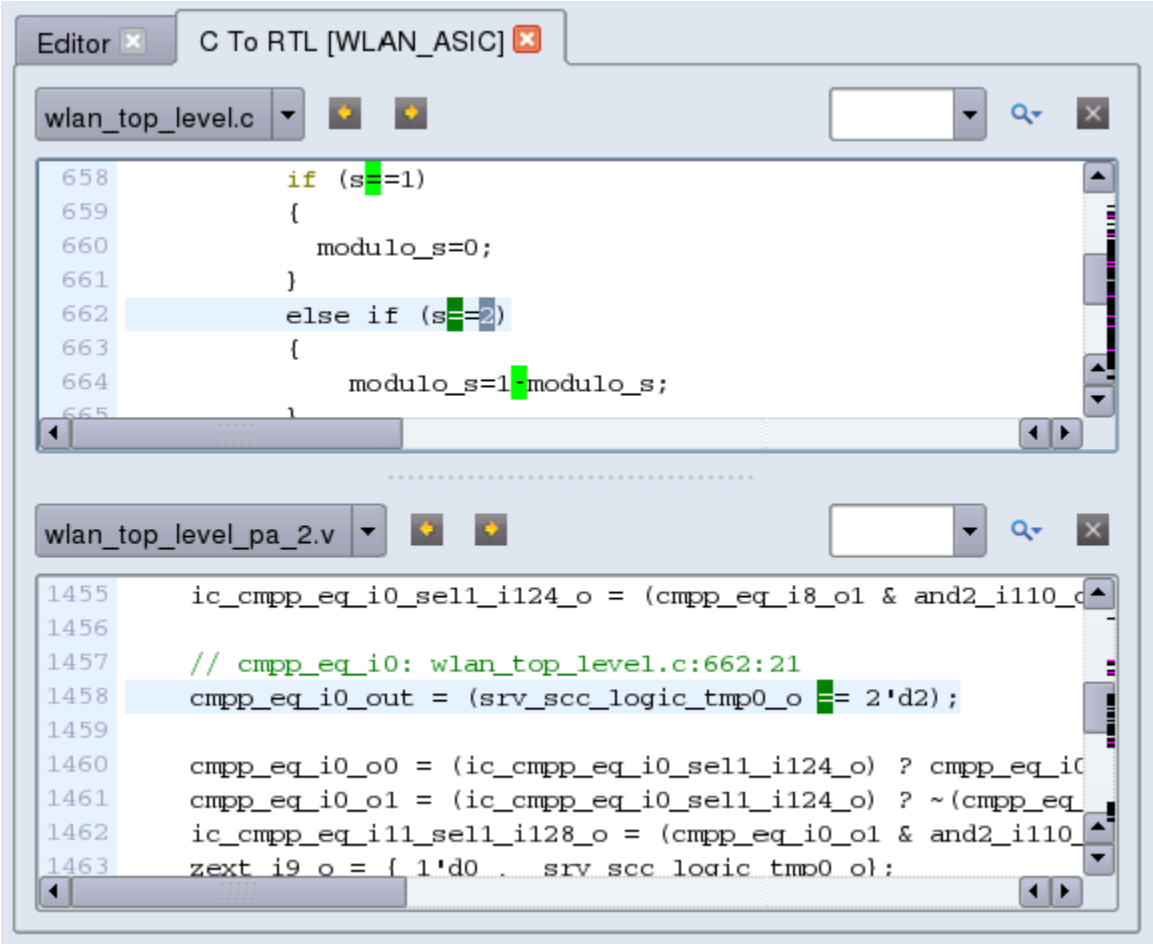
- Task Name
- Loop Number
- Start Cycle

- End Cycle
- Number of Cycles
- Stall Count



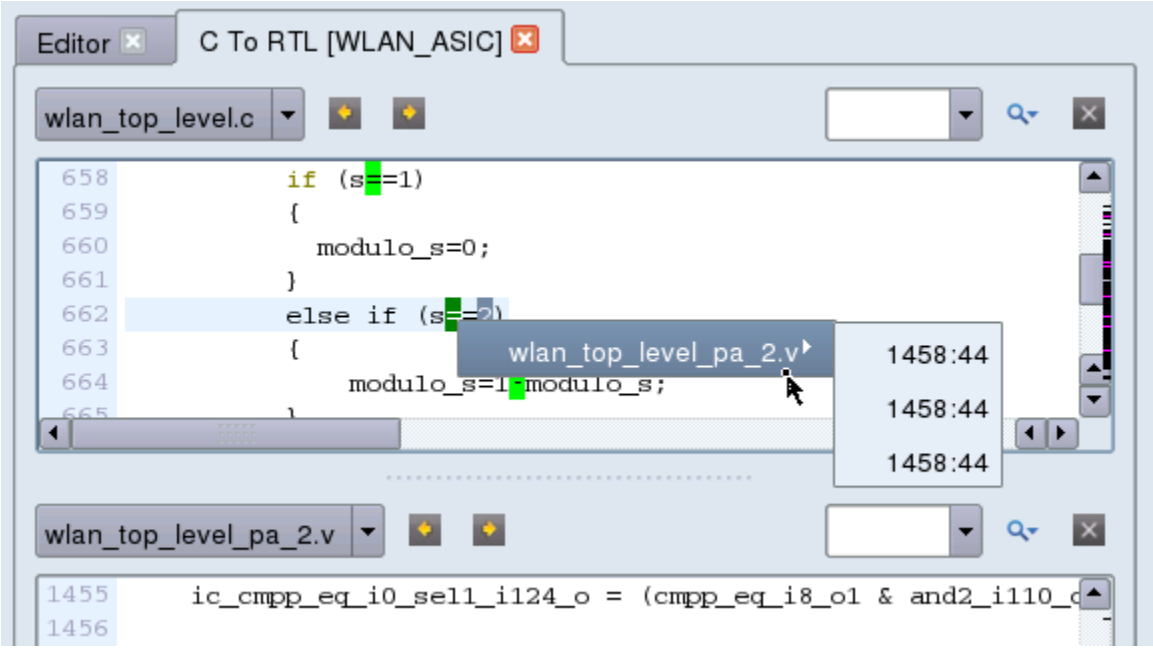
C to RTL Mapping

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

Setting Up Your Design For Synthesis

To use the Synphony C Compiler tool, you need to set up a project and an implementation, as explained in the following sections:

- Getting Started
- Terminologies Used
- Setting Up Your Design for Synthesis

Getting Started

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

- Starting the Tool
- Getting Help

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.

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...	See
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

For help with...	See
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.
- See the product information available on SolvNet.

Terminologies Used

Table 3-1 lists the common terminologies used in this document.

Table 3-1 Common Terminologies Used

Term	Definition
Target	The name of the function or method in the C/C++ source code that is to be synthesized
Data files	The input files used by the C++ program
Result files	The output files generated from the C++ program
Reference files	The files that are compared with the result files generated from the application
Testbench	The C++ code that uses the data files and verifies the application
Synthesizable code	The part of the application to be synthesized
Task	One call to the target (C/C++) function or method

Table 3-1 Common Terminologies Used

Term	Definition
Constraints and directives	Hardware properties that you can specify in the synthesizable code, either using Tcl commands or using #pragma statements
PPA	Top-level RTL block generated from the top-level target
TCAB	Importable RTL block generated for a target that is not a top-level target.

Setting Up Your Design for Synthesis

The following sections describe the setup to synthesize your design using Synphony C Compiler.

Working with Projects

The first step in working with the Synphony C Compiler software is to create a project. The following sections discuss the details of working with projects.

- Creating a New Project
- Opening an Existing Project
- Specifying Parameters

Creating a New Project

Follow these steps to use Synphony C Compiler GUI to set up a project. The equivalent Tcl commands are listed after the procedure.

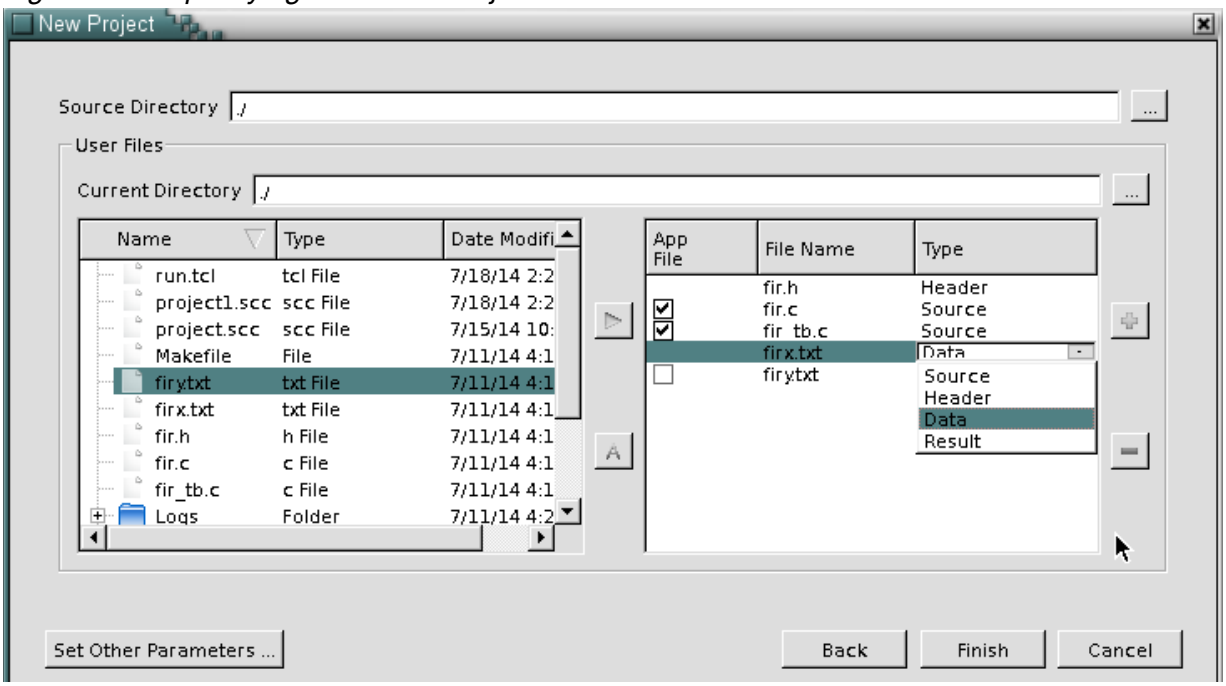
1. Create the project.
   - Select File > New Project
2. Specify the files for the project.
   - In the dialog box that appears, select the source, header, data, and result files from the box on the left and click >. This adds all the selected files to the newly created project.

To auto-populate the user-specified files, click the following icon:

After you auto-populate the files, to change the type of the file,

- Select the file
- Double-click the file type. The drop-down box appears.
- Select the required file type

Figure 3-1 Specifying Files for a Project



3. Click Finish.

Tcl Commands for Creating a Project

The Tcl commands to create project are as follows. If you use a relative file path, the files must be under the project directory.

```
set_defaults -sources "fir.c fir_tb.c"
set_defaults -header "fir.h"
set_defaults -data "firx.txt firy.txt"
set_defaults -results "firz.txt"
```

Opening an Existing Project

You can open an existing project in the following ways:

1. Use the File > Open Project
   - Select the project directory and the project file in the dialog box that opens.
   - Click Open.

   The tool opens the project files, its parameters, the project implementations, and their parameters.
2. Click the Open Project icon in the toolbar and specify the project file as described.
3. File > Recent Projects displays the projects that you worked on. You can select any of the projects to open the project.

   The tool displays a maximum of ten projects.

   Note:
   Only one project can be open at a time. Close a project to open another project.

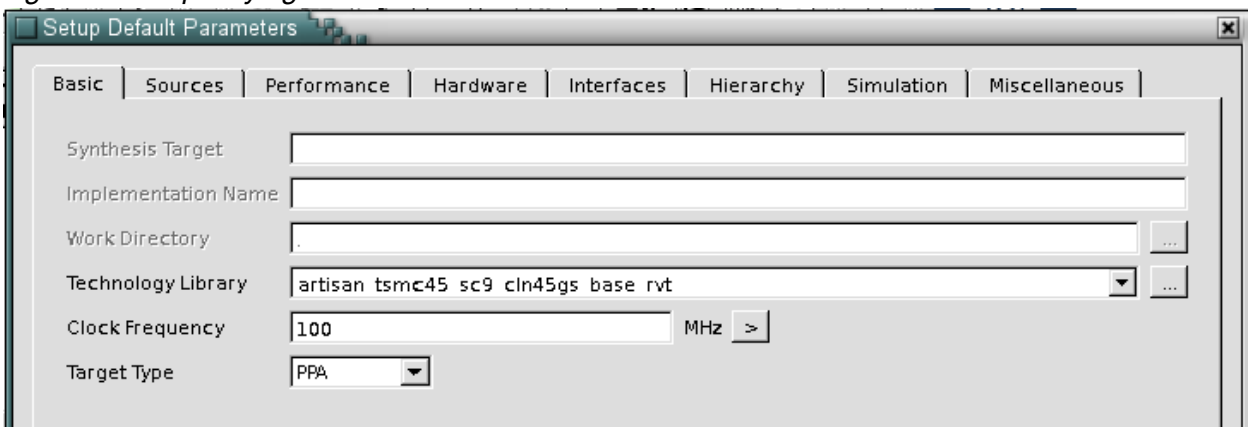
Specifying Parameters

When you create a project, the tool sets the defaults for the parameters. To modify the tool defaults, follow the steps:

1. Select File > Set Default Parameters

   The Setup Default Parameters dialog box opens.

Figure 3-2 Specifying Parameters



2. The various categories of parameters are organized under different tabs. Click the required tab to set the parameters.
3. Click OK.

Working With Implementations

After you create a project, you can create an implementation, which is one design point in the project. You can have multiple implementations for the same project.

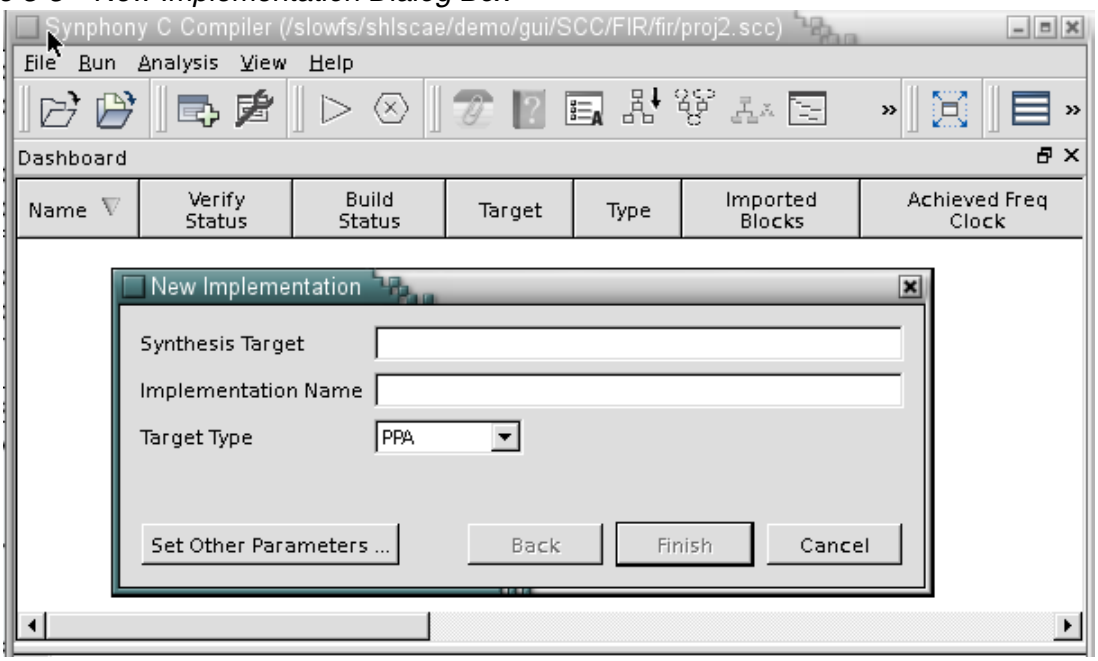
Creating a New Implementation

To create a new implementation

1. Select File > New Implementation

   The New Implementation dialog box opens.

Figure 3-3 New Implementation Dialog Box



2. Specify the target for synthesis.

   Synthesis target is the name of the procedure in the source file, which is the top-level entry point for synthesis. This procedure must be defined in one of the application files you specified while creating the project.

   You can also change the application files.

3. The tool automatically populates the name of the implementation, based on the synthesis target. Optionally, you can specify an implementation name.

4. Select the target type from the Target Type drop-down box. The default target type is PPA

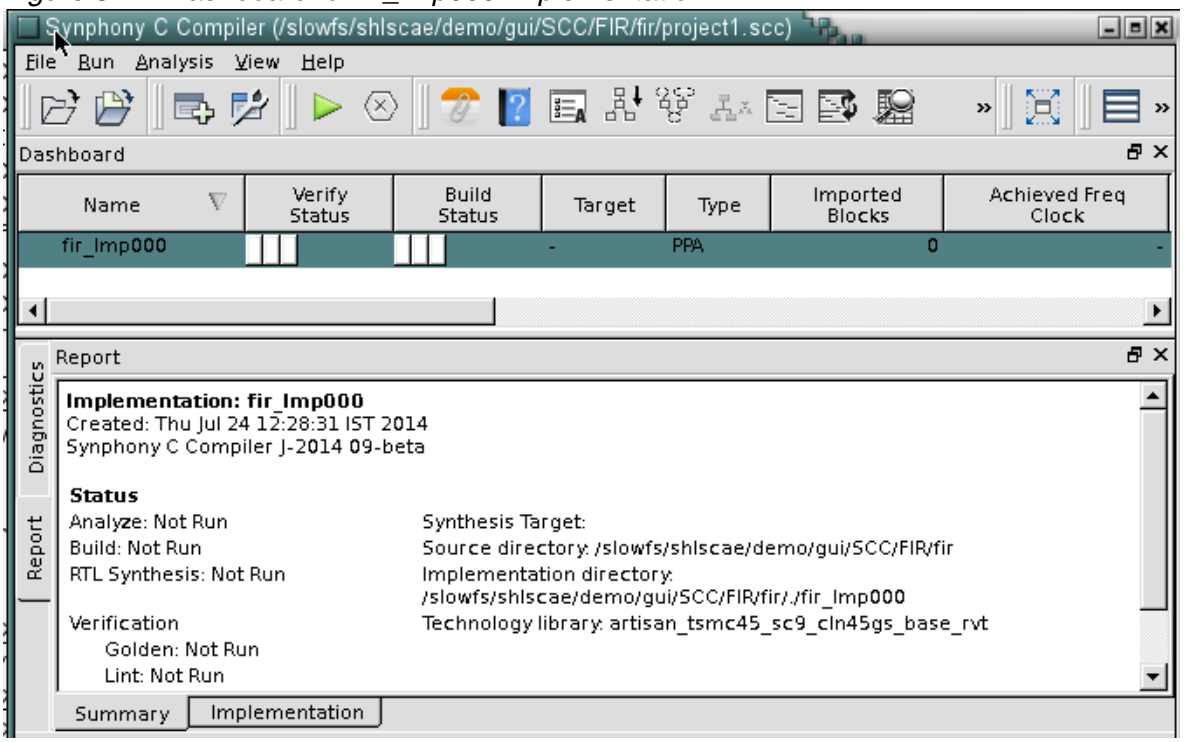
5. By default, the implementation uses the default setting of the parameters in the project. To set the parameters, click the Set Other Parameters button.

For more information about setting parameters, see Setting Parameters.

6. Click Finish.

   The tool creates a new implementation and makes it the active implementation. You can see the active implementation highlighted in the dashboard, as shown in Figure 3-4.

Figure 3-4 Dashboard for fir_Imp000 Implementation



Tcl Commands for Creating a New Implementation

The Tcl commands for creating a new implementation:

```
new_implementation fir_Imp000
set_params -target fir
```

Creating a New Implementation From an Existing Implementation

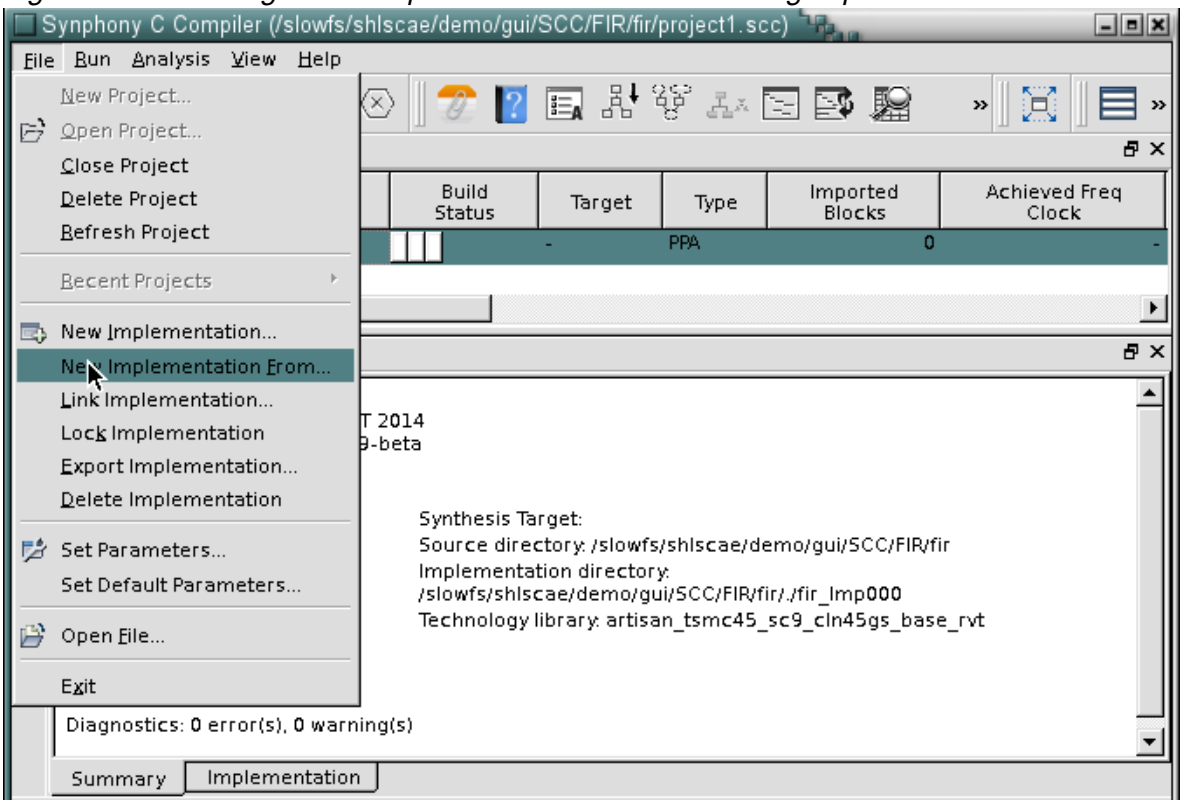
When you create a new implementation, by default, the new implementation inherits the defaults of the parameter settings of the project.

When you create an implementation from an existing implementation, the new implementation inherits all the parameter settings of the existing implementation.

To create a new implementation from an existing implementation,

- Select File > New Implementation From, as shown in Figure 3-5.

Figure 3-5 Creating a New Implementation From an Existing Implementation



- The New Implementation dialog box appears.

  Follow the same steps as creating a new implementation. Optionally, you can change the parameter settings of the new implementation, without impacting the parameter settings of the existing implementation.

Use the following Tcl command to create a new implementation from an existing implementation.

```
new_implementation fir_Impl001 -from fir_Imp000 -params_only
```

Configuring Projects and Implementations

The tool supports several parameters for you to control the type of hardware that is built. When you create a project, all the parameters are set to the tool defaults. You can modify the values of these parameters to suit your requirement.

When you create a project, the project inherits all the tool default parameter values. You can modify the parameter settings for the project, which are the project defaults. The project defaults are inherited by any implementation that you create.

You can modify the parameter settings for each implementation of the project, by resetting the parameters.

The various parameters lets you specify design constraints, performance targets, and settings for simulation.

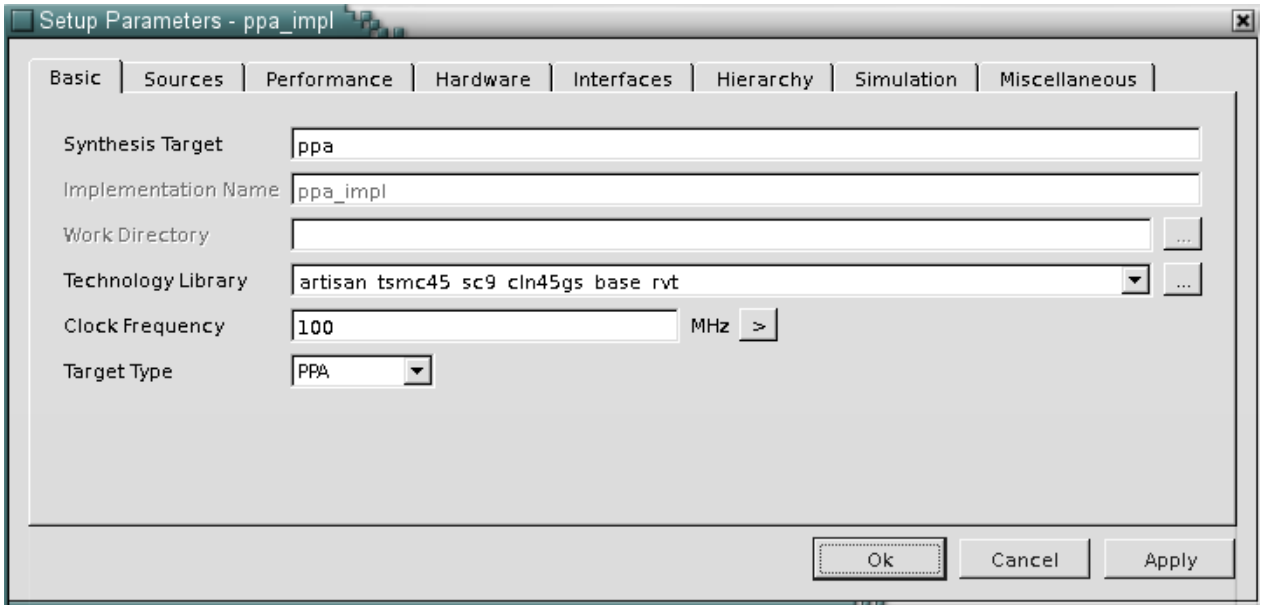
Setting Parameters

Follow these steps to set parameters from the GUI. You can also specify any of these parameters from the command line using the appropriate argument of the `set_params` command. See the *Synphony C Compiler Reference Manual* for the full syntax for this command.

1. Select File > Set Parameters

   The Setup Parameters dialog box appears, as shown in Figure 3-6.

Figure 3-6 Parameters



2. Click the various tabs and specify the values for the parameters.

   For more details about the parameter setting, see Configuring Projects and Implementations. For more details about the Tcl command to set the parameters, see the *Synphony C Compiler Reference Manual*.

3. Click OK.

Specifying Clock Frequency

You must specify a clock frequency for your design. You can specify the clock frequency using the GUI, in one of the following ways.

- When you create a new project, specify the clock frequency in the New Project dialog box.
- When you modify the default parameters settings using the Setup Parameters dialog box or Setup Default Parameters dialog box.

Alternatively, use the `set_params -clock_freq` command.

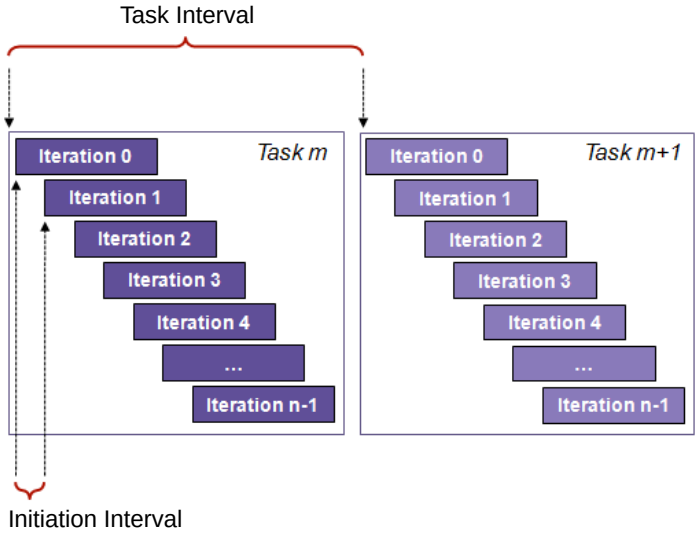
To change the clock frequency for gate-level synthesis, use the `set_params -rtl_clock_freq` command, If you already have created the hardware using the `build` command, run the `reset_rtl_package` command after you modify the `rtl_clock_frequency` variable setting.

Setting Initiation Intervals and Task Iteration Intervals

In addition to the clock frequency target, you must specify throughput targets by setting the rates at which iterations and tasks run. These rates also affect performance.

Figure 3-7 illustrates the task iteration and initiation intervals which together make up the throughput rate. A task is a single execution of a top-level function that is converted to hardware, and the task iteration interval (Task II) is the minimum number of clock cycles between task starts. An initiation interval (II) is the minimum number of clock cycles between successive iterations of a loop. The tool lets you specify these intervals and define the rates at which you want tasks and iterations to launch.

Figure 3-7 Task Interval and Initiation Interval



1. To specify the task interval in the GUI:
   - Select File > Set Parameters

     The Setup Parameters dialog box opens.

- Select the Performance tab.
- Specify the loop parameters.

Use the `set_params -task_ii` command to specify the number of cycles from the command line.

The default setting is Unconstrained, or infinite.

2. To specify a global initiation interval, use one of the following methods:
   - In the GUI, Select File > Set Default Parameters Set Loop II to the number of cycles you want between loop iterations.
   - Alternatively, use the `set_params -ii cycles` command to set the number of cycles between iterations.
3. To specify initiation intervals on a per-loop basis, use one of these methods:
   - Check Reports/report_original_loop_to_line_number.txt to get the loop number. Loop nests are numbered in program order.
   - Use the `set_params -ii "loop=1:2 loop=4:2"` command to set the number of cycles between iterations for that loop from the command line.
   - Use #pragma ii in your C/C++ code to set the number of cycles.

Locking and Unlocking an Implementation

For building large modular designs, when multiple users are involved in implementing various modules, locking an implementation avoids accidentally overwriting the implementation. To lock an implementation,

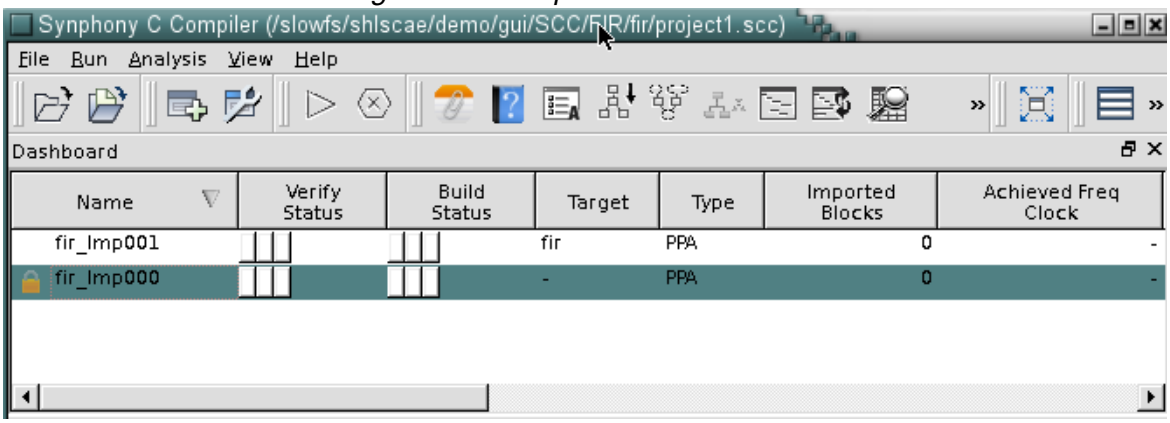
- In the dashboard, select the implementation.
- Select File > Lock Implementation

Alternatively, you can use the following Tcl command to lock an implementation:

```
lock_implementation fir_Imp000
```

In Figure 3-8, the dashboard shows the fir_Imp000 implementation as locked.

Figure 3-8 Dashboard Showing a Locked Implementation



After you lock an implementation, you cannot modify, build, simulate, or delete the implementation. However, you can delete a locked implementation by using the `-force` option of the `delete_implementation` command.

To unlock a locked implementation

- In the dashboard, select the locked implementation.
- Select File > Unlock Implementation

Alternatively, you can use the following Tcl command to unlock a locked implementation:

```
unlock_implementation fir_Imp000
```

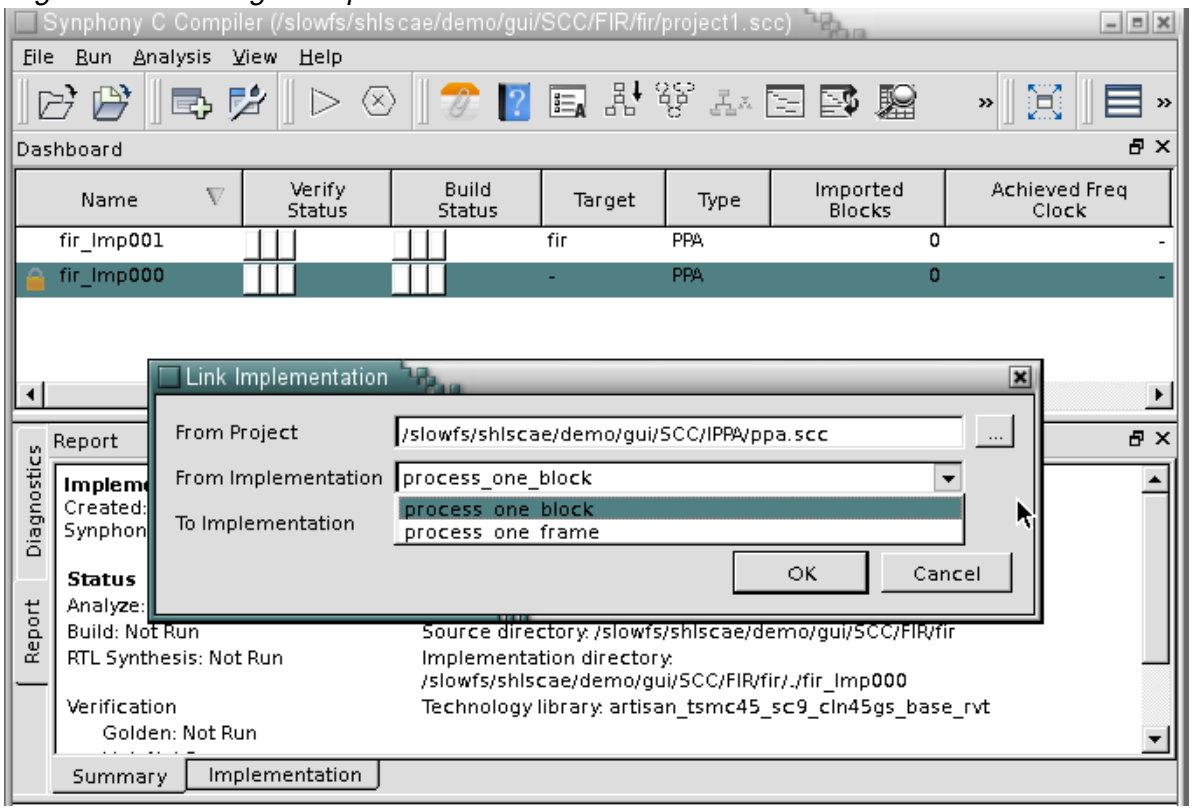
Linking an Implementation

To facilitate reuse of implementations, the tool supports linking an implementation of a project in another project. To link an implementation,

- Select File > Link Implementation

  The Link Implementation dialog box opens, as shown in Figure 3-9.

Figure 3-9 Linking an Implementation



- Browse or specify the From Project.
- The From Implementation drop-down box shows all the implementations of the selected project.

  Select the implementation to be linked.
- Optionally you can change the name of the linked implementation by specifying the name in the To Implementation box.

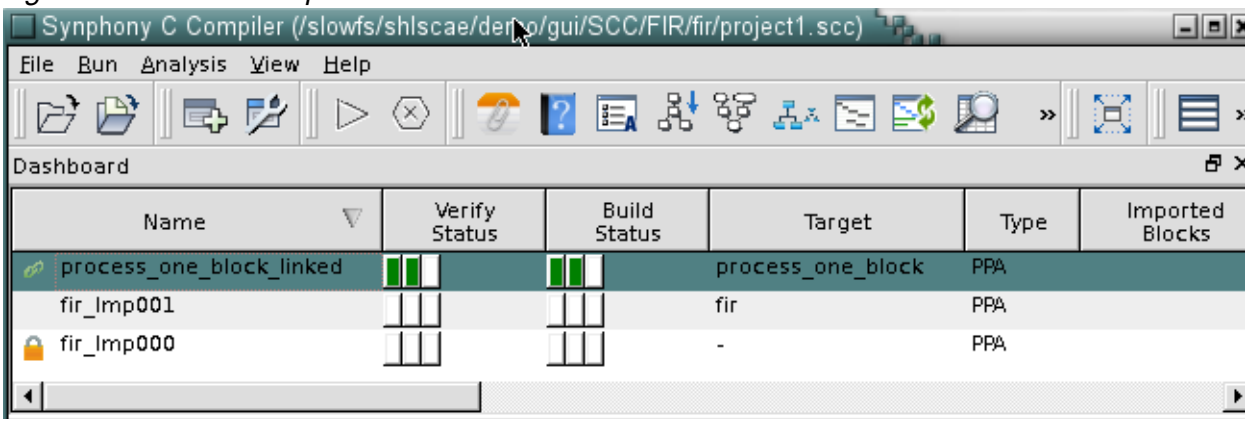
The tool links the implementation to the current project. The dashboard shows the link icon for the linked implementations as shown in Figure 3-10.

You can use a linked implementation for analysis and reporting. But you cannot modify, build and simulate a link implementation.

Alternatively, you can use the following Tcl command to link an implementation.

```
link_implementation /demo/IPPA/ppa.scc process_one_block \
process_one_block_linked
```

Figure 3-10 Linked Implementation in the Dashboard



Exporting an Implementation

Exporting an implementation is useful in archiving the implementation with its settings, including the source, data, and result files.

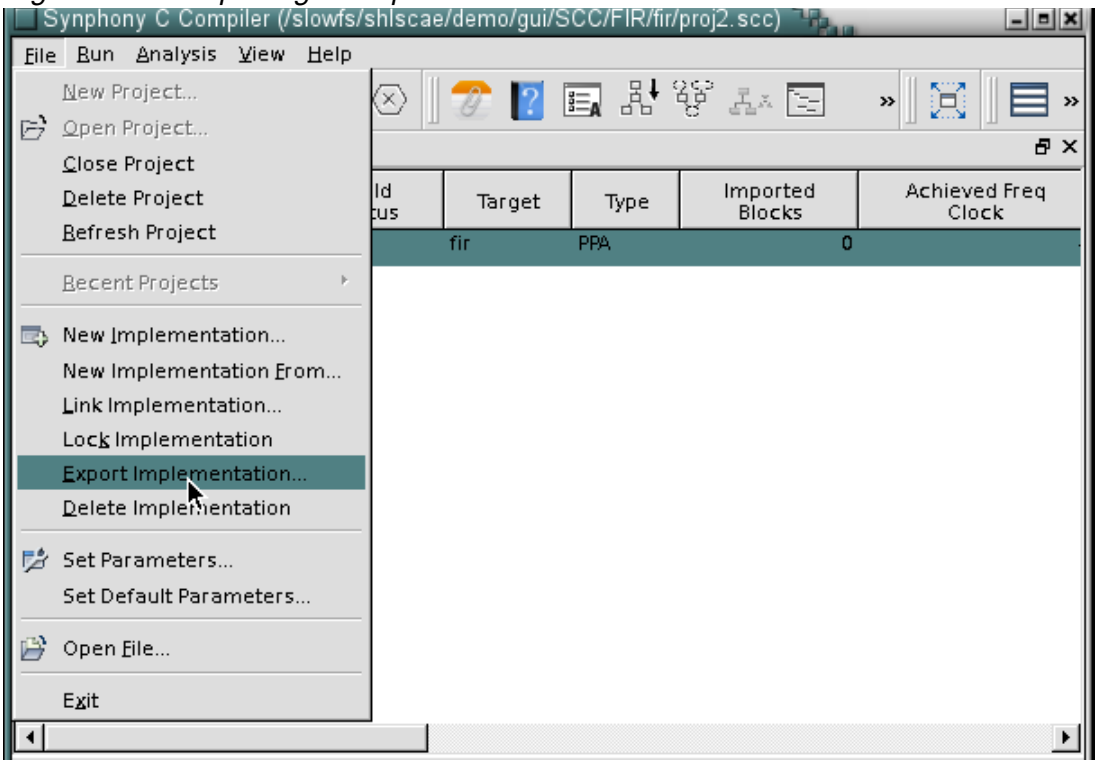
When you export an implementation, the tool creates a *.tar.gz file containing the implementation data. If an existing implementation archive with the same name exists, the tool overwrites the existing file. The file contains

- The project sources and the implementation settings.
- The implementation source, header, data, and results files.
- The implementation constraints.
- The files in the Logs directory, if you have chosen or specified the log option.
- The files in the Reports directory, if you have chosen or specified the reports option.
- The log files, report files, and constraints of the entire hierarchy imported into the implementation.

To export an implementation, select the implementation in the dashboard.

- Select File > Export Implementation as shown in Figure 3-11.

Figure 3-11 Exporting an Implementation



The Export Implementation dialog box appears, as shown in Figure 3-12.

Figure 3-12 Export Implementation Dialog Box

The tool saves all the relevant information as a `tar.gz` file.

Use the following command to export an implementation,

```
export_implementation -filename fir_Impl000.tar.gz -reports yes
```

4

Building and Verifying the Design

This chapter describes the build and simulation operations that the tool performs to implement and verify the design. See the following for details:

- Overview of Build and Verification
- Running the Build Step
- Running C Simulations
- RTL Simulations
- Running RTL Simulation
- Debugging Simulation Issues
- Verifying Corner Cases
- Checking Code Coverage
- The Simulation Regression Flow

Overview of Build and Verification

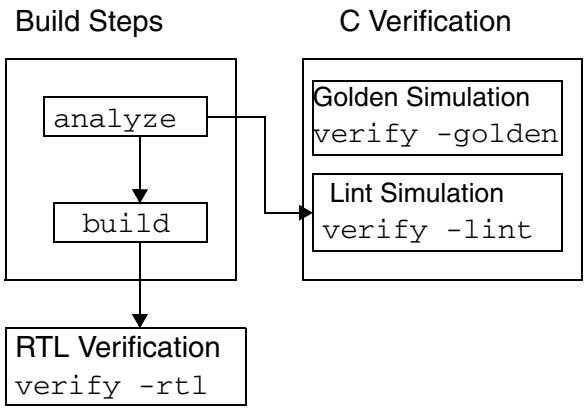
The Synphony C Compiler process can be divided into two complementary operations: build, and verification through simulation, as explained in the following sections:

- Build Steps
- Simulation Steps

Build Steps

You can run each build step and simulation independently. After each step of the build flow, you can run a corresponding simulation to verify your design at that stage.

Figure 4-1 Build and Verification Steps



For a list of all the simulations you can run from the tool, see Simulation Steps. Table 4-1 shows various build steps and where you can get more information:

Table 4-1 Build Steps

Analyze
Build

Simulation Steps

Table 4-2 lists the C, C++, and RTL simulations that you can run in the Synphony C Compiler tool:

Table 4-2 Simulations in Synphony C Compiler

Golden simulation (C, C++)	Running Golden Simulation
Lint simulation (C, C++)	Running Lint Simulation
RTL simulation (Verilog)	Running RTL Simulation
Gate-level simulation (Verilog)	Running Functional Gate-Level Simulation

Generating the RTL

Generating the RTL using Synphony C Compiler involves the following two steps:

- Analyze
- Build

Analyze

Analyze is the first step in synthesis.

To perform the analyze step on an implementation using the GUI

1. Select the implementation in the dashboard.
2. Hover the cursor over the implementation in the dashboard. Right click the implementation to see the implementation context menu.
3. Select Analyze.

Alternatively, use the `analyze` command to analyze the design.

Build

The build step performs the following optimizations based on the constraints that you specified:

- Loop optimizations
- Loop scheduling
- Operation scheduling and parallelism
- Resource allocation and sharing
- RTL generation
- Testbench generation for RTL verification
- Utility script generation for RTL synthesis

To perform the build step on an implementation using the GUI,

1. Select the implementation in the dashboard
2. Select Run > Build > Run

Alternatively, use the `build` command to build the design.

Running the Build Step

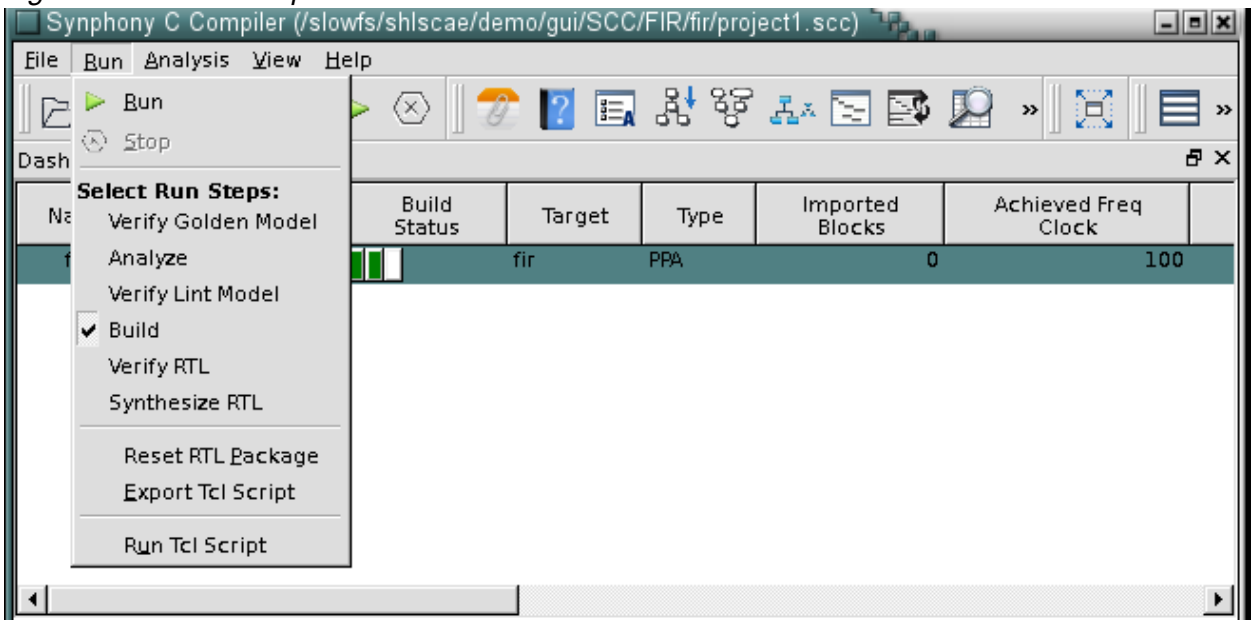
This section describes how to run the Build step. You must set up your project and implementation before you start running the Build step. You must also run golden simulation.

1. Set the parameters for the build step or steps before you run.

   For more information, see Setting Parameters.

2. Run golden simulation, as described in Running Golden Simulation.
3. To run the Build step,
   - Select Run > Build. Click the Run icon:

Figure 4-2 Build Step



- Alternatively, use the `build` command.

The tool runs the build without running any simulations. It generates RTL and a package ready for RTL synthesis.

4. Check the results.
5. Make code changes as required.
   - Edit the code files at the project level.
   - Rerun the golden simulation .

     Note:
     Do not create a new implementation for each run.

Running C Simulations

The SCC tool has four C simulations that you can run to verify your design at various stages. Some simulations have dependencies, and can only be run if the corresponding build phase or a required previous simulation has been completed. Run the golden simulation and lint simulation for all implementations whenever there is a change to the source, in either the driver or application code.

Table 4-3 summarizes the reasons for using each simulation step:

Table 4-3 Simulation Steps and the Reasons to Run the Steps

Simulation	Reasons to run the simulation
Run > Verify Golden Model or `verify -golden`	
Simulates SCC code	Verifies functionality of the original sequential SCC C code
Run > Verify Lint Model or `verify -lint`	
Simulates SCC C code that is augmented with linting functionality	Verifies pragmas such as `bitsize` and `num_iterations`
Run > Analyze and Build or `analyze` and `build`	
Simulates generated C code after the Synthesize step in the design flow	Verifies memory access pattern when memory analysis is turned off

Running Golden Simulation

The first step in working with your C source code is to verify that you correctly specified the set of functions and that the application compiles and runs correctly. Golden simulation uses GCC to compile, and it runs the original C/C++ driver and application code to ensure that it is error-free. The tool treats the golden simulation as the correct behavior, and compares other simulations to this run. You can run golden simulation in different scenarios:

- Run golden simulation as an initial run without constraints, when you do not care about the syntax, but want to test functional correctness before proceeding.
- Run golden simulation later in the design flow for increased coverage.
- Run golden simulation as a part of a flow or another simulation.

The following procedure describes how to run golden simulation.

1. Set up the C/C++ code correctly.
   - Ensure that the code satisfies the application specification for functional correctness.
   - Separate the driver code from the application. Embed the reference code in the driver.
   - Write the applications to be self-sufficient, so that the output RTL can be packaged cleanly without any dependencies on expected directory structure or environment.

2. Set up a project and implementation.

   See Working with Projects, Working With Implementations and Setting Parameters for information about creating projects, creating implementations, and setting parameters, respectively.

   - Make the implementation active by right-clicking and selecting Set Active.
   - If needed, specify options for the simulation run. See Setting Parameters for C Simulation.

3. Run golden simulation.

   - To run golden simulation, select Run > Verify Golden Model
   - Click Run.

   Alternatively, use the `verify -golden` command.

   The Synphony C Compiler tool generates sequential C code that it uses as a benchmark to compare and verify design correctness at subsequent stages in the design process.

4. Check the results.

   See Debugging Golden Simulation Issues for information about problems you could encounter.

5. Make code changes as required.

   - Edit the code files at the project level.
   - Rerun the golden simulation .

     Note:
     Do not create a new implementation for each run.

Running Lint Simulation

Run the lint simulation flow to verify that the original application complies with the Synphony C Compiler coding guidelines, and that the C or C++ code is ready to generate hardware. The tool does not check the driver files during this flow.

1. Before you run the simulation make sure you have done the following:

   - Specify the application files for linting. Do not include the driver files, but you can specify application files that contain some driver code. The tool does not check the driver files.

- Ensure that the implementation is the active implementation.
- If needed, specify options for the simulation run. See Setting Parameters for C Simulation.

2. Run the lint simulation.
   - To run lint simulation, select Run > Verify Lint Model.

     The tool runs the following functions to verify the application files.

`verify -golden`	Runs golden simulation, if it has not already been run.
`verify -lint`	Runs complete compliance checks on the application, including dynamic linting for bit-width ranges, application drivers, and uninitialized arrays and variables,.

     or type `verify -lint` in the console window.
   - Select Run

   - To run lint simulation select Run > Verify Lint Model.
   - Click Run.

   To lint the design, the tool first performs static application source checks, and creates an intermediate representation that is equivalent to the original application. It then executes the intermediate representation and runs dynamic lint checks during this run.

   Your implementation might be a hierarchical block (TCAB or Tightly Coupled Accelerator Block) or a top-level block (PPA or Pipeline of Processing Arrays). For TCAB implementations, the tool limits linting to the code within the TCAB function and other TCABs or inlined functions called from the TCAB. For PPA implementations, the tool lints all the code in the application files.

3. Check the results.
   - Check for lint simulation warnings and errors. Check the warnings in particular, because even if the simulation passes, the warnings can indicate coding issues that could cause problems later.
   - Check for differences between the golden simulation output files and the lint simulation output files.
   - Check for differences between the standard output for the golden and lint simulations.
4. Make code changes as required.

- Edit the code files at the project level.
- Rerun the simulation.

  Note:
  Do not create a new implementation for each run.

RTL Simulations

After synthesizing the C code and implementing the design, the tool generates an RTL package. The package contains all the required files, and you can synthesize and simulate the design outside the Synphony C Compiler tool. For convenience, the SCC tool also lets you run these simulations from the Synphony C Compiler interface. You can run two of these simulations on the output RTL, and one on the post-synthesis RTL:

- SV-UVM simulation
  See Running SV-UVM Simulation.
- Gate-level simulation
  See Running Functional Gate-Level Simulation.

The following table shows how the simulations are different:

Simulation	Reasons to Run the Simulation
Run > Verify RTL or `verify -rtl`	
Simulate SCC-generated RTL using SystemVerilog and UVM testbench	Verifies generated RTL without using a large amount of disk space for data dumps Verifies cycle-accurate performance within and between tasks, with actual hardware transactions between the host processor and the SCC-generated RTL
`verify -rtl -alt_ppa`	
Run functional simulation for a post-synthesis netlist with equivalent ports.	Verifies functionality of post-synthesis netlist

Running RTL Simulation

Run RTL simulation to verify the RTL generated by the Synphony C Compiler tool against the original C/C++ test vectors, using a simulation tool. The default simulator is the Synopsys VCS tool. You can generate RTL targeted for ASIC designs or FPGA designs.

You can run the RTL simulation as part of a flow or as a separate step. You can also use a script to run the simulation on the final RTL package. The following section describes both ways od running simulation:

- Running RTL Simulation
- Running RTL Simulation on the RTL Package
- Running Equivalence Checking on the RTL Package

Running RTL Simulation

During RTL simulation, the tool captures the input stimulus and results statically for each task call of the PPA function being verified. These static vectors are called tiles, and provide the static stimuli for the simulators.

You might want to run RTL simulation as a separate operation or as part of a flow. The following procedure shows you how to run RTL simulation from the SCC tool, using a Verilog simulator for the RTL simulation:

1. Generate the reference C/C++ testbench.
2. Set up the simulator and other tools.
   - To avoid cross-platform issues, set up the simulator to run on the same platform as the SCC tool.

     The tool supports the Synopsys VCS tool. You can also use the Incisive/NC-Verilog from Cadence.
   - Set the environment variable for the Verilog simulator you want to use.

Simulator	Variable to Set
VCS	`$VCS_HOME`
Debussy, Verdi, nLint	`$NOVAS_HOME`
Verplex, Conformal LEC	`$VERPLEX_HOME`

- Make sure you have access to the GCC compiler. The Synphony C Compiler software assumes that the GCC tool is used to compile the C routines invoked from the PLI mechanism.

3. Set up the SCC tool to run RTL simulation.
   - Make sure to start the tool with the appropriate executable. You must use the version appropriate to the kind of RTL you want to generate. For example, if you have an ASIC license and want to generate RTL for an FPGA target, make sure to start the tool with the FPGA executable:
     `synphonycc_fpga`
   - Specify the ASIC or FPGA target.
   - At a minimum, run golden simulation before RTL simulation. The tool uses the output from the golden simulation run to compare results.
   - Set options to generate RTL and an RTL testbench.
4. For RTL simulation, do the following:
   - Specify the tool to use for RTL simulation. See Specifying RTL Synthesis and Simulation Tools.
   - By default, RTL simulation runs tasks in an overlapped or pipelined manner wherever possible. To execute the tasks in a non-overlapped or sequential way, use `-task_overlap never`.
   - Specify other options for the run, as described in Setting Parameters for RTL Simulation.
   - To enable corner-case testing create the config file and enable perturbation. See Verifying Corner Cases for details.
5. Run RTL simulation.
   - Select Run > RTL Simulation

     or type `verify -rtl` in the console window.

   The tool generates RTL and runs simulation on it using the tool you specified. It writes out the results to the detailed performance report.
6. Analyze the results.
   - Check the detailed performance report.
   - Open the dynamic performance graph and check RTL Sim Detail for information about the stalling behavior.
   - Check execution time and stall cycles.

See Verilog RTL Simulation Issues for information about problems you might encounter.

Affect of Source File Modifications on the Project

After you build and simulate your design, if you modify the source file, you can selectively rerun specific steps based on the changes that you made to the source file. For example, if you change the synthesizable code in the source file, rerun the analyze and the build commands.

Running SV-UVM Simulation

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 RTL simulation of the RTL operating with this testbench. This feature is called SV-UVM simulation or just RTL simulation.

The SV UVM testbench accomplishes the same goal as the testbench for RTL simulation, but with some additional features:

- It follows a UVM standard for testbenches; therefore it is more understandable and reusable.
- It runs your 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. Run `verify -golden` and `verify -lint` to find any application errors.
- It compares the DUT results against the reference results using scoreboards.

After you run the `build` and the `create_rtl_package` commands, to run the SystemVerilog (SV) simulation with a UVM testbench, enter the following command:

```
prompt> verify -rtl
```

All command-line arguments of the `verify` command are supported in RTL simulation, except for the following:

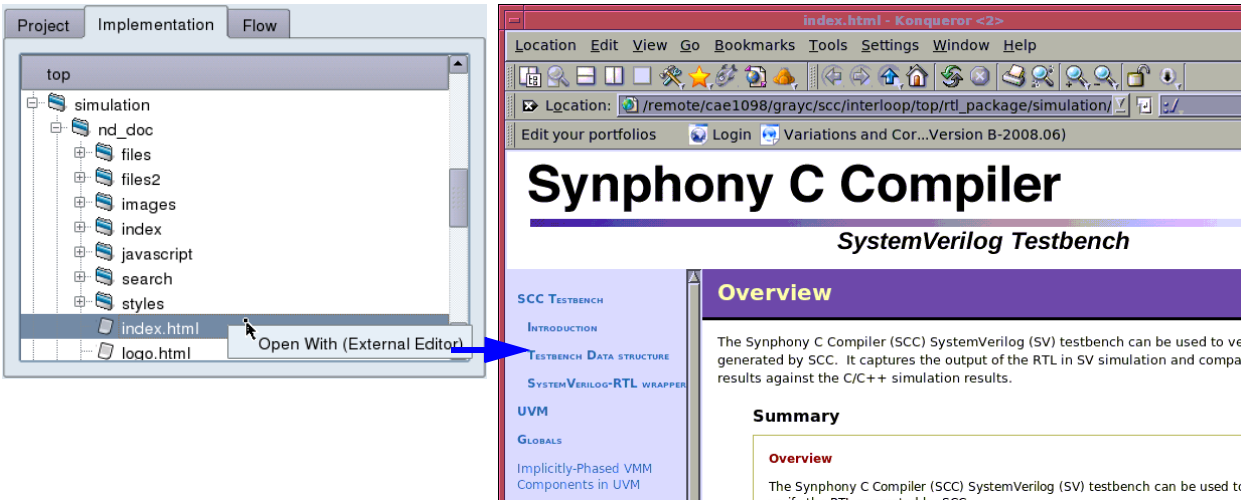
Online Documentation for Testbench

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 4-4. This opens the documentation in your default HTML browser. In the browser window, navigate to the desired topic.

Figure 4-3 SV-UVM Testbench Documentation



Simulation Reporting

During an RTL simulation, the shell console window and the report_summary.txt file report the task II, latency, throughput numbers in a manner similar to an RTL simulation using `verify -rtl` command.

The RTL simulation generates the following report files, which are similar to the corresponding files generated by offline RTL simulation:

implementation-name`_performance_sv.pdf`
`report_verify_rtl_performance_summary.txt`

For more information on SV-UVM simulation, see “SystemVerilog (SV) Universal Verification Methodology” in the *Synphony C Compiler Reference Manual*.

Running RTL Simulation on the RTL Package

The Synphony C Compiler tool includes scripts to run supported simulators to verify an RTL package. The following procedure describes how to use the scripts and run simulation.

1. Specify RTL simulation options for the simulator you are using.

This table lists some `verify -rtl` options for specific simulators:

VCS	`-dump_vpd, -dump_vcd, or -dump_fsdb`
	Set up signal monitoring to facilitate debugging
	`-interactive_on`
	Enables the GUI
	`-vexec_args`
	Lets you specify VCS simulation options like vcs

Cadence Tools	`-dump_vcd or -dump_fsdb`
	Set up signal monitoring to facilitate debugging
	`-vexec_args`
	Lets you specify simulation options like ncverilog

2. The tool generates perturbation config file in the *implementation*`/rtl_package/simulation/uvm` directory:

   `sv_perturbation_config.txt`

   See Running Verification for Corner Cases for details about config file.

3. Set the environment variable for the Verilog simulator you want to use.

Simulator	Variable to Set
VCS	$VCS_HOME
Debussy, Verdi, nLint	$NOVAS_HOME

4. Go to the *implementation*`/rtl_package/scripts` directory, find the script for the simulator you want to use, and customize it as needed for your site and environment setup. Run the script from the rtl_package directory.

Simulator	script
VCS	run_virsim.sh
Debussy, Verdi, nLint	run_debussy. sh run_verdi.sh

- For the Cadence tools, you can also add the following commands to the `rtl_package/lec/runlec.do` file as needed:

```
analyze datapath -merge -verbose -share -qual30 (LEC Ultra only)
analyze abort -verbose
analyze abort -compare
set compare effort complete
compare -single
```

5. Open a Unix shell and run RTL simulation on the package by running the appropriate script from the *implementation*`/rtl_package` directory:

Script	Description
run_debussy.sh run_verdi.sh	Runs the Debussy or Verdi tools. If you run RTL simulation with the `-dump_vcd` option, the script also generates the fsdb waveform file that you can use for debugging.
run_virsim.sh	The script runs the Synopsys VCS interactive simulation debug and waveform viewing tool.

Running Equivalence Checking on the RTL Package

The Synphony C Compiler tool includes a script to run Conformal LEC equivalence checking on the RTL package. Do the following:

1. Generate the RTL package.
2. Set the $VERPLEX_HOME environment variable for the equivalence checker
3. Run the script.
   - Go to the *`implementation`*`/rtl_package/scripts` directory.
   - Customize the run_lec.sh script as needed for your site and environment setup.
   - Run the script from the rtl_package directory.
4. If the checker hits abort points, edit the rtl_package/lec/runlec.do file, and add these commands:

```
analyze datapath -merge -verbose -share // Requires LEC Ultra
analyze abort -verbose
analyze abort -compare
set compare effort complete
compare -single
```

Running Functional Gate-Level Simulation

You can take a synthesized netlist after RTL synthesis and run gate-level RTL simulation on it in the Synphony C Compiler tool. The Synphony C Compiler simulation framework includes scripts that support functional simulation at the gate level. You can substitute a gate-level model or FPGA model that has equivalent ports for the design in the Synphony C Compiler testbench, and then simulate it. The following steps summarize the procedure.

1. Generate a gate-level or FPGA model with ports that are equivalent to the design in the SCC testbench.
2. Set the following options for RTL simulation.
   - Specify the gate-level model to substitute with the `verify -rtl -alt_ppa` command. Provide a full path to the file, or a path that is relative to the RTL package. For example,

     ```
     verify -rtl -alt_ppa path/alt_topvlog_filename
     ```

   - If the model references a technology library use the `verify -rtl -vcompiler_args` options to specify the path to the RTL simulation models in the library or to the library RTL simulation model file:

     ```
     verify -rtl -vcompiler_args "-y libfile_path"
     verify -rtl -vcompiler_args "-v libfile_name"
     ```

   The following example shows a command with all the options:

   ```
   verify -rtl -simulator vcs -alt_ppa synth/run/gateppa.vg \
   -vcompiler_args "-y libfile_path"
   ```

3. Run RTL simulation

   The tool runs functional simulation on the synthesized model. It cannot run timing simulation because it might need to impose the `+delay_mode_zero` simulator option. This sets all delay specifications and module path delays to zero. Therefore, you cannot run timing simulation with the Synphony C Compiler testbench.

Setting Build and Simulation Parameters

There are various parameters you can set for the build and verification operations. Most are optional, but some are required for certain operations. The procedures listed describe the main settings but do not exhaustively describe all the available options. See the *Reference Manual* for a full list of all the options.

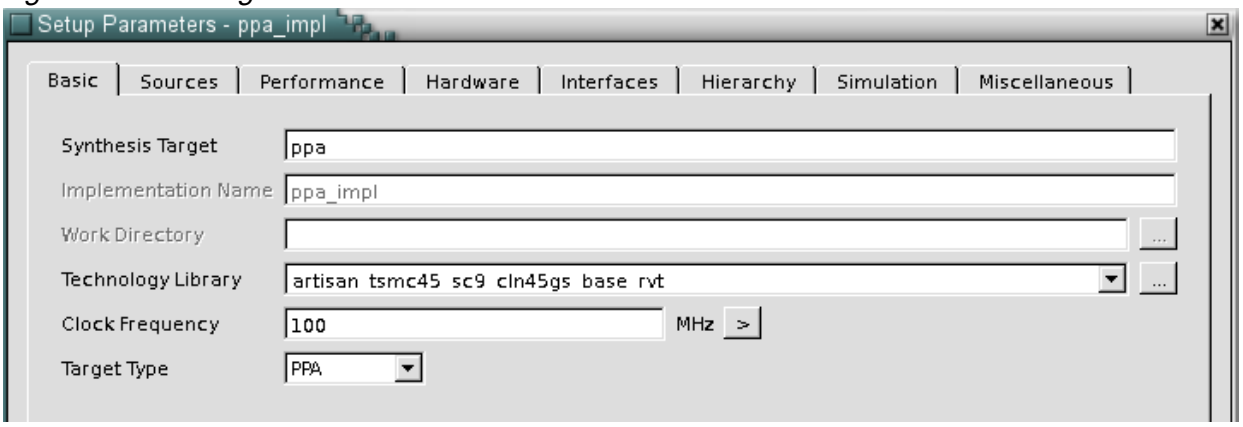
- Setting Parameters for C Simulation
- Setting Parameters for RTL Packaging
- Specifying RTL Synthesis and Simulation Tools
- Setting Parameters for RTL Simulation

Setting Parameters for C Simulation

The SCC tool provides some controls where you can specify runtime options for the different C simulations the tool runs. It lets you specify options for the golden, lint, and schedule simulations. The following procedure describes some of the options you can set, but does not cover all the options. See the *Synphony C Compiler Reference Manual* or the complete list.

1. Open the dialog box for the settings.
   - Open the Parameters dialog box (Implementation-> Configuration New or Current), and click Advanced.
   - Go to the Verification tab, and select the desired settings tab.
2. Determine where to define the parameters.
   - Set general compiler and linker arguments for all C and RTL simulations from the Simulation tab. You can override these general settings as described:
   - To set simulation parameters for all C simulation runs, click the Verification tab and set the parameters. You can set values that applies to all C simulations, but you can then override them by specifying other values on the tabs for each kind of C simulation.
   - To set simulation parameters for a specific C simulation run, click the corresponding tab and set the parameters. The values you set here override the values set on the Common Settings tab. If you do not define anything here, the values defined on Common Settings or Global Settings apply.

Figure 4-4 Setting Simulation Parameters



3. Set options for the run.
   - Enable the runtime operations you want the tool to run, such as Compile and Run.
   - If you want to override the command line arguments you set on the Common Settings or Global Settings tabs and set different options for this particular simulation, define them here.
4. Specify command line arguments to be passed for the run:

C Compiler C++ Compiler	Specify arguments to be passed to the C compiler or C++ compiler. This is equivalent to specifying `verify -compiler_args` or `verify -cppompiler_args`. The tool uses the arguments for PLI compile.
C Linker	Specify arguments to be passed to the final linking phase. This is equivalent to specifying `verify -clinker_args`. The tool uses the arguments for PLI linking.
Execution	Specify arguments to be passed for executing simulation. This is equivalent to specifying `verify -exec_args`.

5. Click OK when you are done.

   The tool uses the specified commands to run the simulation. Where multiple values are defined, it uses the most specific settings; commands defined on the individual simulation tabs override the more general settings on the tabs above it.

Setting Parameters for RTL Packaging

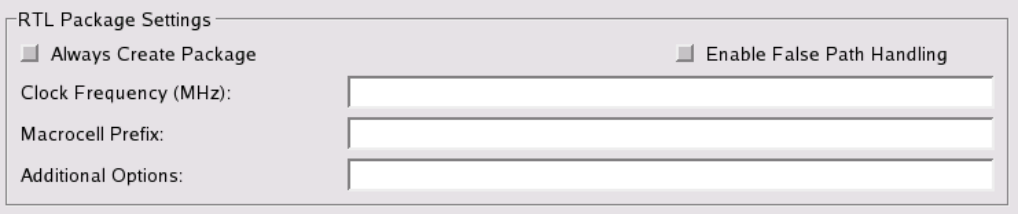
The main parameters to set for RTL packaging are:

1. Open the dialog box for parameters. (Select Implementation-Configure Current | New and click Advanced.
2. Click the Hardware tab.
3. Select the settings you want in the RTL Package Settings section.

   The following describes the main settings.

   - Set the clock frequency for the RTL package.
   - Enable Always Create Package if you want to create an RTL package every time you generate RTL.
   - If you want to analyze false paths, enable Enable False Path Handling.

Figure 4-5 Parameters for RTL Packaging



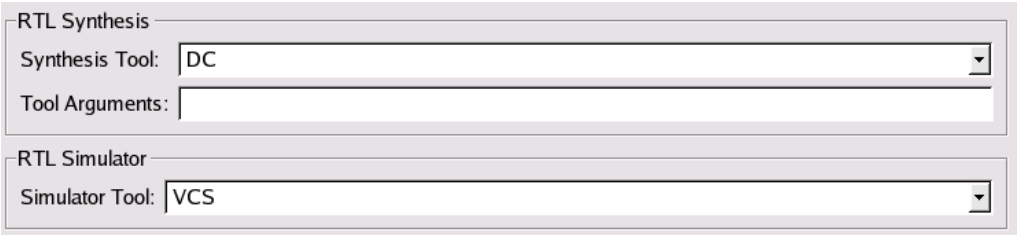
Specifying RTL Synthesis and Simulation Tools

To run a flow that includes RTL synthesis or simulation, you must specify the tools to be used for these operations.

1. Open the dialog box for the settings.
   - Open the Parameters dialog box (Implementation->New or Current), and click Advanced.
   - Go to the Tools tab.
2. To run RTL simulation, set Simulator Tool to the appropriate tool.

   The default tool is VCS. Besides specifying the tool here, you might need to set up the tool as well. For more information, see, Running RTL Simulation on the RTL Package.

Figure 4-6 RTL Synthesis



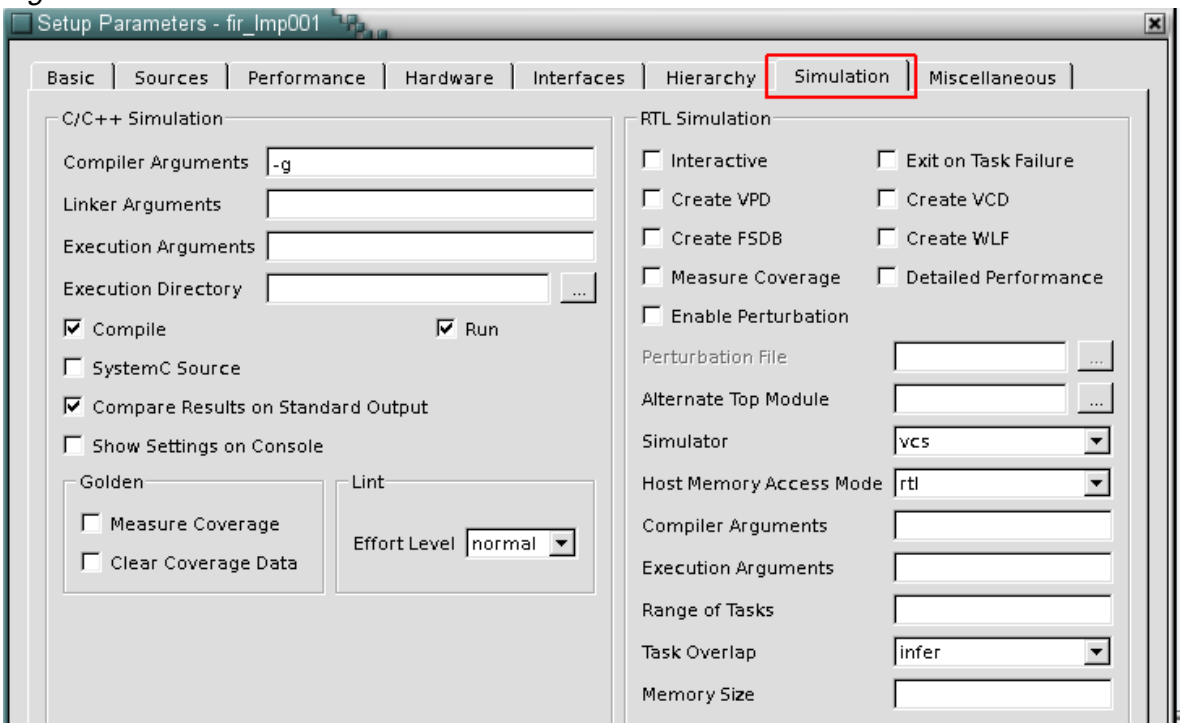
3. To run RTL synthesis, specify the following for the tool:
   - Set Synthesis Tool to the appropriate tool. The default tool is Design Compiler for ASIC designs and Synplify Pro for FPGA designs. Note that the available choices are determined by how you started the tool. Therefore, ensure you start the tool with the FPGA executable if you are working with an FPGA design.
   - Optionally, type RTL synthesis run commands in Tool Arguments. The commands you specify here guide the RTL synthesis run.

Setting Parameters for RTL Simulation

The tool uses the Verilog simulator you specify to run RTL simulation from the Synphony C Compiler interface. The SCC tool provides some controls where you can specify runtime options for the simulator. The following procedure describes some of the options you can set, but does not cover all the options. See the *Reference Manual* for the complete description.

1. Set simulation parameters
   - Select File > Set Parameters
   - Select the Simulation tab.

Figure 4-7 RTL Simulation Tab



2. Specify command arguments for the simulation run.
   - Specify the command arguments as described in Setting Parameters for C Simulation.
   - Specify command arguments for RTL simulation as shown here:

Verilog Compiler	Specify arguments to be passed to the Verilog compiler. This is equivalent to specifying `verify -rtl -vcompiler_args`. The tool passes the options to the compile command line for the Verilog simulator.
Verilog Execution	Specify arguments to be passed to the Verilog simulator. This is equivalent to specifying `verify -rtl -vexec_args`. The tool pas ss es the options to the execute command line for the Verilog simulator.

3. Set options for the run.
   - Enable the required runtime operations such as Compile and Run.
   - Specify the output to generate, such as debug information, performance reports, and FSDB.

- Specify other options for the run in the Command Line Arguments section.
- Specify access for the testbench in the Testbench Options section.

4. If you are going to use perturbation in your simulation run, specify the following:
   - Enable the Enable Perturbation option.
   - Specify the name of the perturbation file to use. See Creating a Config File for information about creating this file.
5. For RTL simulation, set the -task_overlap option to never or always (default).

   Use never to run the tasks sequentially, and always for an overlapped run. The task dumps are created by earlier simulations and are used as stimuli and for verification during RTL simulation.
6. Click OK when you are done.

   The tool passes the specified commands to the simulator and it runs with these commands.

If you already have created the hardware using the `build` command, run the `reset_rtl_package` command after you modify the `rtl_clock_frequency` variable setting.

Debugging Simulation Issues

This section describes how to deal with issues that might come up during different simulation runs. It is assumed that you have created your code correctly, based on the application and driver coding guidelines in *Writing C Applications: Developer's Guide*. This section concentrates on debugging simulation issues, not coding. The issues and their solutions are described under the simulation where it occurs:

- Debugging Golden Simulation Issues
- Verilog RTL Simulation Issues

Debugging Golden Simulation Issues

This section lists simulation issues that could come up with golden simulation (`verify -golden`), and specifies how to deal with them.

Compilation Failure

If you have compilation failures, check the following:

- You have included all source and header files in the project and the implementation
- The `scc.h` file is included in the driver and PPA files.
- You have specified the correct data and result file names.
- You have specified `-lm` option with the `set_params -linker_args` if it was required.
- For applications that use SystemC code, make sure you have specified that the source is SystemC by checking that the implementation parameter is set as follows. The default is yes.

  ```
  set_params -systemc_source yes
  ```

Non-Zero Return Value

Make sure that the golden simulation returns 0 on success. Add a `return` statement to the driver code.

Segfault

Try these techniques:

- Check for a memory error by running the Synphony C Compiler linter:

  ```
  cd <imp>
  analyze
  verify -lint
  ```

- Add `printfs` to the code to isolate the problem.
- Debug with `gdb`, as follows:

  ```
  cd <imp>/simulation
  ln -s ../Work/<app>.golden.
  ln -s ../data/*.
  gdb <app>.golden, or in emacs:M-x gdb ... gdb <app>.golden
  run <execute_args>
  ```

  Note that `gdb` uses sources in `../src` directory. Do not modify the cached source files under `../cache` directory. Instead, modify your sources in the project directory.

Mismatch Between Golden and Reference Results

Use these techniques to identify the problem, and then fix it before rerunning simulation:

- Examine the golden outputs.
  - What is the first byte that does not match?
  - How many bytes mismatch?

  - What changes were made to the code since the last successful run?
- Use `printfs` or `gdb` to help identify the coding error.

Error Message: “Found no data in stream"

This means that a stream read more data than was produced, and you must fix the golden model. Use `printfs` or `gdb` to help identify the coding error.

Warning Message: “Fifo not empty at start of task”

This means that a stream wrote more data than was consumed. The Synphony C Compiler tool requires that all data put into a FIFO in a given task (one call to the PPA `proc`) be removed during the same task. You cannot pass data across tasks using an internal FIFO.

You must fix the golden model. Use `printfs` or `gdb` to help identify the coding error. If you do not fix the error, `verify -rtl` command might fail.

Verilog RTL Simulation Issues

This section describes issues with Verilog RTL simulation:

- RTL Compilation Failure
- Error: Deadlock Detected
- Simulation Timed Out
- Mismatches
- Scoreboard File Dumps in RTL Simulation

RTL Compilation Failure

The most common cause for RTL compilation failure is failure to get a license. Check `Logs/verify_rtl*.txt` and make sure that you can acquire the simulator license.

Error: Deadlock Detected

Check the following:

- Examine the FIFO and loop information in `Logs/verify_rtl*.txt`.
- Make changes to FIFO sizes in implementation/rtl_package/rtl/<target>_define.v.

- Rerun `verify -rtl` command

Simulation Timed Out

Check if you ran synthesis with `infinite_run` or `auto_start_ppa`. The hardware is designed to run forever, but the simulation is finite. You get a timeout error in this case.

Mismatches

Try these techniques:

- Address all unexpected warnings from the `verify -lint` simulation. The lint warnings are in `<`*imp*`>/Logs/verify_lint*.txt`.

Scoreboard File Dumps in RTL Simulation

The UVM scoreboarding mechanism in RTL mechanism (`verify -rtl`) compares the output values of DUT and the REF (C/C++), and prints a summary for every scoreboard failure to the standard output. For more debugging information, use the following commands to print all the REF and DUT output values for every scroreboard object from RTL simulation into pairs of files, `*.ref` and `*.dut`:

```
synphonycc> set_params -vexec_args "+DUMP_OUTPUT_TO_FILE=1";
synphonycc> verify -rtl;
```

The files pertaining to failing scoreboards are printed out to standard output at the end of RTL simulation as follows:

```
Scoreboard: error: scc_0_stream_out_y_data.* scc_1_scalar_out_abc_data.*
are the differing pairs of files, in
<project_dir>/impl/rtl_package/simulation/uvm/scc_scoreboards
```

Viewing the differences between `*.ref` and `*.dut` provides details about the scoreboard failures. The naming convention for the files is:

```
scc_<taskid>_<rtlname>.{ref|dut}.
```

For example, `scc_0_scalar_out_abc_data.ref` contains the REF values for scalar output port `scalar_out_abc_data` in REF task 0.

Verifying Corner Cases

When you verify corner cases, you test vulnerable situations like protocol variations, unexpected signaling or responses, delayed signaling or delayed responses, the capacity status of queues or storage entities, inter- and intra-state machine interactions, difficult or impossible to reach state-transitions, and inter-block communications.

You can define config files to change the runtime behavior or timing relationship of stream control signals driven by the test environment. Changing the behavior or timing of these signals is known as *perturbing* the simulation.

The following procedures describe what you need to do:

- Running Verification for Corner Cases
- Creating a Config File

Running Verification for Corner Cases

The verification of the corner cases is based on DUT interface drivers with perturbation support. The config file is required to specify the interfaces to specify the perturbation and the conditions for perturbation. The following procedure describes how to use this setup to specify stimulus and test corner cases.

1. Develop a test plan to determine what you want to perturb and the results you expect. Include the following:
   - A definition of what you want to test
   - A determination of the extent to which the Synphony C Compiler tool helps with the planned corner case testing
   - A statement of the extent of automation and interaction required to do the testing
   - A definition of the extent of reuse
   - A determination of whether your test output matches the C simulation or whether it requires manual validation
2. Identify the signals that you want to manipulate.

   The following table shows the signals for which the Synphony C Compiler tool has built-in perturbation support, and the ways in which you can manipulate these signals:

Delay signals	StreamReady (instrready0). You can vary the StreamAcknowledge, which responds to the StreamRequest signal, so that it has either a fixed delay or a random delay in its reply to the incoming StreamRequest.

Data signals	nStreamData (instrdata0). You can vary this by either injecting fixed data values or random data values at fixed cycle times, or at signal-relative (expression-based) times.

3. Create the perturbation config file based on your requirements. For more details, see Creating a Config File
4. Run simulation using the perturbation options to the `verify -rtl` command.

   ```
   verify -rtl -perturb_on -perturb_random_seed number \
   -perturb_file filename ...
   ```

   Adding the verbose messaging option to RTL simulation helps in debugging.

   Use waveform perturbators with care, because they can change data inputs for simulation, making the modules inconsistent with the golden results produced by C simulation.
5. Analyze the results.

Creating a Config File

Config files are the mechanism to specify the stream interfaces to be perturbed during simulation and the timing of the perturbations. For every stream interface, the specification consists of the type of perturbation to be enabled, and a range of cycle to bind the random duration of every perturbation and a range of cycles to bind the random duration, during which perturbations are temporarily disabled. The tool generates a default config file for every implementation in the *rtl_package/simulation/uvm/*sv_perturbation.config file.

Checking Code Coverage

When you check code coverage, you see what has not been verified, and you can then add tests to cover these areas. See the following for details:

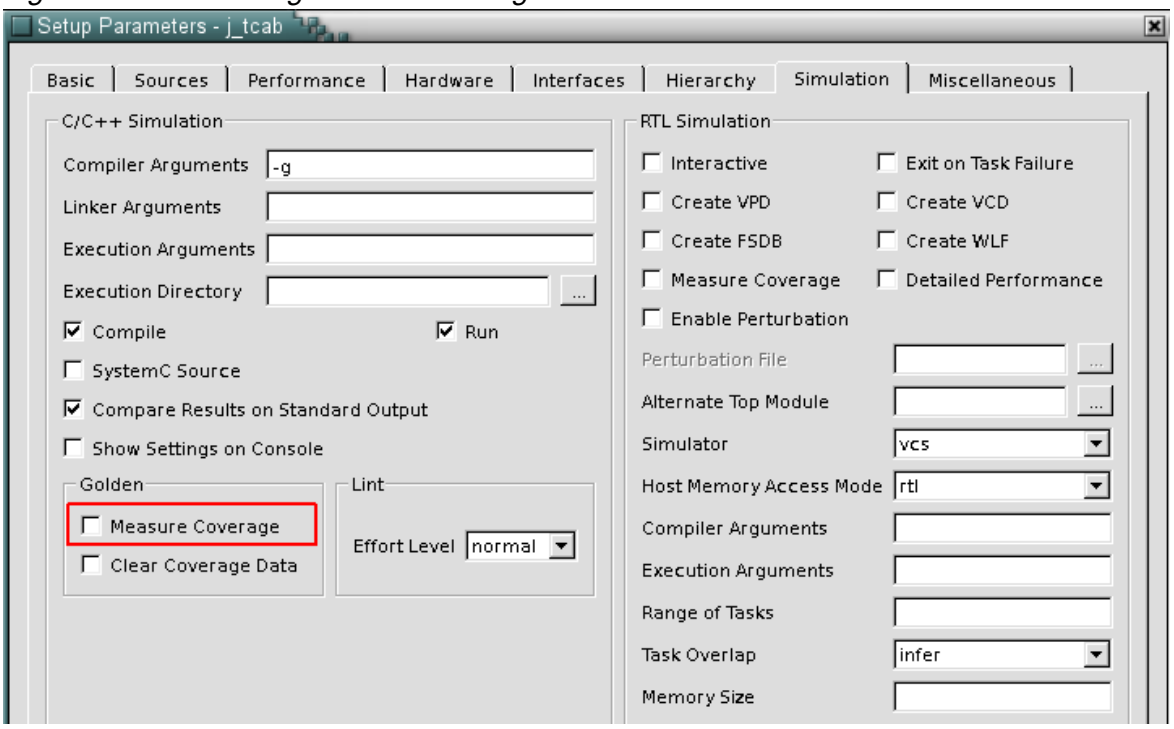
- Checking C Code Coverage with Golden Simulation
- Checking Verilog Code Coverage

Checking C Code Coverage with Golden Simulation

The C code is high-level algorithmic code. Therefore, checking C code coverage is the most efficient way of ensuring that the resulting RTL is also covered. Follow these steps to measure the C code coverage when you run golden simulation.

1. Specify coverage options for golden simulation.
   - Open the dialog box for parameters and select the Simulation tab.
   - Select the Measure Coverage box.
2. Run golden simulation.
3. Analyze the results.

Figure 4-8 Measuring C Code Coverage



The equivalent Tcl command is `verify -golden -ccompiler_args` *`arguments`*

Checking Verilog Code Coverage

The Synphony C Compiler tool lets you use a Verilog simulator and its built-in code coverage features to check the code. There are different kinds of code coverage tests, like statement coverage, branch coverage, expression or condition coverage, FSM coverage and toggle coverage, See the documentation for the Verilog simulator for details of the coverage metrics available.

The following procedure describes how to check if you have reasonably complete if not full coverage of the application at the C code level.

1. You must have a license for the simulator you want to run.

   Use VCS simulator for code coverage . This procedure describes how to use the VCS tool for code coverage.

2. Run RTL simulation with the coverage option.

   - Open the dialog box for parameters.
   - Specify the simulator you want to use.
   - Select the Measure Coverage box.

     You can do the same thing from the command line by specifying `verify -rtl -simulator vcs -coverage_on`.

3. Check the results in *`rtl_package`*`/urgReport/dashboard.html`.

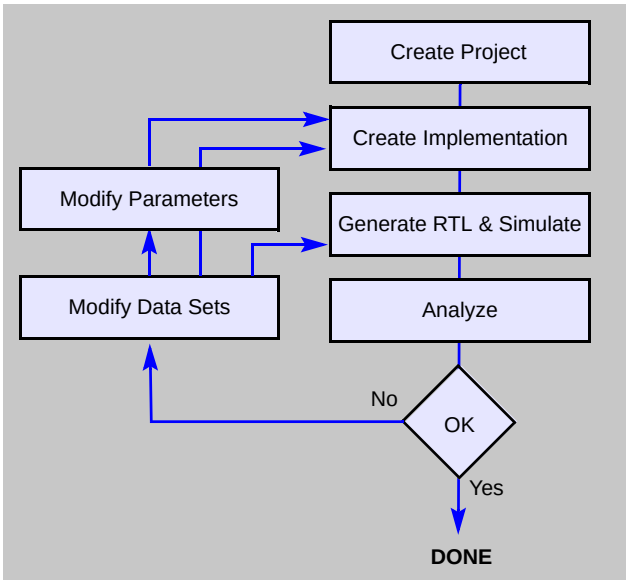
   Analyze the coverage reports and summaries and figure out why the test vectors are not covering the remaining uncovered code.

   - Determine what you can exclude, like unused code or lines that have a reason for not being covered.
   - If you have input or output data streams, consider whether perturbation might help. See Verifying Corner Cases for details about using perturbation modules.
   - Run the simulator in interactive mode, and check uncovered statements and branches to see if you can improve coverage.

The Simulation Regression Flow

During the design phase, you repeatedly refine the Synphony C Compiler C code as you explore different architectures and modify the code to improve quality of results. As a design engineer, you must have a set of regressions that help verify functionality and measure and verify performance.

Figure 4-9 Simulation Regression Flow



The simulation regression flow is focused on the regression process. Use this flow to run multiple data sets and validate your designs. The build and verification steps that the tool executes in this flow build the design, run intermediate simulations, generate RTL, and run RTL simulation.

You can run this flow from the GUI or from a script, but it is more efficient to do it from a script. See the following for details:

- Running Simulation Regressions from a Script
- Running Simulation Regressions from the GUI
- Running Synphony C Compiler EV Regressions

Running Simulation Regressions from a Script

The most efficient way to run simulation regressions is to use a script. Set up the simulations to run across the full regression suite, exercising the functionality and checking for both correctness and performance. You should also cover corner cases.

1. Run one implementation through the built-in RTL Simulation flow (Running RTL Simulation).
2. Create a script.

   The Synphony C Compiler software includes an example `do_test` script (`<`*install_dir*`>/examples/tcl_include/do_test.tcl`). Use this script as an example to create your own.

3. Use the script to run one data set through the implementation for all simulations:

   This creates all the required simulation binaries that remain valid until you update the implementation with the update_implementation command. You do not need to create new binaries for each data set.

4. Organize the other data sets to be run with other configuration files.

   - Set up a separate configuration file for each valid combination of the `data`, `results` and `exec_args` parameters.
   - Make sure you have enough data sets to get good coverage.

   This table lists the simulations you can run for different scenarios:

To verify...	Run these simulations...
Functionality with RTL simulation	RTL Simulation (`verify -rtl`)
Using a full regression	Golden Simulation (`verify -golden`) RTL simulation with your own RTL testbench and the RTL generated by the SCC tool.

Running Simulation Regressions from the GUI

Running simulation regressions from the GUI is a manual process, as you have to initiate each run. It is much easier to use a script, as described in Running Simulation Regressions from a Script.

1. Run one implementation through the built-in RTL Simulation flow (Running RTL Simulation).
2. Run one data set through the implementation for all simulations:

   This creates all the required simulation binaries that remain valid until you recompile the source files. You do not need to create new binaries for each data set.

3. Organize the other data sets to be run with other configuration files.

   - Set up a separate configuration file for each valid combination of the `data`, `results` and `exec_args` parameters.
   - Make sure you have enough data sets to get good coverage.

4. Run the other simulations on all the data sets as required.

5. Analyze the results.

Running Synphony C Compiler EV Regressions

With the Synphony C Compiler 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.

For more information about running EV Regressions, see Running Regressions in the Synphony C Compiler EV.

5

Analyzing the Design

To facilitate the analysis of results and to help with debugging your design, the tool provides a number of reports and graphic views. The following describe how to use them to analyze results after different build phases or for specific tasks.

- Analyzing After Different Build Phases
- Checking the Source Code
- Analyzing Timing and Performance
- Analyzing Resources and Costs
- Checking Power Consumption
- Analyzing System Integration
- Working in Graphical Analysis Views

The views and reports described are primarily for the analysis of results.

Analyzing After Different Build Phases

The tool generates reports and views after different stages in the design process. You can check the design summary after any design phase, but there are other specialized reports and views you can check after different phases. See the following for details about the kinds of information you can analyze after each design step:

- Checking the Design Summary
- Analyzing After RTL Simulation
- Analyzing After RTL Simulation

Checking the Design Summary

Whenever you run build or simulation steps, you can check the design summary to get consolidated information about the design at that stage. For detailed information about these report files, see the *Synphony C Compiler Reference Manual*.

1. After running any build or verification step, you can check the report_summary.txt file in the Reports directory.

   This contains the performance summary. The tool updates it after every build and verification step, and the information available depends on what was run. Check for details about the following:

   - Estimated cost of the design, and the cost after RTL synthesis
   - Implementation of local memories, streams, and loops
   - Hierarchical block properties, if the design is a hierarchical block
   - Constraints that were relaxed
   - Power consumption

2. To check the constraints see the report_constraints.txt file in the Reports directory.

   It contains the following information:

   - Parameters you set
   - Pragmas you specified in the code
   - Performance constraints, like design frequency, task overlap, and task II

Analyzing After RTL Simulation

After running RTL simulation, you can analyze the run using the reports and views described. For information about debugging problems you run into at this point, see Verilog RTL Simulation Issues.

1. Check the design summary, as described in Checking the Design Summary.
2. Review the performance information.
   - Graphically review the performance information using the Dynamic Performance Graph (see Using the Dynamic Performance Graph for more information).
3. For power consumption details, check report_power_theater_sv.txt and report_spyglass_offline.txt.

Analyzing After RTL Simulation

After running RTL simulation, you can analyze the run using the reports and views described:

1. Check the design summary, as described in Checking the Design Summary.
2. Review the performance information.
   - Graphically review the performance information using the Dynamic Performance Graph (see Using the Dynamic Performance Graph for more information).
   - Check the report_verify_rtl_performance.txt file for the loop start and stop events for the RTL simulation run.
3. For power consumption, check report_power_theater_sv.txt.

Viewing RTL Waveforms

To perform RTL simulation, the Synphony C Compiler tool generates a testbench along with the RTL. To aid RTL understanding and debugging, the tool generates an RTL monitor as part of the testbench. RTL monitor is a module hierarchy, which is a replica of interesting RTL hierarchy (for example, PAs, TCABs etc.) containing ports and states, which are at transaction level as opposed to signal-level RTL. For example, a PA monitor contains information about the state of a PA and has transaction-level ports for stream inputs and outputs on the PA. This abstraction makes it easy to understand the RTL structure via schematic viewers and its execution via transaction-level waveforms.

For any debugging of the RTL, use the RTL monitor view instead of the RTL itself.

The RTL monitor hierarchy is a stand-alone set of SystemVerilog modules and can be relocated to the user-specified testench.

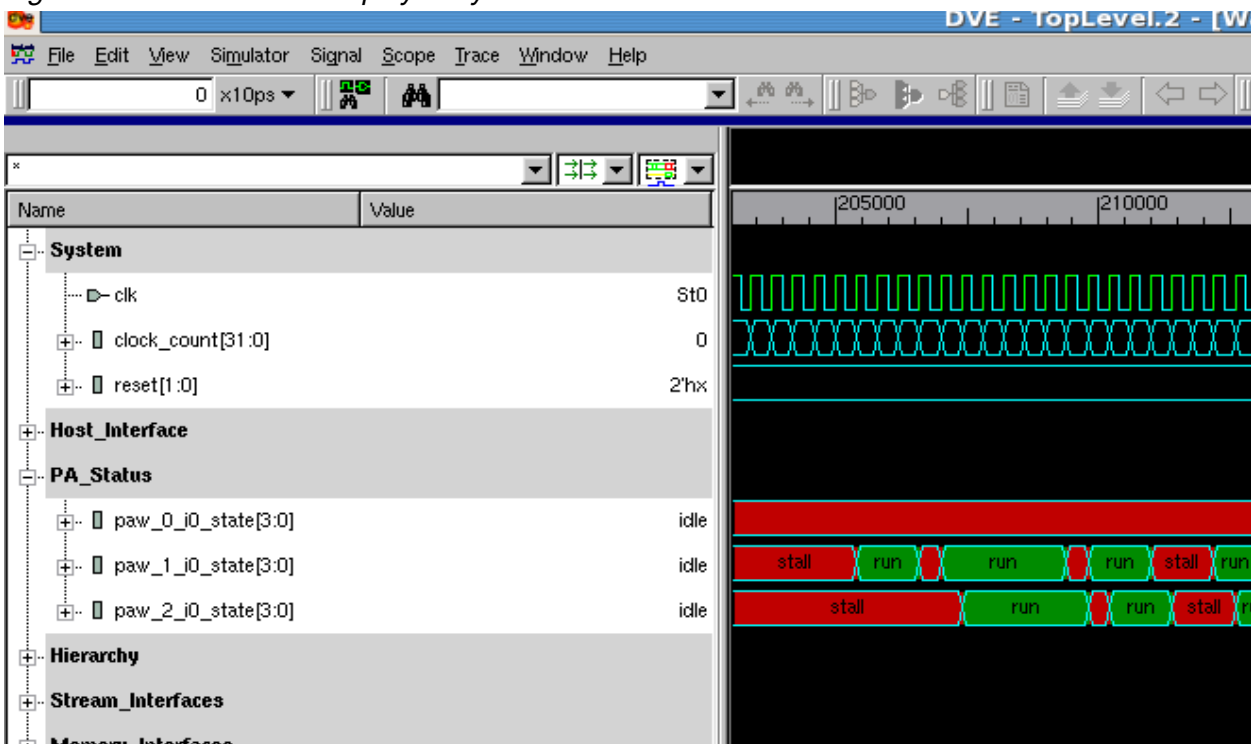
To view monitors, run the RTL simulation with interactive or waveform dump options. Use the `view_waveform` command to open the monitor. It opens a standard viewer (like DVE) and shows monitor contents along with the RTL. You can perform various tasks like adding signals, viewing schematics, and so on. The `view_waveform` command supports multiple viewers. For each viewer, use the following waveform dump option when you run RTL simulation, as shown in Table 5-1:

Table 5-1 Supported Waveform Viewers

Option	Supported Viewer
`verify -dump_vpd`	DVE viewer
`verify -dump_fsdb`	Verdi viewer
`verify -dump_trn`	Simvision viewer

Figure 5-1 displays a monitor view in DVE viewer, showing PA status for each PA, start and done handshakes in transaction form with the corresponding input and output data.

Figure 5-1 Waveforms Displayed by DVE Viewer



Checking the Source Code

Your design implementation depends on the source code you start with. The first step is to ensure that the code is written for an optimal hardware implementation. Typically, you use reports generated at the beginning of the design cycle, to check the code.

1. Open the report_original_loop_to_line_number.txt file and check the following:
   - Information about unrolled loops
   - Location of a loop in the original source code
   - Information about jammed loops

   Make any changes required.
2. Open the report_perfectized_loop_info.txt file, and check how the out-of-loop code was mapped.

   This report contains the exact mapping from the code to the processing arrays in the hardware implementation. Fix any errors you find.
3. Check the report_rate_match.txt file to debug the problems with meeting the task II.

   This file lists the perfectization choices, stream, and hierarchical block constraints. Typically, you do not need to check this file if you worked on the loops after step 2.
4. To check loop connectivity, use the PPA loop graph, as described in Using the PPA Graph.

   Use this to graphically check how the loops connect using memories, streams, and scalars.
5. Check loop sequencing in the Sequentialization view, as described in Using the Sequentialization Graph.

   Use this to graphically check dependencies between loops.

Analyzing Timing and Performance

The tool generates both text reports and graphical views for you to analyze timing. See the following for more information:

- Analyzing Schedules and Timing Paths
- Analyzing Feedback Paths
- Analyzing False Paths
- Checking Performance

Analyzing Schedules and Timing Paths

You can view how operations are scheduled graphically or in a text report. The two do not report or show exactly the same information; therefore, ensure that you check both.

1. To view the schedule of operations graphically, use the Operations Schedule viewer, as described in Using the Operation Schedule Viewer.
2. To check the schedule of operations in a text report, do the following:
   - To display the generated report in the console window, specify the `report_implementation -pseudo_code` command.
   - Check the report_pseudo_code.txt file for opcode information about loops that failed the synthesize step. The file lists the schedule of operations in each loop and the corresponding hardware instance name for the operation.
   - Check the omega values (numbers in square braces) for the number of iterations between the last update and the current access.

Analyzing Feedback Paths

You can view the feedback path information graphically or look through the text reports.

1. To view information about feedback paths graphically, use the Feedback Path viewer and the Operation Schedule viewer together.

   For information about using these windows, see Using the Feedback Path Viewer and Using the Operation Schedule Viewer.
2. To check feedback path information in a report, do the following:
   - To view the feedback reports in the console window, specify the `report_implementation -feedback_path` command.

     The reports list the most critical feedback paths, regardless of whether the design meets timing. If it does not meet timing, you must check the reports and fix the paths.
   - Check for over constrained designs caused by too much packing. This might cause II to fail. You must re-architect the design to work around this.
   - Check the iteration distance, which represents the amount of time available for scheduling the cycle.
   - Check the slack.
   - Look for timing failures.

Analyzing False Paths

You can view the false path information graphically or check the text report.

1. To view information about false paths graphically, use the Operation Schedule and Feedback Path viewers.

   For information about using these windows, see Using the Feedback Path Viewer and Using the Operation Schedule Viewer.

2. To use the text report, check the report_false_paths.txt file after the synthesize phase.

   The file lists information about false paths in the generated RTL. Note that this report only lists false paths caused by functional units shared within a loop. It does not report false paths caused sharing memories or stream hardware.

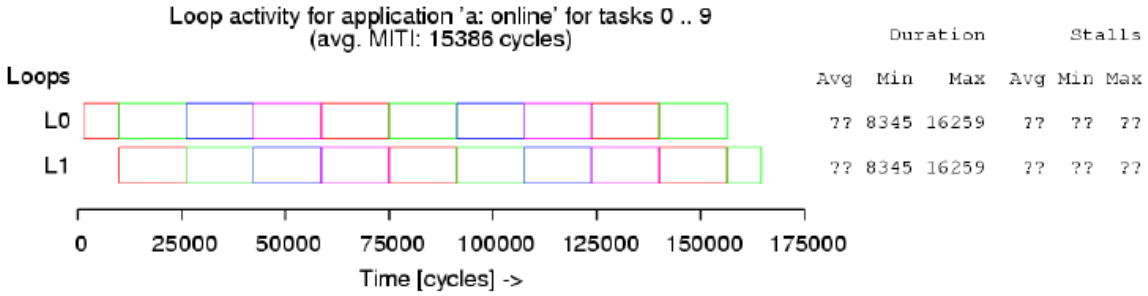
Checking Performance

The Synphony C Compiler tool generates performance reports after different simulations. In addition, you can use the Dynamic Performance graph.

1. Run the simulation.
2. View the performance graphically by doing either of the following steps:
   - Use the Dynamic Performance graph. See Using the Dynamic Performance Graph for details. You can generate this graph after RTL simulation, and event simulation.
   - Use performance charts in PDF or Postscript formats for RTL simulation and RTL simulation. To do this, you need to install the free gnuplot package. From within the tool, specify the `report_implementation -performance` command.

     You see information in performance output, as shown in the following example:

Figure 5-2 Loop Activity



3. Check the appropriate report for details.

Analyzing Resources and Costs

The tool generates both text reports and graphical views for you to analyze the cost of the hardware implementation. See the following for more information:

- Analyzing Cost
- Checking Hierarchy

Analyzing Cost

You can analyze cost graphically or through a text report.

1. Check resource usage.
   - For a graphical view of the design and the resources used, open the Resource Browser. See Using the Resource Browser for details.
   - For the text file equivalent of the Resource Browser, use report_cost.txt. The file lists the resources for the entire PPA or hierarchical block.
   - For detailed information about register resources, view the report_register.txt file.
2. Compare the costs of different implementations with the Relative Cost viewer.

   See Using the Relative Cost Viewer for details.

Checking Hierarchy

The tool lets you analyze the design through a browser or through PDF files. The following let you check the physical locations of hierarchical blocks and view how they are shared.

1. Start with a design with hierarchical blocks.
2. View the PDF files.
   - To view the complete hierarchy, open report_hierarchy_detailed.pdf.
   - To view just the first level of hierarchical blocks, open report_hierarchy.pdf.

   Both files display the hierarchy and show how they blocks are shared.
3. To view the hierarchy graphically, use the Hierarchy Graph, as described in Using the Hierarchy Graph.

Checking Power Consumption

The tool provides report files that you can check for power consumption in your ASIC design.

1. Run RTL simulation.
2. After RTL simulation, open the report_power_theater_offline.txt file and check the power consumption values. The report lists the total power consumption and also breaks it down in terms of internal power consumption, IP core consumption, and so on.
3. If you ran RTL simulation and used the Spyglass linter, check the report_spyglass_sv.txt file for information about power consumption.

Analyzing System Integration

After the synthesize step, you can check various reports to analyze system integration.

- Checking System Integration Reports
- Keeping Address Maps

Checking System Integration Reports

1. Synthesize the design to generate the report files.
2. Check memory mapping in report_address_map.txt.
3. Check port mapping in report_interface.txt.

   Check how the logical ports map to the physical ports and to the original source code.
4. Check boundary timing in report_interface_timing.txt. Use the information in this file to determine if the generated input or output needs to be registered.

Keeping Address Maps

If you want to ensure that you always get the same address map, modify the address map file as described:

1. Synthesize the design to generate the report_address_map.txt file.
2. Edit the Tcl section if required.
3. Use the file as input into the Synphony C Compiler tool.
   - Copy the edited map file to the project directory.

- Add the file with the following command: `set_params -host_address_map fileName`.

Working in Graphical Analysis Views

There are many graphical analysis views that the tool generates at different points in the design cycle. You can use them instead of the text reports. The next sections discusses how to use these analysis views and also discusses some operations you can do in most views:

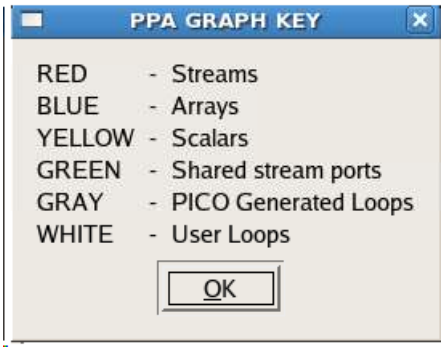
- Performing Common Analysis View Operations
- Crossprobing to C Code and RTL
- Using the PPA Graph
- Using the Sequentialization Graph
- Using the Feedback Path Viewer
- Using the Operation Schedule Viewer
- Using the Resource Browser
- Using the Relative Cost Viewer
- Using the Dynamic Performance Graph
- Using the Hierarchy Graph

Performing Common Analysis View Operations

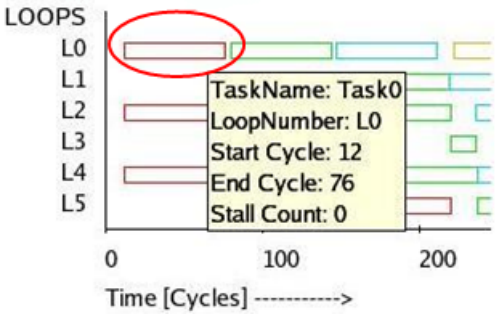
Some of the display and view operations are the same in different views. They are documented together here, for the sake of convenience. Some of the commands described here are not available in all views.

1. For graph views, use the zoom buttons to set the level of display detail you want to see.
2. For graph views, click the icon to display the key to graphical objects and the colors used in that view:

   The tool displays information about interpreting the view. This example is from the PPA Graph view:

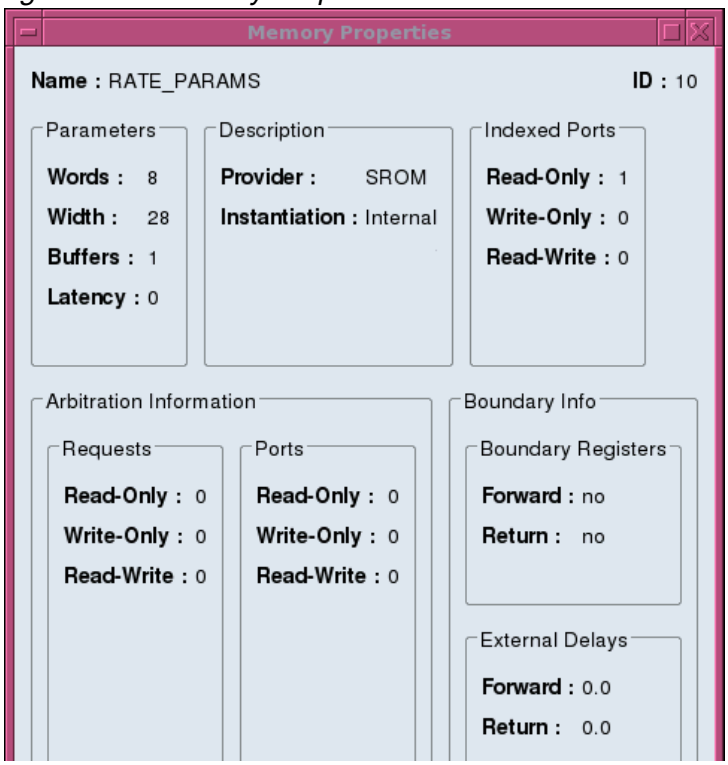


3. To display information about an object do any of the following:
   - Hover over an object with your mouse to display a tooltip with information about that object. The following shows a tooltip for a loop:



   - Right-click an object and select Properties from the pop-up menu to see a list of properties for the object.

Figure 5-3 Memory Properties



- Right-click an object and select Source Context from the pop-up menu to see the corresponding source code for the object. See Crossprobing to C Code and RTL for an illustration of the window.

4. Click an object to highlight and select it.

Crossprobing to C Code and RTL

Many analysis views let you let you crossprobe from the view to the corresponding C code. The PPA graph and Sequentialization graph views also let you crossprobe between the C code and the RTL.

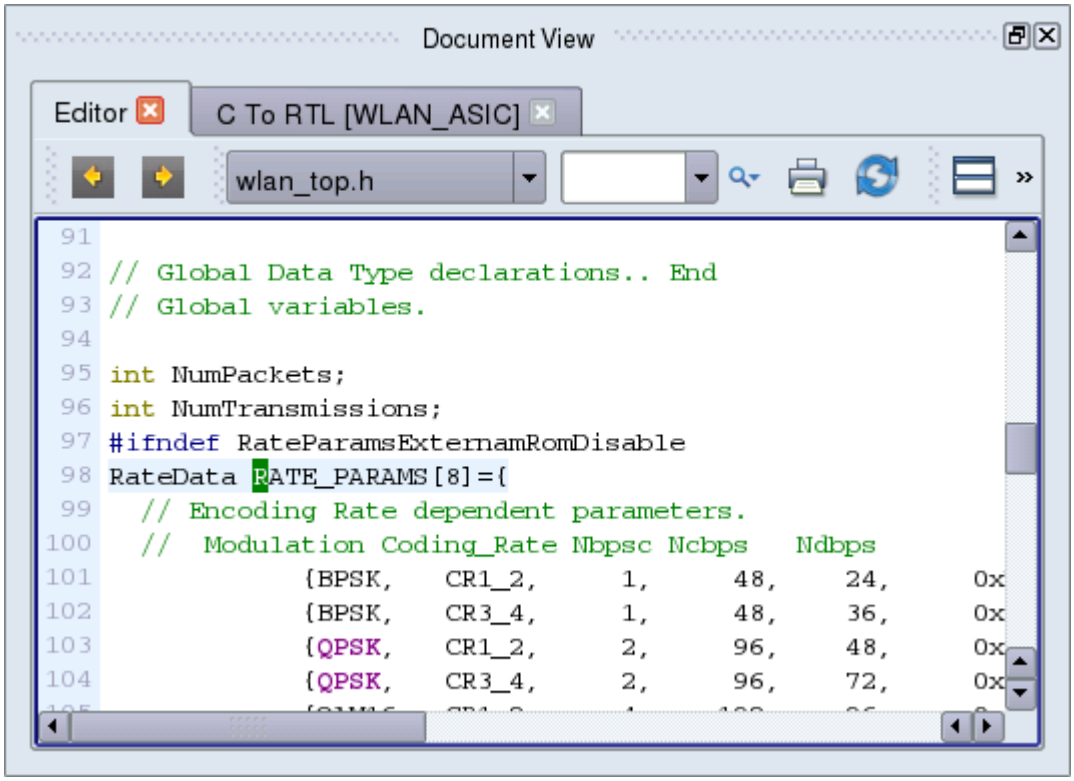
1. In the analysis view, right-click an object.

   This brings up a menu from which you can select crossprobing commands.

2. To crossprobe and see how the object was defined in the C code, select Source Context.

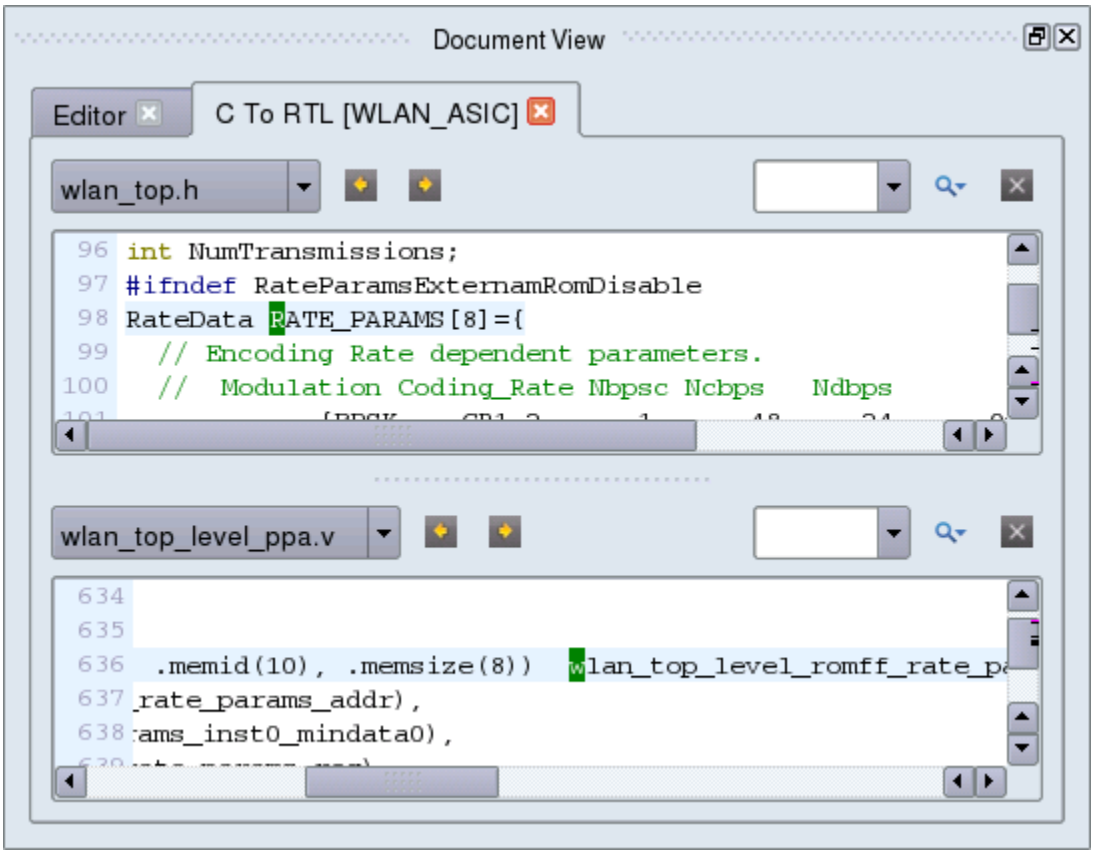
   A window opens with the C code and the cursor jumps to the appropriate location, as shown:

Figure 5-4 Document View



3. In the PPA graph and Sequentialization graph views, to crossprobe from the C to RTL views, right-click an object and select C to RTL Mapping. You see a split window displayed, showing the C code on one side and Verilog code on the other.

Figure 5-5 Viewing C Code and Verilog Code in the Document View



Using the PPA Graph

The PPA Graph is resource-centric, and displays information about the resources being used by each loop. Use this view to analyze the interaction between loops and the rate of execution between loops.

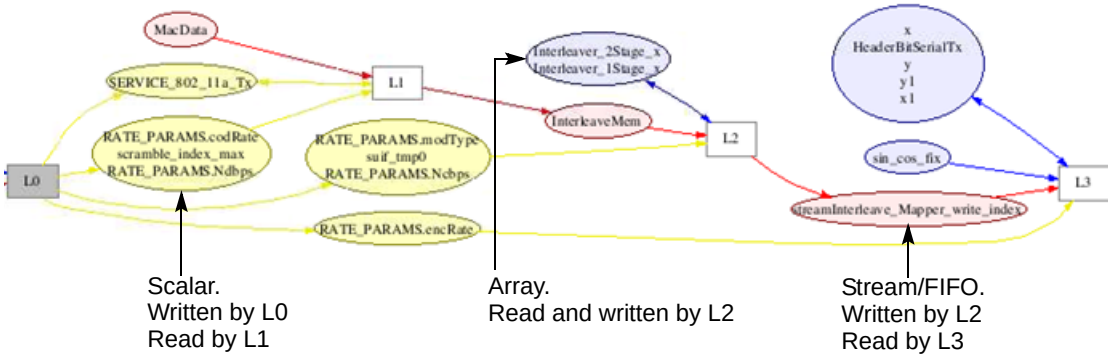
1. Set up the tool properly.
   - Install the Graphviz package. See the installation notes for details.
   - Make sure the dot command appears in the PATH variable.

   You must set these up to generated the loop graphs.
2. At a minimum, run analyze and build.

   You must run these operations to generate the data for the PPA graph.
3. Open the PPA graph by selecting Analyze > PPA Graph or by clicking the icon:

4. Identify shared resources and their dependencies, as shown in this figure:

Figure 5-6 Shared Resources and Their Dependencies



The loops are rectangles, blue ovals represent memories and arrays, red ovals represent streams and FIFOs, yellow ovals represent scalars, and green ovals represent FIFOs that have more than one reader or inter-loop dependency. Display the key if needed.

Using the Sequentialization Graph

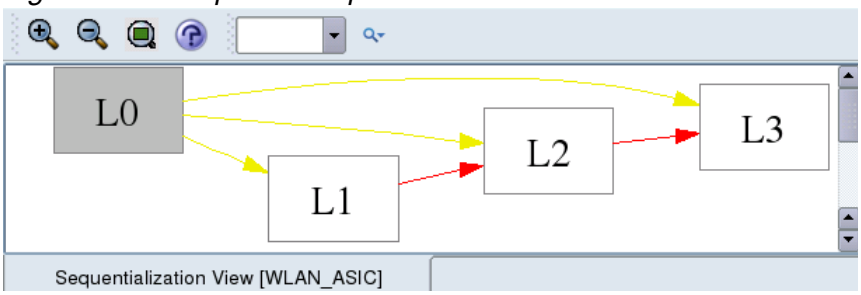
Use this to determine which loop nests are sequential. The graph displays sequentialization caused by data and resource dependencies.

1. Set up the tool properly.
   - Install the Graphviz package. See the installation notes for details.
   - Make sure the dot command appears in the PATH variable.

   You must set these up to generate the loop graphs.
2. At a minimum, run analyze and build.

   You must run these operations to generate the data for the sequentialization graph.
3. Open the sequentialization graph window by selecting Analyze->Sequentialization Graph or by clicking the icon:

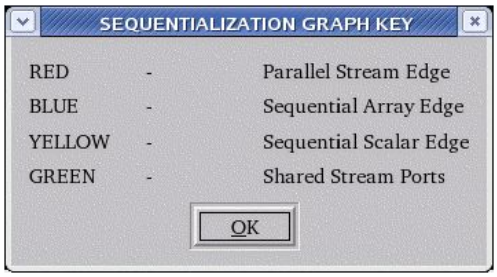
Figure 5-7 Loops and Dependencies



4. Identify dependencies between the loops.

   The view displays the loops as rectangles and the colored arrows show the dependencies: memories are blue, streams are green, and scalars as yellow. Click the icon at the top of the window to display the key as shown in Figure 5-8:

Figure 5-8 Sequential Graph Key



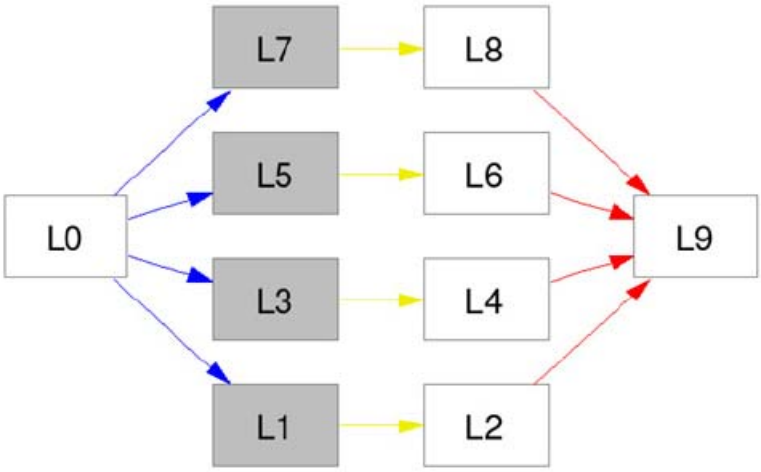
5. Display tooltips and context-sensitive information, and crossprobe as needed.

   See Performing Common Analysis View Operations and Crossprobing to C Code and RTL for details.

Example of Interpreting Dependencies

The following figure shows an example with four perfectized loops (gray boxes) and six user loops (white boxes). It shows four parallel streams of execution that begin at loop L0 and converge at loop L9.

Figure 5-9 Perfectized Loops



Here are the details:

- The perfectized loops only execute after L0 completes, because each perfectized loop accesses a memory updated by L0 (blue arrows).
- L2, L4, L6, and L8 execute after L1, L3, L5, and L7 respectively, because they are updated by scalar variables (yellow arrows) from the perfectized loops.
- L9 executes at the same time as L2, L4, L6, and L8, because it uses streams (red arrows) to communicate with these loops.
- L0 and L0 can execute in parallel because they do not have any data dependencies in common.

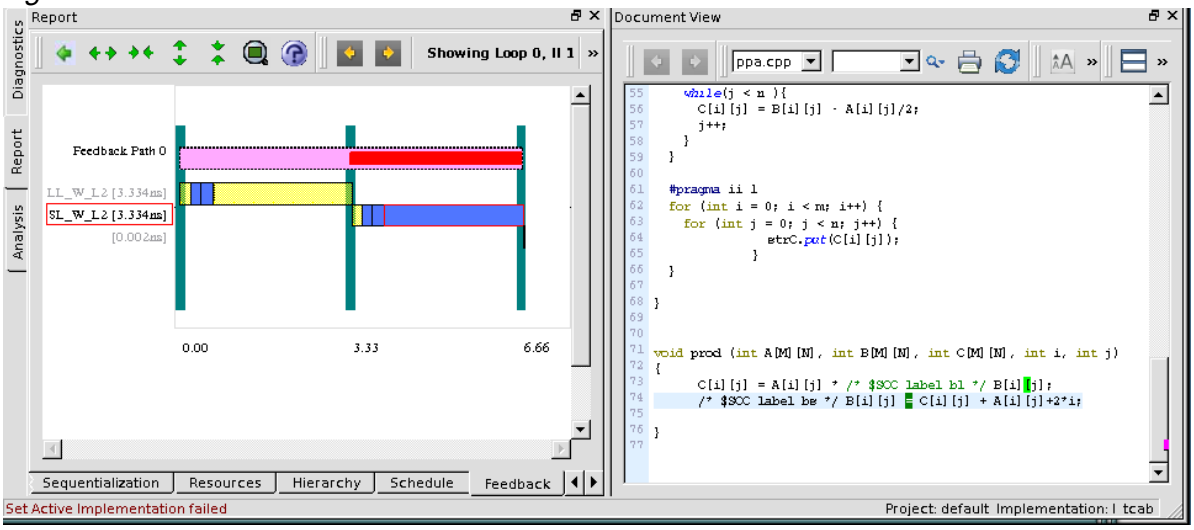
In summary, L0 executes first. The perfectized loops that follow start almost simultaneously. As each perfectized loop finishes executing, the sequentially dependent loop connected to it (L2, L4, L6 or L8) starts executing and streams data to L9.

Using the Feedback Path Viewer

This view lets you check the status of all feedback in the program. It shows the critical feedback paths on a per-loop basis and details about the operations involved.

1. To view the feedback path, select Analysis > Feedback Path Viewer or click the icon:

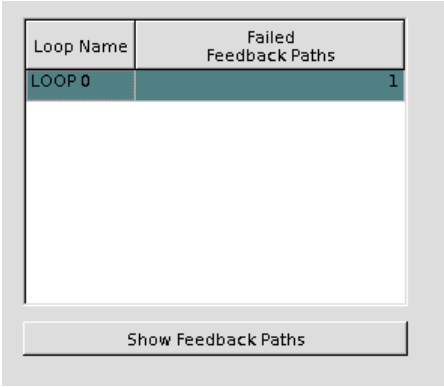
Figure 5-10 Feedback Path View



2. To view a feedback path, do the following:

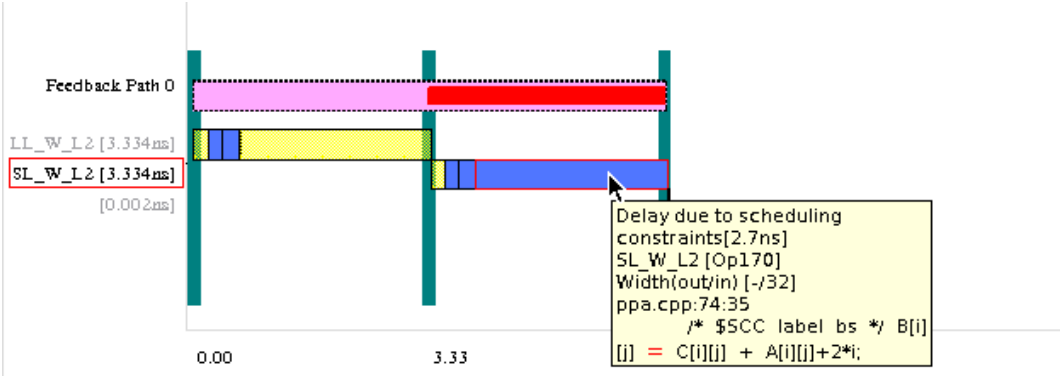
   - Select the loop you want to examine in the loop selection table and click Show Feedback Paths.

Figure 5-11 Loop Selection Table



   - Click a feedback path to expand it and view the instructions beneath. The following shows a path that passed. You can hover over different parts of the view to get information.

Figure 5-12 Details of a Feedback Path



If you hover over an instruction (blue bar), you get information about the opcode and C source. Hovering over the II (vertical teal bar) reveals the II, and hovering over a clock (turquoise vertical bar) shows you the time progress.

You can also right-click an instruction to open the source code.

Figure 5-12 shows a feedback that failed. Note that the operations are in program order. The pink colored bar indicates a failure, and the red bar reflects the negative slack.

3. Use the back arrow at the left of the window to go back to the loop selection table.

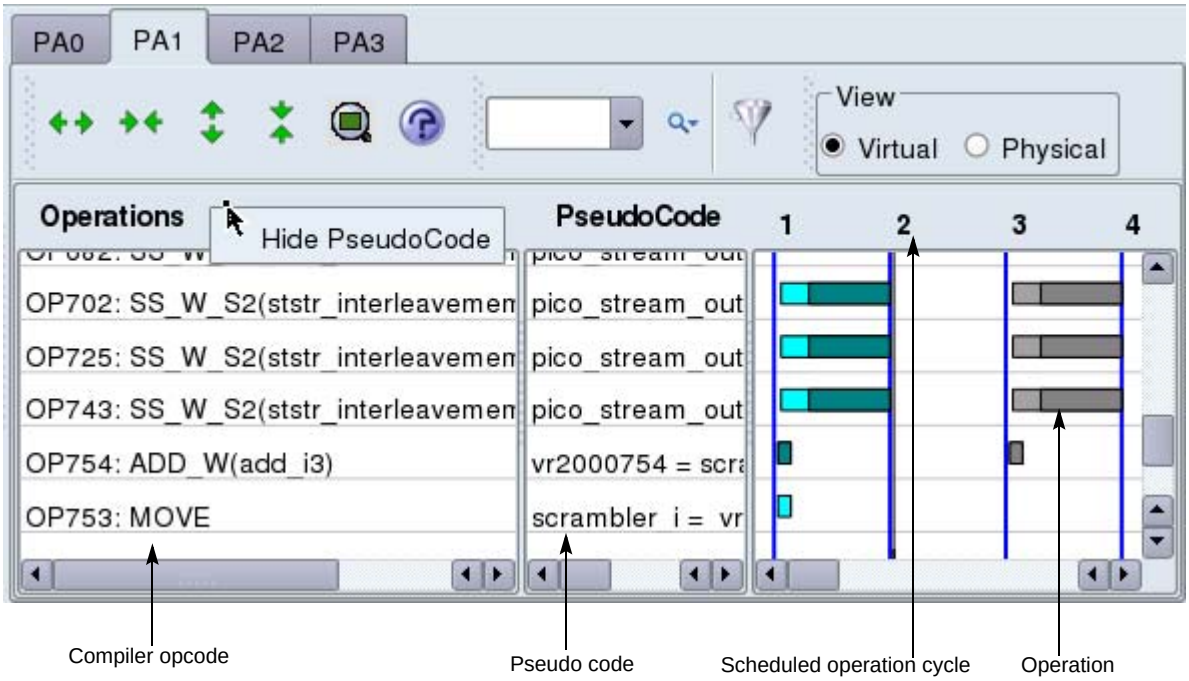
Using the Operation Schedule Viewer

The Operation Schedule view displays the operation schedule and shows the processing arrays (PAs) on a PA-by-PA basis.

1. Start with an active implementation. If needed, select Implementation->Set Active to make an implementation active.
2. To start the Operations Schedule Viewer, select Analyze->Operations Schedule Viewer, or click the icon:

3. The Operation Schedule Viewer displays the operations, pseudo code, and operation cycles for each PA. Each PA is in a different tab, and each tab shows the operations scheduled in the PA. PAs that are successfully scheduled are shown in schedule order. PAs that fail to meet constraints are presented in program order.

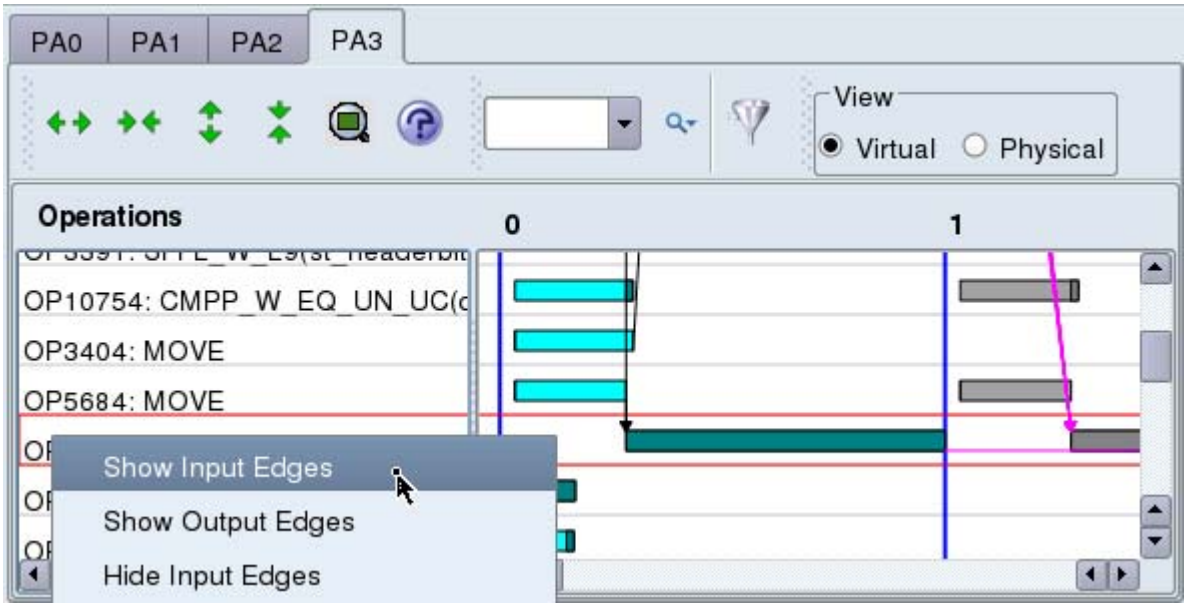
Figure 5-13 Operation Schedule Viewer



Blue indicates the first occurrence of the operation, and gray indicates repetitions, separated by initiation intervals (II) across different iterations. Within a bar, blue indicates the input read operation and green or gray indicates operation execution.

4. Trace dependencies by tracking the input and output edges of an operation.
   - To view all the edges in an implementation, click Show Edges.
   - To view edges on an individual basis, select an operation, right-click, and then select the appropriate command from the pop-up menu to display the required input or output edges. Figure 5-14 shows input edges:

Figure 5-14 Tracking the Input and Output Edges



- To view source content, place the mouse over an operation and a tooltip reports the location of the source.
- To view paths, right-click and choose Show Path. The window displays a sequence of edges, starting from a source operation and ending at a destination operation.

Using the Resource Browser

This view hierarchically lists the hardware resources needed to implement different loops in the input application. This view also shows the hierarchical blocks (TCABs) which are shared across loops at the top level. Use this view to determine the types of memories used in the design and check the cost of implementing functions.

1. There are different ways to open this view:
   - Select Analyze->Resource Browser or the Resource Browser icon:

   - From another analysis view, right-click an object and select Resource Browser from the pop-up menu.

Figure 5-15 Resource Browser

The view consists of a tree view on the left and a table view on the right.

2. Check the aggregate cost for all the loops at the bottom of the tree view.
3. View hardware resources.
   - Expand the tree view as needed and select the object you want in the tree view. The corresponding row in the table is highlighted. The following example shows usage for streams.

Figure 5-16 Using Streams

   - Check resource usage in the table.

4. You can view properties and crossprobe the source code.

   See Performing Common Analysis View Operations and Crossprobing to C Code and RTL for more information.

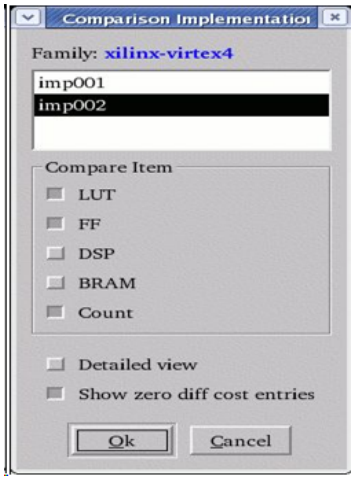
Using the Relative Cost Viewer

Use the Relative Cost view to compare the cost of different implementations. You can customize the level of detail you want to see.

1. Synthesize the implementations you want to compare using the same technology library.
2. Make one of the implementations active by selecting Implementation->Set Active Implementation.
3. Select Analyze > Relative Cost Viewer or click the icon:

   The Relative Cost viewer opens.
4. To compare the cost of implementations, do the following:
   - Click Select comparison implementations. A dialog box opens:

Figure 5-17 Comparing Implementations



   - Select the implementations you want to compare to the active implementation. To select multiple implementations, click them in succession.
   - Enable the resources you want to compare in the Compare Items section.

- Enable Detailed View if you want to see individual functional units. By default, they are grouped together. For example, all select multiplexers are grouped together as select.

  The following figure shows how a macro cell, cmpp, is treated with the two settings. With Detailed View disabled, cmpp is treated as one macrocell, but with it enabled, you see it itemized as different functional units, based on the number of inputs.

- Disable Show zero diff cost entries if you only want to display the entries that contribute towards a cost difference between the two implementation. By default, the tool displays all entries, even if the implementations have the same cost because they have an equal number of instances of a particular kind.

  Figure 5-18 shows the default setting, with the zeroes in the cost column for the zero-cost difference components.

Figure 5-18 Default Setting for Components with Zero Cost Difference

Macrocell Name	Gates i2 relative to i1	Gates i1	Gates i2	Count i2 relative to i1	Count
Merge Register	-63	309	246	-7	
pred compare (!=)	-36	54	18	-15	
pred compare (==)	-36	90	54	-12	
selector (1-hot) (5:1)	-30	30	0	-1	
bitwise and	-14	25	11	-15	
stream load	-13	13	0	-1	
selector (1-hot) (8:1)	-9	9	0	-1	
nor gate (2-input)	-6	11	5	-6	
incr-decr counter (reset=yes,init=no)	-3	3	0	-1	
selector (1-hot) (4:1)	-3	32	29	4	
bitwise xor	-2	2	0	-2	
equality check	-1	2	1	-1	
fast sram load	0	12	12	2	
shift left by const	0	0	0	-9	
shift right by const	0	0	0	-6	
decoder	1	0	1	1	
incrementing counter (reset=yes,init=no)	1	0	1	1	
and gate (3-input)	2	0	2	2	
data compare(<,<=,>,>=)	2	82	84	4	
nor gate (3-input)	2	1	3	2	
inverter	5	96	101	5	

Zero cost entries

- Click OK when you are done.

The window shows the costs, compared to the active implementation.

Using the Dynamic Performance Graph

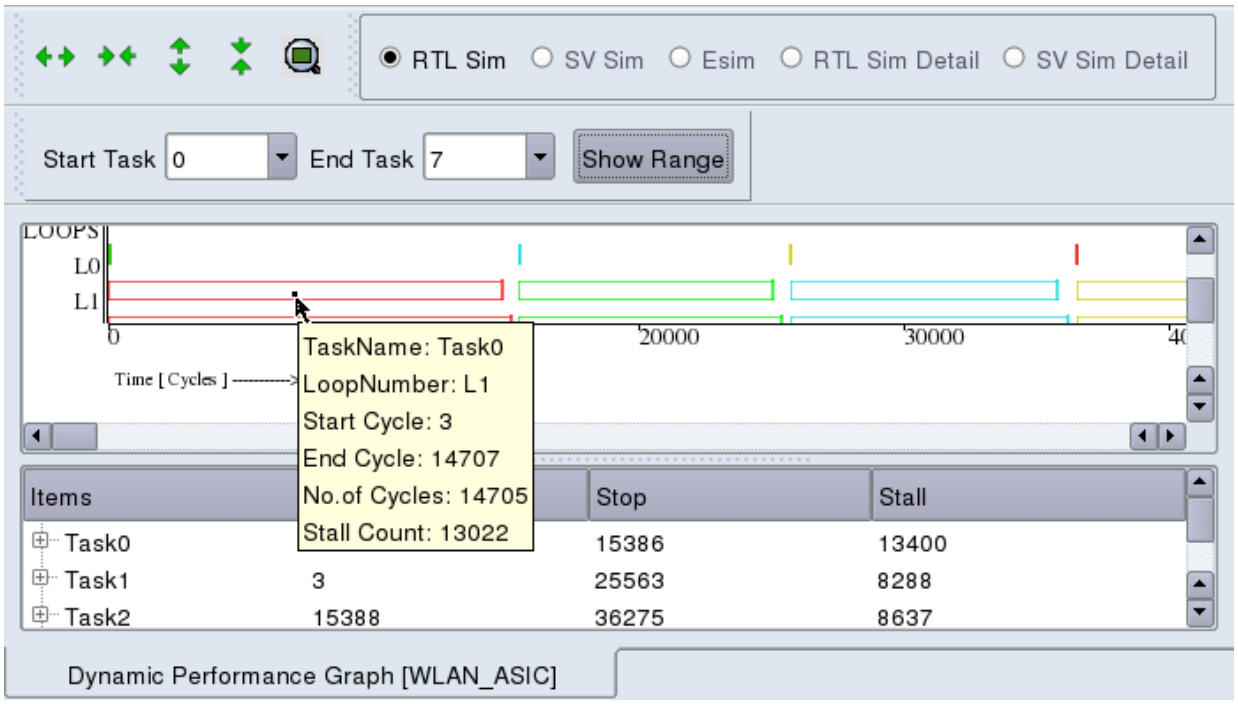
To check design performance in this view, use the following steps. Check if there are overlaps within and across tasks.

1. Run the simulation for which you want to see results: C simulation, RTL simulation, or event simulation.

   If you do not run the simulation, the dynamic performance information is not seen.

2. Select Analyze->Dynamic Performance Graph or the Performance Graph icon to open the view:

Figure 5-19 Performance Graph

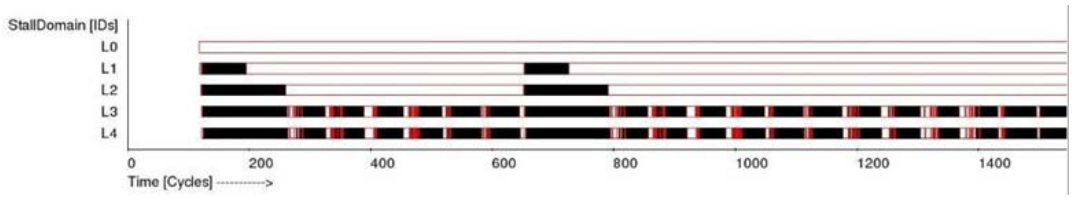


3. Display the performance data.
   - Select the simulation for which you want to see performance results, by enabling the appropriate button at the top. Task Number shows the valid tasks.
   - For C simulation, select a task in Task Number and click Show Range to display the performance for that task.

- For RTL simulation, type task numbers in Start Task and End Task. Then click Show Range.

4. Check the performance data to understand the stalling patterns in the design.
   - Check the graphic diagram of the loops. Black bars indicate stalls.

Figure 5-20 Graphic Loop Diagram



   - Check the task table at the bottom. You can expand tasks to show all the loops in a task and their corresponding cycle numbers.

5. Display tool tips as needed. See Performing Common Analysis View Operations for details.

Using the Hierarchy Graph

The Hierarchy Graph shows a graphical view of the design hierarchy. For a text report, see the report_hierarchy.pdf or report_hierarchy_detailed.pdf files, which show the same information as the graph, as a PDF document.

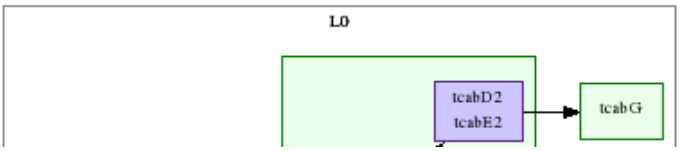
1. Start with a design that has hierarchical blocks.
2. To open the hierarchy graph, select Analyze->Hierarchy Graph or click the icon:

   You can choose to display just one level of hierarchy and hide all lower-level hierarchical blocks, or you can display all the levels of hierarchy.

3. Analyze the hierarchy.

   The following figure shows a hierarchical design with two loops. It has many shared blocks, multiple instances of a hierarchical block (tcabF), and one block that is shared by both loops (tcabZ).

Figure 5-21 Hierarchical Design With Shared Blocks



These are some of the design details revealed in the graph:

- Hierarchy is indicated by the nested boxes. In the example, tcabC is in loop L0, and contains four hierarchical blocks. The tcabD2 and tcabE2 hierarchical blocks are represented with a single box, because they share the same predecessors and successors.
- The colors denote the sharing relationship between the caller and the called function: a green box indicates that the instance has multiple callers, and a blue box indicates that the instance has one caller. tcabz is shared by both loops, and is external to both of them.
- The arrows specify the direction of the calling relationship, with the arrow originating from the caller and pointing to the instance that is called. The number of arrows is generally an indication of multiple calls, but not always. Although there is just one arrow to tcabG, this instance is called by both tcabD2 and tcabE2, as indicated by the caller block. Therefore, use the arrows in conjunction with the color cues.

  The unconnected arrow in L1 indicates that all the callers of tcabY are within that container, L1.

4. For additional information about objects, view properties or the corresponding source code, as described in Performing Common Analysis View Operations and Crossprobing to C Code and RTL.

Using Synphony C Compiler EV

The Synphony C Compiler EV product is simulation intensive. It supports a subset of the Synphony C Compiler commands that are useful in running different simulations. There is no support in this product to rebuild the hardware or simulation models. The use model is:.

1. In the Synphony C Compiler tool, open a project, create an implementation, build the implementation, and create the simulation models using the Synphony C Compiler tool.
2. Invoke the Synphony C Compiler EV product and select the implementation created using the Synphony C Compiler tool, and run simulation using the `verify -rtl` command. This product only supports simulations in run-only mode.

3. The Synphony C Compiler EV product copies an existing implementation into a new one. This feature can be used for running multiple simulations on a given implementation simultaneously.

   Note:
   The platform used for compilation of the simulation executable must be the same as the one used for simulation to avoid any platform-dependent issues.

Running Regressions in the Synphony C Compiler EV

The Synphony C Compiler tool ensures that the result produced at any simulation stage is consistent with all the previous simulations in the selected flow by comparing the output and result files. Running multiple simulations on the same implementation at the same time can produce undesirable behavior. This unwanted behavior enforces a requirement to run multiple simulations serially, one after the other.

The Synphony C Compiler EV product can help you organize your simulations without overwriting one another when the simulations are run in parallel. A good use model, which can be used for running multiple simulations in parallel, is to create multiple copies of an implementation and run different simulations using a category of data sets on each implementation. For example, if there are three modes of operation that need to be run on an implementation (for example, Imp000), perform the following steps to run the regressions.

1. Build implementation Imp000 using all the necessary options to create the correct simulation models in the Synphony C Compiler tool.

2. Create three copies (I1, I2, and I3) of the original implementation using the following commands in the Synphony C Compiler EV product:

   ```
   new_implementation I1 -from Imp000
   new_implementation I2 -from Imp000
   new_implementation I3 -from Imp000
   ```

3. Identify which data set (D1, D2, or D3) and results (R1, R2, or R3) are to be run for each mode. Customize the data results for each implementation using the following commands in the Synphony C Compiler EV product.

   ```
   set_params -data D1 -result R1 (for I1)
   set_params -data D2 -result R2 (for I2)
   set_params -data D3 -result R3 (for I3)
   ```

4. Customize the implementations to run the specific dataset corresponding to each mode by different exec_args values, for example:

   ```
   set_params -exec_args "-d D1" (for I1)
   set_params -exec_args "-d D2" (for I2)
   set_params -exec_args "-d D3" (for I3)
   ```

5. Run the simulations in each of the I1, I2, and I3 implementations using the following command:

```
verify [-golden | -lint] -run_only
verify -rtl -run_only
```

Synphony C Compiler Driver Coding Styles

To verify the RTL generated, you often need to activate different sections of the code (corresponding to different modes and runtime conditions) and look at the coverage obtained to ensure completeness in verification. Verification requires you to change the data input for a particular simulation. There are several ways to change the data input, for example:

1. Code the driver to be able to accept different datasets on the fly. There are several ways to do this, for example:

```
int main (int argc, char *argv) { // driver code as
                    // one of the options, accept the file name.
}
```

2. Set the -exec args (either from simulation command or using the `set_params` command, as shown in the following example:

```
set_params -exec_args "-d data_D1.txt"
verify -golden
```

   Create the simulation models in the Synphony C Compiler tool and specify the options appropriately; use the `-run_only` option in the Synphony C Compiler EV product to vary the different data sets.

6

Synphony C Compiler Output

This chapter describes the Synphony C Compiler Outputs. This chapter is organized as follows:

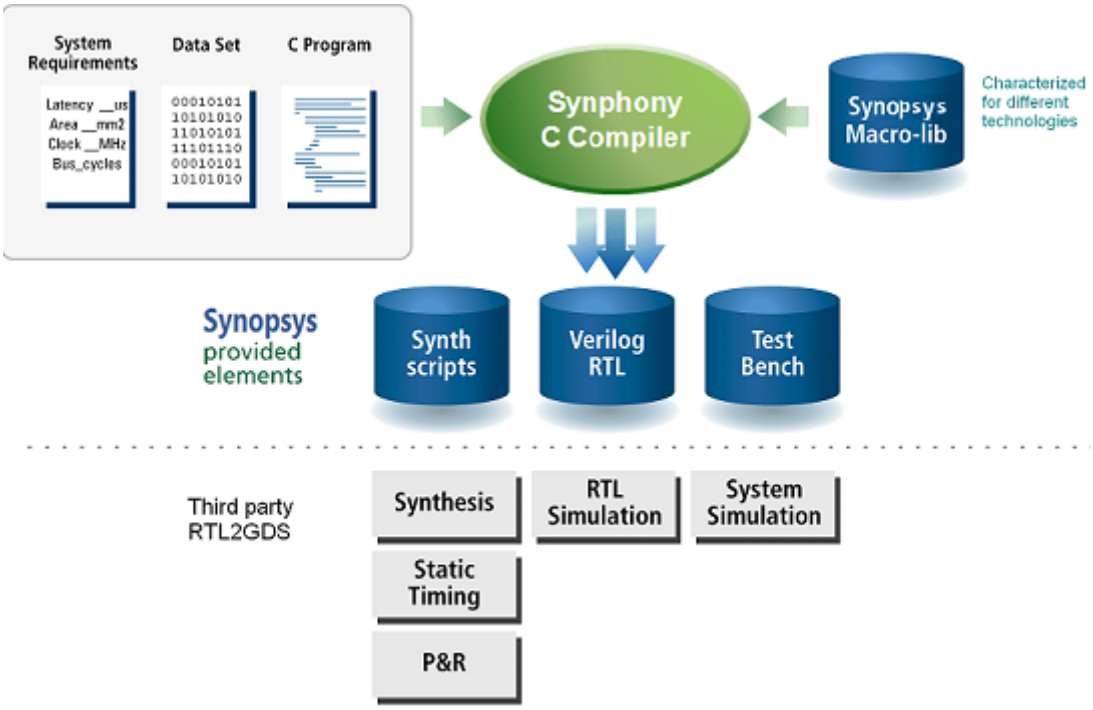
- Overview of Synphony C Compiler Output
- Hierarchy of Generated RTL
- Package Directory Structure
- Directory Structure to Support RTL Simulation
- Corner Case Simulation Support
- Assertions
- RTL Features
- STARC Rule Support

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 RTL simulation of the RTL or to compile and run RTL synthesis. For more information on the directory structures, see "Package Directory Structure" on page 6-4.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 RTL simulation.
cache/	Contains your driver files and data files to test the application. Used for Verilog RTL 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 RTL simulation with the source files in the cache/ directory. Makefiles for running RTL 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> also contains 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 RTL 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 RTL simulation. For more information, see "Directory Structure to Support RTL Simulation" on page 6-7.
simulation/vsim/	Work directory for RTL simulation executables.
simulation/uvm	Contains Makefiles and temporary files used for RTL 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 RTL simulation in VPD format (VCS tool only).
rtl_package/ dump_sv.vcd	It is the waveform dump of the complete RTL 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), re-package the implementation with the new settings and then run `verify -rtl` command.

Directory Structure to Support RTL Simulation

The following section describes the Synphony C Compiler RTL Package directory and file structure under the implementation directory to support RTL simulation from the RTL package. This allows 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 appears in the simulation/<sim_level>/ directory and is copied or linked to the appropriate output directory for the result compare process.

The file names and directories for outputs generated by RTL 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>
```

Corner Case Simulation Support

The Synphony C Compiler RTL 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
hold_max_value	Maximum number of cycles to delay the next stream handshake

Type	What it Implies
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 RTL simulation framework. The monitor assists you with timeout and deadlock detections.

When these conditions are detected, RTL simulation exits with an appropriate message.

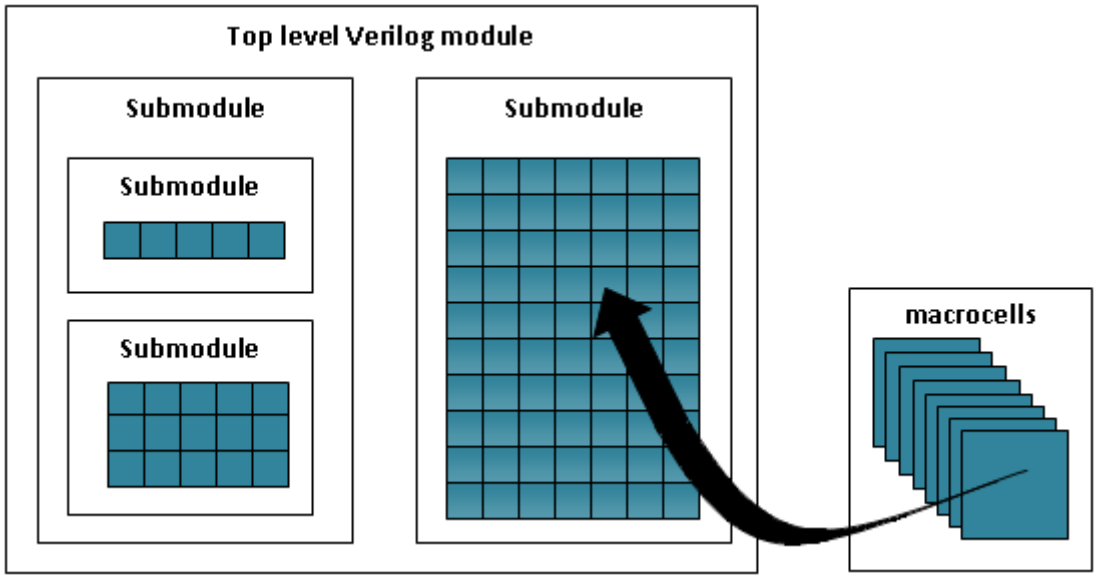
RTL Features

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 might 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 might lead to slightly higher register cost because the tool is not allowed to chain operators as much as it might 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 might be created in the Synphony C Compiler synthesizable RTL, which might 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 might 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 might 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 automatically starts the next task after abort and enable.

Aborting a PPA

The PPA might 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 RTL simulation and perturbation capabilities.

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 might 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 might 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 might 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.

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 stops further execution until a new task is started by the host.

The verification methods 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 might 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.

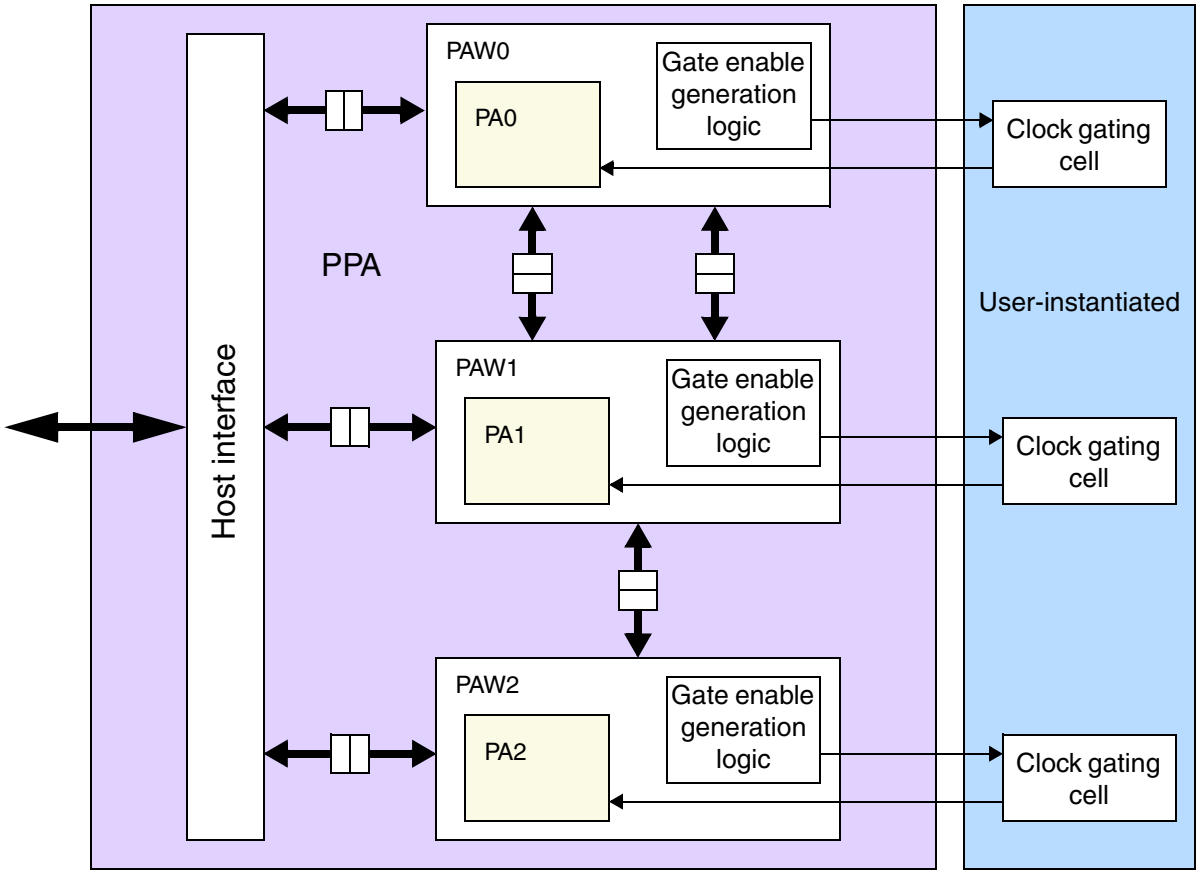
A PA or TCAB is considered “active” when it has a task running on it. PA is additionally considered “active” when it has to register a scalar input. Clock gate enables are asserted whenever the gated block is “active”. Note that, a block is considered “active” even when it is stalling during task execution. 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 might 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 RTL 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`.

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`.

FIFO Implementation (FPGA Only)

You can map stream buffer (FIFO) storage to registers or block RAMs.

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).
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.

Leda Rule	Description
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"
```

7

Building Hierarchical Designs

This chapter describes how to create and use the TCABs (tightly-coupled accelerator blocks) and the importable PPAs to build hierarchical designs.

This chapter contains the following sections:

- Building Hierarchical Designs
- Hierarchical Design Flow
- Building Hierarchical Blocks
- Hierarchical Synthesis Using Importable PPA
- Setting Parameters for TCABs
- TCAB Sharing and Inlining
- Building Hierarchy and Importing Blocks
- Analyzing Hierarchical Blocks

Building Hierarchical Designs

This section describes the building blocks of the hierarchical models and the affect of hierarchy on various optimizations:

- Using Hierarchical Blocks
- Considerations for Hierarchical Design
- Using Deployable TCABs or PPAs for the Top Level

Using Hierarchical Blocks

The Synphony C Compiler tool supports the following structures that you can use to build hierarchical blocks containing multiple levels of hierarchy:

PPA	A pipeline of PAs is a hardware block that contains one or more PAs that is synthesized using a function in the application. A pipeline of PAs can contain multiple level of hierarchy. You can import the pipeline of PAs in any block in the design. A pipeline of PAs can be shared across multiple blocks when they are mutually exclusive. A PA corresponds to a nested loop in the design. A pipeline of PAs can be built as an importable or a deployable pipeline of PAs. For more details about the advantage of using pipeline of PAs as the top-level block, see Using Deployable TCABs or PPAs for the Top Level.

Tightly-coupled accelerator block (TCAB)	A tightly-coupled accelerator block (TCAB) is a shell that contains one processing array synthesized from a function in the application. A processing array usually corresponds to a nested loop in the design. You can use it as a stand-alone module or import it into other hierarchical blocks. A tightly coupled accelerator block can have multiple levels of hierarchy and you can share the block across hierarchical levels, if the procedures involved are mutually exclusive. The higher-level design treats the block as single functional unit with specific input or output. A tightly coupled accelerator block can be built as an importable or a deployable block. For more information about the advantages of using deployable tightly coupled accelerator blocks, see Using Deployable TCABs or PPAs for the Top Level.

Considerations for Hierarchical Design

An alternative to using hierarchy is to inline the function. By default, the tool inlines any function that is not built as a tightly coupled accelerator block or as a pipeline of PAs. Inlining merges the function into the functions that call it, and the inlined function cannot be accessed separately.

Benefits of Using Hierarchy

The following are the benefits of using hierarchical designs.

- Sharing of functions

  The Synphony C Compiler tool supports flexible sharing of blocks across hierarchy of a design. For example, you can create a function for a key functionality, and then create a block (tightly coupled accelerator block or importable pipeline of processing arrays) for the function. You can share the block across hierarchies. The tool can also optimize the sharing of the blocks.

- Clock gating

  In hierarchical designs, you can reduce the power and improve the power optimization by inserting clock gates on the blocks that contain selected parts of the design. Clock gating is particularly effective in hierarchical designs.

- Encapsulation

If you inline functions, your design might end up with a large number of operations that must be scheduled in the same synthesis run. This can cause capacity or runtime issues. Encapsulating operations in TCABs reduces or eliminates this issue, allowing you to create large and complex designs without runtime or capacity issues.

- Fine control of varied constraints

  TCABs or importable PPAs have flexible properties such as variable latency. You can use hierarchy to create hardware blocks with different types of control. For example, you can have regions of the block run at different IIs, or have a parent block stall while running an infrequent child operation.

Limitations of Using Hierarchy

Hierarchical design can limit some optimizations.

- Hierarchy limits hardware sharing
  - Operator sharing
    The SCC tool only shares native C operators like *, /, +, >=, and >> within a given level of hierarchy. The only operations that it shares across hierarchies are TCABs and importable PPAs.
  - Register sharing
    Most registers accessed by a hierarchical level are encapsulated in that level and are not shared with other levels. This means that hierarchy might add to register cost.
  - TCAB sharing
    TCABs and importable PPAs are the operations that the tool shares across hierarchies. A TCAB or an importable PPA is externalized for sharing across hierarchy. If you localize a TCAB or an importable PPA, the TCAB or the importable PPA will not be available for sharing at a higher level.

    For more information, see Sharing Hierarchical Blocks.
- Hierarchy limits compiler optimizations across boundaries

  Inlined functions have no restrictions on the scheduling of the operations in the function. Unlike inlined functions, the schedule for a TCAB or importable PPA is fixed when it is built. The tool schedules operations in a TCAB or importable PPA, independent of the caller's schedule. It then optimizes the caller's schedule around the latency of the imported block as seen by the container. This can prevent some optimizations and lead to higher cost.

Using Deployable TCABs or PPAs for the Top Level

The top-level design can be a deployable TCAB or a PPA. The following table lists the differences between the two types of blocks:

- Interfaces

TCAB	PPA
No memory mapping or host interface. No direct memory ports. Simple control protocol.	You can use the full range of interfaces. You can use full-featured control protocol

- Implementation in the hardware or RTL template

TCAB	PPA
Implemented as a single PA. All code is merged into one loop.	Implemented as multiple and concurrent PAs derived from loop nests and out-of-loop code. This can result in higher performance in some cases.

Hierarchical Design Flow

The Synphony C Compiler hierarchical design flow is based on a bottom-up synthesis approach that has a single project for the entire design and has an implementation for each hierarchical block. The following sections describe the flow:

- Synphony C Compiler Hierarchical Design Flow
- Running the Hierarchical Design Flow

Synphony C Compiler Hierarchical Design Flow

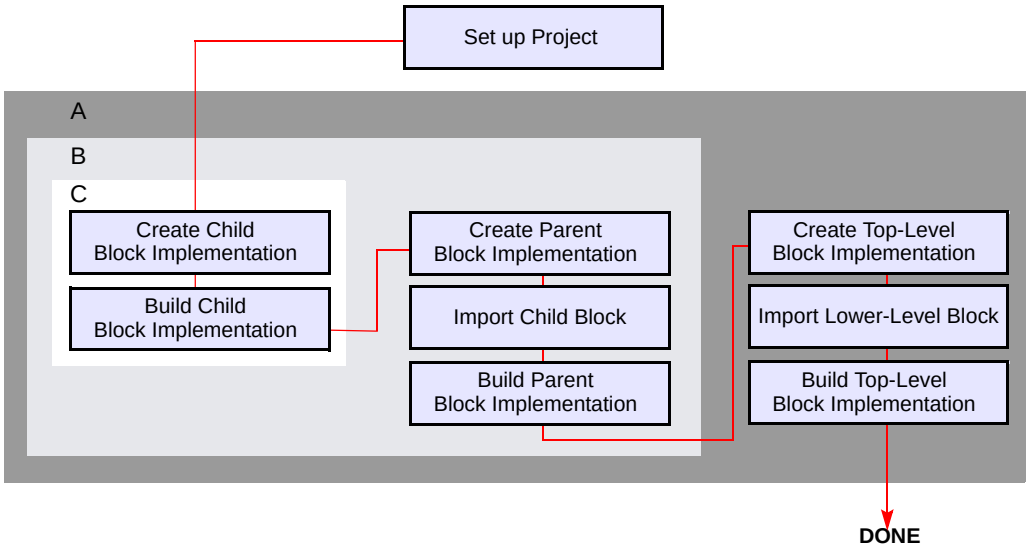
The hierarchical design flow is illustrated in two flow diagrams. The first shows an overview of the bottom-up methodology and does not show any iterations. The second shows how to handle hierarchical iterations necessitated by updates.

- Hierarchical Design Flow With No Iterations
- Hierarchical Design Flow With Iterations

Hierarchical Design Flow With No Iterations

Figure 7-1 provides a high-level view of the ideal hierarchical design flow with no iterations. The red path indicates one straight run through the flow in a three-level hierarchical design, where C is the lowest level, B is a container block, and A is the top level. Note that you must create and compile the child before you can import it into the parent block. Use this basic bottom-up methodology for all hierarchical designs, regardless of the number of levels.

Figure 7-1 High-Level View of A Hierarchical Design Flow



Hierarchical Design Flow With Iterations

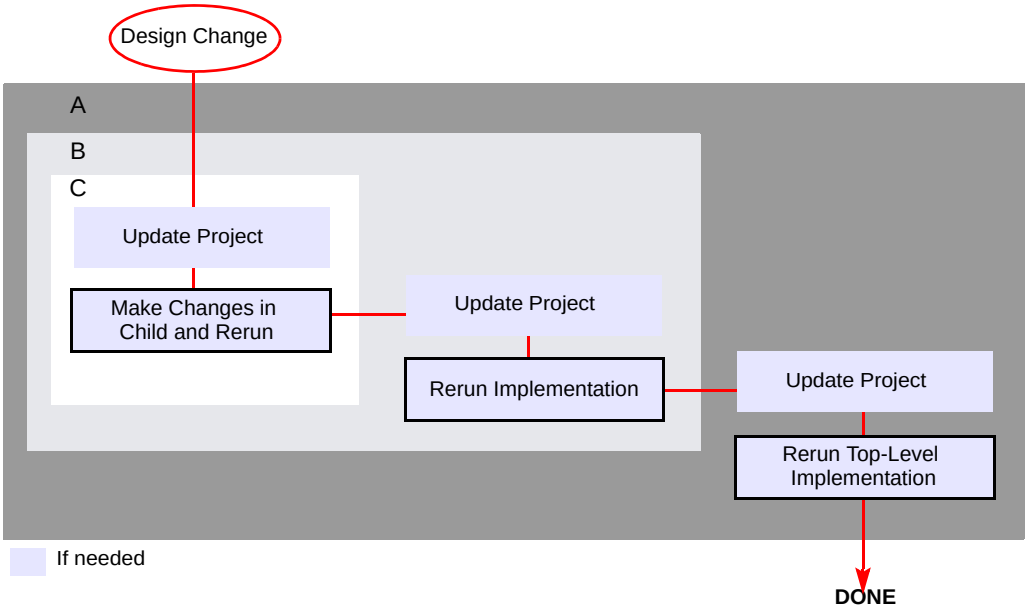
When you need to make changes to a hierarchical level and iterate through the flow, you need to update blocks at other hierarchical levels that are affected by the changes. Changes at one level can affect a child or a parent block. For example, you might have updates to .c and .h files or, you might have modified parameters. Rerun the child block before you resynthesize the next hierarchical level.

Hierarchical Design Flow for Changes at the Child Level

If the change is at the child level, you must resynthesize the block and then update the implementation files for the parent, repeating this process up the hierarchy. It is a bottom-up methodology; therefore, start with the lowest level affected and update and rerun that level before moving up to the next hierarchical level and rerunning it. You might have to update the project if files are added or deleted or if file names change.

Figure 7-2 is a simplified illustration of the design flow. The red arrow marks the path, starting with the changes in the lowest-level block (C) and working through the hierarchical levels above it. This flow shows how to update the parent-level implementation and rerun it before updating the top level.

Figure 7-2 Simplified Design Flow



Hierarchical Design Flow for Changes at the Parent Level

If the change is at the parent level (B) and affects the child (C), you must update the lower level with the changes, resynthesize the lower-level block, and then update and resynthesize the parent level of hierarchy.

Figure 7-3 Iteration Through the Design Flow

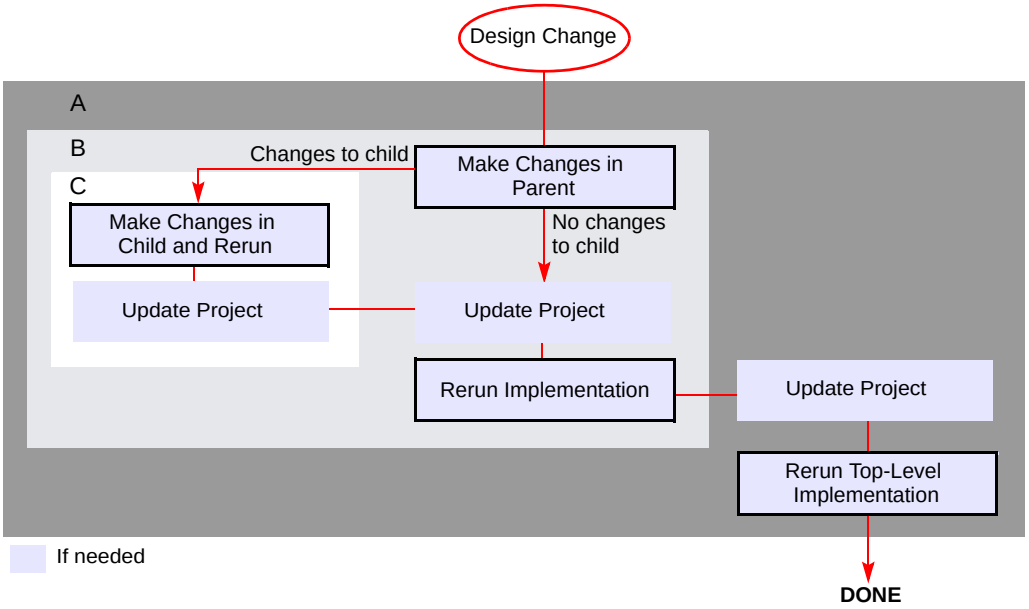


Figure 7-3 is a simplified illustration of an iteration through this design flow. The red arrows mark the path, starting from the change and working through the affected hierarchical levels above and below the parent block. Use this technique on any parent level of hierarchy. You must go down the hierarchy to the lowest level affected and work upwards from there. You must also update the project with all file changes.

Running the Hierarchical Design Flow

Following are the two ways to access the commands to build hierarchical blocks, compile your hierarchical design, and update changes.

- From the GUI

  The GUI provides an interface for you to go through the hierarchical design phases manually. See the rest of this chapter for detailed procedures on using the GUI.

- With a script

  You can run the same commands from a script as from the GUI, but the commands run automatically when you use a script. See Example Script for Hierarchical Design for an example. If you have a large design and need to manage a lot of updates, it might be easier to create and use a script that automatically updates the affected implementations when you make a change.

Example Script for Hierarchical Design

The following example shows the commands you might use to build and use a hierarchical block:

```
# Specify project source and header files
set_params -sources "dct_main.c dct.c streamproc.c"
set_params -headers "dct.h"

# Create TCAB
new_implementation exp1

# Specify TCAB settings
set_params -appfiles dct.c -target dct_transform
set_params -resource_last_use_time 0

# Build the current implementation as TCAB
set_params -type "tcab"

# Build the TCAB
analyze
build

# create top-level implementation
new_implementation exp2
```

```
# Set  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 "exp1"

# Build the top-level hardware
analyze
build
```

Building Hierarchical Blocks

The following sections describe how to set up and configure hierarchical blocks. For information about setting up a hierarchy of blocks, see Building Hierarchy and Importing Blocks.

- Creating and Configuring a Hierarchical Block
- Guidelines for Coding Hierarchical Blocks

Creating and Configuring a Hierarchical Block

Follow these steps to build a hierarchical block.

1. Set up a project and implementation.

   For more details, see Chapter 3, “Setting Up Your Design For Synthesis”.

   a. Select File > New Implementation.

   b. Choose Target Type as PPA or TCAB, based on the type of hierarchical block to be built.

      Alternatively, specify `-type "ppa"` or `"tcab"` with the `set_params` command.

   c. In the Synthesis Target box, specify the function that you want to build as a PPA or TCAB.

      For additional language and syntax guidelines, see Guidelines for Coding Hierarchical Blocks.

      Alternatively, use the following command to specify the target:

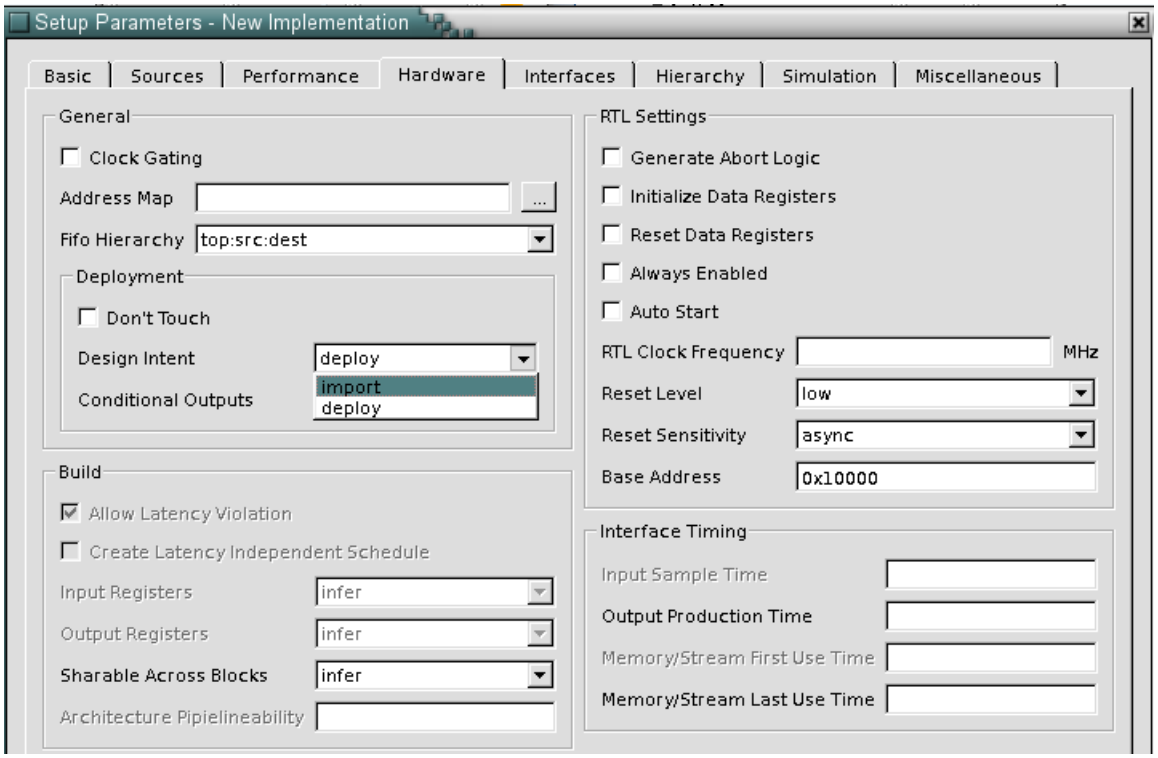
      ```
      set_params -target procedure_name
      ```

   d. Optionally, set the other parameters using the Set Other Parameters button.

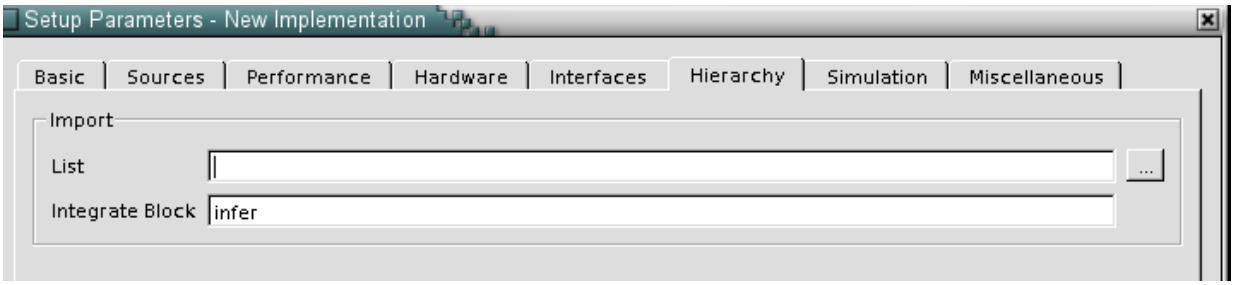
- To build an IPPA, choose import option as the Design Intent, in the the Hardware tab as shown in Figure 7-4.

  Alternatively, use the `set_params -design_intent import` command.

Figure 7-4 Specifying Design Intent for IPPA



- In the List box of the Hierarchy tab, specify the name of the implementations of the hierarchical blocks to import them into the implementation. When you specify multiple names, separate the names by spaces.



Alternatively use the `set_params -import` command.

For more information about importing hierarchical blocks, see Building Hierarchy and Importing Blocks.

2. Run the implementation.

   This builds the implementation for the hierarchical block.

Guidelines for Coding Hierarchical Blocks

The following guidelines apply to both TCAB and PPA procedures, and they use the same syntax. The only difference between the two is in the implementation of multiple loop nests.

- Your code can include the following:
  - Acyclic code or a single nested loop (PPAs and TCABs)
  - A mix of acyclic code and a single nested loop (PPAs and TCABs)
  - Multiple loop nests
    For multiple loop nests, the Synphony C Compiler tool generates TCAB hardware for the final merged loop nest. For a PPA, loop nests are implemented as multiple PAs.
- You can pass parameters to a block.
  The tool implements scalar parameters as input ports and array parameters as memory ports that are connected to the corresponding memory for the array being passed. You cannot pass slices of an array; you must pass the base address of the array.
- The procedure can return scalar values.
  The tool does not support array and pointer return values, but it can return scalar values in the return statement. It returns multiple values through reference parameters, but the procedure must not access the reference parameter beyond the 0 offset.
- You can preserve memory ports with the `preserve_array` pragma.
  The Synphony C Compiler tool might scalarize and eliminate a memory port if the access to the element of the array parameter are done using constant indexes. Use the `preserve_array` pragma if you want to preserve the memory port.
- Do not alias procedure arrays and reference parameters with each other or with a global variable if you are writing any of these variables inside the procedure.
- Specify the width of the memory address port by including the array dimensions in the parameter declaration. The number of elements determines the memory port width. By default, the tool creates memory ports with 32-bit address ports.

  For multiple arrays of different sizes that are passed to different calls, use the `-memory_size` argument:

  ```
  set_params -memory_size "array:arraySize"
  ```

  The `size` parameter is only used for simulation and the tool does not generate hardware for it. Note that you are responsible for ensuring that the procedure does not access the array beyond the specified size.

Hierarchical Synthesis Using Importable PPA

You can create hierarchy using TCAB or importable PPA. A TCAB can have only one PA, where as an importable PPA can have one or more PAs. In some designs, the container needs to know the exact time that a resource is acquired and released by a hierarchical block. In such cases, you can create a shared stall domain TCAB. Generally, you should write a straight-line code that is encapsulated into a hierarchy for sharing.

When the container does not need to know the exact time the resource is acquired and released by the hierarchical block, you can create importable PPAs. The importable PPA feature allows you to build your design hierarchically using PPAs as building blocks. Using PPAs as building blocks has the advantages of multiple loops, automatic multibuffering, and good hierarchical timing insulation. The PPAs communicate using flow-controlled FIFOs, which can be insulated with register delays for better timing between layers of hierarchy. Importable PPAs have few exported properties; therefore, changes to a subblock are less likely to affect the higher-level blocks.

Building a PPA

Use the following Tcl command to specify the block to be built as a PPA:

```
set_params -type ppa|channel|tcab -target target_name
```

If you do not specify the `-type` option, by default, the tool builds the hierarchical block as a PPA.

The *target_name* is the name of the method that you specified, that must be built as a hierarchical block.

A PPA is considered to be importable if it can be imported into another implementation. A PPA is considered to be deployable if it is the top level of the Synphony C Compiler (SCC) design, that is, not imported into another implementation.

Use the following Tcl command to build an importable or a deployable PPA:

```
set_params -design_intent import|deploy
```

When you specify the `-design_intent import` option, the tool builds an importable PPA that can be imported into another implementation.

When you specify the `-design_intent deploy` option, the tool builds a deployable PPA that cannot be imported into another implementation. When you do not specify the `-design_intent` option, by default, the tool builds a deployable PPA.

Importing a PPA

An importable PPA can be imported into another importable PPA, deployable PPA, TCAB, or a CHANNEL. Use the `-import` option to specify the imported implementations as shown in the following example:

```
set_params import "imp1 imp2"
```

where imp1, imp2 are importable PPAs, TCABs, or CHANNELs.

Note:
A CHANNEL can be imported only into a deployable PPA.

Specifying Performance Constraints for Importable PPA

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 or the imported PPA, you might need to specify constraints. These constraints are related to importing a PPA into another implementation and are specified when building or importing the importable PPA. Constraints that are specified at build time are treated as defaults and can be overridden at import time. These constraints do not affect 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. For PPAs these parameters are not part of the hardware; the parameters only control the latency. Therefore, 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 when building the PPA or when 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.

You can specify the parameters that control the latency of the PPA either while building the PPA or while importing the PPA.

To specify the latency while building an importable PPA, use the latency parameters as shown in the following examples:

```
set_params -resource_last_use_time value
set_params -output_production_time value
```

To specify the latency when importing the PPA, specify the `-import` option and the latency parameter option as shown in the following examples:

```
set_params -resource_last_use_time import value
set_params -output_production_time import value
```

The constraints that you specify apply to all the importable PPAs in the container. You can also specify the constraints on specific importable PPA.

To apply the constraints on specific importable PPAs, specify the latency parameters as shown in the following examples:

```
set_params -resource_last_use_time import "ppa_target:value"
set_params -output_production_time import "ppa_target:value"
```

The *ppa_target* is the name of the target that you specified while building the importable PPA using the `-target` option of the `set_params` command.

If you import multiple instances of a PPA, the latency parameter that you specified is applied to all the instances of the PPA.

Advantages of Using Importable PPAs

There are three main advantages of using importable PPAs:

1. Composition

- Composition of multiple PPAs
- The parent and the child PPAs have the same task size

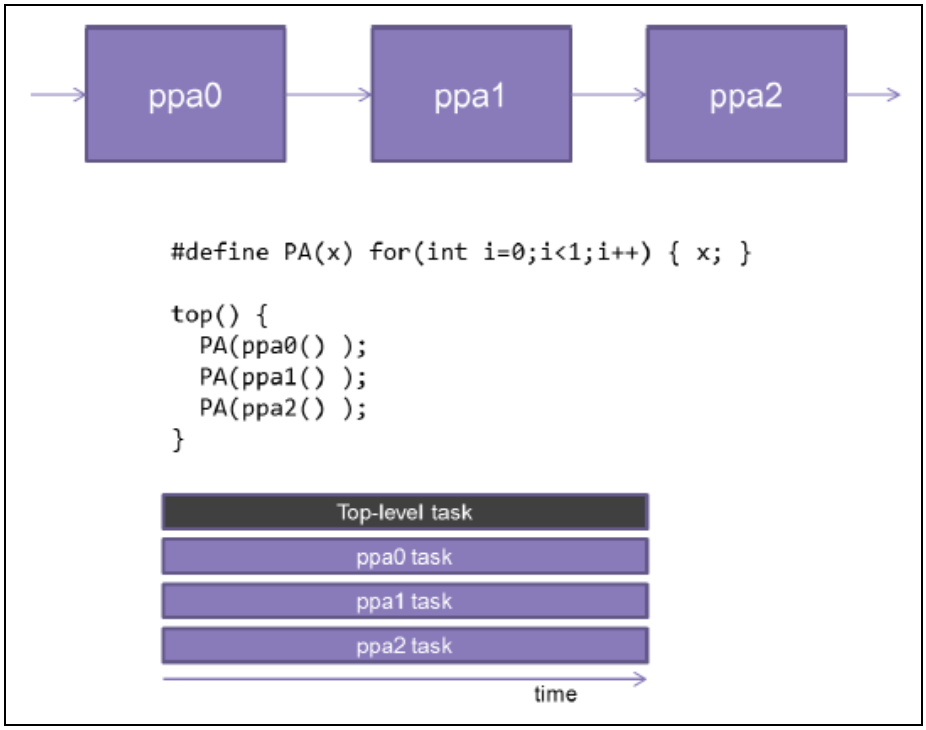
2. Parallelization
   - Outer loop around one or more PPA calls
   - PPAs run in parallel without task overlap
3. Overlap
   - Outer loop around one PPA call
   - PPA runs with task overlap
   - Internally, PPAs use multibuffered communication to enable task overlap

Composition

The composition PPAs are suitable for building large systems with multiple levels of hierarchy for better modularity and runtime. You can build flexible systems that launch the PPAs in different orders or combinations.

Figure 7-5 has a container PPA top that has three PPAs: ppa0, ppa1, and ppa2. Each of these PPAs have the same task size, and they pass the data to other PPAs using streams. Because the three PPAs are inside a separate processing array (PA), it is convenient and does not require using a scheduling constraint and so on. Also, the tool automatically determines if the behavior is sequential or parallel. If the three PPAs are inside the same PA, then it might be required to use scheduling constraints as outlined in the Parallel PPAs.

Figure 7-5 Example of a Composition PPA



To build an importable PPA, use the following command:

```
set_params -design_intent import
```

Optionally, you can use the `-output_production_time` and `-resource_last_use_time` options as shown in the following example:

```
set_params -output_production_time import 1
set_params -resource_last_use_time import 0
```

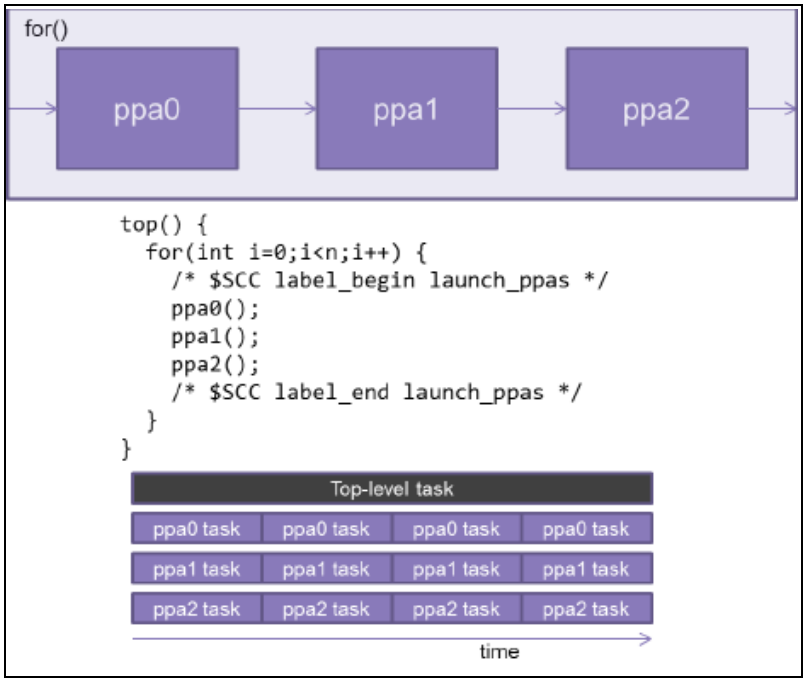
Parallel PPAs

The parallel PPAs are suitable for building parallel systems that iterate over complex blocks such as a video codec.

In Figure 7-6, the importable PPAs ppa0, ppa1, and ppa2 use streams for communication and are called in an outer loop in the container PA. The three PPAs run in parallel and must be launched at the same time in the container PA. Specify this scheduling constraint while building the container implementation as shown in the following example:

```
set_params -wait launch_ppas:launch_ppas:0
```

Figure 7-6 Example of a Parallel PPA



Optionally, you can specify performance constraints that are used while importing the PPA as shown in the following example:

```
set_params -output_production_time 1
```

When you specify this constraint, the tool schedules the done handshake of the importable PPAs in the cycle after the start handshake, and the container waits until the done handshake. The three PPAs finish in the same cycle in the container and do not require task overlap.

If you do not require task overlap, specify the following constraint while building the importable PPA:

```
set_params -task_overlap never
```

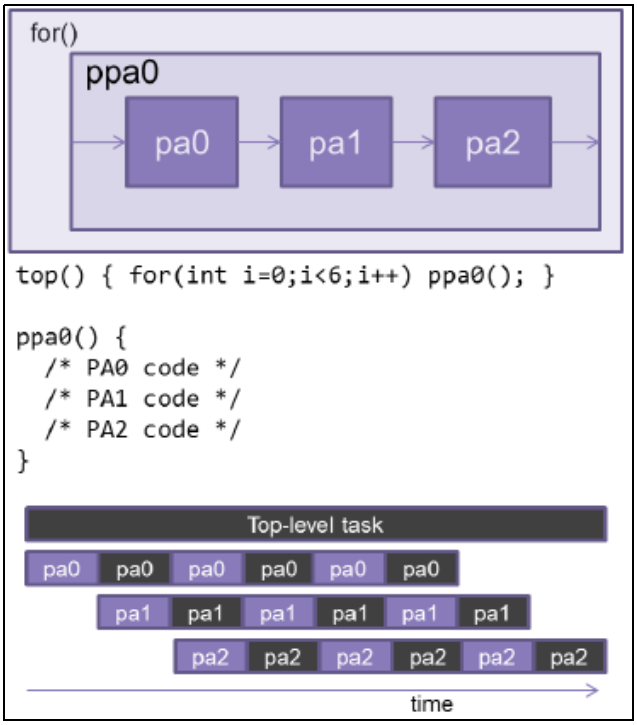
PPAs With Task Overlap

This type of PPAs are suitable for managing multibuffering and task overlap with block-based communication between PAs.

Note:
The tool supports full pipelining in imported PPAs.

In Figure 7-7, the importable PPA ppa0 is called in an outer loop inside the container PA (inside the for loop). The importable PPA is required to have task overlap and has automatic multibuffering.

Figure 7-7 Example of a Task Overlap PPA



You can specify these constraints while building the importable PPA (ppa0) as shown in the following example:

```
set_params -task_overlap always
```

To specify multiple buffers for the shared memories between the PAs, use the `multi_buffer` pragma. For continuous processing in the importable PPA, specify `set_params -continuous_processing always` command.

For the PPA in Figure 7-7, to get the desired task overlap, specify the following options while building the container implementation that imports ppa0.

```
set_params -resource_last_use_time import 3 ( set the desired value)
set_params -ii 1
```

When you specify these constraints, the tool schedules the done handshake 3 cycles after start handshake. Therefore, the container can start 3 tasks before the done handshake for the first task.

Views and Reports

In the GUI and reports, a parent implementation shows the `ippaw` string to indicate an importable PPA. The GUI and text reports show importable PPAs similar to TCABs. An importable PPA is not shown in a PPA graph or a sequentialization graph, unlike the TCABs. You can use the other GUI views for analysis of the importable PPAs.

The report_summary.txt file has additional information for an importable PPA. The report shows "Imported PPA Constraints" and "Imported Blocks" as shown in Example 7-1.

Example 7-1 Report Generated for an Importable PPA

```
Imported PPA Constraints:
+-------------------+----------------------+-----------------------+

| Entity Name        Output Production Time  Resource Last Use Time|

+-------------------+----------------------+-----------------------+

  datapath_ippaw             5                        4


Imported Blocks:
+-------------------+--------------------+-----------+------------+

| Name                 Entity Name         Instances   Block Type |

+-------------------+--------------------+-----------+------------+

  datapath             datapath_ippaw          1           PPA
```

Other reports display similar information for PPA and TCAB. However the the name contains the `ippaw` string appended to the name to distinguish it as an importable PPA.

Features of Importable PPA

This section describes the important features of an importable PPA. An importable PPA supports all the features of a PPA except for the limitations mentioned in Limitations of an Importable PPA.

- Multiple calls to an importable PPA can share the same hardware if the dependences and performance requirements permit and if the calls are in the implementation of the container.
- A PPA can be imported into a TCAB.
- A PPA can be imported into a CHANNEL.

- Calls to an imported PPA with an array parameter can be shared in most cases. However, when the calls pass different arrays and the imported PPA has task overlap, the sharing is not permitted.
- Automatic multibuffering of arrays that are not passed to imported PPAs is supported. Automatic multibuffering of arrays passed to imported PPAs is supported in some cases. For more information, see Limitations of an Importable PPA.
- The memory must be in the focal scope for multibuffering when you use the `multi_buffer` pragma. The tool does not support multibuffering of global memory even if you specify `localize_access` pragma.
- Multibuffering applies to arrays that are written in one PA and read in another. To enable multibuffering across two calls to an imported PPA, the calls must be in separate loops
- The `constant_across_pipelined_calls` (CAPC) pragma specification on an input of an imported TCAB requires that the input not change values while the block is active. The tool issues an error message if this condition cannot be met.

  The `constant_across_pipelined_calls` pragma specified on an input of an imported PPA is treated as a constraint in the container. The tool schedules the container such that the CAPC specification is met. If the specification cannot be met at the target container iteration interval, the tool relaxes the initiation interval constraint and issues an error.

  When an imported PPA with a task overlap is running, it is possible that the value of the CAPC input changes in the container without violating the CAPC specification. One supported case is where a CAPC variable is written outside of a loop that has a `flush` pragma.

  ```
  // task-overlap of ppa_B is > 0
  int ppa_B(int value, …) {
  #pragma constant_across_pipelined_calls value
  ...
  }

  void sppa() {
    int x;
    for(int i =0; i < N; i++) {
     x = …
      #pragma flush
      for (int j=0; j < M; j++) {
       … = ppa_B(x, …); // x is probably CAPC due to flush
      }
    }
  ```

  In this example, the variable x is updated outside the loop that has a `flush` pragma specified. The tool can prove that the value of x is stable across pipelined calls to ppa_B().
- You can use the `set_params -wait` constraints to control the scheduling of PPAs inside a loop.

Example of an Importable PPA

This section describes an example of an importable PPA that is an overlap type of a PPA. In this example, the importable PPA processes one block and is called in a container that processes one frame consisting of many blocks.

In Example 7-2, the function process_one_block() processes one block. The tool builds an importable PPA because the `-design_intent import` option is specified with the `set_params` command.

Example 7-2 Task Overlap PPA - Source File and Tcl Script for Implementation

```
Source file for the PPA (ppa.c)
#include "scc.h"
#define NUM_BLOCKS 4
#define MAX_SIZE 16

int scc_stream_input_get(void) {
static int i=0;
return i++;
}

void scc_stream_output_put(int d) {
static int i=0;
printf("put[%i] = %i\n",i++,d);
}

void block1(char size,int m_out[MAX_SIZE]) {
int i;
for(i=0;i<size;i++) {
  int input = scc_stream_input_get();
  m_out[i] = input;
  }
}
void block2(char size,int m_in[MAX_SIZE]), int m_out[MAX_SIZE]) {
int i;
for(i=0;i<size;i++) {
  int input = m_in[size-i-1];
  m_out[i] = input;
  }
}
void block3(char size, int m_in[MAX_SIZE]) {

int i;
for(i=0;i<size;i++) {
int input = m_in[size-i-1];
scc_stream_output_put(input);
}
}
void datapath(char size) {
int m1_2[MAX_SIZE];
int m2_3[MAX_SIZE];
```

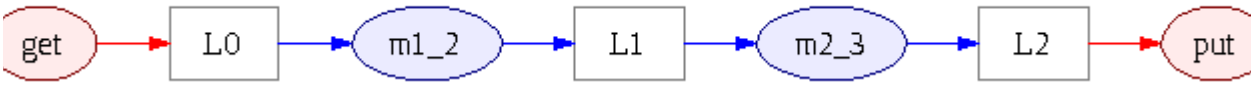
```
#pragma multi_buffer m1_2 2
#pragma multi_buffer m2_3 2
#pragma user_supplied m1_2
#pragma user_supplied m2_3

int i;
block1(size,m1_2);
block2(size,m1_2,m2_3);
block3(size,m2_3);
}

// Process one block
void process_one_block(char size) {
datapath(size);
}
void process_one_frame() {
int i;
const char list[NUM_BLOCKS] = { 16, 16, 16, 16 };
for(i=0;i<NUM_BLOCKS;i++) {
char size = list[i];
process_one_block(size);
}
}
int main(void) {
process_one_frame();
return 0;
}
```

```
#Tcl script to create importable PPA and deployable PPA

set project my_project.scc
if { [file exists ${project}] } {
    open_project ${project}
    reset_defaults
} else {
    new_project -w "." ${project}
}
set_defaults -sources "ppa.c"

set impl process_one_block
if [file exists $impl] { delete_implementation $impl }
new_implementation $impl
set_params -appfiles "ppa.c"
set_params -target process_one_block
set_params -type ppa
set_params -clock_freq 200
set_params -task_overlap always
set_params -design_intent import

set_params -ii 1
verify -golden
```

```
analyze
verify -lint
build
verify -rtl

set impl process_one_frame
if [file exists $impl] { delete_implementation $impl }
new_implementation $impl
set_params -appfiles all
set_params -target process_one_frame
set_params -type ppa
set_params -clock_freq 200
set_params -ii 1
set_params -import "process_one_block"
set_params -resource_last_use_time import 3
verify -golden
analyze
verify -lint
build
verify -rtl -dump_vpd
```

The PPA has three loops and uses multibuffering technique for task overlap. The PPA graph of process_one_block() is shown in Figure 7-8.

Figure 7-8 PPA Graph of process_one_block()

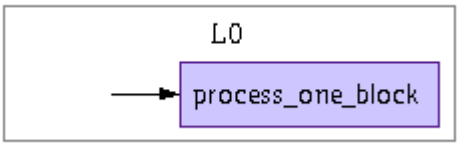


The process_one_frame() is the container function that is built as a deployable PPA and imports process_one_block(). The hierarchy graph of this PPA is shown in Figure 7-9. The process_one_frame() calls process_one_block() in a loop until all the blocks are processed. To have three tasks in parallel, specify

```
set_params -resource_last_use_time import 3
```

The tool schedules the done handshake of process_one_block PPA 3 cycles after the start handshake. This also enables the tool to start 3 tasks before the done handshake of the first task.

Figure 7-9 Hierarchy Graph of process_one_frame()



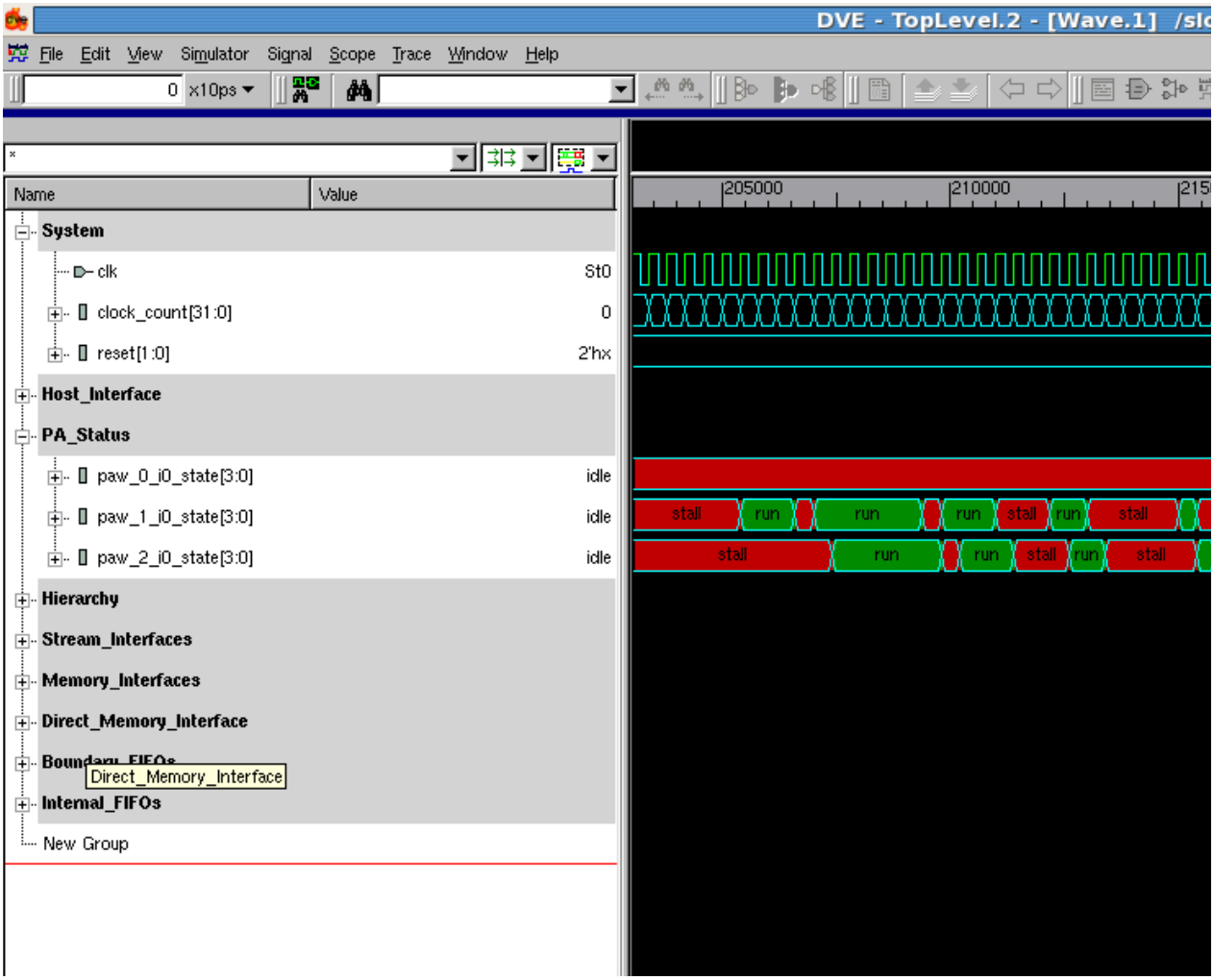
You can view the performance of the design using the Discovery Visualization Environment (DVE), which is part of the Synopsys VCS tool.

Figure 7-10 shows the performance of the design using RTL monitor.

Use the following commands to view the performance in DVE.

```
synphonycc:[doc_ex:dppa]> verify -rtl -dump_vpd
synphonycc:[doc_ex:dppa]> view_waveform
```

Figure 7-10 Viewing the Design Performance Using DVE



Limitations of an Importable PPA

This section lists the limitations of an importable PPA.

1. You cannot import a channel into an importable PPA. Channels can be imported only into a deployable PPA.

2. If a scalar feedback path exists from the output of a PPA call to the input of the same call, the tool does not support task overlap for such a PPA. For example, if you have a PPA call of the form v=ppa(v), there can be no task overlap because the result must be acquired before issuing a new call.
3. When an importable PPA that uses stream or memory resource is shared in the container PA, the container might not run at the desired iteration interval (ii). The tool does not support task overlap of such PPAs.
4. If two importable PPAs use shared memory where one PPA writes the data to the memory and the other reads (or writes) the same memory, the second PPA can acquire the shared memory only at the task boundary of the first PPA, that is, at the done handshake of the first PPA.
5. Multibuffering of arrays passed to the imported PPAs has the following limitation. Consider for example, you are building a PPA that imports another PPA. If the array to be multibuffered is passed to the imported PPA call, certain properties must hold for it to be multibuffered. Either the imported PPA should have no task overlap, or you must not specify continuous_processing for the PA that calls the imported PPA. Otherwise, the array cannot be multibuffered.
6. Use streams for communication between importable PPAs at the same level of hierarchy. If a stream that is meant to be internal to an importable PPA and is declared in global scope, the tool does not check if the PPA is also used as an external stream by another PPA. The tool does not support sharing of streams between different PPAs.

Setting Parameters for TCABs

The following sections describe how to access and set TCAB parameters:

- Specifying Blocks to be Built as TCABs
- Specifying Combinational TCABs
- Setting Pipelining Parameters for TCABs
- Specifying Stall Domains for Hierarchical Blocks
- Specifying External Delays for Hierarchical Blocks
- Controlling the Inference of Boundary Registers
- Specifying Latency for Hierarchical Blocks
- Gating Clocks on Hierarchical Blocks
- Sharing Hierarchical Blocks

For information about setting other parameters , see Setting Parameters. For descriptions of the options, see the *Synphony C Compiler Reference Manual*, especially the descriptions of advanced TCAB parameter settings.

Specifying Blocks to be Built as TCABs

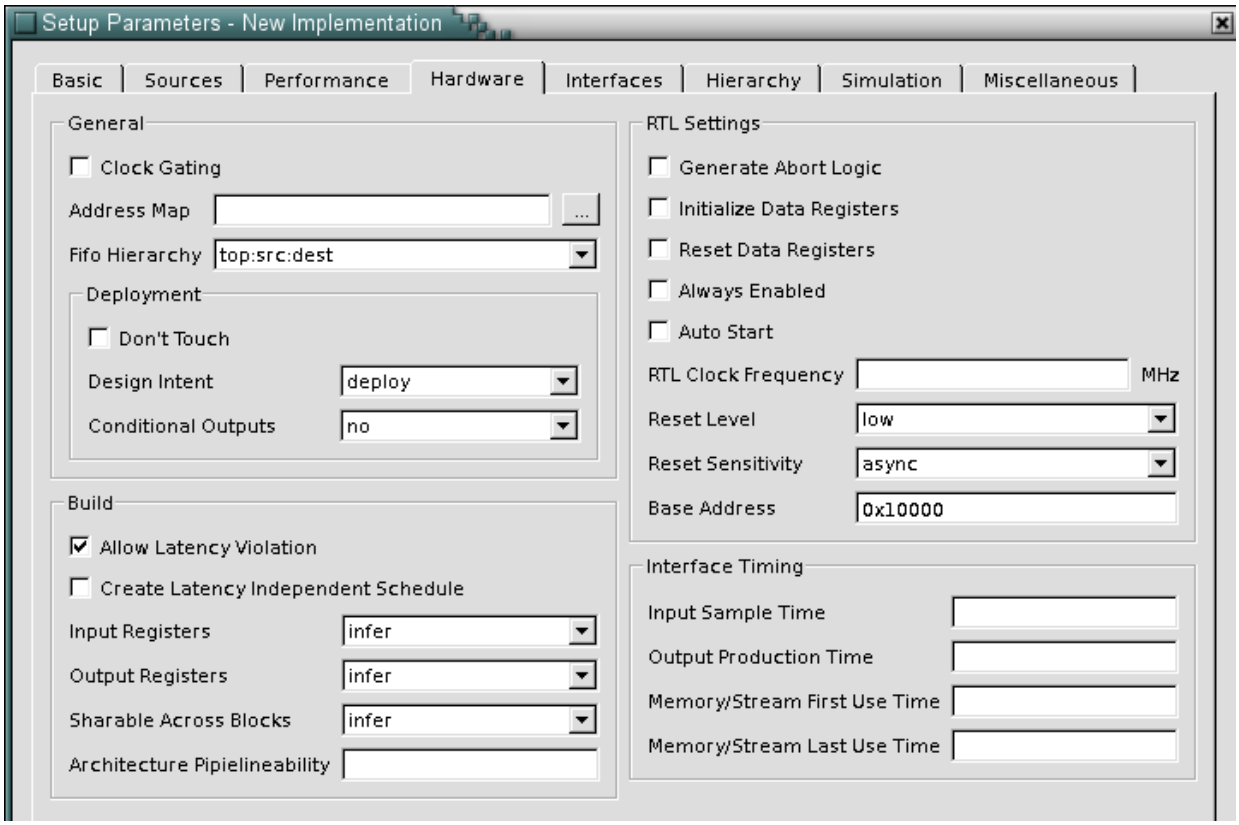
You can set parameters for TCABs in the GUI or using the `set_params` command from the command line. To access the hierarchical block parameters in the GUI, do the following:

1. Select File > New Implementation.
   - To build a TCAB, choose TCAB in the Target Type drop-dowb box.
   - Specify the implementations to be imported into this block in List box, in the Hierarchy tab of the Setup Parameters dialog box.

     Specify only the immediate child blocks to be imported; you do not have to specify all the blocks further down the hierarchy.
   - Use the Hardware tab of the Setup Parameters dialog box to specify other parameters of the TCAB as shown in

     See Setting Parameters for TCABs for various parameters you can set for hierarchical blocks. For information about setting other parameters , see Setting Parameters.

Figure 7-11 Hardware Settings for TCAB



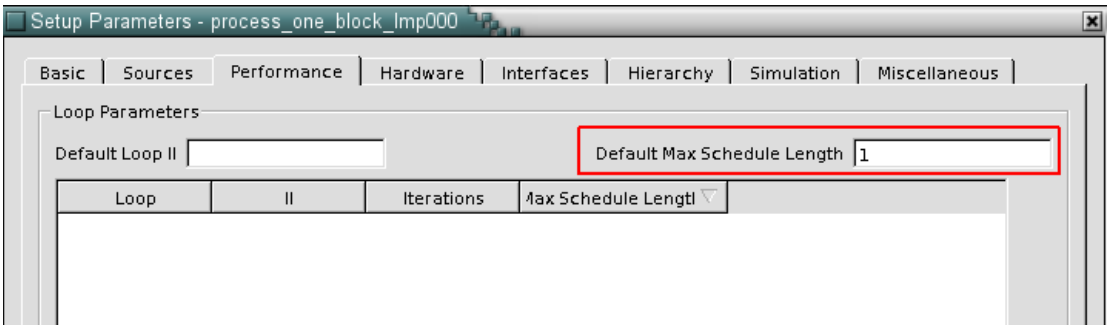
Specifying Combinational TCABs

A combinational TCAB does not have sequential elements between the input and output ports and produces output in the same cycle as the input. If you do not specify a combinational TCAB, the tool implements a sequential one. Unlike combinational TCABs, a sequential TCAB has at least one register element in the paths from input to output. Typically you use sequential blocks for more complex operations. The following procedure shows you how to specify combinational TCABs.

1. Make sure the procedure has acyclic code or loops that can be unrolled.
2. Specify the block as a TCAB, as described in Specifying Blocks to be Built as TCABs.
3. In the Performance tab of the Setup Parameters dialog box, specify a value 1 in the Default Max Schedule Lengthbox, as shown in Figure 7-12.

   This setting prevents II-based sharing.

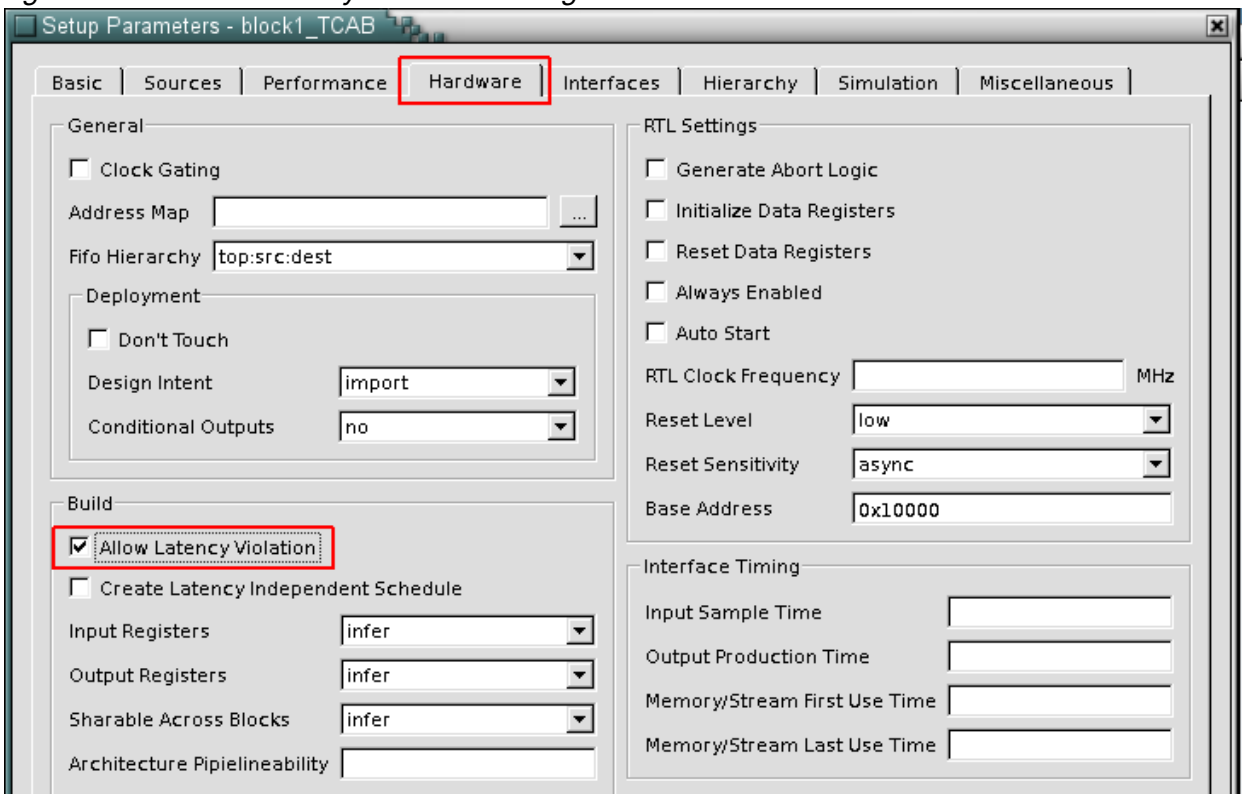
Figure 7-12 Specifying the Maximum Schedule Length



   Alternatively use the `set_params -max_schedule_length 1` command.

4. In the Hardware tab of the Setup Parameters dialog box, ensure that the Allow Latency Violation is turned off, as shown in Figure 7-13.

Figure 7-13 Allow Latency Violation Settings



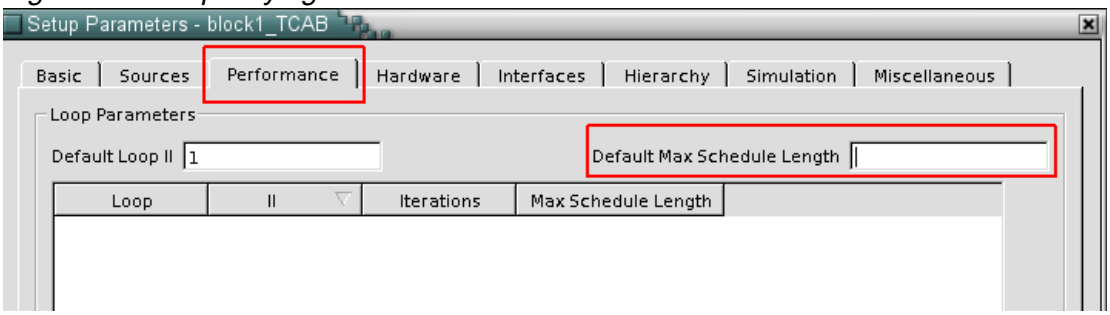
Setting Pipelining Parameters for TCABs

The pipelinability of a TCAB is the minimum number of cycles between successive calls to the TCAB function from a higher-level block. A new task can start only when the previous task has finished input sampling and is draining the module pipeline. You cannot pipeline a TCAB if it uses both the sampling input and flushing pipeline phases, because the next task cannot start until the current TCAB task completes. The pipelinability of a TCAB also affects the II for its container.

You can control the pipelinability of hierarchical blocks as described:

1. Specify the block as a TCAB, as described in Specifying Blocks to be Built as TCABs.
2. Set initiation interval values.
   - In the Performance tab of the Setup Parameter dialog box, set the required value in the Default Loop II box. Alternatively, use the `set_params -ii value` command.

Figure 7-14 Specifying the Initiation Interval Values



3. Specify the architectural pipelinability of the TCAB on the TCAB tab.

GUI (TCAB tab)	Specify the number of cycles in Architectural Pipelinability.
Tcl	`set_params -architectural_pipelinability` *`cycles`*

This sets the minimum number of cycles required between successive invocations of the hierarchical block. The tool generates a shared stall domain block that satisfies this requirement. If it cannot meet the requirement with a shared stall domain TCAB, it creates an independent stall domain TCAB.

4. If you do not want the tool to generate independent stall domain TCABs if it cannot meet the architectural pipelinability requirement, set the following:

GUI (TCAB tab)	Enable Allow Latency Violation.
Tcl	`set_params -allow_latency_violation no`

This command causes the tool to generate an error message when it cannot create shared stall domain TCABs to meet the specified requirement.

5. Set input latencies by specifying positive values greater than one for scalar inputs and memory and stream inputs.
   - From the GUI:

Scalar input	On the TCAB tab, set a value 0 or infer in Input Sample Time.
Memory and stream input	Set a value 0 or infer in Memory/Stream First use Time.

- From the command line:

Scalar input	`set_params -input_sample_time value`
Memory and stream input	`set_params -resource_first_use_time value`

Specifying that the inputs can be sampled after the hierarchical block issue increases pipelinability. See Specifying Latency for Hierarchical Blocks for details.

Specifying Stall Domains for Hierarchical Blocks

A stall domain consists of a set of hardware elements that execute in lock-step with each other and which stall at the same time. A TCAB can operate in a stall domain that it shares with its caller, or in an independent stall domain where it is separate from the domain of the caller.

See Specifying Independent Stall Domain Blocks and Specifying Shared Stall Domain Blocks for more information about setting up these blocks.

Specifying Independent Stall Domain Blocks

If you set up your design as described in Specifying Shared Stall Domain Blocks, the tool creates shared stall domain TCABs by default. You can also specify independent stall domain blocks where the execution of the TCAB is decoupled from that of its caller, potentially leading to improved performance. The following procedure shows how to create independent stall TCABs.

1. Specify the block as a TCAB, as described in Specifying Blocks to be Built as TCABs.
2. Set the number of cycles for output production to a value greater than 0 either in the GUI or from the command line.

GUI (TCAB tab)	For scalar outputs, set Output Production Time to a value greater than 0 (globally or per port). For memory and stream outputs, set Resource Last Use Time to a value greater than 1 (globally or per port).
Command Line	`set_params -output_production_time cycles` `set_params -resource_last_use_time cycles`

If there is a violation, the tool automatically creates an independent stall domain TCAB that issues stalls to the container at the specified output latency and holds the stall until the achieved latency.

3. To force the tool to create independent stall domain TCABs, set this parameter from the GUI or the command line:

GUI (TCAB tab)	Enable Create Independent Stall Domain.
Command Line	`set_params -force_independent_stalldomain_tcab yes`

The tool creates independent stall domain TCABs even if there are no latency violations.

4. If you want to avoid stalls, set the following parameter:

GUI (TCAB tab)	Enable Set Latency Independent Schedule.
Command Line	`set_params -latency_independent_schedule yes`

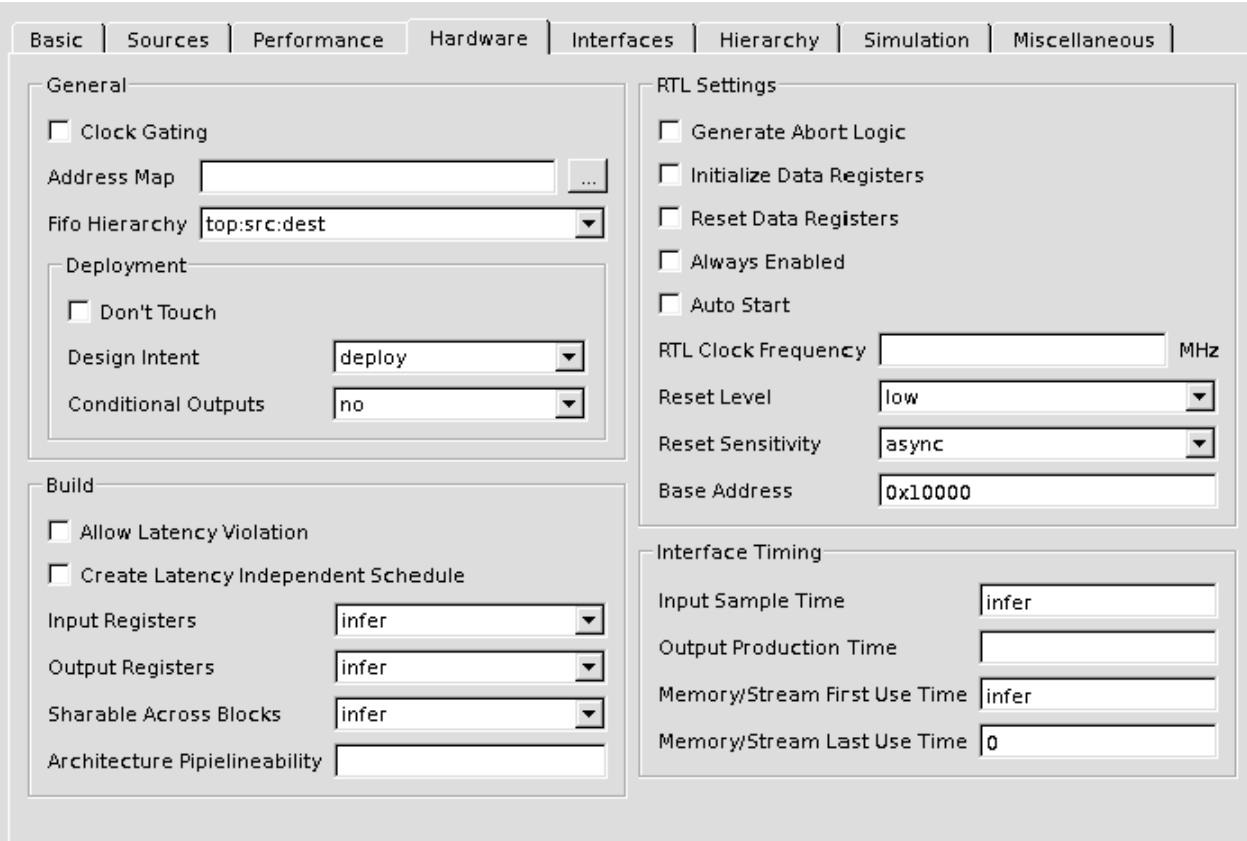
This specifies that the scheduling take the `-output_production_time` or `-resource_last_use_time` into account. Conversely, if you want to create cheaper hardware, you can set this parameter to no.

Specifying Shared Stall Domain Blocks

By default, the Synphony C Compiler tool creates shared stall domain TCABs, if you have the settings described:

1. Make sure your code follows these guidelines:
   - Any loops in the code must be for() loops with fixed bounds.
   - The execution of any loop must be unconditional, without if-around-for scenarios.
2. Specify the block as a TCAB, as described in Specifying Blocks to be Built as TCABs.
3. Check that Create Independent Stall Domain on the TCAB tab is disabled.

Figure 7-15 Specifying Shared Stall Domain



4. Deselect the Allow Latency Violation box on the hardware tab (`set_params -allow_latency_violation no`) if you specified values for the following parameters:

GUI (TCAB Tab)	`set_params` Option
Architectural Pipelinability	`-architectural_pipelinability`
Output Production Time	`-output_production_time`
Memory/Stream Last Use Time	`-resource_last_use_time`

5. If you want to avoid stalls, set the following parameter:

GUI (TCAB tab	Enable Set Latency Independent Schedule.
Command Line	`set_params -latency_independent_schedule yes`

This specifies that the scheduling take `-output_production_time` or `-resource_last_use_time` into account. Conversely, if you want to create cheaper hardware, you can set this parameter to no.

Specifying External Delays for Hierarchical Blocks

You can set timing parameters for the block in the same way as the top level. For a hierarchical block, you can also specify input and output delays to and from the block, as described in the following table. These delays are not the target frequency, which you set in the Configure Implementation dialog box, but reflect the effective timing budget available for the TCAB.

1. Set delays in ns on the block inputs and outputs:

GUI (TCAB tab)	Set values for Scalar Input Delay or Scalar Output Delay. You can set a global or per port value.
Command Line	`set_params -scalar_input_delay` `set_params -scalar_output_delay`

When you set delays globally or on individual inputs and outputs, the tool honors these constraints. If the constraint can only be satisfied by registering the input or output, the tool infers a boundary register.

Controlling the Inference of Boundary Registers

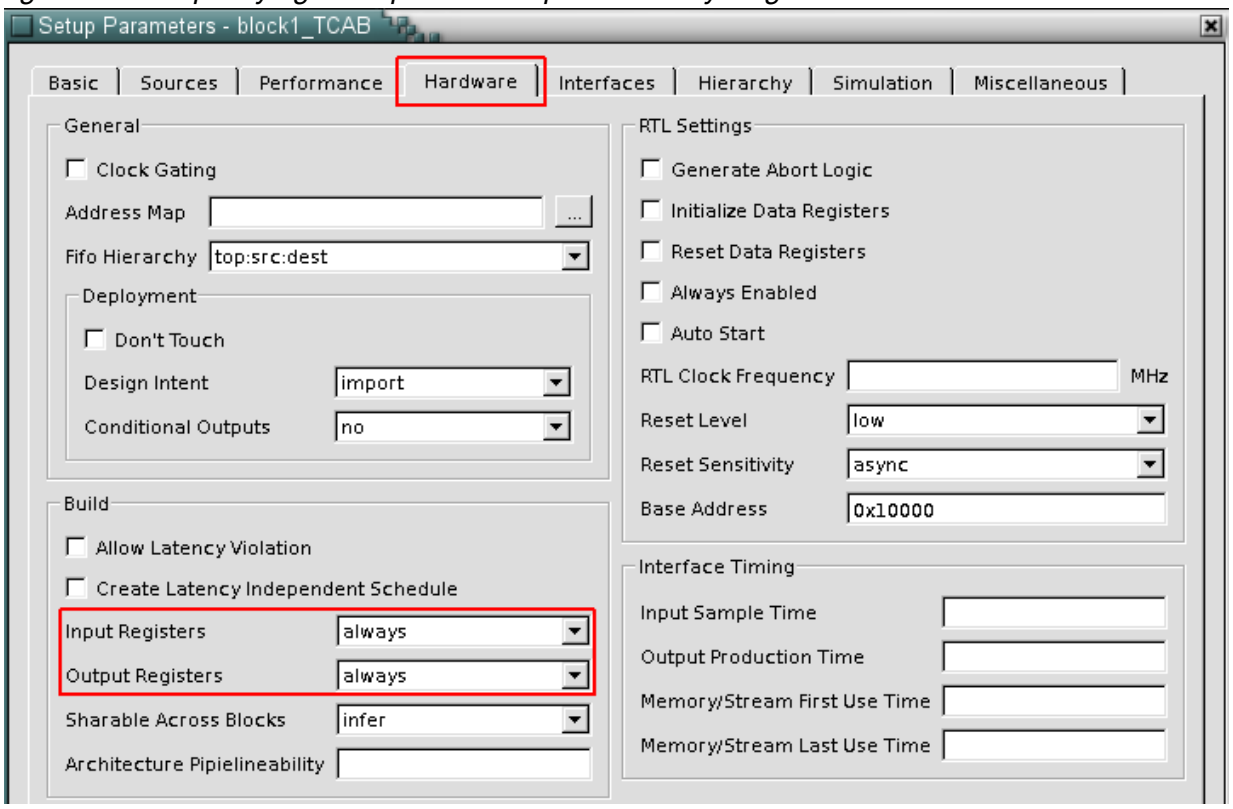
By default, the tool infers and creates boundary registers as needed to meet the delay constraint. Follow these steps to modify the behavior

1. To force the tool to always infer boundary registers for the TCAB,

GUI	Select always in the Input Registers and Output Registers box, as shown in Figure 7-16.

Command Line	`set_params -tcab_input_registers always` `set_params -tcab_output_registers always`

Figure 7-16 Specifying the Input and Output Boundary Register Inference



2. To force the tool to never infer boundary registers for the TCAB,

GUI	Select Input Registers or Output Registers to never.
Command Line	`set_params -tcab_input_registers never` `set_params -tcab_output_registers never`

Specifying Latency for Hierarchical Blocks

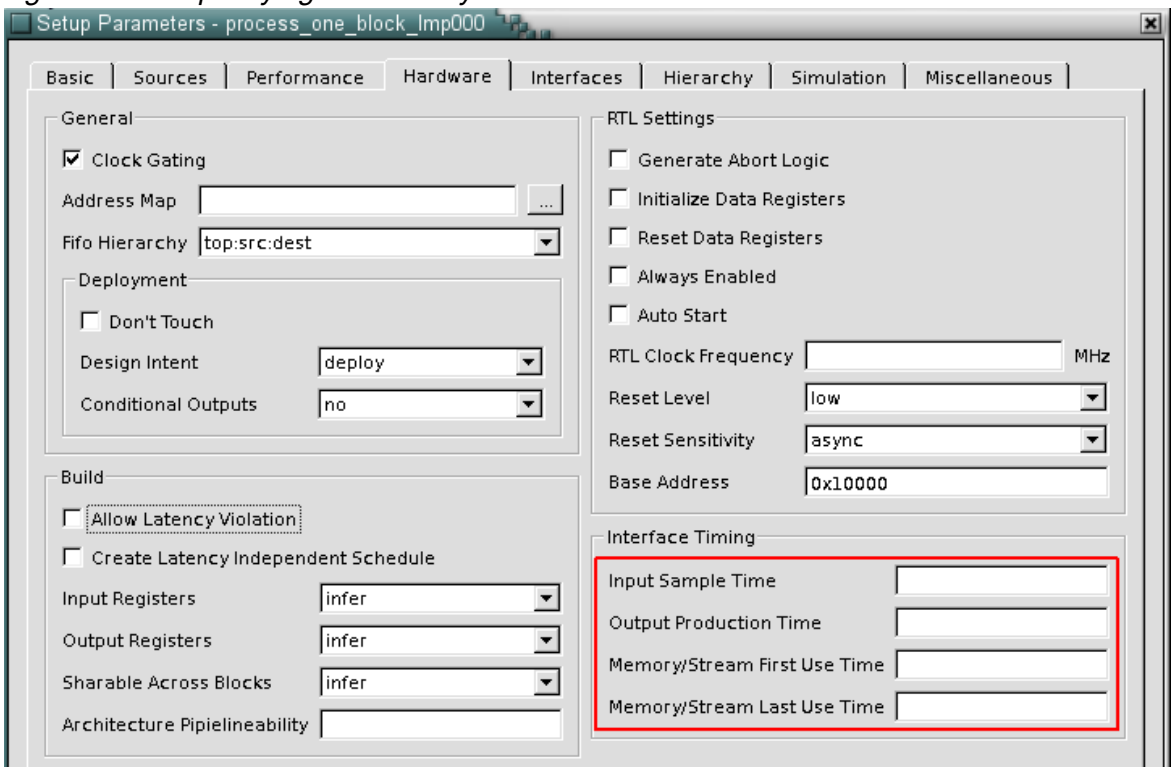
Unlike top-level blocks, you can specify latency on the inputs and outputs of hierarchical blocks. This means that inputs can be sampled after the TCAB is issued, after the specified number of cycles, when they are actually required for data processing. Similarly, you can set

output latency to define the number of cycles from the TCAB start to when the output was last updated or accessed. Specifying these latencies gives the tool more direction on easing feedback around the hierarchical blocks.

1. Specify latency from the GUI or the command line.
   - From the GUI, set latency globally for the TCAB, or for individual inputs and outputs. The value must be greater than 1.

Scalar inputs	Input Sample Time (global or per port)
Scalar outputs	Output Production Time (global or per port)
Memory or stream inputs	Memory/Stream First Use Time (global or per port)
Memory or stream outputs	Memory/Stream Last Use Time (global or per port)

Figure 7-17 Specifying the Latency for the TCAB



- From the command line, set latency globally for the TCAB, or for individual inputs and outputs. The value must be greater than 1.

Inputs/ Outputs	set_params command option
Scalar inputs	`-input_sample_time <cycles>` `-input_sample_time ["<input1>:<cycles> <input2>:<cycles>..."]`
Scalar output	`-output_production_time <cycles>` `-output_production_time ["<output1>:<cycles> <output2>:<cycles>..."]`
Memory and stream input	`-resource_first_use_time <cycles>` `-resource_first_use_time ["<input1>:<cycles> <input2>:<cycles>..."]`
Memory and stream output	`-resource_last_use_time <cycles>` `-resource_last_use_time ["<input1>:<cycles> <input2>:<cycles>..."]`

- Set the output latency limits to one cycle less than the deprecated latency for the architecture.

The tool uses the specified number of cycles as an upper limit. Setting latency values gives the tool more flexibility and makes the design more pipelineable.

2. Specify how you want to handle latency violations.

- Specify whether to allow latency violations or generate stalls with the `set_params -allow_latency_violation yes|no` command.

  In addition to the delays specified in step 1, you might have additional physical latencies because of resource or data dependencies. When it encounters latency violations, the tool creates an independent stall domain TCAB, which stalls the caller. Setting this command to no lets you override this behavior.

- To force the creation of an independent stall domain TCAB, use `set_params -force_independent_stalldomain_tcab yes`.

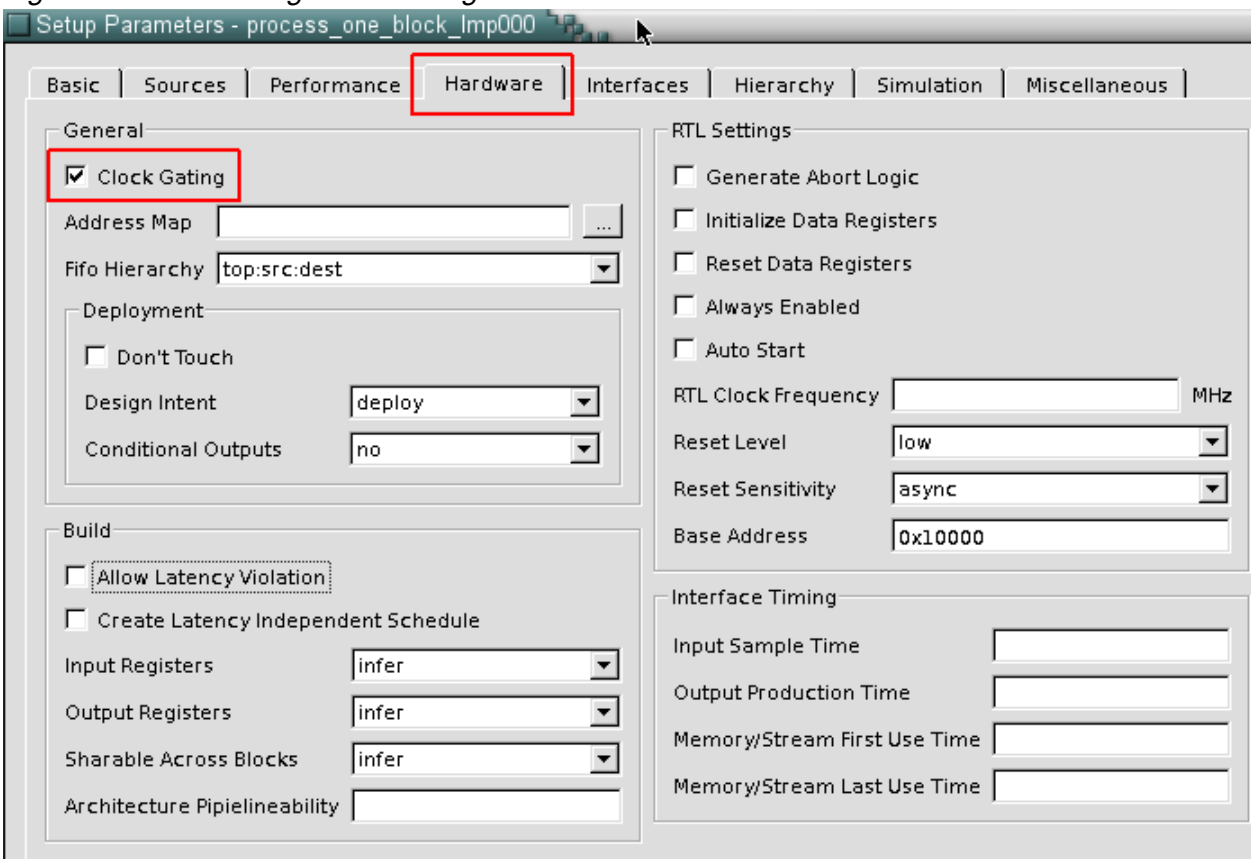
Gating Clocks on Hierarchical Blocks

To reduce power you can enable clock gating in the TCAB.

1. To enable clock gating,

GUI	Set Parameters > Hardware. Select the Clock Gating check box.
Command Line	`set_params -clock_gating yes`

Figure 7-18 Selecting Clock Gating in the TCAB



The tool creates additional clock_gate_enable signals at the TCAB interface to determine when the TCAB is not active, and automatically propagates this port to the higher hierarchical levels. Clock gating can provide significant power savings at a minimal cost, because you gate the clock to the module when it is not in use.

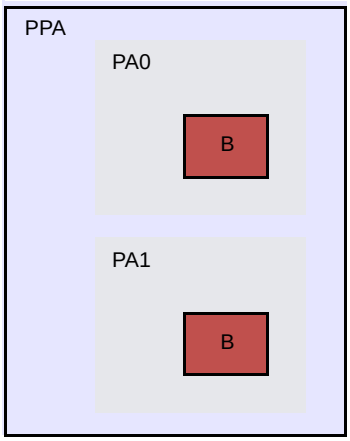
2. Run simulation.
3. Check the results in the clock gating reports.

See the *Synphony C Compiler Reference Manual* for information about the reports.

Sharing Hierarchical Blocks

You can share a TCAB within a PA or across PAs that are mutually exclusive in time. You can share TCABs by calling them at different levels of the hierarchy. In the following figure, you can share B because PA0 and PA1 run sequentially.

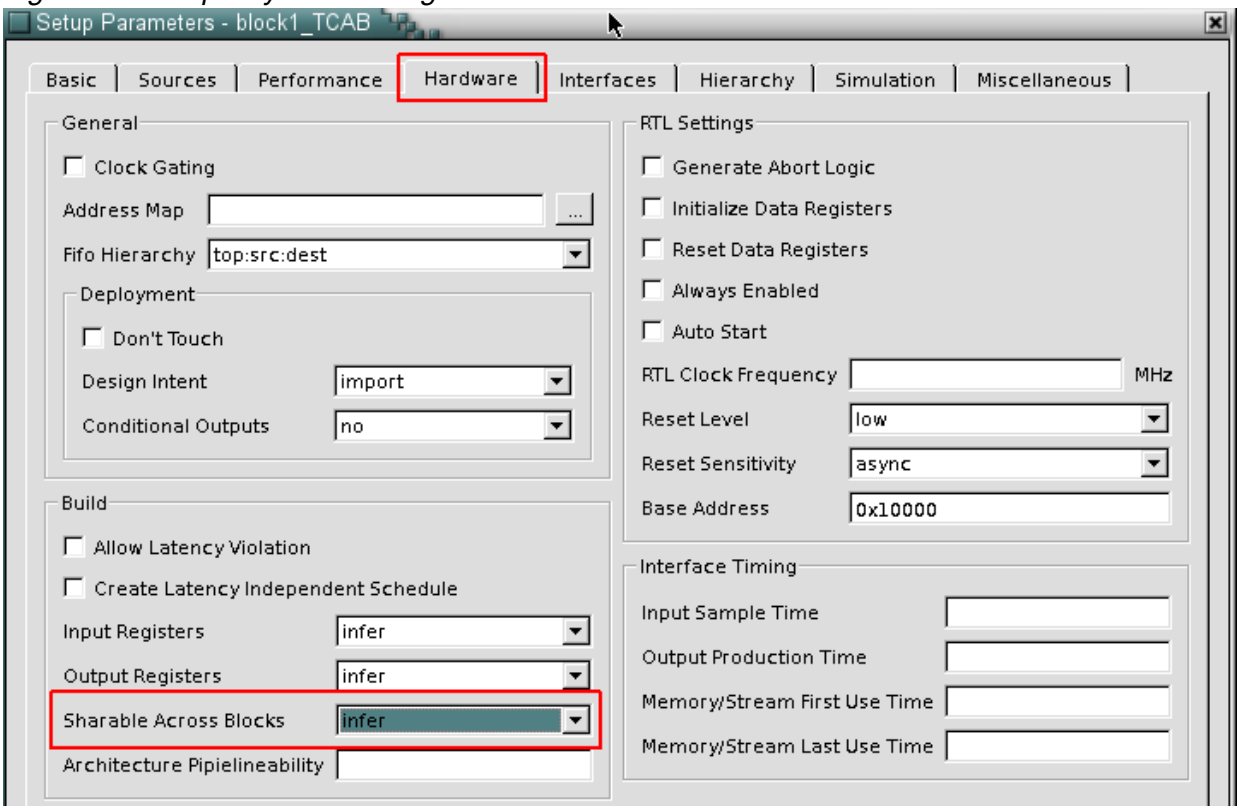
Figure 7-19 Sharable PAs



1. To share TCABs, ensure that your design meets these prerequisites:
   - The containers for the TCABs (PAs) must not overlap in execution, either in the same task, or across different tasks.
   - The calls must be mutually exclusive because the SCC tool only supports multiplexed sharing, not arbitrated sharing.
2. To specify whether the block can be shared, in the Hardware tab of the Setup Parameters dialog box, select infer in the Sharable Across Blocks drop-down box as shown in Figure 7-20.

   Alternatively, use the the `set_params -sharable_across_blocks` command.
   - If you leave the default of infer, you leave it to the tool to decide whether to share the block or not.
   - To share the TCAB regardless of the benefit in cost set this to always. With this setting the tool makes all instances of the block sharable.
   - Change the setting to never if you do not want to share any of the block instances.

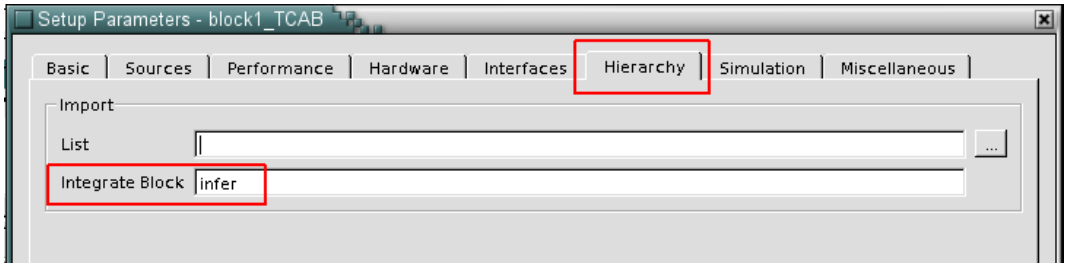
Figure 7-20 Specify the Sharing of the TCAB Across Blocks



3. Import the TCAB into the parent block.

GUI	In the Hierarchy tab of the Setup Parameters dialog box, set the Integrate Block option to infer, localize, or externalize.
Tcl	`set_params -integrate_block infer | localize | externalize`

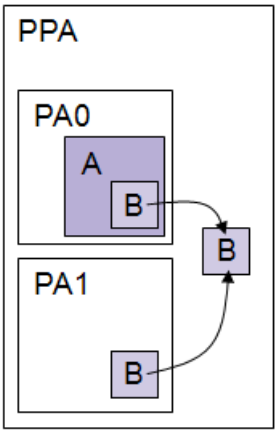
Figure 7-21 Importing a TCAB into the Parent Block



The default for this parameter is infer as shown in Figure 7-21. The tool decides whether the TCAB can be instantiated and shared outside the current container. The tool localizes the TCAB if is not sharable, and externalizes it if it is sharable.

- To create instances of the imported TCAB within the current container only and not share it outside the container, set the parameter to localize.
- To port all instances of the imported TCAB outside the current container, set the parameter to externalize. In this case the TCAB is created where it is localized. Use this setting when you want to share a TCAB at a higher hierarchical level.

  In the following example, PAO and PA1 are mutually exclusive in time. B can be shared by the two blocks if it is externalized. The tool moves the physical copy of B outside the hierarchy so that it can be shared.



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.

Building Hierarchy and Importing Blocks

The hierarchical design model works bottom up, as described in Synphony C Compiler Hierarchical Design Flow. After synthesizing the hierarchical blocks, you can build the hierarchy by importing lower-level blocks into higher-level blocks. The following sections describe the details:

- Building Hierarchy
- Importing Blocks

Building Hierarchy

The following procedure shows how to set up a design with multiple levels of TCAB hierarchy.

1. Create the lowest-level TCAB (TCAB 1) and synthesize it.

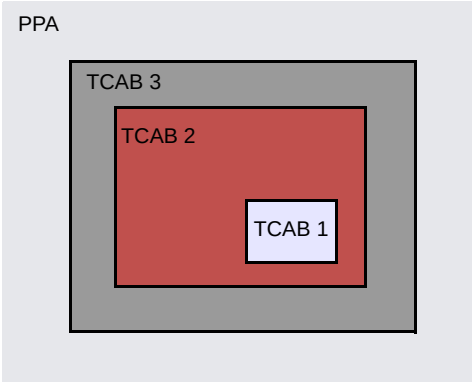
2. Import it into the next level (TCAB 2) and synthesize this block.

   See Importing Blocks for details about importing blocks.

3. Repeat the previous steps to build your hierarchy. In this example, import TCAB 2 into TCAB 3.

   There is no limit to the number of hierarchical levels you can have, and you can share the same TCAB at different hierarchical levels.

Figure 7-22

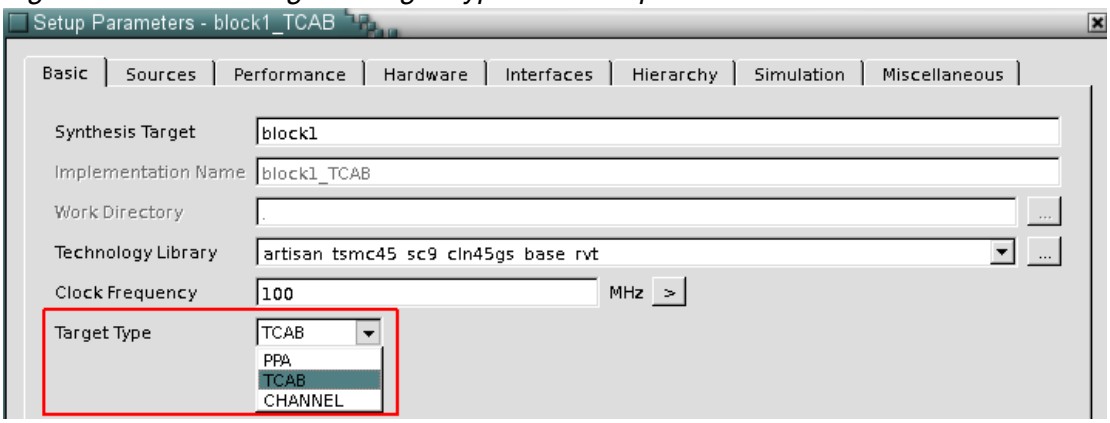


4. For the top-level block specify the target as TCAB or PPA in the Setup Parameters dialog box as shown in Figure 7-23.

   - To implement a PPA, choose PPA as the Target Type to PPA.
   - To implement a deployable TCAB, choose TCAB as the Target Type.

   For more details, see Using Deployable TCABs or PPAs for the Top Level.

Figure 7-23 Selecting the Target Type for the Top-Level Block



Importing Blocks

You can use the following procedure at any level of the hierarchy to import TCAB hierarchical blocks into other TCABs or into the top-level PPA block.

1. Create a single project that contains the source code for the top level and all the hierarchical blocks at all levels of hierarchy.
2. Build the block to be imported as a hierarchical block and synthesize it.

   See Building Hierarchical Blocks for details. The hierarchical design model requires that you build the design from the bottom up; therefore, synthesize lower-level blocks before you importing them into the next hierarchical level.
3. All TCAB function definitions must be available in the parent PA block when you import the TCAB into the next hierarchical level.

   If you modify the TCAB source code, rebuild the TCAB before importing it into a higher-level container.
4. Import a TCAB into a parent block from either the GUI or from the command line.
   - Select Implementation > Configure Implementation. Specify the hierarchical blocks you want to import in Tcab import list.
   - Specify this command from the command line: `set_params -import "<tcab1> <tcab2> <tcab3> …"`

   You only need to specify the immediate child block that you are importing. This means that if you want to import block B into A, and block B contains block C, you only need to specify block B for import into A when you are building A.
5. Check the hierarchy in the Hierarchy graph, as described in Using the Hierarchy Graph.

Analyzing Hierarchical Blocks

The Synphony C Compiler provides reports and views to analyze your hierarchical design. You can also simulate your design to verify your results.

Reports for the Hierarchical Blocks

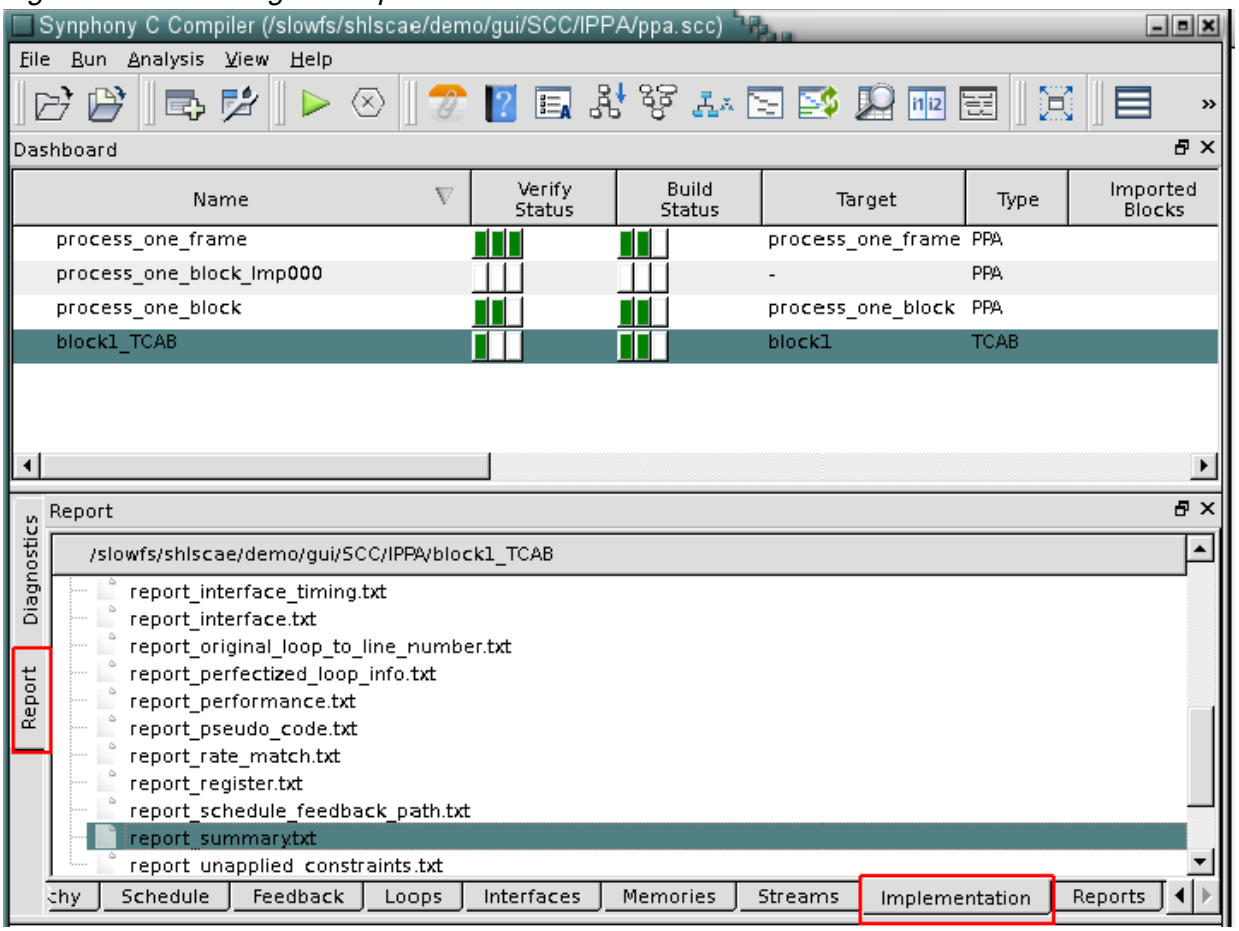
Do generate the reports for the hierarchical blocks

1. Build the implementation.

   The tool generates a summary report in the *implementation*/Reports directory.

2. You can view the report in the following ways:
   - Select the Implementation tab in the Report window. Select the Report tab. Double-click `report_summary.txt` file.

Figure 7-24 Viewing the Report



   - Use the `report_implementation -summary` command.
   - Open the *implementation*`/Reports/report_summary.txt` file in a text editor.

   This file displays the TCAB properties, as shown in the following. For detailed information about the report, see the *Synphony C Compiler Reference Manual*.

```
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
Scalar Input Sample Times:
+---------------+---------+-----+-------------------------------+
| Architectural | Physical| Reg | Item(width, C name)           |
| Cycle #       | Cycle # |     |                               |
+---------------+---------+-----+-------------------------------+
|             0 | 0       | no  | _scc_tcab1_t1_a_in_0(8, t1_a) |
|             0 | 0       | no  | _scc_tcab1_t1_b_in_1(8, t1_b) |
|             0 | 0       | no  | _scc_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:
+----------------+----------------+-----+---------------------+
| Architectural  | Physical       | Reg | Item(width, C name) |
| Cycle # [Spec] | Cycle # [Init] |     |                     |
+----------------+----------------+-----+---------------------+
|            1[] |           1[1] |  no | _scc_ret_tcab1(8)   |
+----------------+----------------+-----+---------------------+
Cycle # is the cycle during which the data is available to the
container.
Resource First Use Times:
     None.
Resource Last Use Times:
     None.
```

3. Check the following in the report:
   - Latency (Maximum physical output production/Release last use time)
   - Targeted and achieved pipelinability (Architectural Pipelinability and Physical Pipelinability).
   - Sharing (Stall domain). If the TCAB has an independent stall domain, the report lists the reason for creating it.
4. Check that the TCAB was imported correctly by checking the report file in the parent implementation.

   The report file is called `report_imported tcab.pdf` and is in the *implementation*`/Reports` directory.

8

Designing Efficient Hardware In C/C++

You can increase the efficiency of your hardware by coding the design in C or C++ in an optimal way. Code that is optimized for the Synphony C Compiler tool differs from traditional sequential C programming in that it is written to take advantage of the parallel optimizations that the tool can execute and it accounts for system architecture requirements.

The following describe some tips and techniques for doing this.

- Coding Guidelines in Synphony C Compiler Tool
- Building Memories
- Building FIFOs
- Building Shift Registers
- Optimizing the Build Steps
- Influencing Perfectization
- Flushing the Pipeline
- Optimizing Memories for Rate Matching
- Optimizing for Area
- Optimizing Performance
- Coding to Maximize Parallel Optimizations
- Task Processing

- Using Templates
- Correlating Timing With ASIC Synthesis

Coding Guidelines in Synphony C Compiler Tool

This section describes in detail the following about coding guidelines used in Synphony C Compiler Tool:

- General Coding Restrictions
- Supported C/C++ Syntax
- Unsupported C/C++ Syntax
- Interfaces
- Arrays
- Streams
- Synthesizable C/C++ Code

General Coding Restrictions

When you code your design in C, be aware of the following restrictions:

- Use arrays instead of pointers.
- Use structured code. Do not use goto statement.
- Do not use floating point notation.
- Do not use unions.
- Do not use volatile declarations.
- Do not write code with an outer loop around a sequence of loops.
- Use subfunctions. The modular nature of a subfunction is easier to understand, debug, and maintain. In addition, you can use it as a hardware subfunction without additional cost, because the tool automatically inlines functions that are not explicitly created as hierarchical blocks.

Supported C/C++ Syntax

Synphony C Compiler tool supports a wide range of C/C++ syntax, to describe the hardware to be synthesized. The supported syntax is described in the following sections:

- Data Types
- Modifiers

- Enum and Typedef
- Parameterized Data Type
- Arrays
- Class and Structure
- Functions and Methods
- Unsupported C/C++ Syntax

Data Types

The Synphony C Compiler tool supports the following data types:

Unsigned		Signed	
`char`	8 bits	`char`	8 bits
`short`	16 bits	`short`	16 bits
`int`	32 bits	`int`	32 bits
`long`	32 bits	`long`	32 bits
`long long`	64 bits	`long long`	64 bits

The Synphony C Compiler tool does not support the float and double data types in the synthesizable code.

For more details about the unsupported data types and syntax, see Unsupported C/C++ Syntax.

Modifiers

Synphony C Compiler supports the following modifiers:

- `extern`

  Use this modifier to declare globally scoped variables

- `const`, `signed`, and `unsigned` modifiers for static variables

  Implements a single copy of the variable in the hardware

The Synphony C Compiler tool does not support `register` and `volatile` modifiers in the synthesizable code.

For more details about the unsupported data types and syntax, see Unsupported C/C++ Syntax.

Enum and Typedef

The Synphony C Compiler tool supports the specification of enumeration constants for the supported data types, using the `enum` keyword. The enumeration constants must be in the [-2^{31} to 2^{31}] range. When your `enum` definition is outside this range, Synphony C Compiler converts the constants to unsigned values.

The Synphony C Compiler tool supports the `typedef` keyword to create new data types for all the supported data types and structures.

Parameterized Data Type

The Synphony C Compiler tool supports width-parameterized integer data types, for example, use s_int<n> and s_uint<n>, where 1<=n<=64.

To use the parameterized data type, include the `scc_sint.h` file in your synthesizable source code.

The following example shows how to use the parameterized data type.

```
#include <scc_sint.h>

s_uint<4> a;
s_uint<8> b;

a = 15;
b = a*a;
```

Synphony C Compiler supports the conversion of a parameterized data type to another data type and vice versa. The following example shows how to convert an unsigned integer to a parameterized data type.

```
unsigned int a;
s_uint<32> b;

b = a;
a = b.to_uint();
```

Fixed-Point Data Type

The Synphony C Compiler tool also supports the fixed-point data type. The syntax to define a fixed-point data type is as follows:

```
s_ufixed <int wl=32, int iwl=32, s_q_mode Q=S_TRN, s_o_mode, 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>
```

The following example represents a 10-bit fixed-point data type having 2-bit integer part.

```
s_fixed<10,2>a;
```

SystemC Data Type

The Synphony C Compiler tool does not support SystemC data types in the synthesizable code. Use the parameterized data type in your synthesizable code. However, Synphony C Compiler supports SystemC data types in the testbench code.

For more details about the unsupported data types and syntax, see Unsupported C/C++ Syntax.

Arrays

The Synphony C Compiler tool supports single and multidimensional arrays of various supported data types, arrays of structure, and class objects.

If you index a multidimensional array beyond its declared bounds, the result is undefined. This behavior differs from the standard ANSI-C semantics.

Pointers

The Synphony C Compiler tool supports pointers in the synthesizable code, if the pointer can be statically resolved, as shown in the following example:

```
int a[100];
int *q = &a[x];
```

The Synphony C Compiler tool supports arithmetic operations of pointers only if the pointers point to the same array.

Synphony C Compiler supports the comparison of pointers of the same data type. If you compare pointer of different data types, Synphony C Compiler produces an undefined result.

```
int a[10], b[10];
int *ap = &a[i]
int *bp = &b[i];
x = (ap < bp)
```

The tool supports the comparison of null pointers, as shown in the following example:

```
int *p=NULL:
if (condition)
  p=&a[i];
if(p!=NULL) {
   //code
```

```
}
```

For more details about the unsupported data types and syntax, see Supported C/C++ Syntax.

Functions and Methods

Synphony C Compiler supports functions and methods.

You can specify a function or method as a target for synthesis. The tool synthesizes all the functionality inside the target into hardware and writes out an RTL description of the hardware. By default, the top-level module of the generated hardware is called a PPA (pipeline of programmable arrays).

Functions and methods that are not specified as targets for synthesis are inlined. You can also define hierarchy in the synthesized hardware by specifying multiple functions or methods as targets. For more details about hierarchical synthesis, see Chapter 7, “Building Hierarchical Designs.

The tool does not initialize global and static variables to zero value in the synthesized hardware. You must explicitly initialize the variables that you declare. You can set the reset_data_register parameter to “yes” to initialize all the data registers in the synthesized hardware. The scalar variables and arrays that are not part of the synthesizable code must be initialized in the testbench code.

Class and Structure

The Synphony C Compiler tool supports functions, data encapsulation, constructors, and destructors inside the C++ `class` and `struct` type elements.

When you define a global class object, the tool does not synthesize the hardware for the constructor and destructor of the class object; the constructor and destructor are considered as testbench code.

Unsupported C/C++ Syntax

The Synphony C Compiler tool does not support the following C/C++ syntax in the synthesizable code:

- Float and double data types
- Register and volatile modifiers
- Pointers as members of struct and class in the synthesizable code
- Pointers to pointers and array of pointers

- Arithmetic operations of pointers where the arithmetic elements point to different scalar arrays
- Pointer to data members in a class
- Union construct and SystemC data types as fields of structures
- Bit field specifications for class and structures
- Loops in constructors and destructors
- Virtual functions
- Multiple inheritance
- Exceptions
- RTTI (typeid, dynamic_cast)
- Dynamic memory allocation
- The `export` keyword
- Dynamic initialization
- User-defined names that begin with the following prefixes: scc, SCC, pico, PICO

  Synphony C Compiler reserves these names for internal use. Do not use them for defining file names, variable names or the typedef names.

Interfaces

The Synphony Compiler tool creates the interfaces of the top level module in the generated RTL hardware when parameters of a function or method is specified as a target for synthesis.

System and Control Interface

The Synphony C Compiler tool generates hardware containing default system and control interfaces using the parameters specified in the target function or method. This enables the integration of the hardware with an external system. Table 8-1 shows the various interface ports, their defaults and, and control interface generated by the tool:

Table 8-1 Tool Interfaces, Default, and Control

Interface	Default	Control
Clock input	Positive edge	Edge sensitivity

Table 8-1 Tool Interfaces, Default, and Control

Interface	Default	Control
Reset input	ASIC: active low, asynchronous FPGA: active high, synchronous	Level sensitivity
start handshake done handshake		Controls the execution of the hardware block

Scalar Variables and Parameters

For the scalar variables that are parameters of the target function or method, the Synphony C Compiler tool generates wires or registers based on the operation of these elements. To control the bit-size of the hardware element corresponding to the scalar variable, use the s_int or s_uint type or the `bitsize` pragma statement as shown in the following example:

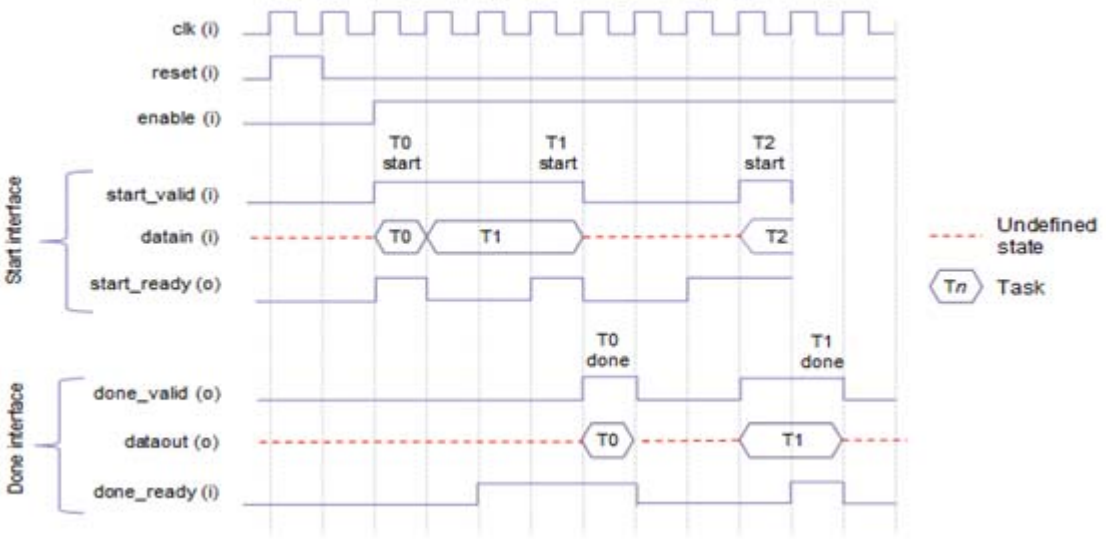
```
#pragma bitsize scalar_var_1 bit 8
```

Type of scalar defined	Hardware generated by Synphony C Compiler
Global scalar variable, scalar parameter (pass by reference), scalar class member	Input wire (if read by C/C++ function) Output wire (if written by C/C++ function)
Scalar parameter (pass by value)	Input wire
Scalar return value	Output wire

The read and write operations of the scalar input and output signals must be synchronized with the handshake (control) signals:

- The input wires corresponding to the scalar variables must be valid before the start (handshake) signal of the PPA; datain signal in Figure 8-1.
- The output wires corresponding to the scalar variables must be read (sampled) before the done (handshake) signal of the PPA; dataout signal in Figure 8-1.

Figure 8-1 Control HandShake Signals



Constant Scalar Variables

The tool creates a register within the synthesized hardware for a scalar variable that is an input parameter of the target function or method.

If the scalar variable is a constant across all calls to the target function or method, this register is not necessary. Use the following pragma to inform the tool about the variable:

```
#pragma stable var_1
```

The input wire corresponding to the scalar variable must be held constant during the execution of the generated hardware block.

Class Member and Global Scalar Variables

The input and output wires generated by the Synphony C Compiler tool for the class members and global scalar variables might not be accessible outside the hardware block. To prevent the tool from creating interfaces for these blocks, use the following pragma:

```
#pragma localize_access yes var_1 ppa_1
```

Struct Member Scalar Variables

The struct parameters are also considered as scalar variables. In the RTL, Synphony C Compiler creates a port for each member of the structure. To create a single port for a structure, specify the packed pragma as shown in the following example:

```
#pragma packed struct_var_1 yes
```

Arrays

When you declare the array variables, the Synphony C Compiler tool generates different types of memories in the hardware. By defauit, the tool creates a single memory for an array. Multidimensional arrays have their address spaces mapped to the flat address space of the memory.

For example, array element M[2][10] would be mapped to location 20 of the memory synthesized for the array. Synphony C Compiler can split the array M[2][10] into two different memories, M_0[10] and M_1[10], when you declare the array M

- In the scope of a procedure and M is accessed with a constant in the left most dimension. For example, M[0][i], M[1][i], and so on.
- In the global scope and specify the `memory_structure` pragma:

  ```
  #pragma memory_structure M bank
  ```

The tool can create wires or registers for the array you declare, when you specify the `scalarize` pragma. However, this can create large multiplexers in the hardware that increases the number of operations.

The tool can also create wires or registers for arrays, when the scope of the array is local and the array is accessed with a constant address only. Specify the following pragma to stop the tool from creating wires or registers for the arrays.

```
#pragma scalarize array_var1 no
```

To control the type, placement, and properties of the memory generated, use the array declaration and the pragma specification.

Memory Type

Table 8-2 shows the type of memory generated based on the array and pragma specification.

Table 8-2 Type of Memory Controlled by the Array and Pragma Declaration

Type of memory in the hardware	Array and pragma declaration
ROM	Array defined with `const` modifier. Example: `const int x[100]`
ROM	Using the `rom` pragma. Example: `#pragma rom x`
SRAM	All other array declarations

Note:
The Synphony C Compiler tool generates an error message when you attempt a write operation to the arrays that generate ROM memory in the hardware.

Memory Placement

Table 8-3 shows the pragmas that you can specify to control the placement of the memory in the hardware.

Table 8-3 Placement of Memory Controlled by the Pragma Declaration

Placement of memory in the Hardware	Default	Pragma declaration
External	If size greater than 16 words default is 256 bits	`#pragma user_supplied var_1` (for ASIC) `#pragma user_supplied_fpga var_2` (for FPGA)
Register-based internal memory	If size less than 16 words, default is 256 bits.	`#pragma internal_fast var_3`
External, when X > size Internal, when X <= size		`set_params -ram_synthesis_threshold size` `set_params -rom_synthesis_threshold size`
Internal block RAM		`#pragma internal_blockram size` (for FPGA only)

Memory Properties

Use the commands shown in Table 8-4 to control the properties of the memory generated by the Synphony C Compiler tool.

Table 8-4 Commands to Control Memory Properties

Memory property	Command or constraint

Latency
(default 1 cycle)
	Control the latency of all external memory
	`set_defaults -memory_access_latency latency_value`
	
	Control the latency of individual external memory
	`#pragma memory_access_latency array_var1 value`

Table 8-4 Commands to Control Memory Properties

Memory property	Command or constraint

External delay (default 0 ns)	Control the delay of all external memory
	`set_defaults -memory_forward_path_delay value`
	`set_defaults -memory_backward_path_delay value`
	
	Control the delay of individual external memory
	`#pragma memory_forward_path_delay array_var value`
	`#pragma memory_backward_path_delay array_var value`

Boundary registers (default `never`)	Control registers of all external memory
	`set_defaults -memory_forward_boundary_register value`
	`set_defaults -memory_backward_boundary_register value`
	
	Control the register of individual external memory
	`#pragma memory_forward_boundary_register array_var value`
	`#pragma memory_backward_boundary_register array_var value`
	
	where, value can be `always|never|infer|datapath`

Ports	`#pragma read_write_ports array_var combined`
	`#pragma read_write_ports array_var combined 2`
	
	`#pragma read_write_ports array_var separate`
	`#pragma read_write_ports array_var separate 1 readonly 1 writeonly`

Streams

The Synphony C Compiler tool allows you to use streams in your C/C++ code. These streams are synthesized into FIFO streams and stream interfaces in the generated hardware.

The stream data type in Synphony C Compiler tool is a C++ class that contains the `get()` and `put()` methods to transfer the data in and out of the stream.

The following example shows how to declare an integer type stream variable:

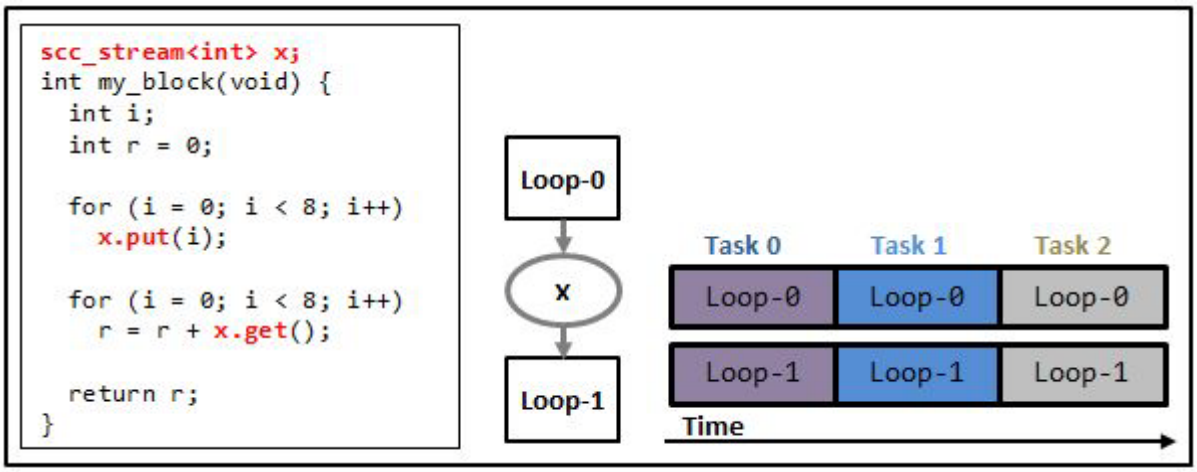
```
scc_stream <int> stream_var_1;
```

An internal stream can be used to parallelize the execution of loops specified sequentially in C/C++. Figure 8-2 shows C/C++ code for two loops communicating via a stream. Synphony C Compiler allows the two loops to execute in parallel in the generated hardware and introduces a FIFO between them to manage the data flow. Figure 8-2 also illustrates the FIFO and how the generated hardware executes in time.

By default, the length of the FIFO is 0. To increase the length of the FIFO use the `fifo_length` pragma as shown in the following example:

```
#pragma fifo_length var_name depth
```

Figure 8-2 Two PAs in Parallel, Each With an Internal FIFO



An external stream generates a data interface in the hardware that is either an input or output port with valid handshake signal ports.

Synthesizable C/C++ Code

You can specify the algorithm that can be converted into hardware using the C/C++ language, in one or more files. The coding style of the algorithm affects the implementation of the hardware . The following section describes the coding requirements to create efficient hardware.

Include scc.h header file in your source file as shown in the following example. This is a prerequisite to synthesize your code.

```
#include "scc.h"
```

To use C++ syntax in the testbench code, define the entry point as follows:

```
extern "C" int main () {
...
}
```

Coding Requirements

To generate fast and predictable hardware use the architecture-driven methodology when specifying your algorithm in the C/C++ language, and follow these coding guidelines:

- Use a modular approach to define your algorithm. Define procedures for each functional block where a top-level procedure invokes lower-level procedures. This enables resource sharing and parallelism.
- Define and use variables whose scope is local to the procedure where they are used.
- Define parameters for the tool to generate memory and other data interfaces in the hardware as expected.
- Define loops whose exit criteria is predictable.
- To obtain an efficient hardware of two functional blocks that run in parallel, define a loop for each functional block and use stream variables to share data between the loops.
- Specify appropriate constraints and directives to get good QoR.
- Avoid using pointers. Use arrays instead.
- Avoid defining loops that contain a sequence of loops.
- Avoid using unsupported syntax.

Example Source File

Source File

Table 8-5 describes the content of the source file, fir.c.

Table 8-5 Contents of the Source File fir.c

Statements	Purpose and use
`#include "scc.h"` `#define NTAPS 15`	Includes the scc.h file. This is a tool-specific header file that is required for synthesizing the C code into hardware.

Table 8-5 Contents of the Source File fir.c

Statements	Purpose and use
`int fir (int in)` `{`	Input is a function parameter. The return value of the function is the output.

`#pragma bitsize in 12`
`#pragma bitsize fir 12`
`#pragma bitsize window 12`
	The `bitsize` pragma controls the size of the input and output ports.

`const int taps[NTAPS] = {` `-12, -36, -57, 0, 210, 553, 885, 1024, 885, 553, 210, 0, -57, -36, -12 };`	The use of const in the array definition creates ROM in the hardware.

`static int window[NTAPS] = {`
`0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0`
`};`
 	
`unsigned short i;`
`#pragma unroll`
`for (i = NTAPS-1; i > 0; i-- ) {`
`window[i] = window[i-1];`
`}`
`window[0] = in;`
 	
`int sum = 0;`
`#pragma bitsize sum 16`
`#pragma unroll`
`for (i = 0; i < NTAPS; i++) {`
`sum = sum+((taps[i]*window[i]) >> 12);`
`}`
`if (sum < -2048) sum = -2048;`
`else if (sum > 2047) sum = 2047;`
`return sum;`
	The use of `static` causes the variable to be converted into a register in the hardware that stores values across different tasks.
	
	The `unroll` pragma causes the tool to convert the loop into repeated lines of code and execute the operations in a single cycle.

Testbench File

Table 8-6 describes the content of a testbench file.

Table 8-6 Contents of the Testbench File fir_tb.c

Statements	Purpose and use
`#include "scc.h"`	Includes the scc.h file. This is a tool-specific header file that is required for synthesizing the C code into hardware.

Table 8-6 Contents of the Testbench File fir_tb.c

Statements	Purpose and use
`extern "C" int main(int argc, char **argv)`	Use `extern "C" int main` as the entry point of the testbench code.

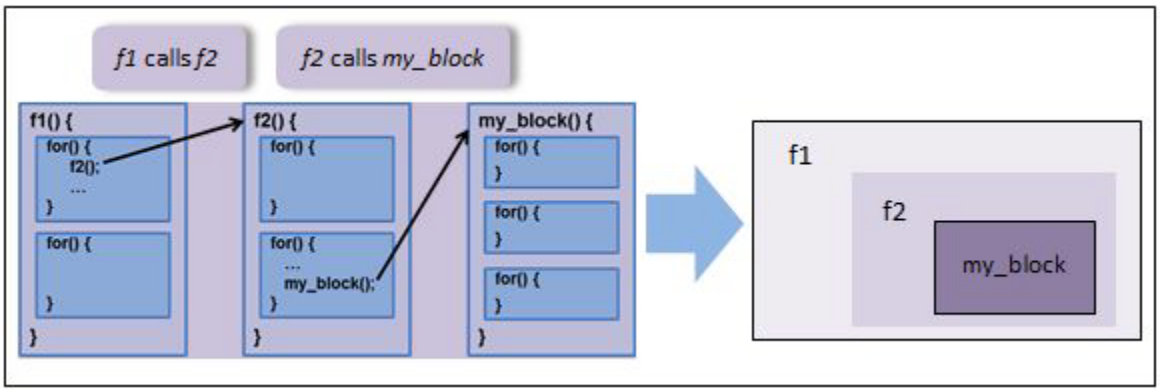
`{`
`for (i = -0; i < 100; i++) {`
`printf("%d\n", fir(i));`
`}`
`return 0;`
`}`
	Invoking the the synthesizable function fir.

Hierarchical Synthesis

When synthesizing hierarchical designs, the Synphony C Compiler tool can synthesize individual blocks lower in the hierarchy and import these blocks when synthesizing the top-level design. A TCAB or an IPPA is a lower-level hierarchical block that is meant to be integrated into another block that is higher in the hierarchy. The Synphony C Compiler tool uses an import mechanism to integrate individual blocks into the top-level block.

Consider the hierarchical design shown in Figure 8-3. The design has multiple levels of hierarchy, the top-level function f1() calls another function f2(), which calls another function my_block(). Each function can have multiple loops defined.

Figure 8-3 Hierarchical Design



For such a hierarchical design, the tool allows you to synthesize each procedure as an importable hardware block. When you synthesize each procedure, you can import the hardware blocks created for the procedures called by the current target procedure.

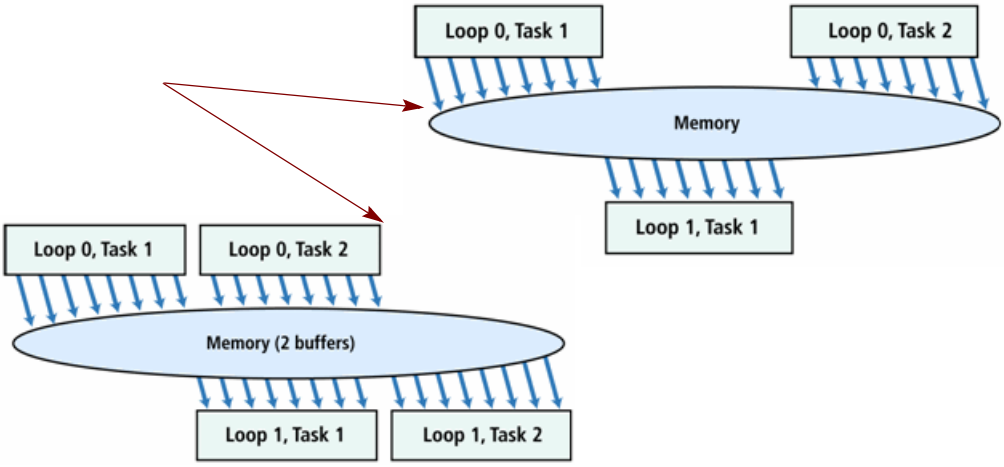
Building Memories

The Synphony C Compiler tool can map arrays to memories. Your design can have internal or external memories and they can be RAM or ROM memories. The tool includes various parameters and pragmas that let you determine the kind of memory that the tool implements.

The following example code and figure show the coding style to implement memory. The tool implements single-buffered or double-buffered memory based on the performance constraints that are specified, like task II.

```
for () {
     x[i] = a;
}
for () {
b = x [j];
...
```

Figure 8-4 Coding Style to Implement Memory



See the following for information about building memories:

- Coding Arrays and Mapping Memories
- Mapping C Arrays to Memories in ASIC Designs
- Mapping C Arrays to Memories in FPGA Designs
- Specifying Memory Properties

For information about optimizing memories, see the following

- Optimizing Memories for Rate Matching
- Optimizing Local Memories

- Minimizing Memory Ports

You might want to use streams instead of memories in certain situations. See Using FIFOs Instead of Memories for a discussion of the advantages and disadvantages of each.

Coding Arrays and Mapping Memories

In general, the Synphony C Compiler tool maps the following C variable types to FIFOs and memories in hardware.

C variable type	Hardware implementation
Array, declared at file/procedure	Local memories; could be third-party external SRAM
Array, declared at file/procedure, and internal_fast	Local memories; SCC register-based synthesizable RAM
Scalar, declared at file	Generally implemented as registers; could be a wire tied to a constant input if it is declared at file scope and initialized as a constant.
Constants	ROMs
Stream, with SCC constructs	FIFOs

Mapping C Arrays to Memories in ASIC Designs

This procedure shows you how to map arrays to memories in ASIC designs. You can specify whether the arrays are mapped to internal memories that can be synthesized and optimized, or whether they are hard macros that are implemented outside the PPA. You can also specify ROM and RAM implementations.

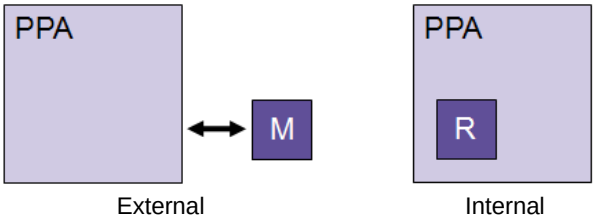
1. Set size thresholds of memory with the `set_params -ram_synthesis_threshold` and `set_params -rom_synthesis_threshold` commands.

   These commands set array size thresholds of ROM and RAM respectively. The tool only implements memories if the arrays are larger than the minimum threshold; arrays below the limit are implemented as registers.

   If you do not specify a threshold, the tool uses the following defaults:

   RAMs: 16 words, 256 bits
   ROMs: 256 words, 4096 bits

2. To specify RAM, do the following:
   - If you declare the array with `#pragma user_supplied`, the tool implements it as an external RAM, as shown here:



   - To specify internal RAM, declare the array with #pragma internal_fast. The tool generates internal flip-flops or register-based internal memory, depending on the design. Write takes one cycle and read is combinational. The tool implements the following as internal RAM:

```
int Y[10][10];
#pragma internal_fast Y
```

3. To specify ROM, do the following:
   - Declare the array with the const modifier or #pragma rom. This implements an external or internal ROM, depending on the other settings you specify. If you do not specify the modifier or the pragma, the tool implements the array as RAM, not ROM.

     To specify an external ROM, declare the array with #pragma user_supplied. The tool implements it as external ROM. The following is synthesized as external ROM:

```
const int Z[200] = {10, 20,};
```

     See Specifying Memory Properties for other parameters you can set for external RAMs.

   - To specify an internal ROM, declare the array with #pragma internal_fast. The tool generates internal flip-flops or register-based internal memory, depending on the design. The following example is implemented as synthesized (internal) ROM:

```
const int C[10][10] = {{5, 3, 2, 1,}, {3, 0,},};
#pragma internal_fast C
```

4. If you want to pack the array into a single wide memory, use #pragma packed.

   Concatenate memories and pack array elements into a single word to save on memory decoding. Making arrays wider with this pragma can lower the required bandwidth in some cases. Use it for very wide memories.

Mapping C Arrays to Memories in FPGA Designs

This procedure shows you how to map arrays to memories in FPGA designs. You can specify that the arrays be mapped to internal memories that can be synthesized and optimized, or that they be implemented outside the PPA as external memories. You can also specify ROM and RAM implementations.

1. Set size thresholds of the memory with the `set_params -ram_synthesis_threshold` and `set_params -rom_synthesis_threshold` commands.

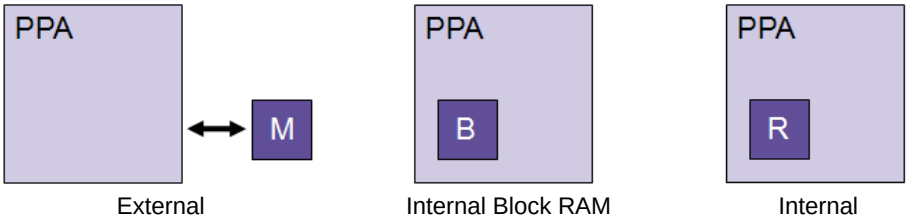
   These commands set array size thresholds of ROM and RAM respectively. The tool only implements memories if the arrays are larger than the minimum threshold; arrays below the limit are implemented as registers. If you do not specify a threshold, the tool uses the following defaults:

   RAMs: 16 words, 256 bits
   ROMs: 256 words, 4096 bits

2. To specify RAM, do the following:

   - Do not use the const modifier or #pragma rom. Either of these implements the array as ROM, not RAM.
   - To specify external RAM, declare the array with #pragma user_supplied_fpga. The tool implements it as an external RAM. Use this for hard macros. The following code is implemented as external RAM:

     ```
     int X[200];
     #pragma user_supplied_fpga X
     ```



   - To specify internal RAM, declare the array with #pragma internal_fast. The tool generates internal flip-flops or register-based internal memory, depending on the design. Writes take one cycle, and reads are combinational flip-flops. The software can optimize internal RAMs. The tool implements the following as internal RAM:

     ```
     int Y[10][10];
     #pragma internal_fast Y
     ```

- To specify internal block RAM, declare the array with #pragma internal_blockram. The tool generates internal flip-flops or register-based internal memory, depending on the design. The tool implements the following as internal block RAM, which is the default:

```
//internal block RAM
int X[200];
```

  It implements the following as block ROM:

```
// internal block RAM
const int Z[200] = {10, 20, };
```

3. To specify ROM, do the following:

   - Declare the array with the const modifier or #pragma rom. This implements an external or internal ROM, depending on the other settings you specify. If you do not specify the modifier or the pragma, the tool implements the array as RAM, not ROM.
   - To specify an external ROM, declare the array with #pragma user_supplied. The tool implements it as external ROM. The following is synthesized as external ROM, which is the default:

```
const int Z[200] = {10, 20,};
```

   - To specify an internal ROM, declare the array with #pragma internal_fast. The tool generates internal flip-flops or register-based internal memory, depending on the design. Writes take one cycle, and reads are logic.

     The following example is implemented as synthesized (internal) ROM:

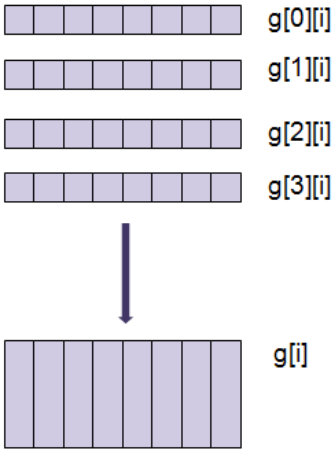
```
const int C[10][10] = {{5, 3, 2, 1,}, {3, 0,},};
#pragma internal_fast C
```

   - To specify internal block RAM, declare the array with #pragma internal_blockram. The tool generates internal flip-flops or register-based internal memory, depending on the design. The internal block RAM can be optimized.

4. If you want to pack the array into a single wide memory, use #pragma packed.

   Concatenate memories and pack array elements into a single word to save on memory decoding as illustrated in Figure 8-5. Making arrays wider with this pragma can lower the required bandwidth in some cases. Use it for very wide memories.

Figure 8-5 Concatenating Memories Using Pragma



Specifying Memory Properties

You can fine-tune the kind of memory that is implemented by setting various memory properties, as described here.

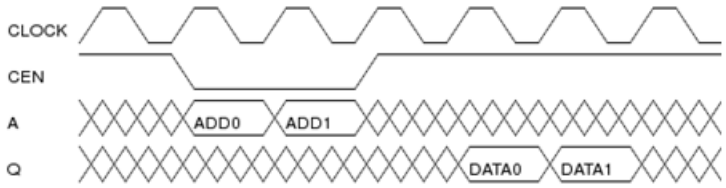
1. Specify the kind of memory.

   See Mapping C Arrays to Memories in ASIC Designs and Mapping C Arrays to Memories in FPGA Designs for information about specifying external memories and synthesized memories.

2. Define latency for external memories.

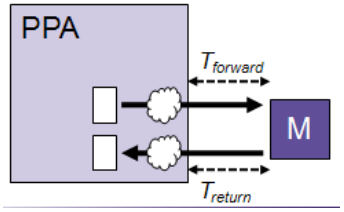
   - To define an intrinsic latency for all external memories, select Setup Parameters dialog box. Click the Interfaces tab, followed by the Memory tab. Set the value in Memory Access Latency. Alternatively, set it at the command line using the `set_params -memory_access_latency` command. Figure 8-6 shows memory latency set to 3:

Figure 8-6 External Memory with Latency Value 3



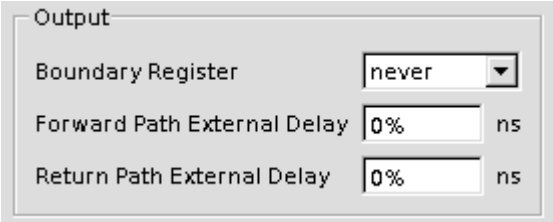
   - To override the global setting and set latencies for individual memories, set a value of 1 or greater using `#pragma memory_access` *`array latency`* in the source code. The default is 1.

3. Specify the external delay allocated to external memories.



- To set the delay globally, go to the Interfaces tab as described in the previous step. Set the delay values in Forward External Path Delay and Return Path External Delay, as shown in Figure 8-7.

Figure 8-7 Setting the Delay Globally



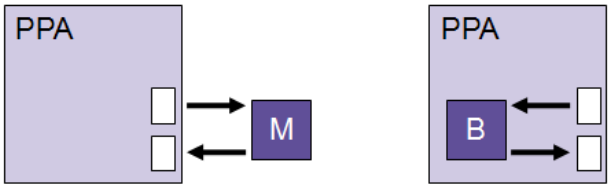
Alternatively, set it at the command line with the `-memory_forward_path_external_delay` and `-memory_return_path_external_delay` arguments to the `set_params` command.

- Override the global settings or define values for individual memories by specifying the following pragmas. The default is 0.

```
#pragma forward_path_external_delay <array> <delay>
#pragma return_path_external_delay <array> <delay>
```

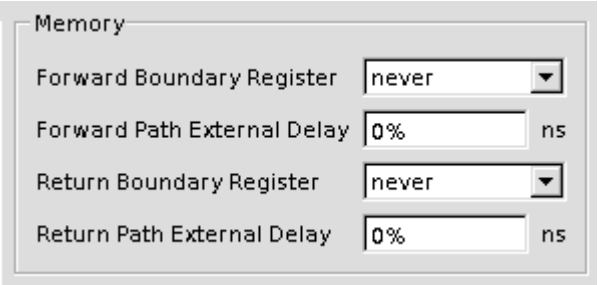
4. Set boundary register placement for external memories and for internal block RAMs in FPGA designs.

Figure 8-8 Register Placement for External Memories and Internal Block RAM



- To insert boundary registers globally, go to the Interfaces tab as described in Step 2. Set the delay values in Forward Boundary Register or Return Boundary Register to the setting you want: always, infer, never, or datapath, as shown in Figure 8-9.

Figure 8-9 Specifying Global Insertion of Boundary Registers



Alternatively, set it at the command line with the `-memory_forward_boundary_register` and `-memory_return_boundary_register` arguments to the `set_params` command.

- Override the global settings or define values for individual memories by specifying the following pragmas in the source code. The default is `never`.

```
#pragma forward_boundary_register [always | never |infer | datapath]
#pragma return_boundary_register [always | never |infer | datapath]
```

Building FIFOs

Streaming interfaces are common in embedded structures. IP blocks might be optimized for a particular global memory organization and for the system bus. You can also use streams to communicate between IP blocks that are pipelined together. You can build interfaces for external and internal streams, as described:

- Using FIFOs Instead of Memories
- Building External FIFOs
- Building Internal FIFOs

Using FIFOs Instead of Memories

Reduce cost by replacing memory with a stream. Streams are less expensive than memory, because they uses cheaper units to enable access to the stream. In addition, when you use internal streams instead of memories for inter-PA communication, the PAs can overlap tasks. If you used memories, the PA execution would normally be sequential.

The advantage that memories have over streams is that they allow random access, whereas stream data is streamed sequentially. With a memory you can randomly access any data in the memory by supplying the address.

Use streaming data in the following cases:

- When processing blocks have to run in parallel within a task
- When the read/write access pattern between blocks is sequential. Figure 8-10 shows a memory that transfers data between loops sequentially. This access pattern makes it an ideal candidate for this memory to be converted to a stream.

Figure 8-10 Sequential Data Transfer Between Loops

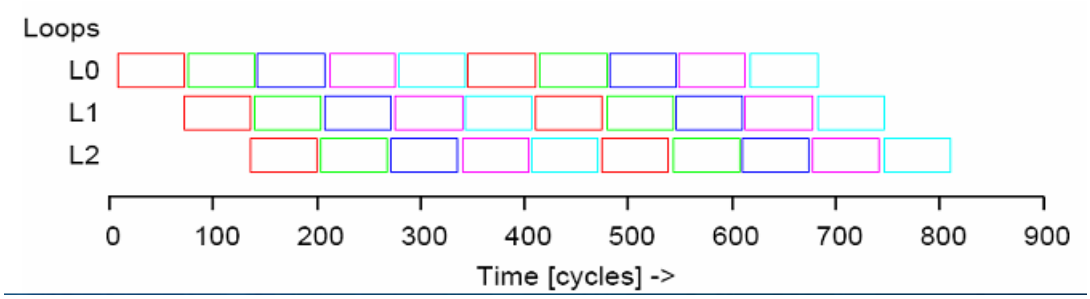
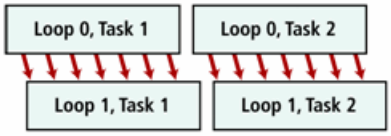


Figure 8-11 and the following example show the coding style to implement streaming data:

Figure 8-11 Coding Style to Implement Streaming Data




```
for () {
     scc_stream_output_x(a);
}
for () {
b = scc_stream_input_x();
)
```

Use memory in the following cases:

- When processing blocks can run sequentially
- When processing blocks have to run in parallel across tasks
- When the read/write access pattern between blocks is complex or random.

Building External FIFOs

External FIFOs define the communication between driver and PPA by streaming data between the global memory and other blocks on the system. For external streams in C++, use the scc_stream class to functionally emulate receiving from and transmitting to external streams; in C designs, use pragmas. The details are described in the following sections:

- Building External Streams in C++ Designs
- Building External Streams in C Designs

Building External Streams in C++ Designs

Use the scc_stream class to define blocking or non-blocking streams in C++ designs. This method allows you to pass stream objects to the PPA function, making it easy to encapsulate and reuse code.

1. Define the stream using the scc_stream<> class.

   The tool instantiates a FIFO.

   See the *Synphony C Compiler Reference Manual* for details of the syntax.

2. In the driver code, specify how the FIFO is to be read and written.
   - If you know the amount of data at the input, use the put and get calls to preload input streams and read them back, respectively.
   - If you do not know the amount of data, do all the loading before calling the PPA and all the unloading after it completes. You can use the nb_can_get call to test the FIFO.
3. If you want to do other operations other than reading and writing while the PPA is running, do the following:
   - Create a new class that is derived from the scc_stream class.
   - Write your own put and get functions to achieve the behavior you want.
4. Follow these guidelines when simulating non-blocking streams:
   - For `verify -golden`, vary the calls to test different behaviors of the stream functions. The behavior varies with the actual data in the internal FIFO; therefore nb_can_get returns `true` if there is data and `false` otherwise. Derive a class from scc_stream and override the methods to mimic unavailability of data, even though the FIFO is preloaded.
   - When you run the `verify -rtl` command, mismatches between certain types of coding in the golden simulation and the RTL simulation might occur. One example is an application that counts the number of times a call to nb_can_get() returns false. There is no valid comparison between the two simulations. Therefore, in case of such a mismatch, run the golden simulation and debug it, and run the RTL simulation and debug it separately.

Building External Streams in C Designs

For C procedures, use the `scc_stream_input_` and `scc_stream_output_` prefixes. These procs must be part of the driver code and the PPA code must contain the corresponding function prototypes. You can only build blocking streams in C designs.

1. Use the internFIFO(<name>, <type>) to declare external streams. This macro is defined in the `scc.h` header file.

   The FIFO macro automatically generates the following streaming functions:

   ```
   <type> scc_stream_input_<name>(void)
   void scc_stream_output_<name>(<type>)
   ```

   These functions model an infinite FIFO for simulation; it is not synthesized. The tool synthesizes a FIFO for each external stream. Each FIFO is composed of flip-flop based registers.

2. Specify the number of FIFO entries by defining these pragmas:

   ```
   #pragma fifo_length scc_stream_output_<name> <size>
   #pragma fifo_length scc_stream_input_<name> <size>
   ```

   The default number of storage elements is 0. This indicates a simple boundary register between PAs, and a register to handle stalling cases. Increasing the length adds FIFO storage elements in addition to the boundary register.

3. Use the bitsize pragma to set the width of the streaming data input or output. Set the pragma on the input or output.

   - Check that the two filters are now running in parallel

Building Internal FIFOs

Internal FIFOs stream data between loops. You can also build a FIFO that passes data to and from the same loop. This requires both ends of the stream to be in the same loop.

The details of defining internal FIFOs in C++ and C designs are described:

- Building Internal Streams in C++ Designs
- Building Internal Streams in C Designs

Building Internal Streams in C++ Designs

Use the scc_stream class to define blocking or non-blocking streams in C++ designs. This method allows you to pass stream objects to the PPA function, making it easy to encapsulate and reuse code. See the *Synphony C Compiler Reference Manual* for details of the syntax.

1. For C++ designs, use the scc_stream class to define internal streams.
2. Use the put() and get() calls to write and read the stream data.

Building Internal Streams in C Designs

Use the following procedure to define streams in C designs. You cannot have internal non-blocking streams.

1. For C designs, use FIFO(<name>, <type>) to declare internal streams.

   The FIFO macro automatically generates the following streaming functions:

   ```
   <type> scc_stream_input_<name>(void)
   void scc_stream_output_<name>(<type>)
   ```

   These functions model an infinite FIFO for simulation; it is not synthesized. The tool synthesizes a FIFO for each internal stream. Each FIFO is composed of flip-flop based registers.

2. Specify the number of FIFO entries by defining these pragmas:

   ```
   #pragma fifo_length scc_stream_output_<name> <size>
   #pragma fifo_length scc_stream_input_<name> <size>
   ```

   See the *Synphony C Compiler Reference Manual* for details.

   The default number of storage elements is 0. This indicates a simple boundary register between PAs, and a register to handle stalling cases. Increasing the length adds FIFO storage elements in addition to the boundary register.

3. For FPGA designs, you can specify how to implement the FIFO with these pragmas:

   ```
   #pragma internal_fast <name>
   #pragma internal_blockram <name>
   ```

   See the *Synphony C Compiler Reference Manual* for details.

4. Use the bitsize pragma to set the width of the streaming data input or output. Set the pragma on the input or output.

   See the *Synphony C Compiler Reference Manual* for details.

5. To share FIFOs, do the following:

- Use the same stream name to share FIFOs across different loops. This introduces sequentialization.
- If you know your design, you can suppress sequentialization in the same task or across tasks with these pragmas:

```
#pragma no_inter_loop_stream_analysis <stream_name>
#pragma no_inter_task_stream_analysis <stream_name>
```

See the *Synphony C Compiler Reference Manual* for details.

Building Shift Registers

Your design can have sequential inner loops that represent functionality that should not be implemented iteratively in hardware, like the shifting of registers or summation across filter taps in a high performance FIR filter. The tool builds shift registers automatically, but you can guide the tool by eliminating ambiguities where the tool would not be able to make a deduction.

The following list some guidelines for coding shift registers:

- Wherever possible, try to code the register so that a constant width is evident to the compiler. This means that the constant is known at compile time.

  The SCC tool instantiates two types of shifters: variable amount and constant amount. Variable-amount shifters must implement a data path for every possible amount. Therefore, they can be expensive. By contrast, constant registers are usually low cost. This is an example of code that you can optimize:

```
x >>= shift_amount;
```

  If you know that this variable can only be 6 or 3, code the shift register explicitly to make this evident:

```
if(shift_amount==3)  x >>= 3;
else                 x >>= 6;
```

- Code the shift register so that the constant shifts are evident when loops are unrolled.

  The following example extracts bytes from a 32-bit word. If this loop is unrolled, it automatically generates shift-by-constant functional units, because 8*i is always constant after unrolling.

```
for(i=0;i<4;i++) {
     …
     byte = (word >> (8*i)) & 0xff;
     …
}
```

  However, if the loop remains iterative, code the shift register as shown in the following example. This code makes it explicit that there are only four possible values for i and they are all constant. The result is four shifts by constant and four extractions.

```
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;
…
}
```

Optimizing the Build Steps

You can code your design to make it easy for the compiler to go through its operations and implement hardware. The following table lists the main build steps in the design flow and the operations performed at each stage. It also lists other sections with more information about how you can influence these optimizations.

Operations	For more information, see...
Inlining Loop unrolling	Optimizing the Build Run
Perfectization Rate matching	Optimizing the Schedule Run Influencing Perfectization Flushing the Pipeline Optimizing Memories for Rate Matching
Hardware allocation Instruction scheduling RTL generation	Optimizing the Synthesize Run

Optimizing the Build Run

During the build stage, the tool automatically inlines all procedure calls in the PPA function, and unrolls all marked loops. You can influence the build stage by specifying the loops to unroll.

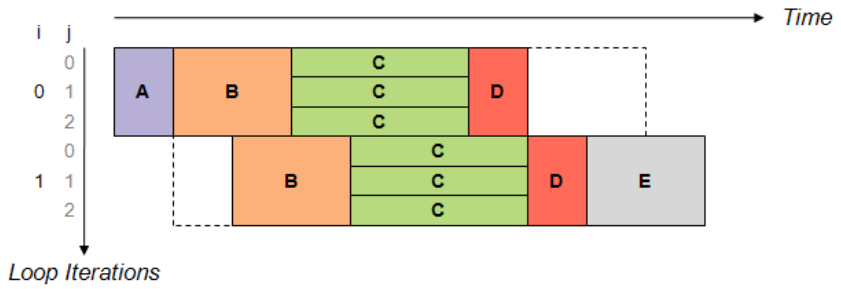
1. In the code, mark loops to be unrolled with the #pregame unroll pragma.

   Unmarked loops are not unrolled, and become PAs that are synthesized. The following code and Figure 8-12 shows the loops unrolled. The II applies to the iterative loop; therefore, you can usually increase performance by the unroll factor:

```
A;
for (i = 0; i < 2; i++) {
```

```
        B;
#pragma unroll
        for(j = 0; j < 3; j++) {
              C;
        }
        D;
}
E;
```

Figure 8-12 Loop Unrolling



1. Use inlined functions.

   Inlined functions can produce more efficient hardware because the code inside them can be optimized along with code outside it. However, they might increase runtime.

2. After the build stage, check the results by looking at the following files:

ppa_inline.c	Transformations after inlining
ppa_unroll.c	Transformations after unrolling `report_implementation -loop_unrolling`
Unrolling report	Generate the report with `report_implementation -loop_unrolling`. The unrolling report lists the loops that were unrolled.
ppa_build.c	All other build optimizations

Optimizing the Schedule Run

The following procedure shows how you can influence the operations that the tool automatically runs as part of the Schedule stage.

1. Influence the perfectization stage by doing the following:
   - Set the clock frequency, task iteration interval (Task_II), and initiation interval (II). The tool takes these settings into account when it perfectizes the code.
   - Optimize your code, as described in Influencing Perfectization.

2. Influence the results of rate matching by banking or multi-buffering memories.

   See Optimizing Memories for Rate Matching for details.

3. Generate reports and check the results of the schedule stage optimizations:

Perfectization	Open <*ppa*>_perfected.c, and check the perfectization results. Use report_ implementation -perfectized_loop_info command to generate a report, and check the summary perfectization report. To compare changes to the original, generate a report with report_ implementation -original_loop_to_line_number command, and check the line number for each loop in the original code, before perfectization.
Rate Matching	Generate a summary report with report_implementation -rate_match command and check the rate matching results.

Optimizing the Synthesize Run

The hardware allocation and scheduling operations the tool performs as part of the Synthesize stage are timing-driven.

1. Set timing goals by setting the clock frequency, task initiation interval, and initiation interval.
2. Check the generated RTL after synthesis.

Influencing Perfectization

The tool automatically performs perfectization during the schedule stage. Perfectization is the process of converting the code in the PPA procedure into perfect loop nests, while still meeting the specified task iteration interval. A perfect loop nest is one where all the code is inside the innermost loop and there is no code between consecutive for statements in the nest. For perfectization, the tool uses these two techniques:

- It moves code that is outside loop nests or between loops to the innermost loop.
- It wraps out-of-loop code into a separate loop nest with a single iteration.

The following describe how you can affect the perfectization process:

- Manually Perfectizing Code
- Code Sinking Examples

- Loop Unrolling Example
- Loop Distribution Example
- Iteration Space Extension Example

Manually Perfectizing Code

The tool automatically selects the best implementation for your code and performance target, but you can manually perfectize your code by using any of the techniques described in this procedure:

1. Check results in `Reports/report_original_loop_to_line_number.txt` and `Reports/report_perfectized_loop_info.txt` to determine what to manually perfectize.
2. Manually add single iteration outer loops.
3. Use any of the four hand-coding techniques described in the following table to override the automatic perfectization process or specify your choices.

Technique	Pros	Cons
Code sinking (Code Sinking Examples)	Easiest to do Best for simple operations	Most expensive hardware Could lead to underutilized resources Could change code semantics Could sequentialize loop nests and affect performance
Loop unrolling (Loop Unrolling Example)	Higher performance; good for expensive operations Best if innermost loop is small enough Resource sharing	Not always possible Higher cost
Loop distribution (Loop Distribution Example)	Good for expensive operations Performance targets set per loop	Additional hardware overhead for streaming and control No resource sharing across loops Not always possible because of data dependencies

Technique	Pros	Cons
Iteration space extension (Iteration Space Extension Example	Good for expensive operations	Analysis and rewriting

See the subsequent sections for examples that use the techniques listed in the table. Some examples see the following imperfect loop nest, which has a task iteration interval of 4000:

```
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;
}
```

Code Sinking Examples

With code sinking, you move imperfect out-of-loop code into the innermost loop, while maintaining the semantics of the original code. This technique is best suited to simple operations like shift, mask, add, subtract, and compare, where it is also the most cost-effective.

Example of Effective Code Sinking

If you apply code sinking to the example in Manually Perfectizing Code, you move two statements to the beginning of the inner loop under the j==0 condition, and one statement to the end of the inner loop under the j==3 condition.

```
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;
          }
     }
}
```

Example of Ineffective Code Sinking

Do not use code sinking where it affects performance by sequentializing shared data. Take the example of a PPA with two loop nests c0 and c1 that run in parallel and are computed after a certain number of cycles.

```
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);
}
```

You can sink the out-of-loop code and perfectize the first loop as shown in the following example, but this causes the first loop to define c1 and the second loop to use it. This sequentialization can make it hard to achieve your performance targets.

```
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);
}
```

Loop Unrolling Example

Loop unrolling replaces the original loop with one copy of the loop body per iteration. Any dead code created by unrolling is eliminated. Loop unrolling reduces iterations by exposing parallelism at no additional cost. This technique works best when the performance requirements and trip counts after each unrolling result in a small initiation interval. Ideally, the II should be between 1 and 8.

If you apply the loop unrolling technique to the example in Manually Perfectizing Code, you get the following:

```
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;
}
```

Task_II is set to 4000, which translates to four cycles per iteration. As a result, the tool time-multiplexes the nine multiply operations into three multipliers. This is a tremendous savings over the five or six multiplier result achieved with code sinking.

Loop Distribution Example

With the loop distribution technique, you create a new perfect loop nest for the imperfect code. To enable parallelism, you also need a mechanism for the loop nests to communicate. Unlike loop unrolling where you can share resources, there is no resource sharing across loops. However, you can set different performance targets for each loop.

If you apply loop distribution to the example in Manually Perfectizing Code, you get the two loop nests shown in the following example. Note that this code also includes 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 j==0 and j==3 conditions:

```
// 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 = scc_stream_input_x();
          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 * scc_stream_input_z();
}
```

Task_II is set to 4000, which translates to four cycles per iteration for loops 0 and 2, and one cycle per iteration for loop 1. This implementation results in three multipliers, one each for loop 0 and loop1 (100% utilized), and one for loop 2 (25% utilized). This is the same result as loop unrolling, with the streaming and control overhead.

Iteration Space Extension Example

In some cases, the other techniques do not offer effective solutions. Take the example of algorithms like FFT or DCT that perform frequency/time or frequency/spatial domain transformations. They often handle DC and AC components differently, sometimes coding this as a first step for DC and then a loop for AC:

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

If encode_DC and encode_AC share a memory or common operations like multiplications, neither of the two solutions the tool automatically considers work.

- If it uses loop distribution and wraps the out-of-loop code in two PAs, each PA requires separate load and store units, multipliers, and ports.
- If the tool sinks the out-of-loop code, it is inefficient in terms of memory access calls, multiplications, and ports.

In this case, the best thing to do is to manually modify the code and extend the iteration space. With the following coding, there is just one load, one store, and one multiply for each iteration, and these resources can be shared between the encode_DC( ) and encode_AC( ) functions because the calls are now mutually exclusive:

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

Flushing the Pipeline

To achieve high performance, Synphony C Compiler builds a pipeline of operations. At any given cycle, there could be many loop iterations running. Each pipeline is in its own stall domain and any stall condition causes the whole pipeline to stall. If there are constraints on the external environment, you might have to flush the pipeline of old iterations before starting an operation that might cause a stall.

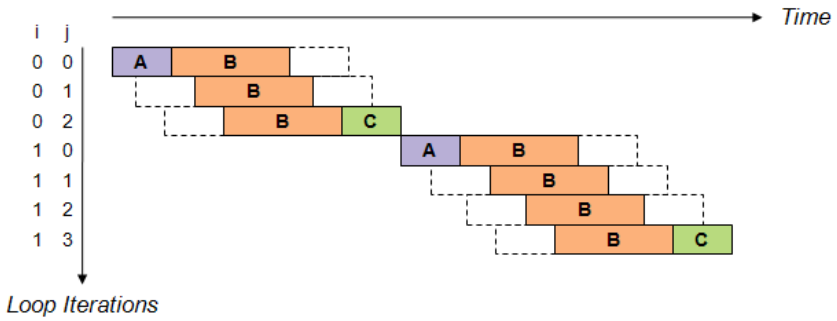
1. To flush a pipeline, add the #pragma flush pragma to the loop in your code:

   Figure 8-13 shows that the loop is flushed between outer loop iterations as specified in this code:

   ```
   for (i = 0; i < 2; i++) {
        A;
   #pragma flush
        for(j = 0; j < i+3; j++) {
             B;
   ```

```
        }
        C;
    }
```

Figure 8-13 Loop Flushing



2. Run RTL simulation and check the throughput results.

   You must do this to check the throughput because the compiler does not consider flush cycles when it determines the IIs for each loop to achieve throughput.

The following are examples of scenarios where pipeline flushing is useful:

- Flushing the Pipeline to Avoid Stalls
- Flushing the Pipeline to Relax Feedback Constraints
- Flushing the Pipeline to Enable Resource Sharing

Flushing the Pipeline to Avoid Stalls

In the following example of a 720p video (720x1280 pixels), the design has a one-tap filter that outputs results to the next block in a video pipeline. The input is received in bursts of 1280 consecutive values, Since the inner loop is not flushed before the next iteration of the outer loop, the input stream could stall waiting for the next burst of 1280 values.

```
for(rscc_streamrows=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 following shows the code with #pragma flush added to the inner loop to ensure that the pipeline is flushed before the next iteration of the outer loop:

```
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); }
    }
```

Flushing the Pipeline to Relax Feedback Constraints

The following loop has an II >=3, because of the latency of the multiplier. When j==0, the result of multiplication is used in the addition. This means that addition must be scheduled at least three cycles after the multiplication. When j==999, the result of the addition is used in the multiplication of the next iteration.

```
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();
        }
    }
```

You can relax this constraint on pipelining in the outer loop by flushing the pipeline at the end of inner loop iterations with #pragma flush, as shown in the following example. The tool introduces the idle cycles needed to completely flush the pipeline and get II=1 for the loop:

```
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();
    }
}
```

Flushing the Pipeline to Enable Resource Sharing

Take the example of algorithms like FFT or DCT that perform frequency/time or frequency/spatial domain transformations. They often handle DC and AC components differently, sometimes coding this 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);
    }
}
```

Assume that encode_DC and encode_AC share a memory with the DC value written to element 0 and AC values to elements 1-63. Also assume that encode_DC and encode_AC each do one read and one write to the shared memory, and that each performs one multiplication, and that II is set to 1 for this loop. This would require four ports on the memory,

because there would be four accesses to memory for encode_DC and encode_AC and two multiplications.

However, if you flush the pipeline with #pragma flush as shown in the following example, you only need one load, one store, and one multiply on each iteration, and the resources can be shared by encode_DC and encode_AC.

```
for (blocks=0; blocks < nBlocks; blocks++)
     encode_DC();
     #pragma flush
     for(i=1;i<64;i++) {
          encode_AC(i);
     }
}
```

Optimizing Memories for Rate Matching

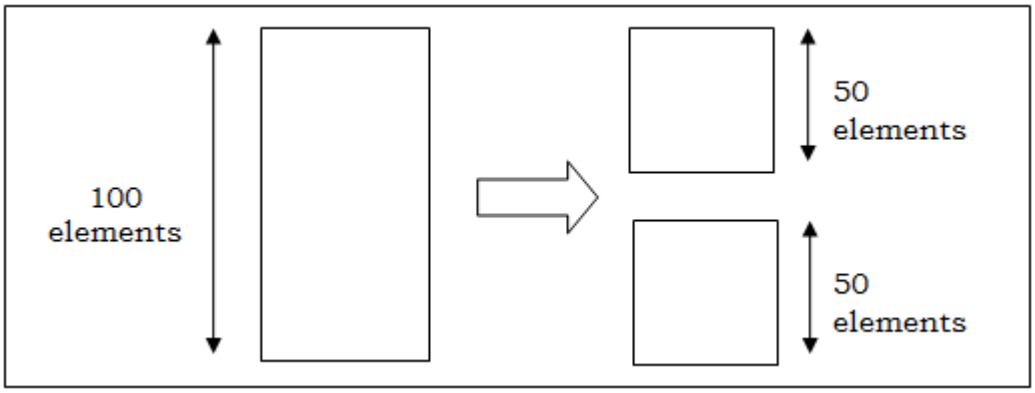
As part of the schedule run, the tool performs rate matching. After perfectization is complete, rate matching determines the relative rates or initiation intervals (II) of all loops in the design. The following techniques influence rate matching because you try to minimize dependencies and reduce sequentialization.

- Banking Memories
- Multibuffering Memories

Banking Memories

Banking is where the tool automatically splits a single multi-dimensional array into multiple physical structures to reduce the porting cost, as shown in Figure 8-14. Banking improves bandwidth and performance by identifying and modifying code that would otherwise generate multi-port SRAMs.

Figure 8-14 Banking Memories



If you do all of the following, the tool automatically generates banked memories during the rate matching stage of the schedule run:

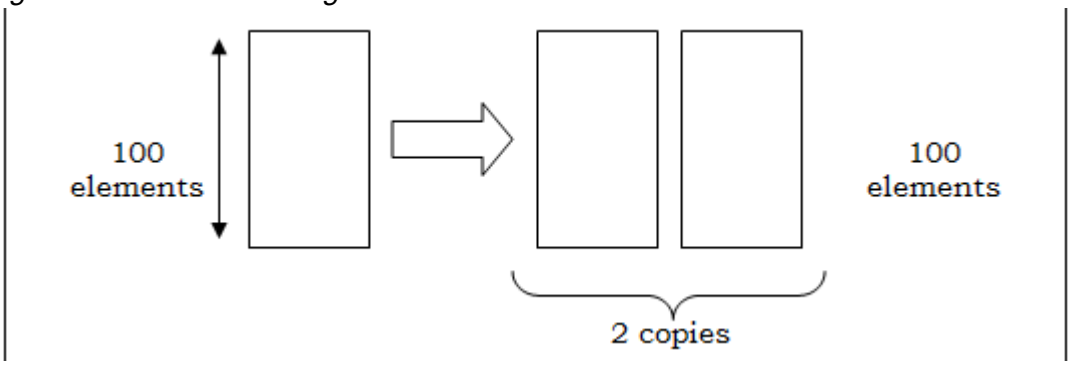
1. Make the desired number of banks the extent of the first array dimension.
2. The tool generates two banks for A in the following code:

   ```
   #pragma memory_structure A bank,
   ```

3. To prevent banking, specify `#pragma preserve_array` in the code.

Multibuffering Memories

Figure 8-15 Multibuffering Memories



Multi-buffering is a technique that minimizes task sequentialization by making copies of shared arrays so that task dependencies can be eliminated. It enables more task overlap by storing the shared arrays in different physical locations for subsequent tasks to use. The tool creates these copies automatically.

1. Manually direct multi-buffering by declaring #pragma multi_buffer on an array in the code.

   See the *Synphony C Compiler Reference Manual* for details.

2. If you set a low task II, the tool might create a multi-buffer to meet the performance requirement.

Optimizing for Area

If area is your primary focus, you can use the following optimizations to make sure you achieve the best results.

1. Use local optimizations.
   - Check the bitsizes of major functional units and registers.
   - Examine the C code for anomalies. In particular, check whether the tool inferred registers instead of wires, or if multiple registers are mapped to the same variable.
   - Check for high cost units like dividers, multipliers, and shifters.
   - Use constant specification where possible.

   For additional information about local optimizations, see the following:

Bit Sizing	Optimizing Bit-Widths
Register inference	Inferring Registers
Constant specialization	Specifying Constants

2. Optimize memory.
   - Check memory properties like size, porting, buffers, and arbitration.
   - If necessary, minimize array references using common sub-expression elimination (CSE).

   For additional information about memory optimizations, see the following:

Memory optimization	Banking Memories Multibuffering Memories Optimizing Local Memories
C to RTL mapping	Mapping C Arrays to Memories in ASIC Designs Mapping C Arrays to Memories in FPGA Designs Specifying Memory Properties

Memory properties	Specifying Memory Properties

3. Share hardware resources.
   - Use mutual exclusion coding.
   - Increase II when performance allows it.
   - Move shared hardware into the same loop by jamming sequential loops or merging parallel loops.
   - Use hierarchical blocks.

   For additional information about resource sharing, see the following:

Mutual exclusion	Exploiting Mutual Exclusion
initiation intervals	Setting Initiation Intervals and Task Iteration Intervals
Loop jamming	Jamming Sequential Loops
Loop merging	Merging Parallel Loops
Hierarchical blocks	Chapter 7, “Building Hierarchical Designs”

Optimizing Bit-Widths

When you specify bit-widths, the tool uses the information to reduce the size of registers, memories, FIFOs, data paths, and so on, thus reducing area. The tool also dynamically checks for overflows when you specify bit size and run lint simulation. The original code has pure C or C++ behavior and does not include overflow checking. The Synphony C Compiler software uses declarations and directives to optimize bit-widths.

1. Analyze your design and check bitwidths.
   - Run lint simulation and check for overflow and underflow warnings.
   - Check the Resource Browser for bitwidth over estimation.
   - Check interfaces, internal FIFOs and state variables, and add #pragma bitsize where needed, as described in the following steps.
2. Specify an upper bit-width limit for memories, streams, input and output scalars, internal FIFOs and state variables.

If you do not specify bit-widths, the tool uses the following C/C++ data type defaults as the bit-width limits:

char	8 bits
short	16 bits
int	32 bits
long long	64 bits

3. Specify bit-width limits in the following ways:

Per variable	#pragma bitsize *variable width* (*variable* can be a scalar, array, stream, or procedure name)
By type definition	#pragma bitsize u1 1 (typedef unsigned char u1)

See the *Synphony C Compiler Reference Manual* for details.

You can also use `s_int<>` and `s_uint<>` to force bitwidth behavior.

The tool optimizes bit-widths of variables and constants according to the limits you set. It propagates bit-widths where possible. It always preserves the C/C++ semantics.

This pragma dynamically checks for overflows when the original code has pure C behavior, and is verified during the lint run. If you apply pragma bitsize to an array that is later scalarized, the pragma is automatically propagated to the scalarized elements.

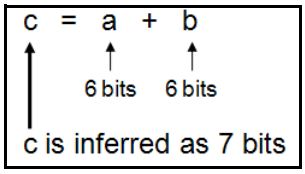
4. To optimize bit-widths of registers and functional units, use the following declarations and directives in the source code:
   - Use C native data type bitwidths as the upper bound (char-8, short-16, int-32, long long-64).
   - Use `#pragma bitsize` to set the upper limit. This pragma is verified during the lint run.
   - For literal constants, use minimal 2's complement representation.

   The tool automatically propagates bit-widths throughout the design. It propagates the bit-width of operands forward, backward, and through operators. See the following section, "Examples of Bit-Width Propagation."

Examples of Bit-Width Propagation

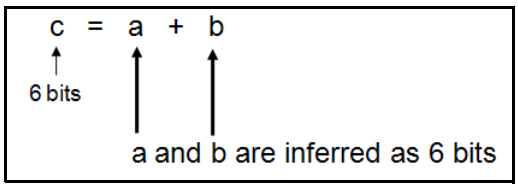
Forward Propagation

```
unsigned int a, b, c;
#pragma bitsize a 6
#pragma bitsize b 6
c = a + b;
```



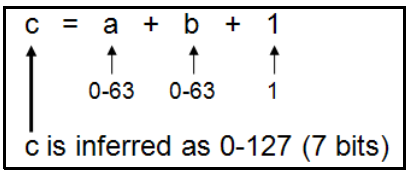
Backward Propagation

```
unsigned int a, b, c;
#pragma bitsize c 6
c = a + b;
```



Range Propagation

```
unsigned int a, b, c;
#pragma bitsize a 6
#pragma bitsize b 6
c = a + b + 1;
```



Inferring Registers

The compiler tries to eliminate the need for registers, but it might infer one if the code implies state.

1. Check the report_register.txt files to see the registers instantiated by the tool.
2. For the inferred registers, ensure the following:
   - Cover all cases by making sure that all if-else statements are complete or include a default statement in the switch case.
   - Make sure all variables are declared within the scope of the loop iteration.

     The compiler infers registers for the following example, because the int z variable is defined outside the loop, and the if-else assignments are incomplete.

```
int b[10]
int c[10];
void ppa(unsigned char type) {
     int i;
```

```
        int z;
        for(i=0; i<10; i++) {
            if (type == 0)
                z = b[i];
            else if (type == 1)
                z = -b[i];
            c[i] = z;
        }
    }
```

You can fix it by doing either of the following:

Make the assignment to z unconditional, as with this else statement:

```
    if (type == 0)
        z = b[i];
    else
        z = -b[i];
```

Move the declaration of z into the loop iteration:

```
    for(i=0; i<10; i++) {
        int z;
        if (type == 0)
            z = b[i];
        else if (type == 1)
            z = -b[i];
```

Specifying Constants

The SCC tool automatically propagates and reduces constant expressions. Therefore, use constants where possible.

1. To introduce constants in builder macro cells, do either of the following steps:
   - In the source code, set `#define MODE 2`.
   - In the Parameters dialog box select the Verification tab, and then select the Global Settings tab. Set C Compiler to `-g - DMODE 2`.
2. Unroll loops with `#pragma unroll`.

   Unrolling makes many variables into explicit constants, and this helps reduce area.
3. Reduce the cost of expensive units by explicit coding.

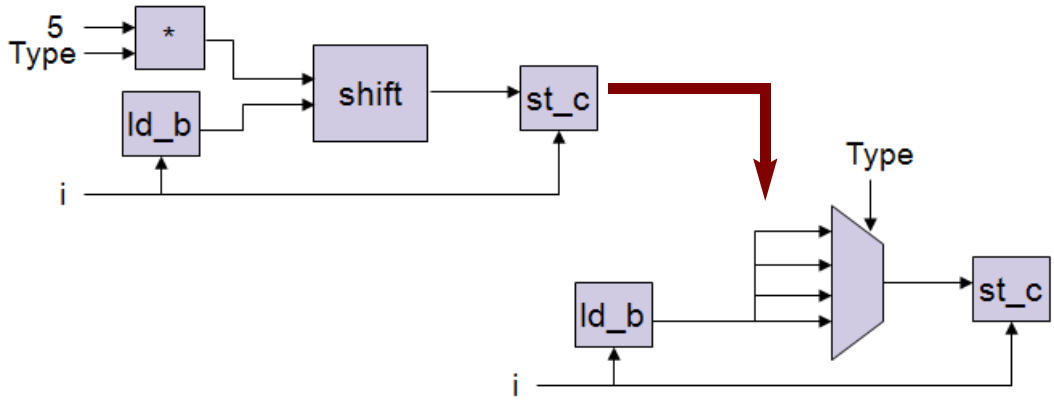
   The following code implies a variable shifter:

```
long long b[10];
long long c[10];
void ppa(unsigned char type){
#pragma bitsize type 2
    int i;
    for(i = 0; i < 10; i++) {
        c[i] = b[i] >> (5*type);
```

```
        }
}
```

However, only a small number of shifts is possible. You can rewrite each option as a constant shift, as shown in the following example, to minimize the cost of the shifter.

Figure 8-16 Minimizing the Cost of the Shifter




```
#include "scc.h"
long long b[10];
long long c[10];
void ppa(unsigned char type){
#pragma bitsize type 2
     int i;
     for(i = 0; i < 10; i++) {
          switch(type) {
          case 0: c[i] = b[i] >> 5*0; break;
          case 1: c[i] = b[i] >> 5*1; break;
          case 2: c[i] = b[i] >> 5*2; break;
          case 3: c[i] = b[i] >> 5*3; break;
          default: scc_assert(0);
          }
     }
}
```

4. Use banking to reorganize memory and reduce area cost. See Banking Memories for details.

Optimizing Local Memories

The tool automatically runs advanced optimizations on local arrays that are implemented as internal memories: scalarization, banking, and multi-buffering. You can influence these automatic optimizations for all memories.

Scalarization is an optimization where the array elements are implemented as individual registers. This provides the tool more flexibility to eliminate the individual elements, optimize

them for bit size, and schedule them independently. By contrast, an internal_fast memory is treated as a single array, and is never scalarized.

- Scalarize arrays when the array is intended to convey a combinational or wiring function rather than a state.
- Scalarize arrays when the array describes an extraction and multiplexing function rather than a state.

See Comparing Scalarization and internal_fast Memories for a discussion of the advantages of scalarization.

The Synphony C Compiler tool automatically scalarizes local memories based on its internal heuristics to reduce area. These heuristics depend on the size of the array and the number of accesses to the array that are constant at compile time after inlining and unrolling. Specify `scalarize` pragma to control the scalarization of the array.

If a pragma is applied to an array name that the tool scalarizes, it gets automatically propagated to the scalarized elements. For example, if you set `#pragma bitsize` on the array name, each scalarized element of the original array has the same pragma applied to it.

For banked memories, set up the code as described in Banking Memories.

Specify multi-buffered memories with `#pragma multi_buffer`. See Multibuffering Memories for details.

Note:
Setting `#pragma preserve_array` on the array name prevents the tool from scalarizing or banking the array. If you set this pragma on a ROM array, the pragma disables the constant substitution of initialization values and bit resizing optimizations that are based on initialization values.

Comparing Scalarization and internal_fast Memories

Scalarization has some advantages and disadvantages when compared to internal_fast memories:

Scalarized arrays	internal_fast memory
Completely separates the components; therefore, they have no common dependences and can be analyzed independently.	References must be analyzed along with other accesses to the memory.
Can be implemented as a wire if the scalarized variable is accessed inside only one loop nest, and the physical implementation does not require storage.	Always implemented in registers.

Scalarized arrays	internal_fast memory
Automatically support task overlap if scalarized arrays are shared across loops.	Do not support task overlap unless the arrays are multi-buffered.
Duplicated storage if scalarized arrays are shared across loops, with one output register per writer loop and one input register per reader loop.	Only requires the original array to be implemented in registers.

Minimizing Memory Ports

The number of memory references can affect the cost and performance of the design. Multiple ports are expensive; therefore, the tool tries to minimize porting using automatic techniques like reducing the number of references and sharing ports across sequential loops. The following procedure describes manual techniques you can use to reduce multi-porting.

1. Assist common sub-expression elimination (CSE).

   The tool runs CSE routines automatically to remove multiple references with the same array indexes, but you can manually fix coding where the tool would not be able to prove that the accesses are the same.

2. Concatenate memories into wider memories using #pragma packed.

   Packing might reduce porting requirements. The tool analyzes array access patterns and determines the bandwidth required to meet the performance. In some cases, the required bandwidth might be lower if the arrays are made wider. See Mapping C Arrays to Memories in FPGA Designs for more information about packing memories.

3. Specify arbitration for ports.

   - Run event simulation with the esim command. See the *Synphony C Compiler Reference Manual* for details about the command.
   - Determine the best porting for the performance you require.
   - Specify the porting constraints with `#pragma read_write_ports`. See the *Synphony C Compiler Reference Manual* for more information.

   Arbitration is required when there are more access requests than the number of ports available. The tool uses a fixed priority arbitrator and stalls the stall domains of requestors that did not get access to memory.

Exploiting Mutual Exclusion

The SCC software generates parallel hardware if it can prove that the conditional blocks are mutually exclusive. This can improve timing on designs with mutually exclusive conditional blocks. You can assist the tool by coding to make mutual exclusion explicit, as described:

1. Avoid overlapping conditions to make parallelism explicit.

   For the following example, only one of the conditions `(0==x), (1==x), (2 ==x)` is true on a given iteration. The tool can evaluate these conditions in parallel instead of sequentially, because there are no overlapping conditions.

   ```
   for (i = 0; i < 10; i++) {
        if (0 == x) {
             a[i] = b[i];
        }
        else if (1 == x) {
             a[i] =c[i];
        }
        else if (2 == x ) {
             a[i] = d[i];
        }}
   ```

2. Use structured if-else or switch-case coding. Make sure of the following for switch-case coding:

   - All case labels must be at the highest level inside the switch statement. Do not enclose the case statements of a switch-case block in an if-block or any other block.
   - End case blocks with an unconditional break. If there is no break statement, the tool might not be able to prove that the operations are mutually exclusive.

   This is an example of a mutually exclusive switch case statement:

   ```
   switch (x) {
        case 0: A; break;
        case 1: B; break
   }
   ```

3. Use full case statements.

   A full case statement is one where at least one case statement executes. Full case is most effective for variables that are defined in every block of the switch statement. The following example has a default defined; therefore, one case statement always executes:

   ```
   switch (x) {
        case 0: A; break;
        case 1: B; break
        case 2: C; break;
        case 3: D; break
        default: break;
   }
   ```

In some cases, using full case statements is not enough to make parallelism explicit. You might also have to specify reachability, as described in the next step.

4. Specify reachability using the `scc_assert` construct.

   By specifying reachability, the `scc_assert` construct tells the tool whether a particular block can be optimized. In the following example, y can be optimized because the default block is unreachable:

```
#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];
}
```

   You can also use the scc_assert construct with if-then-else statements in which one of the blocks is unreachable. For example:

```
#include "scc.h"
if(i==0) {
     packetlen = scc_stream_input_plen2();
}
else if(i == (packetlen_delay>>4)) {
     packetlen_delay = 32;
}
else if (i == 1) {
     packetlen_delay = packetlen;
}
else scc_assert(0);
```

Jamming Sequential Loops

Loop jamming or loop fusion is a technique for combining smaller sequential loops into one bigger loop, thus reducing loop overhead. The larger loop also provides the compiler with more opportunities for optimization. Usually, this approach works best on for loops, where the loop indexes are constrained in the iteration statement.

The tool automatically jams loops that are inside a TCAB. You can specify loop jamming for loops that are not in a TCAB with an option to the build command.

The following example shows a candidate for loop jamming, where there is only one addition operation in each loop:

```
for(i=0;i<N;i++) {
```

```
     a[i] = b[i] + c[i];
}
for(i=0;i<N;i++) {
     d[i] = a[i]*( e[i] + 1.0 );
}
```

This is the same example after loop jamming, which allows resources to be shared:

```
for(i=0;i<N;i++) {
     a[i] = b[i] + c[i];
     d[i] = a[i]*( e[i] + 1.0 );
}
```

Merging Parallel Loops

Merge parallel loops into a single loop to allow resource sharing in mutually exclusive operations. Merging loops increases the amount of work per iteration (granularity) and reduces loop overhead. Loop merging increases reuse. The following is an example of loop merging:

Before Merging

```
for( ) {
     ...
     A( );
     ...
}
for( ) {
     ...
     B( );
     ...
}
```

After Loop Merging

```
for( ) {
     ...
     A( );
     B( );
     ...
}
```

The following are some guidelines for merging parallel loops:

- As a rule of thumb, the best candidates for parallelization are loops with high iteration counts.
- For multiple nested loops, choose the outermost loop that is safe to parallelize. This approach generally gives the coarsest granularity.
- Ensure that work can be evenly distributed to each thread. If this is not possible because the outermost loop has a low iteration count, an inner loop with a large iteration count might be a better candidate for threading.

Optimizing Performance

There are two levels of performance problems, the block level and the operation level:

- Meeting Throughput Goals
- Resolving Memory Bottlenecks
- Analyzing Feedback Paths

The first two topics in the list describe block-level performance issues; and the third describes operation-level issues.

Meeting Throughput Goals

The following procedure discusses how to meet your throughput goals:

1. Check the throughput.
   - Use the loop sequentialization graph to identify sequential and parallel loops at the block or loop nest level.
   - Check the PPA graph to identify shared resources and dependencies.
   - Check the summary report for the schedule length and the total cycles for each loop, and for information about pipelinability and latency for TCABs.
   - Use the performance charts to check loop and task overlap and identify stalls.
2. Code the loop nest to enable desired performance.
3. Code to support your parallelism strategy:
   - Set the task size and task overlap, if required
   - Determine whether to use streams or memory connections.

Resolving Memory Bottlenecks

There are various memory issues that can degrade performance. This section lists some of the causes that you can try and avoid them when you architect your design:

- Check for single-port memories with multiple memory accesses. This can cause performance bottlenecks.
- Try not to use large local memories, because they are inefficient in the hardware implementation.
- Check shared arrays, because they might result in unnecessary sequentialization.
- Eliminate false bottlenecks like the following:
  - Use common sub-expression elimination to remove redundant accesses.

- Check if conditional memory accesses are causing false high bandwidths.
- Examine unanalyzable indexes to see if they are causing false dependencies.

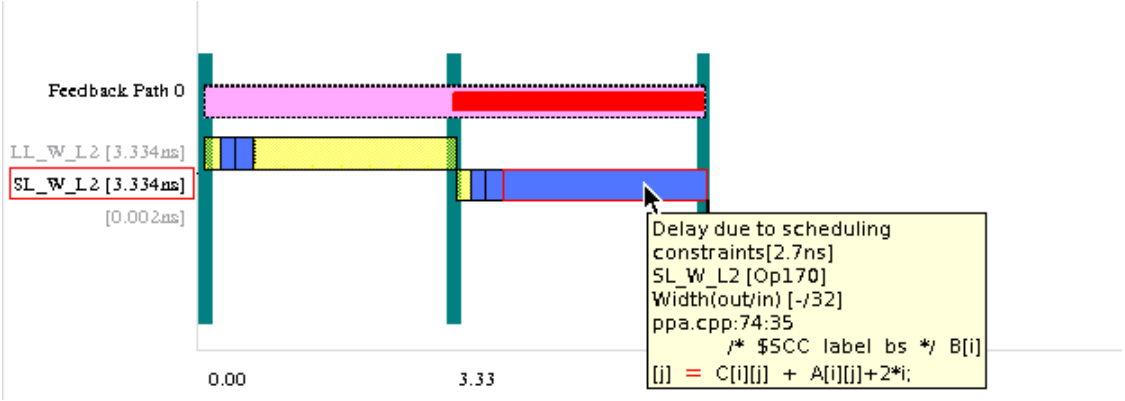
- Address bottlenecks by increasing bandwidth. You can use intra-task multi-buffering or use virtual dual porting.

Analyzing Feedback Paths

A feedback path is cyclic timing path between iterations of a loop nest. It can span one or more iterations; this is called the feedback path depth.

1. Analyze the feedback using the following reports and views and identify the critical feedback paths:
   - In the GUI, open the Feedback Path Viewer by clicking the icon.

Figure 8-17 Report Feedback Path

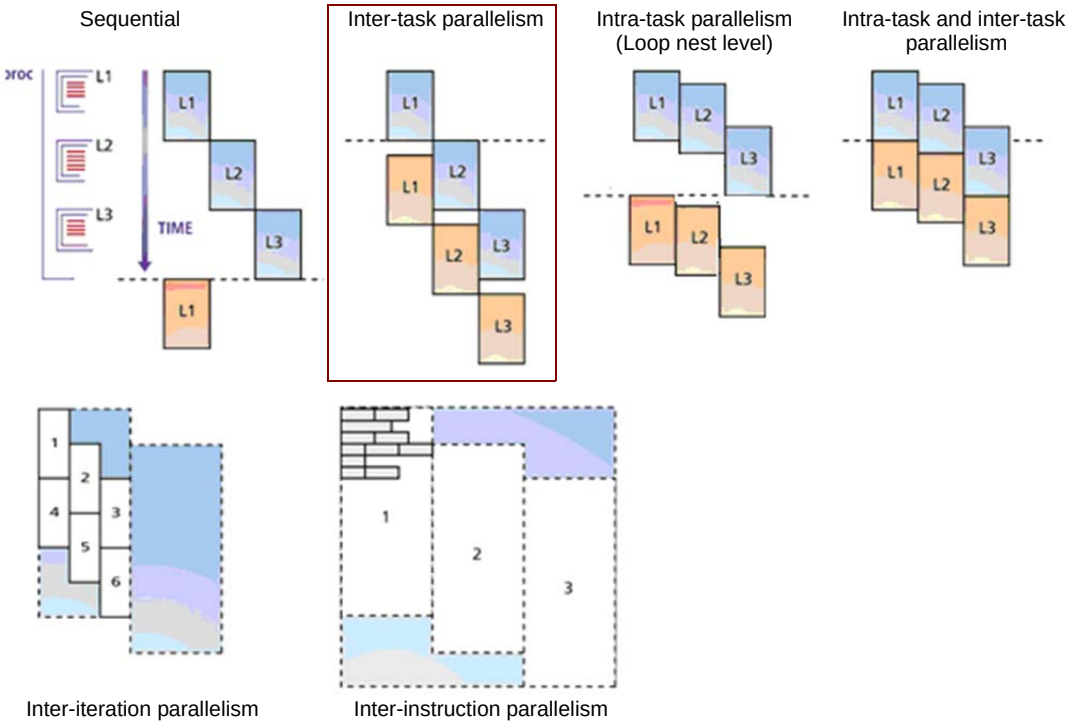


   - Check the `report_feedback_path.txt` report.
2. Use the following techniques to address feedback path issues:
   - Relax constraints.
   - Reset the clock frequency to a lower value.
   - Recode your design.

Coding to Maximize Parallel Optimizations

To improve performance, the Synphony C Compiler tool leverages parallelism at different operational levels. The following figure shows parallel optimizations at different levels. For most of the levels, the tool runs automatically to optimize the design. The only one you can influence is inter-task parallelism, and you do this by defining task overlaps, which is the maximum number of concurrent tasks the tool can run.

Figure 8-18 Coding for Parallel Optimization



You can code your design to take advantage of the parallelizing transformations automatically executed during the Synphony C Compiler run. The following are some tips:

- Use loop nests
  Use procedures with one or more compute-intensive loop nests with large iteration spaces. These yields better results than procedures with straight-line code with no loops or many small loops.
- Reduce stall cycles
  Loop nests communicate with each other through streams, that are implemented as FIFOs in the hardware. The tool does not automatically size FIFOs, but you can restructure the code to reduce the number of stall cycles. You can also increase FIFO size with fifo_length programs.
- Avoid sequential semantics

- Do not read from or write to the same stream in multiple loop nests (stream port sharing), as this practice ties up the stream port resource. This restriction does not apply to point-to-point FIFO streams that are written from one loop and read in another loop.
- You can share input scalar variables for read-only access across loop nests, because this does not sequentialize the loop nests. However, the tool automatically inserts buffering if you read a scalar variable in one loop nest and read/write it in another.
- You can share memory arrays for read-only access across loop nests because this does not sequentialize the loop nests. However, if you write a memory array in one loop nest and read/write it in another, the tool sequentializes these loop nests so that the latter loop nest finishes its accesses to the current task before the next task overwrites the array. For locally declared memory arrays, you can specify multiple buffers (#pragma multi_buffer), and the tool automatically generates the associated hardware to manage access to the buffers efficiently.

- Managing persistent state across tasks
  Do not use globally scoped scalar variables that translate to input or output scalar configuration registers. At the global level, declare small data arrays that are accessed with constant indexes. These array variables are accessible within the hardware just like scalar variables and would preserve state across tasks.

Loop Index Expressions

Synphony C Compiler accepts an extended collection of expressions involving loop indexes if they can be mapped to linear expressions, or in other words, there are no terms of the form `i*j` where `i` and `j` are loop indexes.

The syntax of the expressions is described as follows:

```
exp ::= loop-index
  | INT
  | INT * exp | exp * INT
  | exp + exp | exp - exp
  | exp / INT | exp % INT
  | exp & INT | INT & exp
  | exp '|' INT | INT '|' exp
  | exp ^ INT | INT ^ exp
  | exp >> INT | exp << INT
  | ~exp

pred ::= exp == exp
  | exp != exp
  | exp >= exp
  | exp <= exp
  | exp > exp
  | exp < exp
```

```
| exp || exp
| exp && exp
| !exp
```

where `INT` is a literal integer constant and `loop-index` represents a loop index variable.

The first four forms together generate the affine expressions. In addition, the tool accepts bitwise operators and bit-shifting if one of the operands is a literal integer constant, or in other words, the grammar indicates the position of the literal constant. For example, the tool recognizes that `ar[i % 2]` does not alias with `ar[1 - i % 2]` or with `ar[1 + i & 2]` in a particular iteration of the i-loop and that `ar[i & 0x3]` never aliases with `ar[i & 0xf4]`.

For affine expressions, the tool performs a form of symbolic analysis. A variable assigned in the loop can, in some cases, be used in an access expression or predicate expression. The tool constructs the affine expression from a syntactically nonaffine form. For example,

```
for (...) {
  int i1 = 3 * i + 2;
  ar[i1 + 4] = ...;
}
```

The expression `i1 + 4` can be reduced to an affine expression `3i + 6` by propagating the expression associated with `i1` to its point of use. The tool simulates such propagation without actually doing the transformation during analysis.

A limitation of the extended operator support is that this form of propagation is not available. The tool analyzes these expressions only when they involve the loop index variables or loop-invariant symbols.

Task Processing

In this section, the usability aspects such as constraint specification and relevant reports are explained. This is followed by an example that shows various task overlap and continuous processing scenarios.

See “Task Processing in PPAs” in the *Synphony C Compiler Reference Manual* for basic information about task processing and continuous processing.

Constraints for Task Processing

The following constraints apply to task processing:

1. Generally, you require a particular task_ii for your design (task_ii is the number of cycles between consecutive tasks). This can be specified using:

   `set_params -task_ii` *`number`*

Use the `-ii` specification for each PA as shown in the following, rather than specifying the task_ii.

The PPA that you design can have a single PA or multiple PAs. For each of the PAs, you can compute the initiation interval (number of cycles between consecutive iterations of the loop) based on the task_ii of the PPA. This can be specified using:

```
set_params loop_num -ii number
```

2. With the specified constraints, the Synphony C Compiler tool generates the hardware without any task overlap. The hardware can have only one task in flight and the next task starts after completion of the current task. If the task_ii that you require needs task overlap, enable task overlap using:

```
set_params -task_overlap always
```

3. For task overlap to happen, your design might need to have multi-buffering of memories in your design. You can specify the `multi_buffer` pragma on the memories that need to be multi-buffered. The `multi_buffer` pragma is explained in greater detail in the *Synphony C Compiler Reference Manual*.

4. After performing Step 3, you get inter-PA task overlap. See “Task Processing in PPAs” in the *Synphony C Compiler Reference Manual* for the different task overlap scenarios. If the performance goal (task_ii) of your design requires inter-PA and intra-PA task overlap, then you can design continuous processing hardware. This can be set using:

```
set_params -continuous_processing always
```

Synthesize Reports for 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 the build flow, you must 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_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_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.

RTL Simulation Reports

You can view the reports generated by running RTL simulation to gain an understanding of the throughput and dynamic performance achieved in hardware. You can also view the dynamic performance graph in the GUI to find out about the performance of your hardware.

The `report_verify_rtl_performance_summary.txt` report gives summary information about task ii and task latency for the PPA as well as each PA in the design. There are four values reported for task ii and task latency:

- `min`: The minimum value for task ii and task latency among the tasks run
- `max`: The maximum value for task ii and task latency among the tasks run
- `avg`: The average value for task ii and task latency among the tasks run
- `mode`: The most frequently occurring value for the task ii or task latency among the tasks run for the design

The `report_verify_rtl_performance.txt` file gives more details on the performance of the design for each task. For every task, it gives the start cycle and end cycle as well as the PAs in the design. It also reports the stall count for every PA in all the tasks. This report also gives information on potential FIFOs to be resized for achieving higher performance in your design. See “Task Processing in PPAs” in the *Synphony C Compiler Reference Manual* for an explanation of FIFOs in your design.

In some cases, you might be required to resize the FIFOs in the design to achieve a certain task_ii. You can do this by editing the *`PPA_name`*`_defines.v` file in the `rtl_package/rtl` folder.

Example of `report_verify_rtl_summary.txt`:

```
Delivered Task II (min/avg/max/mode): 64/67/67/67 cycles
 (16 sequential tasks)
Delivered task latency (min/avg/max/mode): 133/136/136/136 cycles
 (16 tasks)
Loop0 delivered Task II (min/avg/max/mode): 64/67/67/67 cycles
Loop1 delivered Task II (min/avg/max/mode): 67/72/128/67 cycles
Delivered throughput: 4225352 sequential tasks/second
Total PPA simulation duration: 1135 cycles
```

Example of `report_verify_rtl.txt`:

```
Task 14
=========
    PPA, started at cycle              935
    PPA, finished at cycle            1070
    Stall Count for task 14 :            9

    Loop0, task 14 started at cycle    935
    Loop0, task 14 stopped at cycle   1005
    Stall Count for Loop 0        : 3

    Loop1, task 14 started at cycle    999
    Loop1, task 14 stopped at cycle   1070
    Stall Count for Loop 1        : 6

*****************************************************

Task 15
=========
    PPA, started at cycle             1002
    PPA, finished at cycle            1134
    Stall Count for task 15 :            6

    Loop0, task 15 started at cycle   1002
    Loop0, task 15 stopped at cycle   1072
    Stall Count for Loop 0        : 3

    Loop1, task 15 started at cycle   1066
    Loop1, task 15 stopped at cycle   1134
```

```
    Stall Count for Loop 1        : 3

*****************************************************
Stall Contributions for PAs: Not available
Cycle Count for task[s] 0-15
total cycles including testbench:    11190
system_start,  PPA started at cycle     13
system_finish, PPA finished at cycle  1150
system_cycles,  PPA ran for   1138 cycles
```

RTL Simulation Problems

In some applications, there are deadlocks or performance might not be met because of insufficient FIFO sizes. Here are some general guidelines for debugging performance issues.

A description of the FIFOs used in your design is available in `Reports/ppa_all_fifos.pdf` and `Reports/ppa_sync_fifo.pdf`. See “Task Processing in PPAs” in the *Synphony C Compiler Reference Manual* for details. You can change the sizes of the FIFOs by editing the `<PPA_name>_defines.v` file in the `rtl_package/rtl/` folder.

To debug deadlocks in an offline or RTL simulation:

- After you run `verify -rtl` command, the tool generates report_verify_rtl_performance.txt. In case of deadlock the report mentions potential FIFOs to be resized in your design for avoiding deadlocks. Resize the FIFOs till you avoid deadlock and meet performance. You can resize the FIFOs by editing the `<PPA_name>_defines.v` file in the `rtl_package/rtl` folder.
- In some cases, simulation fails because of deadlock and gives the following message:

  ```
  ERROR: Empty Deadlock Detected. Please increase the pipelinability of
   the following loop(s) using synthesize_auxopt 'per_iter_task_ii_lb'
  ```

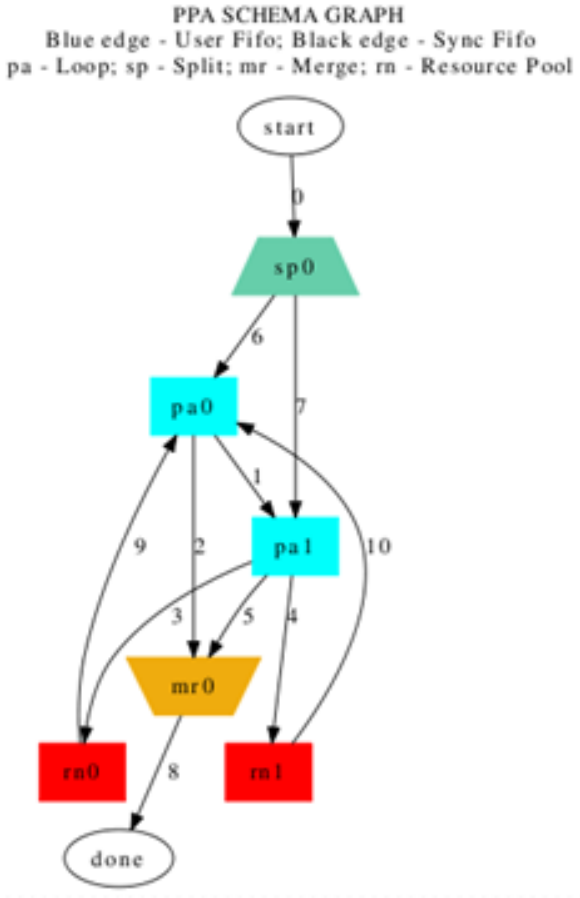
  This must be resolved by increasing the loop pipelinability. For details, see “Task Processing in PPAs” in the *Synphony C Compiler Reference Manual*.

Resizing FIFOs for Better Performance

To understand the FIFOs used in your design, see the `Reports/ppa_all_fifos.pdf` and `Reports/ppa_sync_fifos.pdf` files. These reports show the FIFOs used in the design. Some FIFOs could be user-defined and some could be generated by SCC and might correspond to a resource in the hardware. You can see the `report_pseudo_code.txt` file to understand the scheduling of the resources in your design. The FIFOs corresponding to these resources are scheduled according to the resource schedule. Alternatively, you can see the `Work/report_cp.txt` file to understand the FIFOs used in your design, the acquire-time, and release times of the resources in the design.

The following figure shows an example of `ppa_all_fifos.pdf` report for a design. For more information, see “Task Processing in PPAs” in the *Synphony C Compiler Reference Manual*.

Figure 8-19 FIFO Report Example



Example Design

In this section, the task processing of the PPA is explained using a matrix multiplication example. The application is done as a two loop design having PA0, PA1:

- PA0: Reads elements of the matrix, does scaling and level shifting of each element and stores them into two memories A, B
- PA1: Does matrix multiplication A*B and sends the output using streams

We take a 4x4 matrix and our goal is to get a task_ii of 64 cycles at 300 MHz. The code for the matrix multiplication is shown in the following figure.

PPA Code

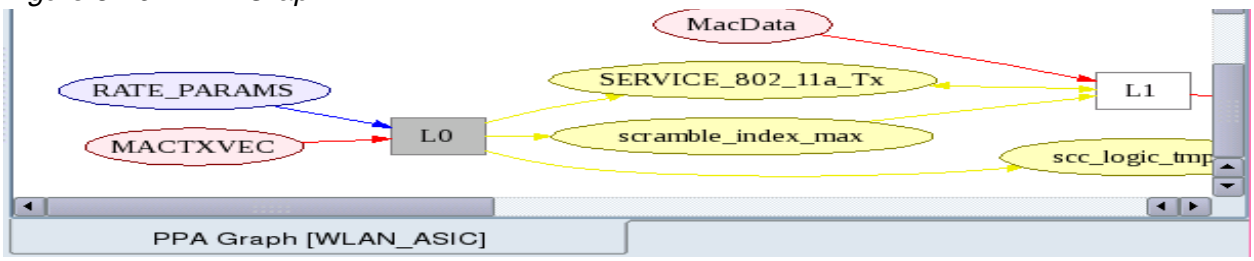
```
#define M 4
#define N 4
#define P 4

void MMul(
  scc_stream<int> & strA,
  scc_stream<int> & strB,
  scc_stream<long long int> & strC,
  char scale1, char scale2,  char shift)
{
  int A[M][N];
  int B[N][P];

  #pragma ii 4
  for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
      A[i][j] = scale1*strA.get()+shift;
      B[i][j] = scale2*strB.get()+shift;
    }
  }


  #pragma ii 1
  for (int i=0; i < M; i++) {
    for (int j=0; j < P; j++) {
     long long sum = 0;
      for (int k=0; k < N; k++) {
        sum  = sum + A[i][k] * B[k][j];
      }
     strC.put(sum);
    }
  }
}
```

Testbench Code

```
int main(void)
{
  scc_stream <int> strA;
  scc_stream <int> strB;
  scc_stream <long long int> strC;

   for (int i = 0; i < 16; i++) {
    cout << "A: " << endl;
    for (int i = 0; i < M; i++) {
      for (int j = 0; j < N; j++) {
         int aij = rand() & 0xFF;
         strA.put(aij);
         cout << aij<< "     ";
      }
      cout << endl;
    }

    cout << "B: " << endl;
    for (int i = 0; i < N; i++) {
      for (int j = 0; j < P; j++) {
          int bij = rand() & 0xFF;
              strB.put(bij);
          cout << bij<< "     ";
      }
      cout << endl;
    }

    MMul(strA,strB,strC,1,1,0);

    cout << "C: " << endl;
    for (int i = 0; i < M; i++) {
      for (int j = 0; j < P; j++) {
        int cij = rand() & 0xFF;
        cij = strC.get();
        cout << cij<< "     ";
      }
      cout << endl;
    }
  }
  return 0;
}
```

The PPA graph obtained for the matrix multiplication is shown in the following figure:

Figure 8-20 PPA Graph



The following subsections describe how this design can be implemented with no task overlap, with inter-PA task overlap, with inter-PA and intra-PA task overlap, and with resizing the FIFOs for better performance. Each successive implementation offers higher performance.

No Task Overlap

By default, SCC does not create hardware for task overlap.

The loop statistics and task processing section of report summary for this scenario is as shown:

```
Loop Statistics:
-------+----------+----------+------------+---------+-----------------
  Loop      II      Schedule     Target      Task     Pipelining Loss
  Name   Achieved    Length    Iterations   Latency    Between Tasks
-------+----------+----------+------------+---------+-----------------
   PA0          4          8           16       68*                6
   PA1          1          3           64       66*                3
-------+----------+----------+------------+---------+-----------------
* implies that the task latency is 1 more than the number shown due
  to registered output handshake.

Task Processing:
   Task Overlap : Yes
   Continuous_processing : No
   Reasons for PA Pipelining Loss Between Tasks :
     PA Task pipelining is turned off
     PA Scalar Input register optimization (1 cycle) :  PA0
     PA Output handshakes are registered (1 cycle)
```

For the example, you can see the performance after synthesizing your design in Dynamic Performance graph. This is shown in the following figure.

Figure 8-21 Dynamic Performance Graph

Items	Start	Stop	Stall
Task0	0	135	0
Task1	136	271	0
Task2	272	407	0
Task3	408	543	0

Inter-PA Task Overlap

The foregoing code generates single copies for memories A and B. To run the hardware with task overlap, you need to double buffer memories A and B. You can use multi_buffer pragma to get double buffering of A, and B:

```
int A[M][N];
#pragma multi_buffer A 2
int B[N][P];
#pragma multi_buffer B 2
```

Use the `set_params -task_overlap always` command to enable task overlap.

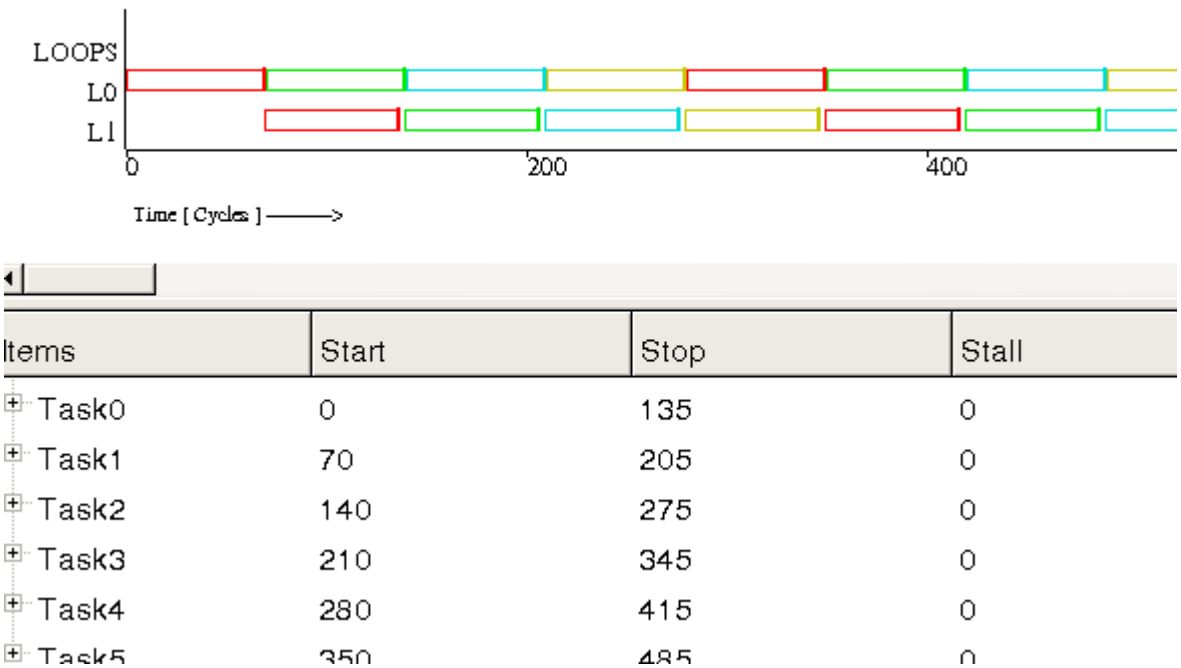
With the changes applied, the hardware generated by Synphony C Compiler tool is capable of running with task overlap. The loop statistics and task processing section of report summary for this scenario is as shown:

```
Loop Statistics:
-------+----------+----------+------------+---------+-----------------
  Loop      II      Schedule     Target      Task     Pipelining Loss
  Name   Achieved    Length    Iterations   Latency    Between Tasks
-------+----------+----------+------------+---------+-----------------
   PA0          4          8           16       68*                6
   PA1          1          3           64       66*                3
-------+----------+----------+------------+---------+-----------------
* implies that the task latency is 1 more than the number shown due
  to registered output handshake.

Task Processing:
```

```
Task Overlap : Yes
Continuous_processing : No
Reasons for PA Pipelining Loss Between Tasks :
  PA Task pipelining is turned off
  PA Scalar Input register optimization (1 cycle) :  PA0
  PA Output handshakes are registered (1 cycle)
```

The dynamic performance graph of the design with the specified changes is shown in the following figure.

Figure 8-22 Dynamic Performance Graph With Specified Changes



Items	Start	Stop	Stall
Task0	0	135	0
Task1	70	205	0
Task2	140	275	0
Task3	210	345	0
Task4	280	415	0
Task5	350	485	0

Inter-PA and Intra-PA task overlap

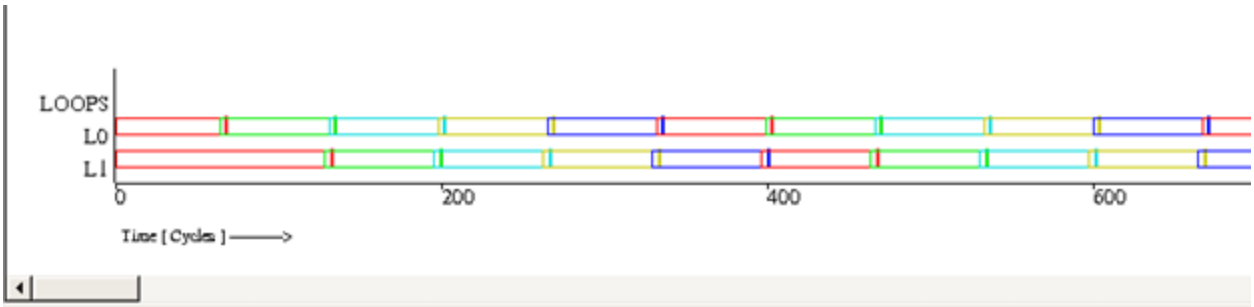
To get both inter-PA and intra PA, task overlap, you need to design your PPA for continuous processing. You can use the `set_params -continuous_processing always` command for designing this kind of PPA. The loop statistics and task processing section of report summary for this scenario are as follows.

```
Loop Statistics:
-------+----------+----------+------------+---------+-----------------
  Loop      II      Schedule     Target      Task     Pipelining Loss
  Name   Achieved    Length    Iterations   Latency    Between Tasks
-------+----------+----------+------------+---------+-----------------
   PA0          4          8           16       68                 0
   PA1          1          3           64       66                 0
-------+----------+----------+------------+---------+-----------------
```

```
Task Processing:
   Task Overlap : Yes
   Continuous_processing : Yes
   Reasons for PA Pipelining Loss Between Tasks :
     None
```

The dynamic performance graph of the continuous processing PPA is shown:

Figure 8-23 Dynamic Performance Graph of a Continuous Processing PPA



Items	Start	Stop	Stall
Task0	0	132	67
Task1	64	199	9
Task2	131	266	9
Task3	198	333	9
Task4	265	400	9
Task5	332	467	9
Task6	399	534	9
Task7	466	601	9

From this graph, you can notice that the achieved task_ii is still not meeting the goal of 64 cycles. This means that the FIFOs could be resized for better performance.

Resizing FIFOs for Performance

You can run `verify -rtl -detailed_perf_report` command to see which loop is stalling. After running RTL simulation, you can look at the `report_verify_rtl_performance.txt` file, which gives the potential FIFOs to be resized. You can start changing the FIFOs mentioned in this report to get better performance. In some cases, you might want to understand the FIFOs in the design. See the `Reports/ppa_all_fifos.pdf` report for a graphical view of the FIFOs in the design.

Another way to resize FIFOs is by doing calculations manually. You can see the `Work/report_cp.txt` file to get the release and acquire times of the resources and then

increase the appropriate FIFO sizes. For this example, after running `verify -rtl -detailed_perf_report`, PA0 stalls. In PA0, the release of memory “A” is in cycle 61, memory “B” in Cycle 64. In PA1, both “A” and “B” are acquired in cycle 1. Since PA0 stalls, the reason could be that it cannot release the resource (“A” has to be released first).

Note the following:

- Task 0: “A” is released at cycle 61, “B” at 64.
- Task 1: “A” is released at cycle 64+61 = 125, “B” at 128
- At task 1, “A” cannot be acquired because PA1 can start at cycle 128.

To avoid this stall, you can increase the FIFO size corresponding to A by 1 in mmul_define.v. That is, change:

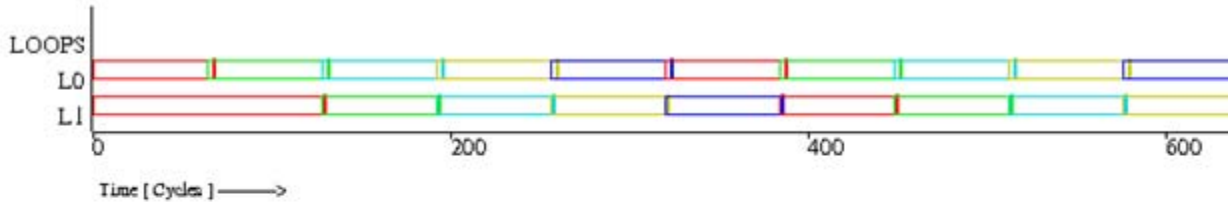
```
define MMul_fifo_0_pa0_pa1_len 0
```

to

```
define MMul_fifo_0_pa0_pa1_len 1
```

The performance of the hardware after resizing the FIFO is shown in Figure 8-24. This shows a task_ii of 64 for the hardware.

Figure 8-24 Hardware Performance After FIFO Resizing

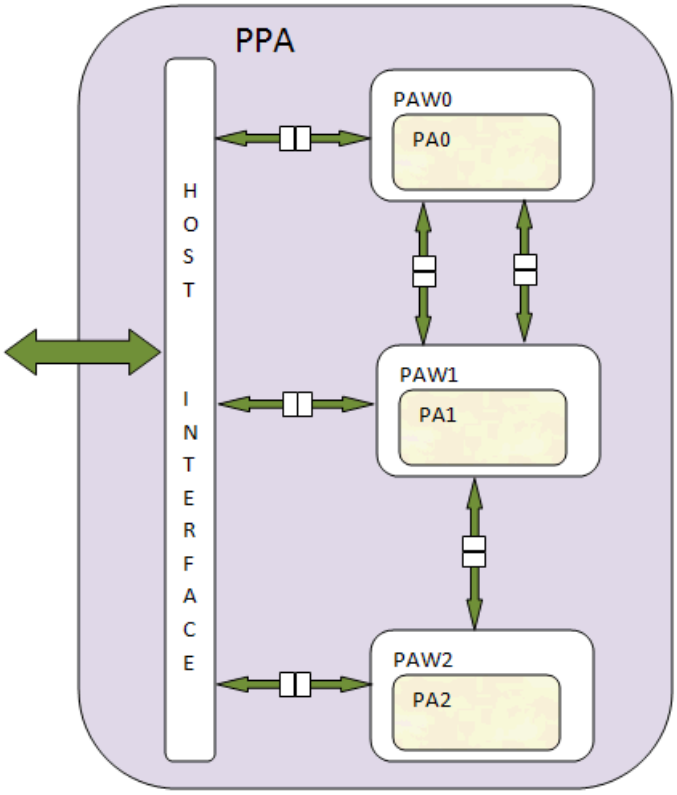


Items	Start	Stop	Stall
Task0	0	129	64
Task1	64	193	0
Task2	128	257	0
Task3	192	321	0
Task4	256	385	0
Task5	320	449	0

Using Templates

The Synphony C Compiler template makes it easy to create ready-to-integrate IP blocks. The template architecture exploits parallelism at all levels at minimal cost. Using templates offers a repeatable design process where timing is understood and has been tested in many designs. Templates are predictable, and you know the kind of implementation you can get from a particular coding style. The following figure shows the architectural templates for a top-level PPA.

Figure 8-25 Architectural Templates for a Top-Level PPA

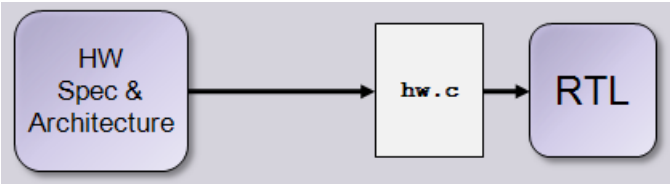


See the following for guidelines for working with templates:

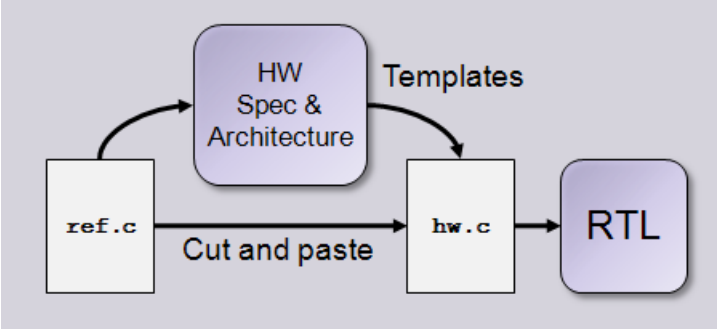
- Working with Templates
- Creating a Template

Working with Templates

1. To design a template from system-level architecture specs and performance constraints, do the following:
   - Define the hardware architecture you want.
   - Copy an existing template or code a C template as described in Creating a Template.
   - Directly code the algorithms in synthesizable C.



2. To design a template from reference code, do the following:
   - Define the architecture you want.
   - Select or code a C template as described in Creating a Template.
   - Cut and paste reference algorithms (kernels) into synthesizable C.



Creating a Template

When you create a template, you apply architectural thinking. You understand how the tool maps the code to hardware; therefore, you envision the parallel operations and create a template that is based on streaming data flow, memory access, and hardware sharing. Focus on the kernel as much as possible. The following steps provide a basic methodology for creating a design template.

1. Start with a block diagram of your system and its performance constraints.
2. Identify external interface constraints like streams, memories, and configuration registers.
3. Identify task granularity.

   For example, one task, possibly running infinitely is coarse grain; a sequence of finitely-sized tasks is medium grain; one task per data element is fine grain.
4. Partition and map blocks to loops.

   Collapse feedback. Identify block-level hardware sharing opportunities.
5. Identify inter-block data flow, like memory architecture, FIFO sizing, and scalar dependencies.
6. Identify intra-block control flow. For example, identify variable loop bounds, block bypass, and early exit conditions.
7. Code the template and verify performance.

Correlating Timing With ASIC Synthesis

When you start using a new technology, you might not have characterization data for the ASIC library, and you need to develop a correlation between the SCC target frequency and the target frequency for the ASIC synthesis tool.

This procedure describes how to develop this data for the Design Compiler synthesis tool. You can also use this methodology to adjust targeting inaccuracies with previously characterized data. The goal is to achieve a correlation with the back end tool in the +/-15% range.

To correlate timing, you can start with a similar representative design or start from scratch:

- Working With a Representative Design
- Working Without a Representative Design

Working With a Representative Design

If you already have one or more characterized designs that are similar to your design, you can use that as your starting point to compute a conversion factor from SCC frequencies to Design Compiler frequencies.

1. Use the following formula to derive the optimal SCC target frequency (fp) for the given application:

```
fp = fs * k
```

In this formula, k is the SCC target frequency used for the representative design, or the DC target frequency used for the representative design.

After the application development is complete, you can further optimize area by performing a final mini-exploration.

2. To perform a mini-exploration, do the following:
   - Synthesize the design using SCC for clock frequencies in the range [fp-20%, fp+20%] with 5% steps.
   - Synthesize the designs with frequency fs in Synopsys.
   - Pick the one which results in the best area.

Working Without a Representative Design

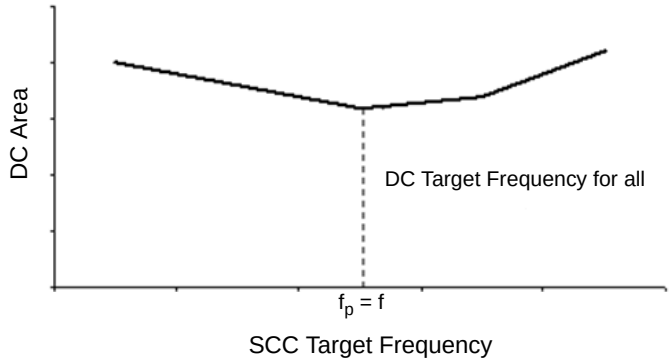
The following procedures provide the guidelines to determine the Synphony C Compiler frequency for an uncharacterized design:

1. Derive the optimal SCC target frequency (fp) by doing the following:
   - After you have coded a significant portion (over 50% of the functionality) of your design, target a range of clock frequencies around the Design Compiler target clock frequency fs.
   - In the Design Compiler tool, run ASIC synthesis and synthesize the designs with different SCC frequency targets at frequency fs .
   - Select the SCC frequency that results in the best area, and use it as the SCC target frequency for the rest of your application development.
2. Aim for minimum area when the SCC tool targets a particular clock frequency (f), and increased area when the tool targets clock frequencies that are higher or lower than that frequency.

   Note that some implementations might fail in the SCC tool due to timing and feedback issues. Some implementations might pass SCC but the generated RTL might still not meet timing in the Design Compiler tool at frequency.
3. If you run across these issues, then use f as the optimal SCC target to get minimal Synopsys area.

   In other words, fp = f.

Figure 8-26



- Note that due to non-linearities, you might see more than one local minimum, in which case, pick the one with smallest area that specifies the specified criteria.
- If there is no such point, then you probably need to either modify your application (e.g., relax some tight feedback paths, adjust bitsizes) or modify some SCC constraints/settings. If there is a point that has minimal area but doesn't meet the second criterion specified, then try a finer-grained exploration of SCC target clock frequency around that point.

To minimize the time spent in discovering fp, you might choose to follow the following implementation sequence of SCC target frequencies: fs, fs+10%, fs-10%, fs+20%,.... and stop as soon as you discover a point that meets the specified criteria.

You can also use a more sophisticated implementation sequence, e.g., binary search within the range [fs-50%, fs+50%,], to find the optimal fp.

After the application development is complete, you can further optimize area by performing a final mini-exploration.

1. To perform a mini-exploration, do the following:
   - Synthesize the design using SCC for clock frequencies in the range [fp-20%, fp+20%] with 5% steps.
   - Synthesize the designs with frequency fs in the Design Compiler tool.
   - Pick the one which results in the best area.

9

Characterizing ASIC Macro Cells

The ASIC version of the Synphony C Compiler tool includes the Novo tool to characterize macro cells. Macro cells such as adders and multipliers are the basic building blocks of the Synphony C Compiler tool. The Novo tool is a separate executable that is integrated into the Synphony C Compiler ASIC software. The Novo tool characterizes the macro cells for the specified standard cell library, and generates a set of area-delay tradeoff values that are required for timing-driven and optimal-area synthesis.

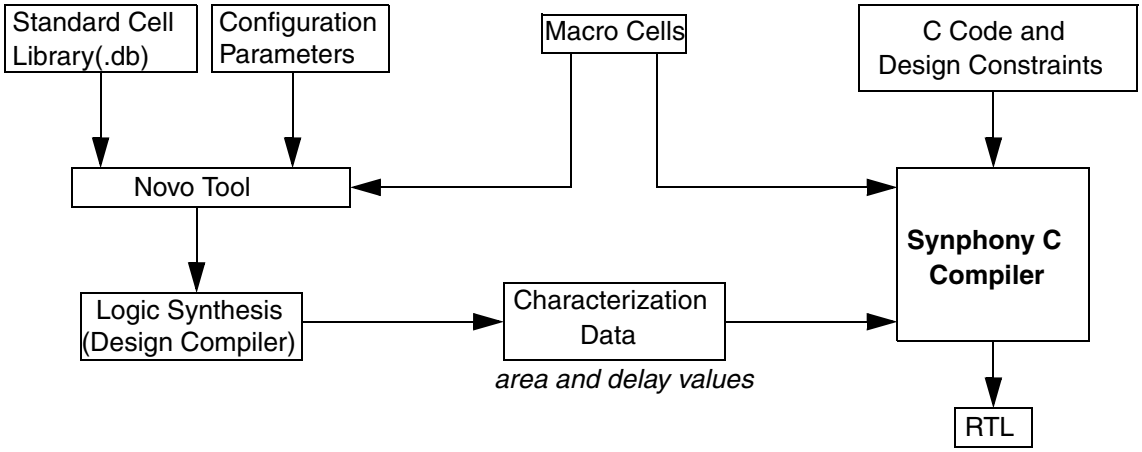
This chapter contains the following sections:

- ASIC Characterization Flow
- Using Characterization Data
- Validating Characterization Data

ASIC Characterization Flow

The characterization process involves interactions between the Novo, Synphony C Compiler, and Design Compiler tools. Figure 9-1 shows the inputs required for characterization and the flow of data between the tools:

Figure 9-1 Characterization of Macro Cells



The Novo tool uses the standard cell library, configuration parameters, and macro cells that you specify as the inputs to generate a set of characterization data for macro cells by using the Design Compiler tool. The characterization data consists of area and delay values for each macro cell. The data is stored in Synphony C Complier's internal representation.

To design hardware for a specific technology using the Synphony C Compiler tool, you must first run the Novo tool to generate characterization data for the specific technology, from the standard cell library. For a specific technology, generating the characterization data is a one-time activity. However, you might need to regenerate the characterization data when the Synphony C Compiler or the Novo tool is updated.

The following sections describe how to generate characterization data:

- Setting Up the Environment for Characterization
- Performing Characterization
- Debugging the Characterization Process
- Performing Incremental Characterization

Setting Up the Environment for Characterization

The following procedure describes how to generate the characterization data for the macro cells using the Novo tool.

1. Set up the environment.
   - Use the following environment setup:

Platform	Linux RedHat Enterprise Server 3.0 or 4.0
Logic synthesis	Design Compiler Ultra. Use the `.db` files
Batch processing	LSF 7.0.3.108010, SGE 6.2, or any other version that uses a similar job submission method
Shell	BASH
Scripting language	Perl 5.6.x or later version
Disk space	10 gigabytes

   - Set the appropriate paths, license settings, and environment variables for the Novo, Design Compiler Ultra, and LSF tools.

   The following example shows the environment variable settings required for running characterization:

```
##########################################################
#!/bin/bash
# LSF settings

export MANPATH=/tools/lsf/man:$MANPATH

export PATH=/tools/lsf/bin:$PATH
export LSF_ENVDIR=/tools/lsf/conf
# SGE settings
export MANPATH=/tools/sge/man:$MANPATH
export PATH=/tools/sge/bin:$PATH
export SGE_ENVDIR=/tools/sge/conf
# DC path settings
export PATH=/mnt/tools/programs/synopsys/synthesis \
B-2008.09-SP2/linux/syn/bin/:$PATH
# SYNPHONYCC LICENSE SETUP
export LM_LICENSE_FILE=$LM_LICENSE_FILE:/tools/licenses/synopsys/
license.dat
##########################################################
```

2. To check the environment settings, use the following command:

   ```
   synphonycc_novo -check_env
   ```

   This command verifies your environment and prints the settings at the command prompt. For more details about the various options supported by the Novo tool, see the *Synphony C Compiler Reference Manual*.

3. To set up the configuration file

   - Copy the config.template file from *SCCinstallDir*/docs to your working directory.
   - Set the other configuration parameters appropriately. See the *Synphony C Compiler Reference Manual* for detailed information about the required parameters.

   The Design Compiler tool uses some of these parameters to run logic synthesis on macro cells.

4. To verify your installation, run the following command.

   ```
   synphonycc_novo -config-file file \
    -remote-method lsf|sge|local \
    -synthesis-licenses number \
    [-remote-options "moreLSF/SGE_Options"]
    -install-test
   ```

   This command identifies any issues with your path and LSF settings.

Performing Characterization

To perform characterization, run the following command after setting up the environment:

```
synphonycc_novo -config-file file \
-remote-method lsf|sge|local \
-synthesis-licenses number \
-characterization \
   [-remote-options "moreLSF/SGE_Options"] \
[-macrocell-list-file fileName] \
[-verbose level]
```

The Novo tool uses the configuration parameters to characterize a subset of parameter values for each macro cell. It runs the Design Compiler tool (RTL synthesis) multiple times on the subset of macro cells using different clock constraints, and generates a set of area and delay tradeoff values.

When you specify `lsf` or `sge` with the `-remote-method` option, the Novo tool runs multiple jobs in parallel.

Note:
Characterization is a lengthy process and the Novo tool might require more than a day to complete. You can monitor the progress by checking the CHARACTERIZATION_TRACKER_SUMMARY file in the working directory and CHARACTERIZATION_TRACKER file in the logs directory.

When the characterization process is complete, the Novo tool generates characterization data required for the Synphony C Compiler tool.

If you encounter failures during characterization, follow these steps.

1. Debug the characterization run, and make the necessary changes.

   For more information, see Debugging the Characterization Process.

2. Run incremental characterization.

   For more information, see Performing Incremental Characterization.

   You can interrupt the characterization process by pressing Ctrl+C. You can then continue characterization by running it incrementally.

After the characterization process is complete, it is a good practice to validate the characterization data. For more information, see Validating Characterization Data.

Debugging the Characterization Process

A characterization process can fail because of disk or network problems, and problems with the LSF, SGE, or Design Compiler tools. You can interrupt a characterization process by pressing Ctrl+C. This section describes how to debug the characterization process.

1. Before you invoke the Novo tool, specify these settings to help you debug the issues encountered during characterization:

   - Set the `characterization_preserve_scratchdirs` variable to `1` in the Configuration file.
   - Specify the `-verbose` option with a value of `10`, with the `synphonycc_novo` command. With this setting, errors are printed to the screen and in the log files.

2. Monitor the progress of the characterization process by periodically checking the CHARACTERIZATION_TRACKER_SUMMARY file in the installation directory. For a more detailed report, see the CHARACTERIZATION_TRACKER file in the logs directory. These two files provide the status of the characterization process. For more details, see the *Synphony C Compiler Reference Manual* .

3. Follow these steps to debug the design.
   - Check the CHARACTERIZATION_TRACKER_SUMMARY and CHARACTERIZATION_TRACKER log files.
   - Check the following files for additional information:

logs/characterization.log	Problems found during the characterization stages
*characterization.cellName**	Snapshots of the synthesis runs and intermediate characterization data.

4. If you see disk space warnings during characterization, allocate more space and restart characterization.

Performing Incremental Characterization

Because characterization is a lengthy process, the Novo tool supports incremental characterization. During incremental characterization, the tool characterizes macro cells that were not characterized or failed characterization in the previous run. To run incremental characterization,

- Rerun characterization in the same working directory as the previous run.
- For the incremental run, use the same options as the previous run.
  - Changing the options triggers a complete rerun instead of an incremental rerun. For example, changing the name of the LSF or SGE queue causes a rerun of characterization.
  - Changing the options such as pico_tech_lib_db can lead to unpredictable results. If you have to change the options, start a new run instead of an incremental run.

  Note:
  You can change the number of synthesis licenses (`-synthesis-licenses`) without affecting the incremental run.

Using Characterization Data

After you generate and validate the characterization data in the Synphony C Compiler tool, follow these steps to build the target.

1. Set the path to the characterization data.

   Specify the path to the characterization data using the `set_params` command, as shown in the following example:

   ```
   set_params
   -techlib SCCInstallDir/scc/results/characterization/
   artisan_tsmc65gp-rvt_adv10a
   ```

2. Compile and build the target using the Synphony C Compiler tool.

Validating Characterization Data

After characterization, validate the generated data by running designs through the Synphony C Compiler and Design Compiler tools. Follow these steps to validate the characterization data:

1. Read the AreaDelay.pdf file, the datasheet generated by Novo tool after characterization. Specifically check the `sregn_ninit`, `addw`, and `mpyw_1_Stage` macro cells.
2. Use (2-4) representative designs of different sizes, ranging from 20K to 200K gates, for the validation process.
3. Run the designs in Synphony C Compiler.
4. Run logic synthesis on the RTL design generated by Synphony C Compiler.
5. Validate the timing closure by comparing the timing results from the Synphony C Compiler and Design Compiler tools.

   For more details, see Validating Timing Closure.

6. Validate area optimization by checking the results from Synphony C Compiler.

   For more details, see Validating Area Optimization.

Validating Timing Closure

To validate timing closure for the characterization data, follow these steps:

1. Run the Synphony C Compiler tool at various frequencies, say 150 Mz, 200 Mz, 250 Mz using the target library that was used to generate characterization data.

Select designs without tight feedback paths; otherwise the Synphony C Compiler might fail at higher frequencies. You might require a lot of time to rework your application code to eliminate tight feedback paths.

2. Run Design Compiler to synthesize the design. Choose the same frequencies so that the results are comparable.
3. Compare the results between Design Compiler and Synphony C Compiler tools.

   Note:
   Synphony C Compiler sometimes achieves a lower clock frequency than RTL synthesis. This is not necessarily a timing closure issue. This difference is because Synphony C Compiler tool models the design at a high level of abstraction.

If the RTL Synthesis fails timing, contact Synopsys support with the RTL synthesis critical path and the characterization data.

Validating Area Optimization

To validate the area values generated by the Novo tool, follow these steps:

1. Generate data points for area.
   - Run a design in the Synphony C Compiler tool at 3 different clock frequencies.
   - For each RTL Synphony C Compiler generates, run RTL synthesis at the lowest target frequency. In our example, run RTL-200 and RTL-250 to target150 MHz in RTL synthesis. This gives you three data-points where you have targeted 150 MHz in RTL synthesis as shown in Table 9-1:

Table 9-1 Data Points for Area

Design	Synphony C Compiler target	Design Compiler target
RTL-150	150 MHz	150 MHz
RTL-200	200 MHz	150 MHz
RTL-250	250 MHz	150 MHz

   - RTL-150 should have the smallest area.

These steps ensure that the Synphony C Compiler tool generates the most area-efficient RTL for the targeted frequency.

2. Repeat the same experiment comparing RTL-200 and RTL-250, both targeted at 200 MHz in RTL Synthesis. The RTL-200 should produce a smaller area.

If you get unexpected results, contact Synopsys Support for help, making sure to supply the results of all the RTL synthesis runs and characterized library data.

A

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 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 A-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 • 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 A-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` 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 A-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 is 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_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 .

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.

Basic Settings

-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. The default 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 RTL 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 SystemC. 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 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 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 might 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 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).

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 might 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 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 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 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 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 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.

Index

A

analysis views 10
  graph keys 10
  properties 11
Architecture Analyzer
  Code Tree View 21
area optimization
  validating ASIC characterization 8
area, optimizing 43
arrays
  mapping to ASIC memories 19
  mapping to FPGA memories 21
  mapping to memories 18
ASIC characterization
  tracking run 5
  validating 7
assertopms 9
autostart 16

B

banking 41
bit widths 44
boundary registers
  TCABs 34
build
  optimizing 31
build command 6
build flows 2
  running 4
build phases
  analyzing 2

C

C code
  crossprobing 12
C simulations 5
  parameters 17
C to RTL Mapping 66
clock frequency 10
clock gating 18
  TCABs 35
clocks 11
code coverage 27
code sinking
  defined 35
coding guidelines
  PPAs 11
  TCABs 11
coding style 1
  restrictions 3
console window 14
constants 47

conventions for documentation xix
corner cases 26
cost
  analyzing 8
  Relative Cost view 23
  Resource Browser 21
create_implementation command 13, 14
crossprobing 12
  source code from analysis views 12
customer support xx

D

dependencies 16
deployable TCABs
  defined 3
  specifying 44
design summary 2
desogm for test 13
document view window 15
double buffering 42
Dynamic Performance Graph 64
Dynamic Performance graph 25

E

e timing closure 12
equivalence checking with LEC 15
error signaling 15
estimate_power command 8
export_implementation command 9
external delays
  TCABs 34

F

false paths
  analyzing 7
Feedback Path viewer 17
feedback path viewer 50
feedback paths
  analyzing 6
  definition 55
  using pipeline flushing 40
  viewing 17
FIFOs 22, 25
  compared to memories 25
  external 26
  internal 28
File menu 4

G

gate-level simulation 16
golden simulation 6
  checking code coverage 28
  debugging 22
GUI 2

H

hierarchical blocks. See also TCABs
hierarchical design 1
  iterating 6
hierarchical design flow 5
  making changes 6
  running 8
  sample script 8
hierarchy
  analyzing 8, 45
  building 41, 43
  Hierarchy graph 26
  limitations 4
Hierarchy Graph 37, 26

I

II (initiation interval)
  specifying 11
implementation parameters
  basic 10

implementations
  comparing cost 23
  creating with Tcl commands 8
independent stall domains
  specifying for TCABs 31
infinite run 16
iterations
  extending space 38

L

latency
  TCABs 35
lint simulation 7
local memories 48
loop distribution 37
loop fusion 52
loop jamming 52
loop merging 53
loop unrolling 36

M

memories 18
  ASIC 19
  bottlenecks 54
  code optimizations 41
  FPGA 21
  local 48
  specifying properties 23
memory arbitration 21
memory ports 50
menus 4
multi-buffering 42
multiplier pipeline stages 31
mutual exclusion 51

N

Novo
  tracking runs 5
  validating characterization 7

O

Operation Schedule Viewer 40, 19
output 2

P

package directory 4
parallel optimizations
  coding for 56
parallelization 56
perfectization
  defined 33
  techniques 34
performance
  analyzing 5
  checking 7
  Dynamic Performance graph 25
  optimizing in hardware 53
pipeline flushing 38
pipelining
  specifying for TCABs 29
power
  clock gating 18
power consumption 9
PPA Graph 30, 14
PPAs
  coding guidelines 11
  compared to TCABs 5
  specifying 44
Preprocess
  optimizing 31
projects
  creating withTcl commands 5
properties
  viewing 11

R

rate matching
  defined 41
registers
  inferring 46
Relative Cost view 23
Relative Cost Viewer 62
report_implementation command 14
resets 11
resizing FIFOs for performance 70
Resource Browser 56, 21
resource sharing
  using pipeline flushing 40
resource usage
  PPA graph 14
  Resource Browser 21
ROMs 20
RTL
  crossprobing 12
  features 10
RTL package
  equivalence checking 15
  setting parameters 19
RTL simulation 10
  debugging 24
  RTL package 13
rtl_synthesize command (ASIC) 18
rtl_synthesize command (FPGA) 19
Run menu 6

S

scalarization 48
SCC
  output 2
  related tools 5
Schedule
  optimizing 32
scheduling
  analyzing 6
  Operation Schedule view 19
  Sequentialization graph 15
Sequentialization Graph 25
Sequentialization graph 15
shell window 14
shift registers 30
simulation
  debugging 22
simulation regression flow 29
simulations 3
  analyzing 2
SolvNet
  accessing xx
  documentation xviii
source code
  checking 5
  viewing from analysis views 12
stall domains
  defined 31
  specifying independent TCAB 31
  specifying shared TCAB 32
STARC support 23
streams
  building external FIFOs 26
  building internal FIFOs 28
Synphony C Compiler EV 32, 38
synphonycc command 2
Synthesize
  optimizing 33
system integration
  analyzing 9
  reports 9

T

task interval, specifying 11
task overlap
  analyzing 25
  inter-PA 68
  inter-PA and intra-PA 69

task processing 58
  constraints 58
  RTL simulation problems 64
  RTL simulation reports 62
  synthesize reports 59
TCABs
  analyzing 45
  boundary registers 34
  clock gating 35
  coding guidelines 11
  combinational 28
  compared to PPAs 5
  defined 3
  external delays 34
  externalizing 41
  importing 45
  independent stall domain 31
  latency 35
  sequential 28
  shared stall domain 32
  sharing 39
  specifying 26
templates 72
  creating 73
throughput 11
  meeting goals 54
timing
  analyzing 5
timing closure
  validating ASIC characterization 7
timing correlation 74
tooltips
  analysis views 11

V

verification flows 2
verify command 30
Verilog
  checking code coverage 29
Verilog simulation
  setting parameters 20
Verilog simulations 9
View menu 9