Cycle Model Studio

Version 11.4

User Guide

arm

101108_1104_00_en

Cycle Model Studio

User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
0903-00	01 August 2017	Non-Confidential	Update for 9.3
0905-00	01 February 2018	Non-Confidential	Update for 9.5
1000-00	01 September 2018	Non-Confidential	Update for 10.0
1001-00	15 January 2019	Non-Confidential	Update for 10.1
1100-00	01 May 2019	Non-Confidential	Update for 11.0
1102-00	15 January 2020	Non-Confidential	Update for 11.2
1102-01	01 October 2020	Non-Confidential	Documentation update
1103-01	20 January 2021	Non-Confidential	Documentation update
1104-00	12 February 2021	Non-Confidential	Release 11.4

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

developer.arm.com

Progressive terminology commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used terms that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive terms. If you find offensive terms in this document, please contact *terms@arm.com*.

Contents
Cycle Model Studio User Guide

Preface

This preface introduces the *Cycle Model Studio User Guide*.

It contains the following:

About this book

This guide describes how to use the Cycle Model Studio user interface to compile RTL as a Cycle Model and generate SystemC and Platform Architect components.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction

Introduces Cycle Model Studio, including a high-level view of its functionality and platform requirements.

Chapter 2 Compiling RTL into a Cycle Model

Getting started with Cycle Model Studio, including creating your first project and configuring commonly-used compiler settings using the GUI.

Chapter 3 Creating components for specific simulation environments

Cycle Model Studio supports creating Cycle Model components for both SystemC and Platform Architect simulation environments. This section describes generating these components and making changes to component ports.

Chapter 4 Advanced features

Describes advanced features of the GUI.

Glossary

The Arm® Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the *Arm® Glossary* for more information.

Typographic conventions

italic

Introduces special terminology, denotes cross-references, and citations.

bold

Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`

Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace`

Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`

Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`

Denotes language keywords when used outside example code.

`<and>`

Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS
: Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *Cycle Model Studio User Guide*.
- The number 101108_1104_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Other information

- *Arm® Developer*.
- *Arm® Documentation*.
- *Technical Support*.
- *Arm® Glossary*.

Chapter 1
Introduction

Introduces Cycle Model Studio, including a high-level view of its functionality and platform requirements.

It contains the following sections:

1.1 Cycle Model Studio functionality

Cycle Model Studio is a graphical tool designed for the generation of hardware-accurate software models.

Cycle Model Studio simplifies the task of compiling an RTL hardware model into a Cycle Model. It generates platform-specific components for simulation environments (such as Platform Architect and SystemC), and tunes Cycle Models for optimal performance during simulations.

The Cycle Model Compiler is part of Cycle Model Studio. The compiler takes an RTL hardware model as input, and creates a high-performance linkable object (the Cycle Model), which is cycle- and register-accurate. The Cycle Model provides an API for interfacing with your validation environment. For details about the Cycle Model Compiler, see the *Cycle Model Compiler User Manual*.

Cycle Model Studio shortens the design flow process by running RTL and software debugging tasks concurrently, so that your final product is available sooner. Cycle Model Studio also finds RTL debug issues during software debug.

The Cycle Model Studio GUI manages all aspects of the Cycle Model compile and runtime processes.

1.2 Cycle Model Studio flow

This section provides a high-level description of the Cycle Model Studio functions and workflow.

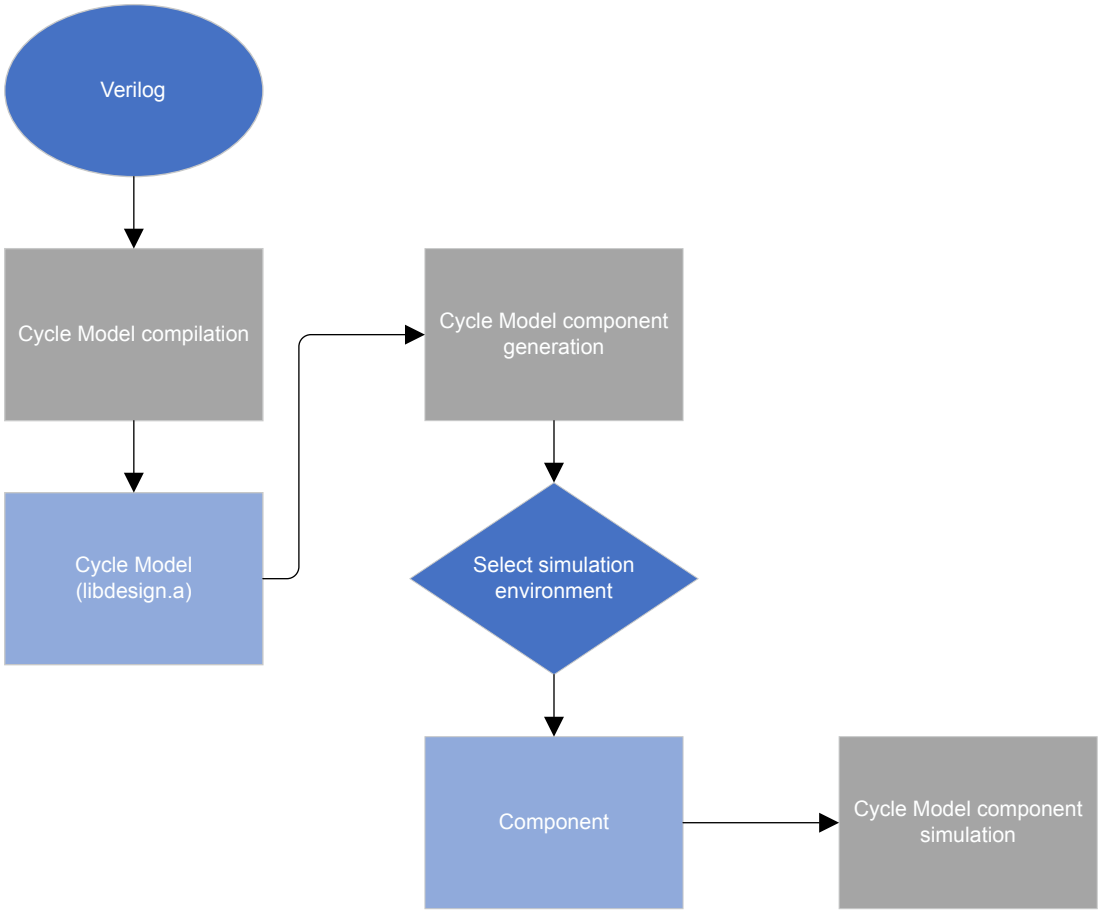

Figure 1-1 Cycle Model Studio process flow

After you create a project in Cycle Model Studio:

1. Add the Verilog design and library files to the project.
2. Compile the project using the Cycle Model Compiler. This creates the Cycle Model.
3. Generate a component that is compatible with your simulation environment. Cycle Model Studio supports Platform Architect and SystemC.
4. Configure the generated component. This enables you to specify the connection interface between your Cycle Model and your simulation environment.
5. Run and tune your simulation using the component.

1.3 Simulation dependencies and requirements

The compiled Cycle Model has certain dependencies in order to simulate properly.

- Cycle Models must be able to find `libstdc++.so` from GCC 4.8.3 or later. Add the directory that contains `libstdc++.so` to the `LD_LIBRARY_PATH` environment variable.
- If you are compiling custom code and using a third-party simulation tool, use the GCC version provided by the simulation tool. The GCC version provided by the simulation tool must be GCC 4.8.3 or later.
- Ensure only a single GCC version is included within your environment to avoid library conflicts.

1.4 Compiler use of options and directives

The Cycle Model Compiler reads the following files, in order, and generates a Cycle Model for the design.

1. Options files
2. Directives files
3. Verilog design and library files

Options files

Options files contain command options that provide control and guidance to the Cycle Model Compiler (sometimes called switches). Use the Cycle Model Compiler **Properties** to configure parameters to apply when compiling Cycle Models. These parameters display when **Cycle Model** or **RTL Sources** is highlighted in the Project Explorer view. They are organized into a variety of categories.

Directives files

Directives files contain directives that control how the Cycle Model Compiler interprets and builds a Cycle Model. Directives are compiler commands that can be contained in a directives file or embedded in Verilog source code. Directives control how the Cycle Model Compiler interprets and builds a linkable Cycle Model.

Cycle Model Studio manages two types of compile directives:

- Net directives apply to one or more nets.
- Module directives apply to one or more modules.

For more information, see the *Cycle Model Compiler User Manual*.

Verilog design and library files

Verilog design and library files describe the golden RTL of the hardware design.

1.5 Data collection in Cycle Model Studio

Arm periodically collects anonymous information about the usage of our products to understand and analyze what components or features you are using, with the goal of improving our products and your experience with them. Product usage analytics contain information such as system information, settings, and usage of specific features of the product. They do not include any personal information.

Host information includes:

- Operating system name, version, and locale.
- Number of CPUs.
- Amount of physical memory.
- Screen resolution.
- Processor and GPU type.

Note

To disable analytics collection for all tools running in the environment, set the environment variable `ARM_DISABLE_ANALYTICS` to any value, including 0 or an empty string. This setting is not saved in persistent storage. It must be reset at subsequent invocations of the tool.

Chapter 2
Compiling RTL into a Cycle Model

Getting started with Cycle Model Studio, including creating your first project and configuring commonly-used compiler settings using the GUI.

It contains the following sections:

2.1 Start Cycle Model Studio

After installation, launch Cycle Model Studio from the command line.

Prerequisites

Before launching Cycle Model Studio, ensure you have installed all components successfully, and that the appropriate environment variables have been set. See the *Cycle Model Installation Guide* (101106) for instructions.

Launch Cycle Model Studio

To start Cycle Model Studio, from your working directory, enter:

```
> ${CARBON_HOME}/bin/modelstudio
```

The GUI launches.

Next steps

2.2 Create a new project on page 2-17

2.2 Create a new project

This section describes how to create a new Cycle Model Studio project.

Prerequisites

Launch Cycle Model Studio. See *2.1 Start Cycle Model Studio* on page 2-16.

Create a new project

Note

The options available from the **Project** menu allow customization of component settings, but do not include RTL source. For access to the RTL files to further configure the Cycle Model, you can import an existing `.ccfg` file into a project. See *4.2 Import a configuration file* on page 4-47 for details.

1. Select **File** > **New** > **Project**. The **New Project** dialog appears:

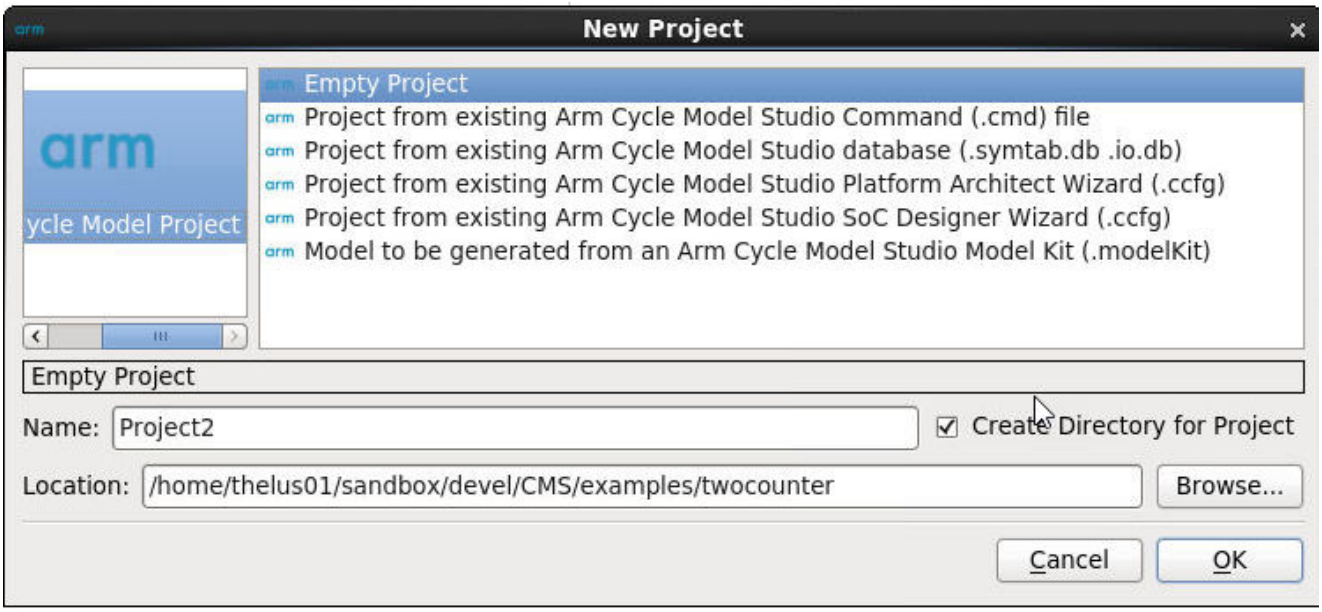

Figure 2-1 New Project dialog

2. Select the type of project you want to create:
   - **Empty Project** - Start a new project from scratch, without using an existing project as a template.
   - **Project from existing Arm Cycle Model Studio Command (.cmd) file** - Start a new project using an existing command file containing RTL source files and specific compile settings from a previous Cycle Model compile.
   - **Project from existing Arm Cycle Model database (.symtab.db)** - Start a new project using an existing database file from a previous Cycle Model.
   - **Project from existing Arm Cycle Model Studio Platform Architect Wizard (.ccfg)** - Customize an existing configuration file from Platform Architect.
3. In the **Name** field, enter the name for your new project.
4. In the **Location** field, browse to the location where you plan to store your new project.
5. To automatically create a new directory using the name of the project, select the checkbox **Create Directory for Project**. The project files are placed in that directory.
6. Click **OK**.

Next steps

2.3 Add RTL source files on page 2-18

2.3 Add RTL source files

After creating a project, add RTL sources to the project.

Prerequisites

Create a project. See *2.2 Create a new project* on page 2-17.

Add RTL source files

1. Select **Project** > **Add RTL Source(s)...** The **Select RTL Source(s)** dialog appears.

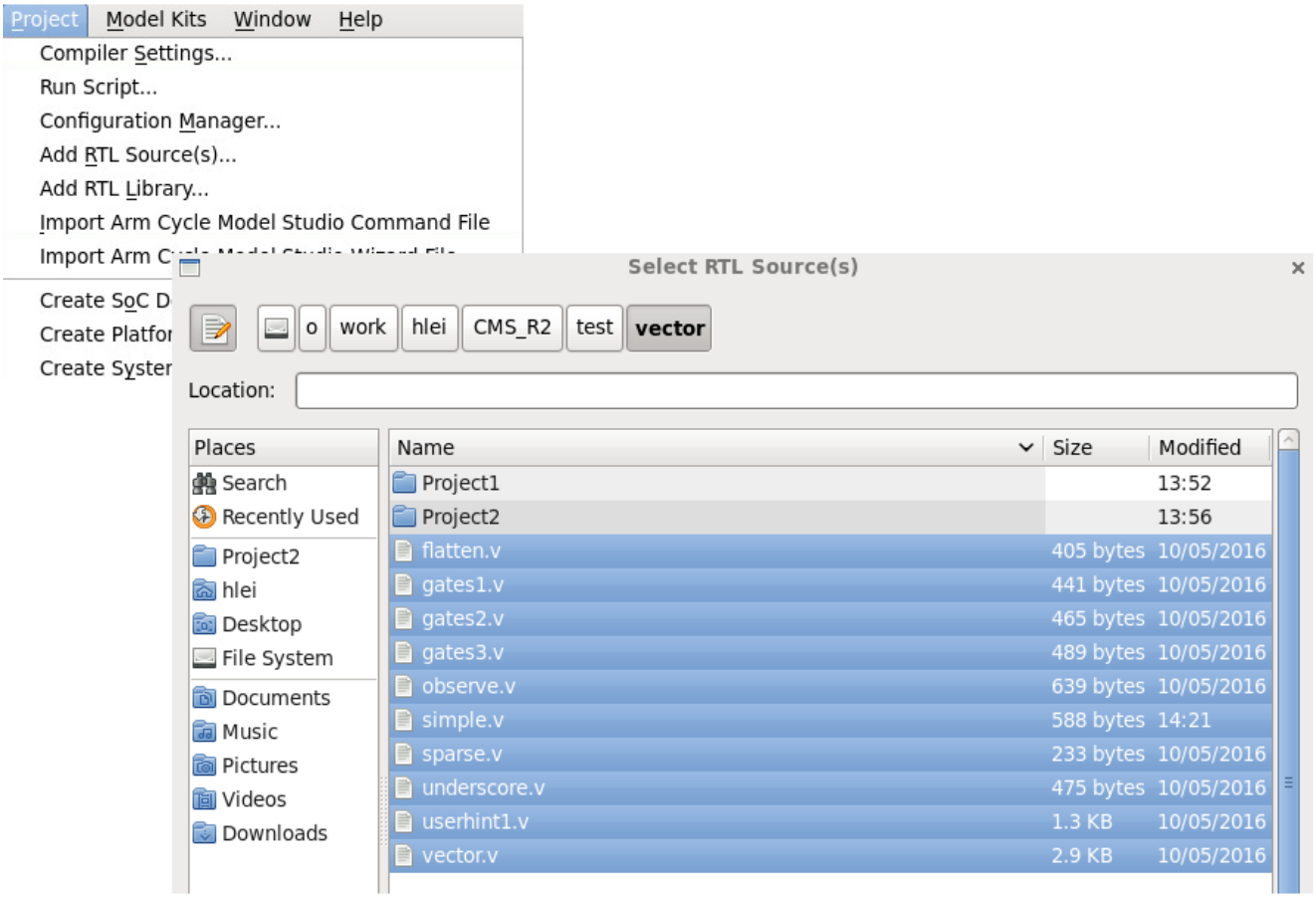

Figure 2-2 Selecting RTL source files

2. Browse to and select your project's RTL source file or files.
3. Click **Open** to add the RTL files to your project. The files are listed in the Project Explorer:

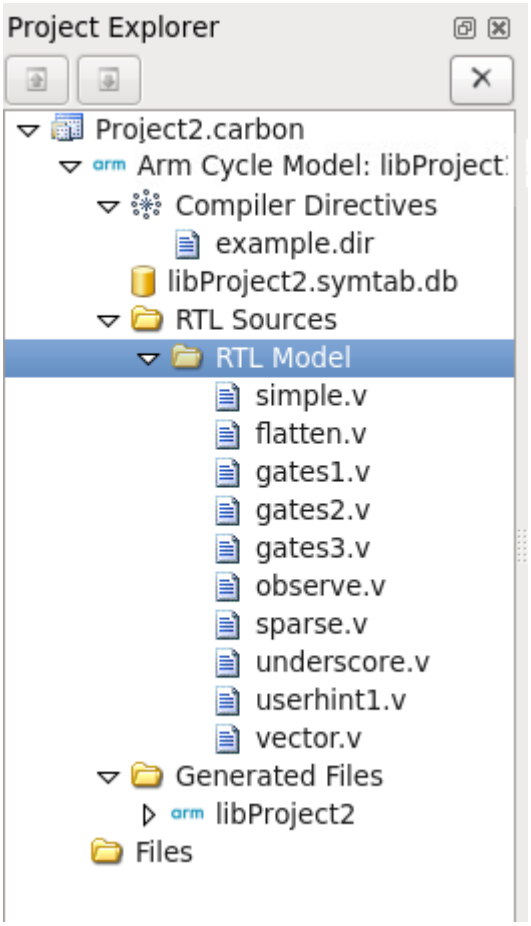

Figure 2-3 RTL source files added to project

Note

The compilation of the project is affected by the order of RTL Source files in the Project Explorer. You can move the files up or down in the Project Explorer tree view using the **Move Up** and **Move Down** buttons.

Next steps

Specify compiler options. See *2.4 Define compiler options* on page 2-20.

2.4 Define compiler options

The default compiler settings allow you to compile your project immediately into a Cycle Model. This section describes setting some commonly-used compiler options using Cycle Model Studio.

The **Compiler Properties** dialog includes the complete set of compiler options. A brief description of the selected option appears at the bottom of the dialog. For complete details about compilation options, see the *Cycle Model Compiler User Manual*.

Select **Project** > **Compiler Settings** to access the **Compiler Properties** dialog:

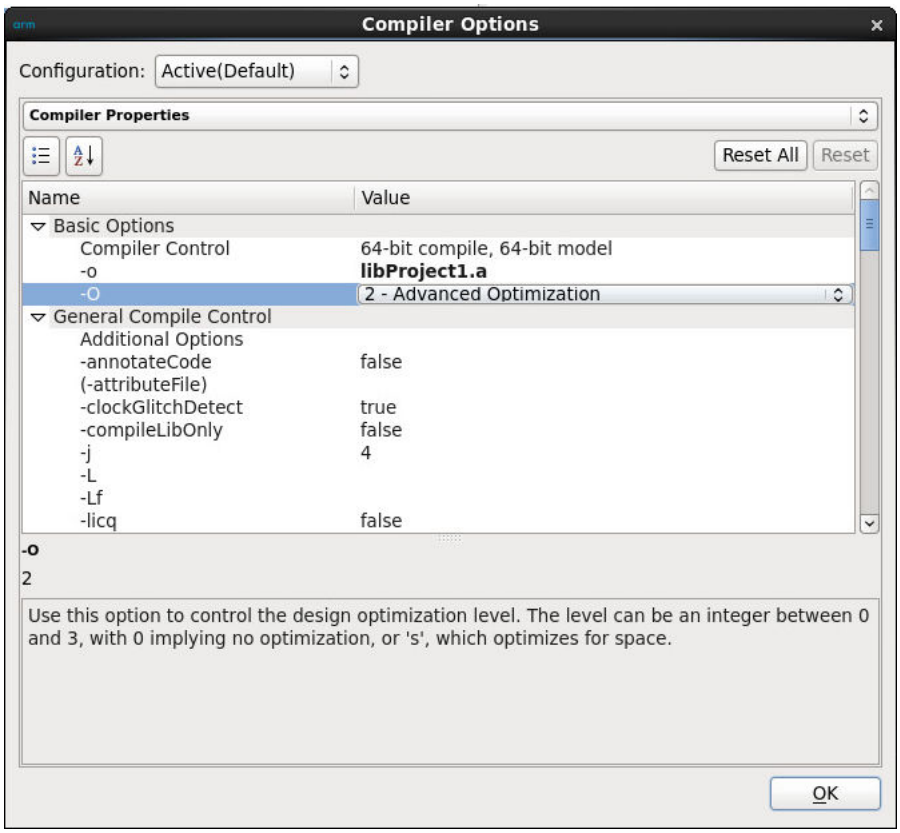

Figure 2-4 Compiler Options dialog

This section contains the following subsections:

- *2.4.1 Name the Cycle Model (optional)* on page 2-20.
- *2.4.2 Set the Verilog language variant (optional)* on page 2-22.
- *2.4.3 Pass command-line options to the compiler (optional)* on page 2-24.
- *2.4.4 Pass directives to the compiler (optional)* on page 2-26.
- *2.4.5 Hide internal signals in a generated Cycle Model (optional)* on page 2-27.
- *2.4.6 Enable license queuing (optional)* on page 2-28.
- *2.4.7 Increase debug visibility into design nets (optional)* on page 2-29.

2.4.1 Name the Cycle Model (optional)

The Cycle Model name defaults to `lib<project_name>.a`.

Usage notes

- You can use alphanumeric characters, period (`.`), hyphen (`-`), underscore (_), and plus sign (`+`) characters.
- You must use the `lib` prefix for the file name.

- Do not use white spaces in the string.
- Do not use existing system library names (for example, `libc` or `libm`).

Procedure for renaming the files associated with the Cycle Model

To change the name:

1. Select **Project** > **Compiler Settings** to access the **Compiler Properties** dialog.
2. In the **Basic Options** section, select the `-o` option. Do not confuse the lower-case `-o` option with the upper-case `-O` option.
3. Enter the new name in the **Value** column:

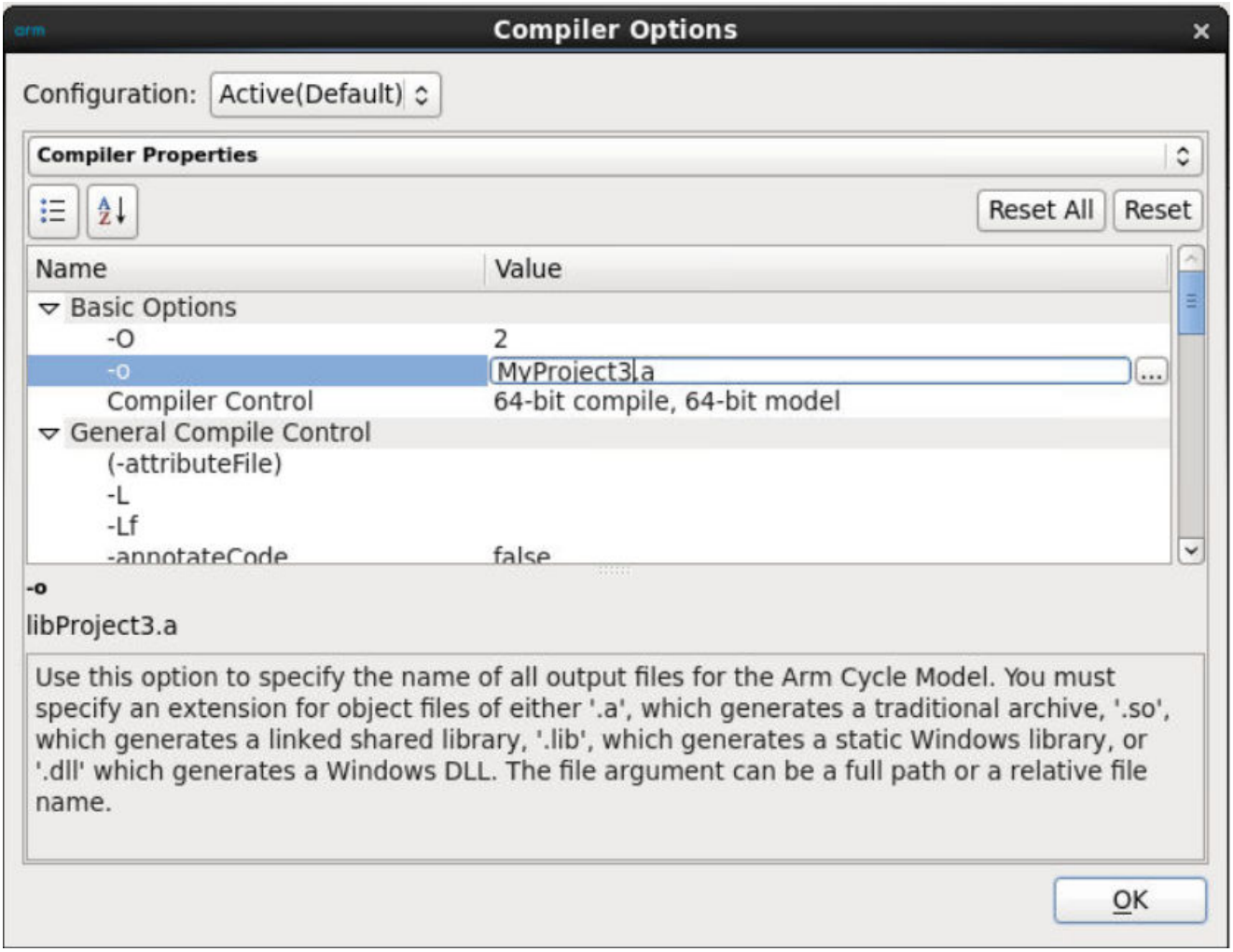

Figure 2-5 Changing the Cycle Model name

4. Click **OK** to close the **Compiler Properties** dialog.

Procedure for setting a different output file location

The compiler uses the name specified as a basis for additional output files. To specify a different directory for the output files:

1. In the `-o` row, click in the **Value** column. This enables the **Browse** (**…**) button.
2. Click **Browse** (**…**) to launch the **Output File** dialog.
3. Navigate to the directory you want to hold the output files.
4. Click **Save**. The **Output File** dialog closes.
5. Click **OK** to close the **Compiler Properties** dialog.

2.4.2 Set the Verilog language variant (optional)

By default, the Cycle Model Compiler uses Verilog 2001 during compilation. The Cycle Model Compiler also supports Verilog 95 and SystemVerilog (1800-2012).

You can set the Verilog variant for the selected project, or at the compiler level for all projects.

Procedure for selected project

To change the Verilog variant for the selected project:

1. Select **Project** > **Compiler Settings** to access the **Compiler Properties** dialog.
2. In the **Verilog Options** section, select the desired variant from the **VerilogMode** menu:

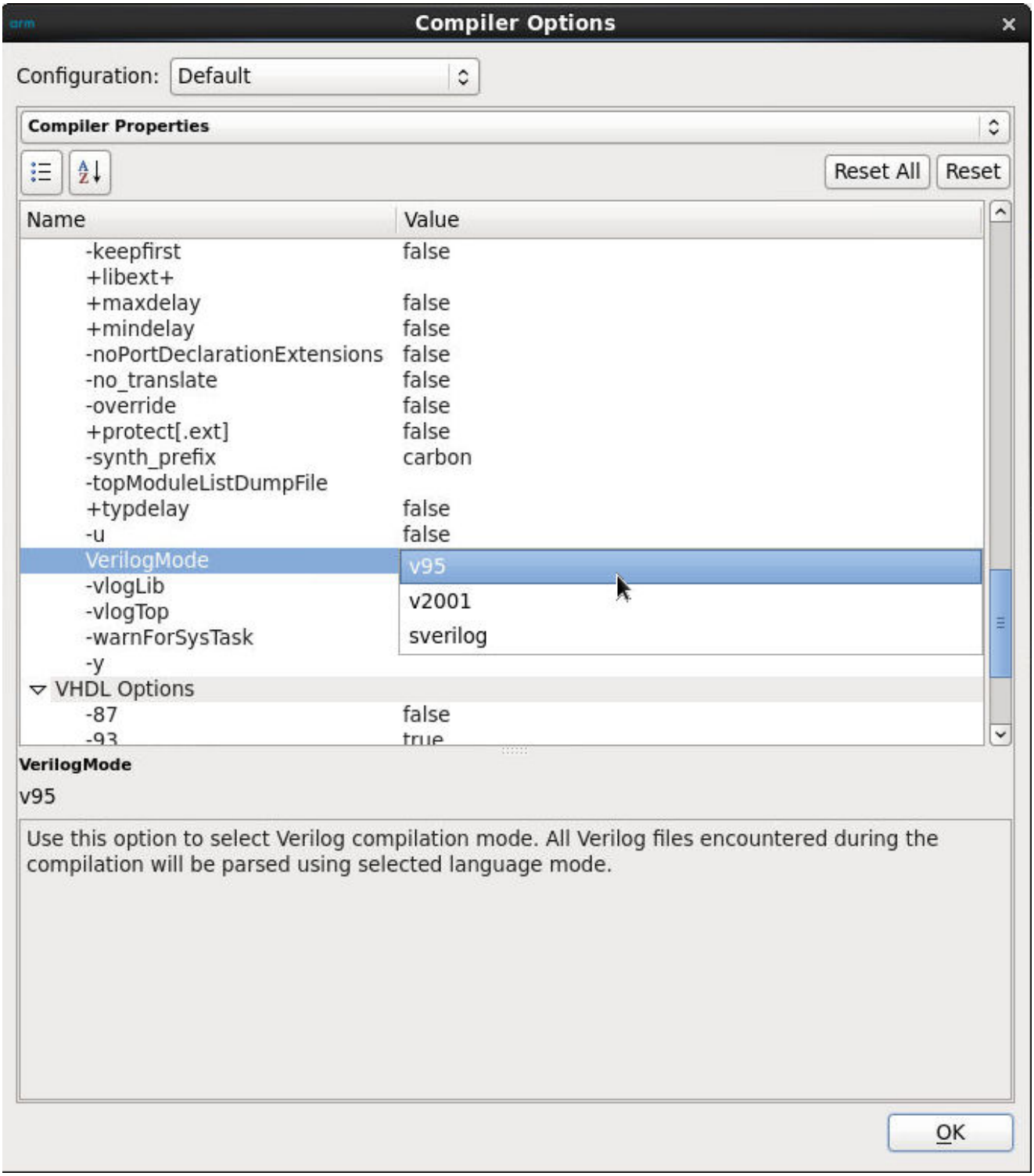

Figure 2-6 Setting the Verilog variant

3. Click **OK** to close the **Compiler Properties** dialog.

Procedure for setting default for all projects

To change the Verilog variant that the Cycle Model Compiler uses by default for all new projects:

1. Select **Edit** > **Preferences** to access the **Preferences** dialog.
2. Select the desired setting from the **Verilog Default mode for new projects** menu:

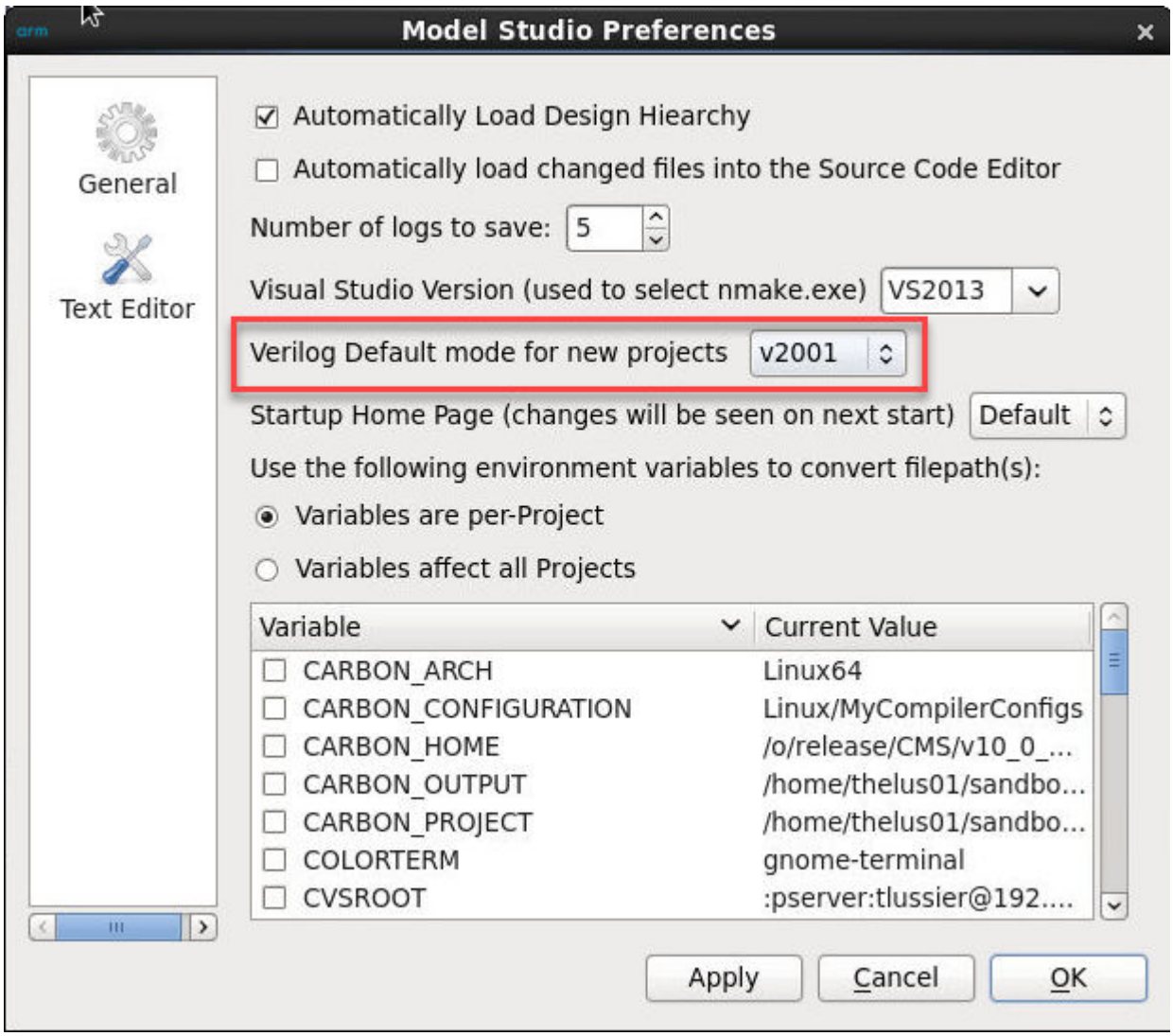

Figure 2-7 Preferences dialog

3. Click **OK** to close the **Preferences** dialog.

Existing projects continue to use the variant initially specified for the project.

2.4.3 Pass command-line options to the compiler (optional)

The Cycle Model Compiler reads command line options from a file that you specify.

Using the **-f** option, you can specify a file name that includes command-line options. The compiler reads the command-line options from the file, treating the options as if they have been entered on the command line.

Use the extension `.cmd` for command files.

Prerequisites

Review important usage notes related to command files in the *Arm® Cycle Model Compiler User Guide* (101050).

Procedure for including a file of command line options

To specify a command file:

1. Select **Project** > **Compiler Settings** to access the **Compiler Properties** dialog.
2. In the **Input File Control** section, click **-f**. The **Value** field and **Browse (…)** button become active:

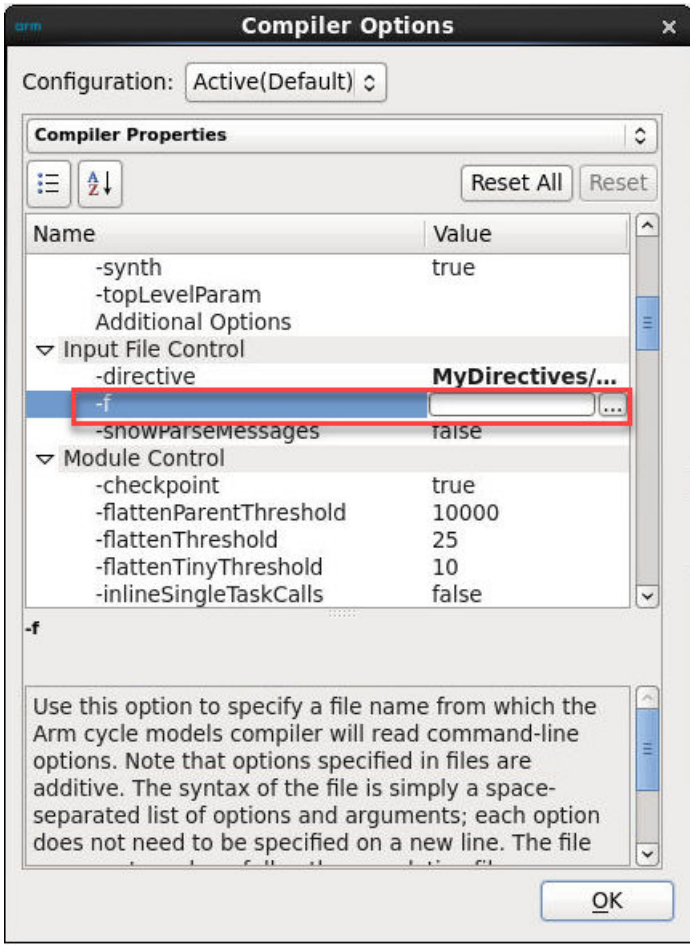

Figure 2-8 Specifying a command file

3. Click **Browse** (**…**). This displays the **-f** dialog. Browse to add one or more command line files.
4. If you specify more than one command file, use the arrows to change the order in which files are passed to the compiler:

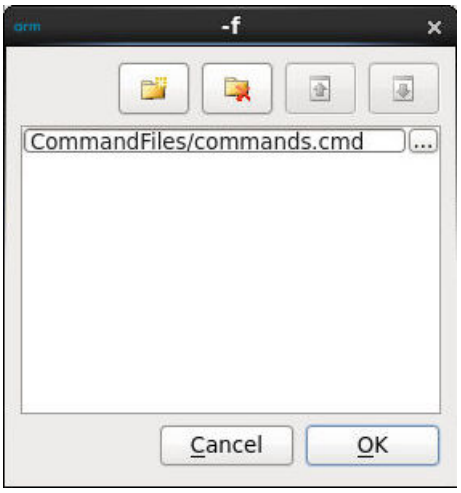

Figure 2-9 -f dialog

5. Click **OK** to close the **-f** dialog.
6. Click **OK** to close the **Compiler Properties** dialog.

2.4.4 Pass directives to the compiler (optional)

Directives are compiler commands that can be contained in a directives file. Directives help control how the Cycle Model Compiler interprets your design.

Prerequisites

Review important limitations and guidelines related to directives in the *Arm® Cycle Model Compiler User Guide* (101050).

Procedure for including a list of directives

To enable license queuing:

1. Select **Project** > **Compiler Settings** to access the **Compiler Properties** dialog.
2. In the **Input File Control** section, click **-directive**. The **Value** field and **Browse** (**…**) button become active:

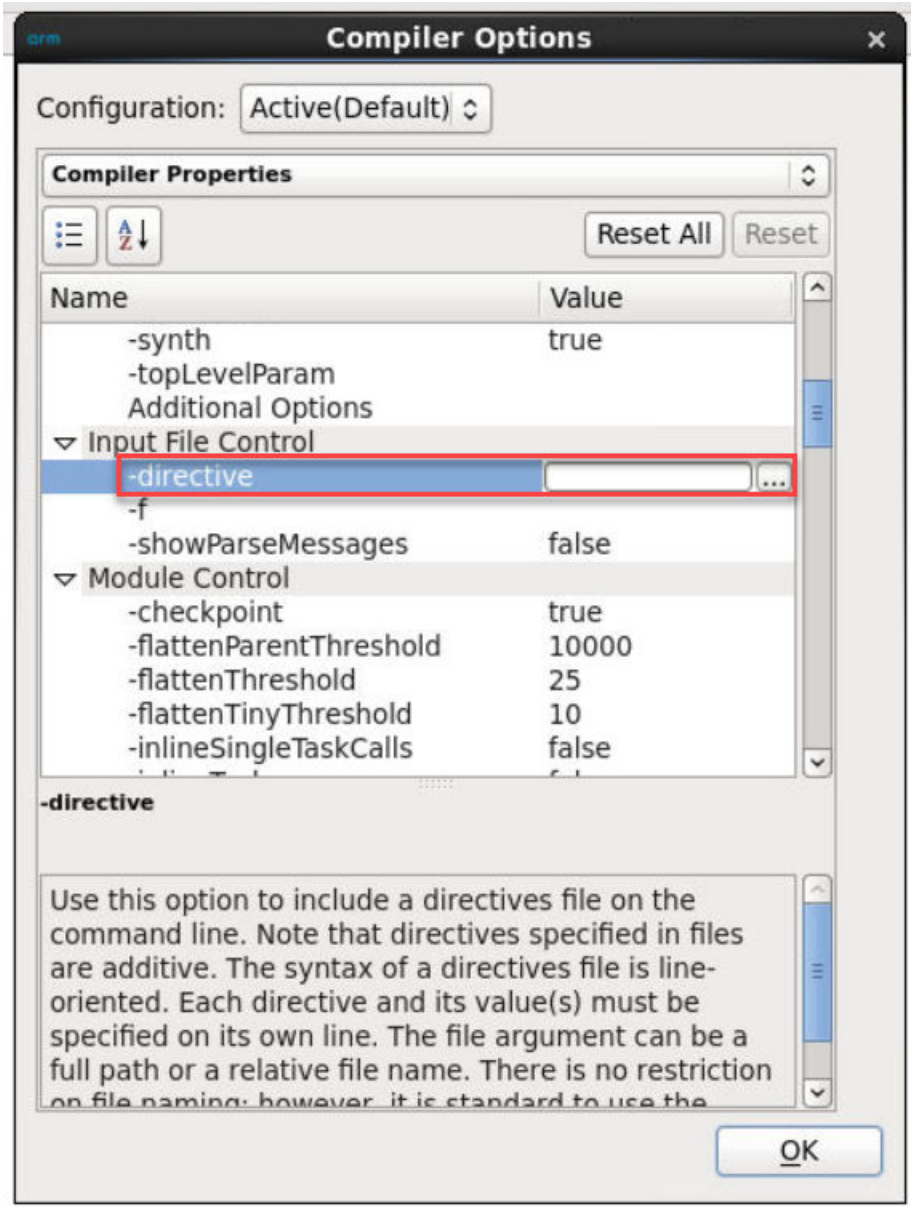

Figure 2-10 Specifying directives

3. Click **Browse** (**…**). This displays the **-directive** dialog. Browse to add one or more directives files.
4. Use the arrows to change the order in which files are passed to the compiler:

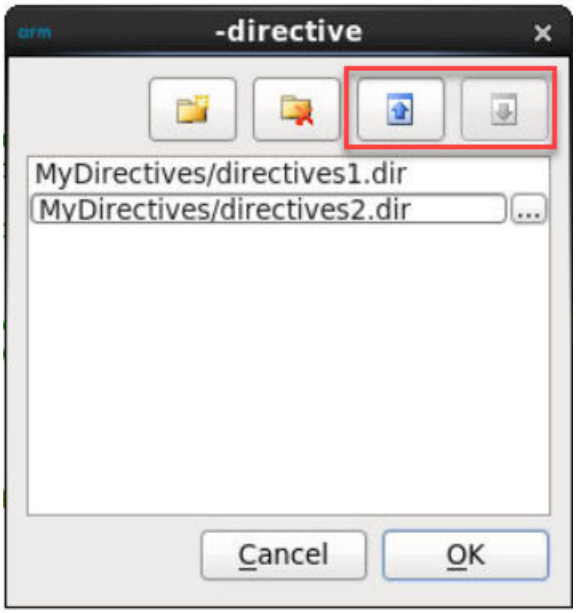

Figure 2-11 -directive dialog

5. Click **OK** to close the **-directive** dialog.
6. Click **OK** to close the **Compiler Properties** dialog.

2.4.5 Hide internal signals in a generated Cycle Model (optional)

If you are creating Cycle Models for internal use, the **-noFullDB** option makes the internals of the design invisible to users of the model.

-noFullDB specifies that waveforms generated by the Cycle Model include only those marked with the `observeSignal` net control directive. For more information about the `observeSignal` directive, see the *Arm® Cycle Model Compiler User Guide* (101050).

Procedure for enabling -noFullDB

To enable the **-noFullDB** option:

1. Select **Project** > **Compiler Settings** to access the **Compiler Properties** dialog.
2. In the **Output Control** section, change the **-noFullDB** setting to **True**:

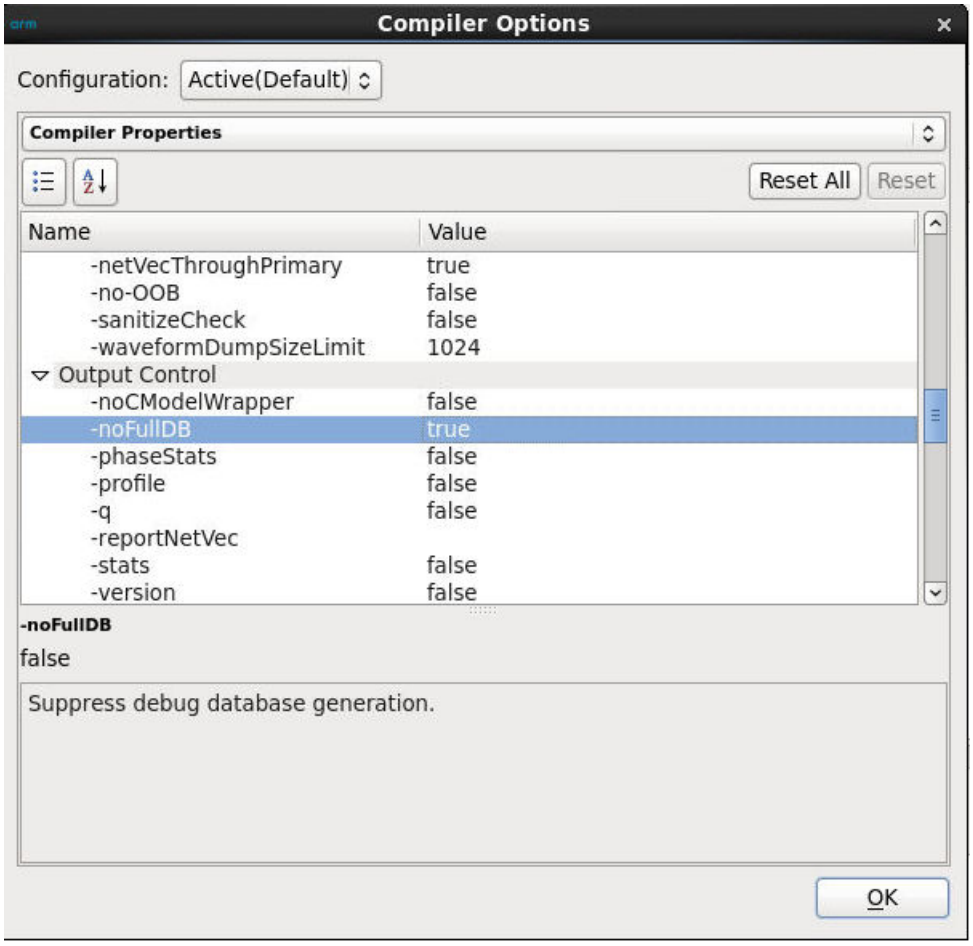

Figure 2-12 Enabling -noFullDB

3. Click **OK** to close the **Compiler Properties** dialog.

   After you generate the Cycle Model, waveforms output by the Cycle Model include only signals that were labeled `observeSignal`.

2.4.6 Enable license queuing (optional)

When license queuing is enabled for the Cycle Model Compiler compilation process, the Cycle Model Compiler waits for a license to become available, rather than exiting immediately if all licenses are in use.

License queuing is disabled by default.

Procedure for enabling license queueing

To enable license queuing:

1. Select **Project** > **Compiler Settings** to access the **Compiler Properties** dialog.
2. In the **General Compile Control** section, set **licq** to **True**:

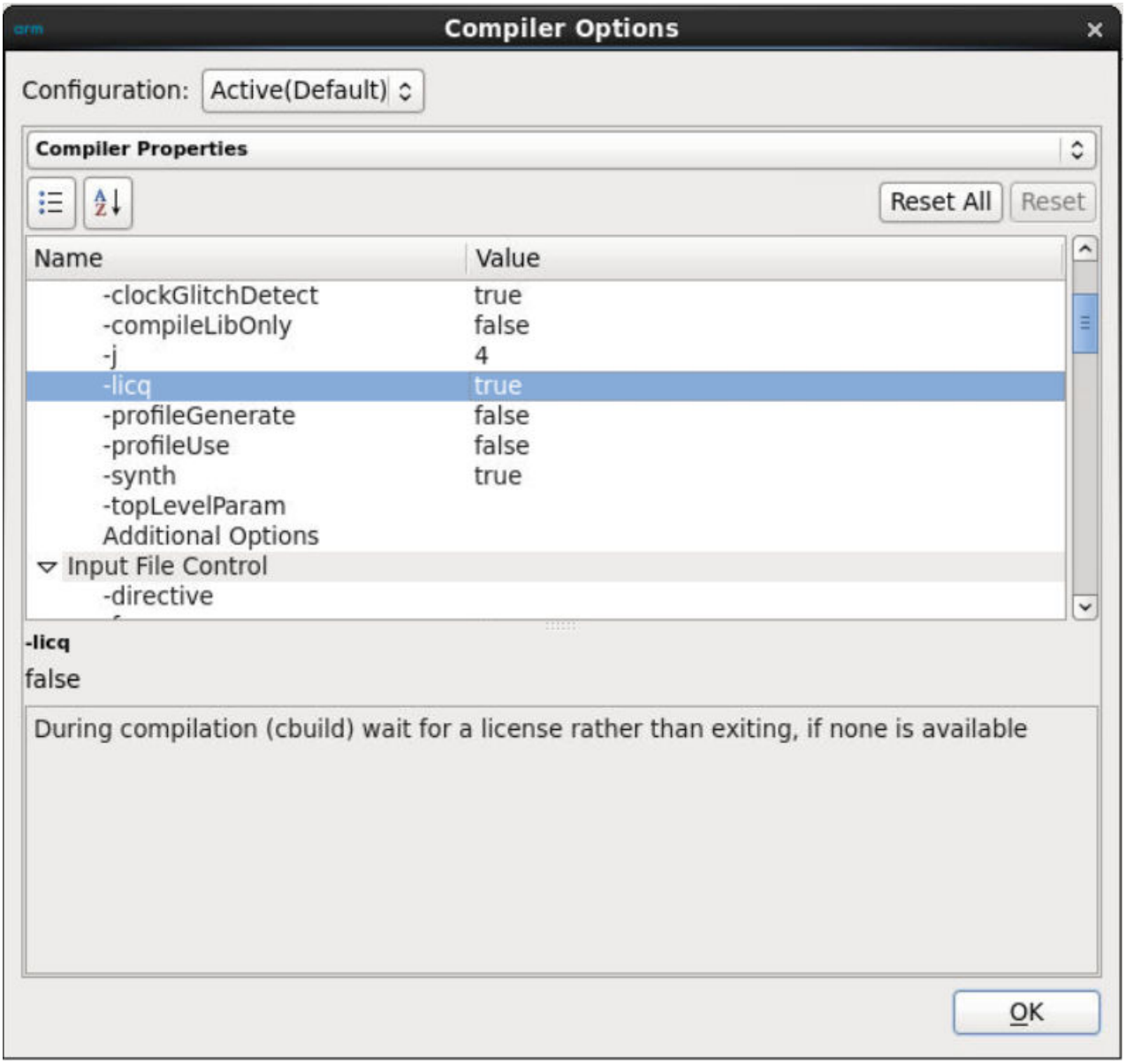

Figure 2-13 Enabing license queuing

3. Click **OK** to close the **Compiler Properties** dialog.

You can also enable license queuing for the Cycle Model Compiler using the `CM_ENABLE_LICENSE_Q` environment variable. See the licensing appendix of the *Arm® Cycle Model Studio Installation Guide* (101106).

2.4.7 Increase debug visibility into design nets (optional)

For debugging purposes, you can increase the visibility of design nets using the **-g** option. **-g** is disabled by default.

The **-g** option disables optimizations that reduce debug visibility to design nets. See the *Arm® Cycle Model Compiler User Guide* (101050) for more information about using this option.

Procedure for enabling -g

To enable the **-g** option:

1. Select **Project** > **Compiler Settings** to access the **Compiler Properties** dialog.
2. In the **Net Control** section, change the **-g** setting to **True**:

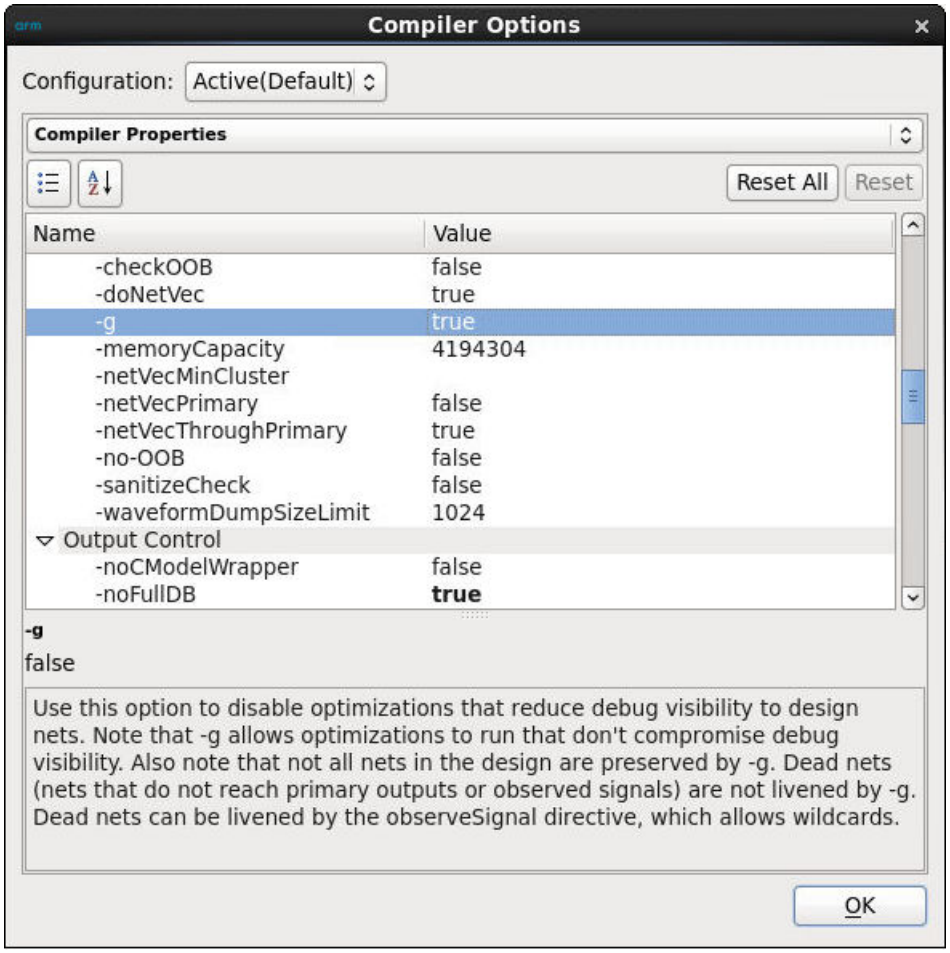

Figure 2-14 Enabling -g

3. Click **OK** to close the **Compiler Properties** dialog.

2.5 Compile the Cycle Model

When your project is ready, compile it into a Cycle Model.

Prerequisites

- Add RTL. See *2.3 Add RTL source files* on page 2-18.
- Set compilation options. See *2.4 Define compiler options* on page 2-20.

Compiling

Before you compile, check for errors in environment variables and paths. Use **Build** > **Check** to check the project components.

When you are ready to compile, click **Compile**:

Figure 2-15 Compile button

The compiler uses the currently-selected Configuration (and all defined settings) to perform the compilation.

If you plan to assign directives to the nets in your design, use **Build** > **Explore Hierarchy**. See *4.3 Assign directives to nets* on page 4-48.

To stop an in-progress compilation, select **Build** > **Explore Hierarchy** or click **Stop**.

Next steps

Generate a component that is compatible with your simulation environment. See *Chapter 3 Creating components for specific simulation environments* on page 3-32.

Chapter 3
Creating components for specific simulation environments

Cycle Model Studio supports creating Cycle Model components for both SystemC and Platform Architect simulation environments. This section describes generating these components and making changes to component ports.

It contains the following sections:

3.1 Create a SystemC component

How to generate a SystemC component using Cycle Model Studio.

Prerequisites

Before you generate a component for SystemC:

- You must have a compiled Cycle Model. See *Chapter 2 Compiling RTL into a Cycle Model* on page 2-15.
- Set the environment variable `SYSTEMCHOME` to the base directory of your SystemC installation.
- Ensure that the Cycle Model output libraries (`lib<design_name>.a`) are static.
- To ensure that the validation engineer can access important signals in the component, use the `observeSignal` and `depositSignal` directives to mark any desired registers or internal memories as observable or depositable.

  Mark internal registers intended to be I/O ports in the component (that is, those to be connected to a transactor or to a port) with `scObserveSignal` or `scDepositSignal`. Be aware that this can slow down performance.

  See *4.3 Assign directives to nets* on page 4-48for more information.

Generating the component

To generate the component:

1. From the Project Explorer, right-click the Cycle Model to display the context menu.

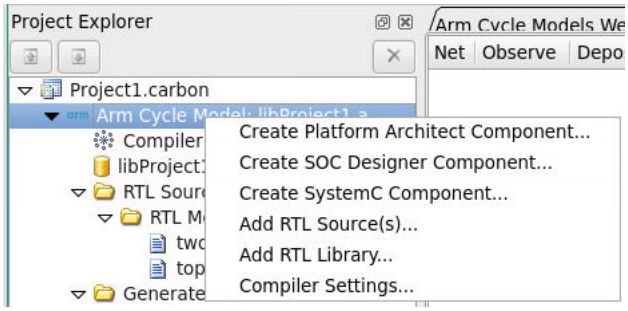

Figure 3-1 Cycle Model context menu

2. Select **Create SystemC Component**. You can also click **SystemC** on the toolbar.

The new component and its associated output files are displayed in the Project Explorer:

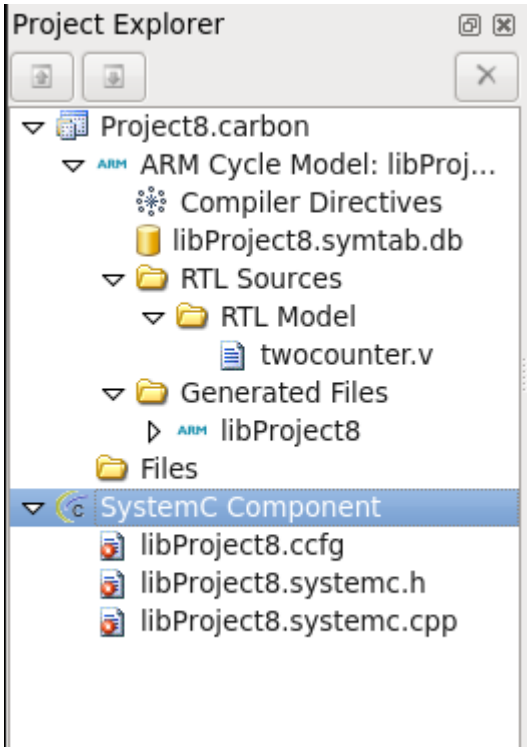

Figure 3-2 Created SystemC component

The component properties view shows basic information about the component. By default, the SystemC module name is the same as the top module name:

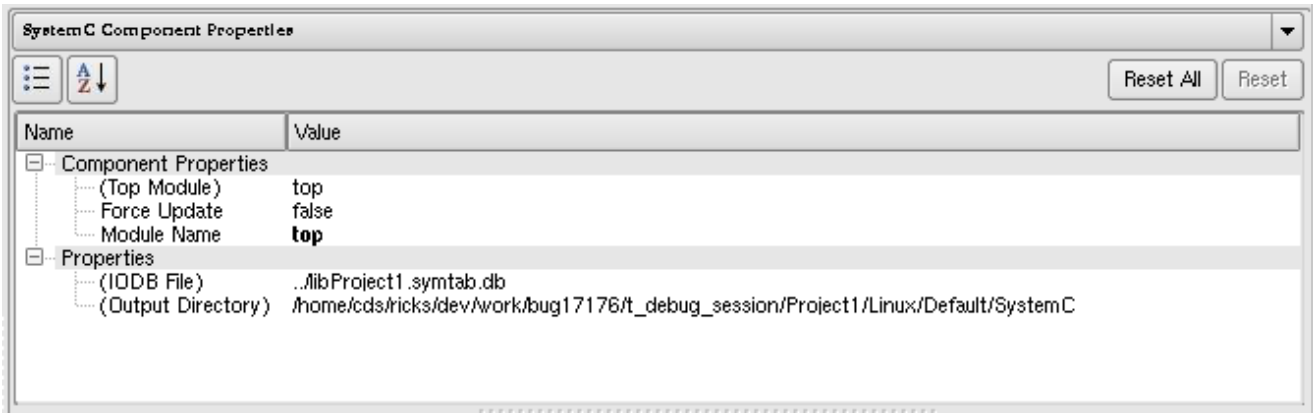

Figure 3-3 SystemC component properties

Next steps

Make changes as needed to the Cycle Model component and complilation settings. Then, recompile.

Related information:

- *2.4 Define compiler options* on page 2-20
- *4.3 Assign directives to nets* on page 4-48

3.2 Create a Platform Architect component

How to generate a Platform Architect component using Cycle Model Studio.

Map data types and define which internal information inside the Cycle Model to export to the Platform Architect environment. Then compile the Component and generate interfaces.

Prerequisites

Before you generate the component for Platform Architect:

- You must have a compiled Cycle Model. See *Chapter 2 Compiling RTL into a Cycle Model* on page 2-15.
- If they have not already been defined, set the necessary environment variables:
  - `COWAREHOME` - *`Platform Architect installation directory`*
  - `CARBON_HOME` = *`Cycle Model Studio installation directory`*
  - `COWAREIPDIR` = *`Platform Architect IP directory`*. This is optional. If not specified, then Cycle Model Studio uses `$COWAREHOME/IP` to identify available Platform Architect transactors.
  - `COWAREIPLIB` = *`Platform Architect installation directory`*. For version-specific instructions, refer to Section 4.5.2.1, COWAREIPLIB Implementation Notes on page 130.)
- Ensure that the Cycle Model output libraries (`lib<`*`design_name`*`>.a`) are static.
- To ensure that the validation engineer can access important signals in the component, use the `observeSignal` and `depositSignal` directives to mark any desired registers or internal memories as observable or depositable.

  Mark internal registers intended to be I/O ports in the component (that is, those to be connected to a transactor or to a port) with `scObserveSignal` or `scDepositSignal`. Be aware that this can slow down performance.

  See *4.3 Assign directives to nets* on page 4-48> for more information.

Generating the component

To generate the component:

1. From the Project Explorer, right-click the Cycle Model to display the context menu.

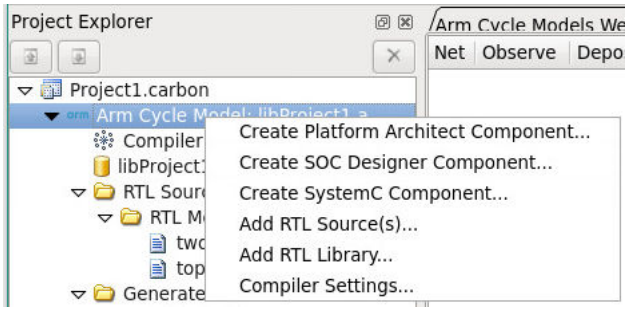

Figure 3-4 Cycle Model context menu

2. Select **Create Platform Architect Component**. You can also click **Synopsys Platform Architect** on the toolbar.

The new component and its associated output files are displayed in the Project Explorer:

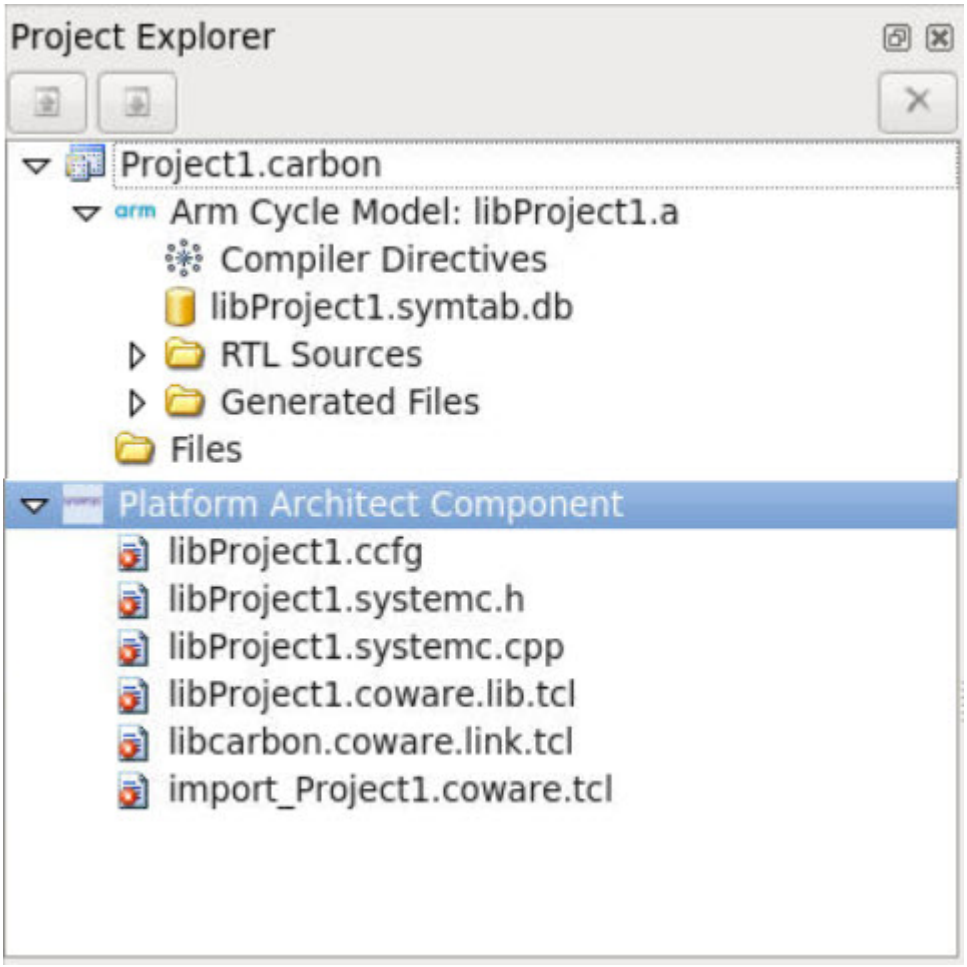

Figure 3-5 Created Platform Architect component

The Component Properties view shows basic information about the component. By default, the SystemC module name is the same as the top module name:

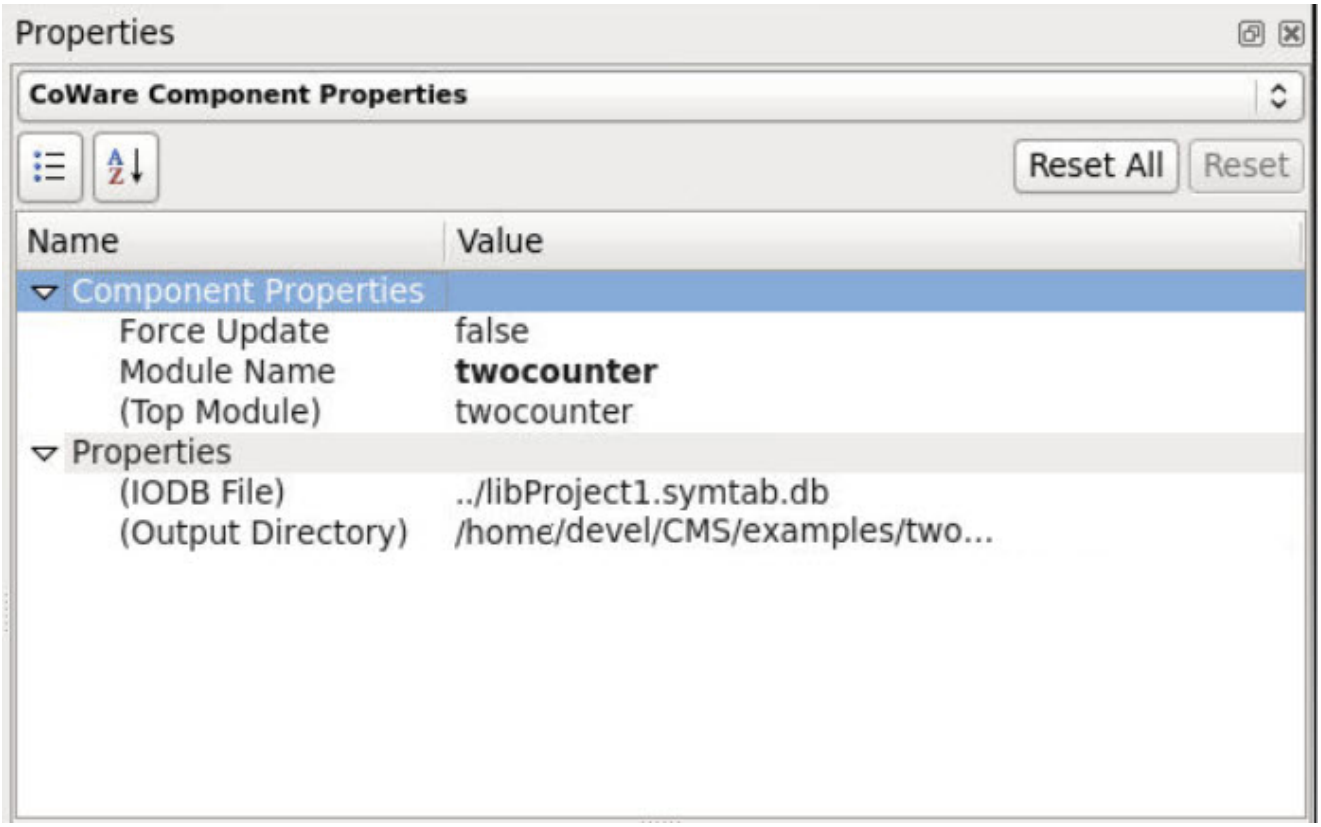

Figure 3-6 Platform Architect component properties

Next steps

Make changes as needed to the Cycle Model component and complilation settings. Then, recompile.

Related information:

- *2.4 Define compiler options* on page 2-20
- *4.3 Assign directives to nets* on page 4-48

3.3 Component Generator output files

The Cycle Model Studio Component Generator outputs several file types, including configuration, source, `make`, and component files.

Output files appear in the Project Explorer panel nested below the new component.

The following figure shows the process flow from Cycle Model to component:

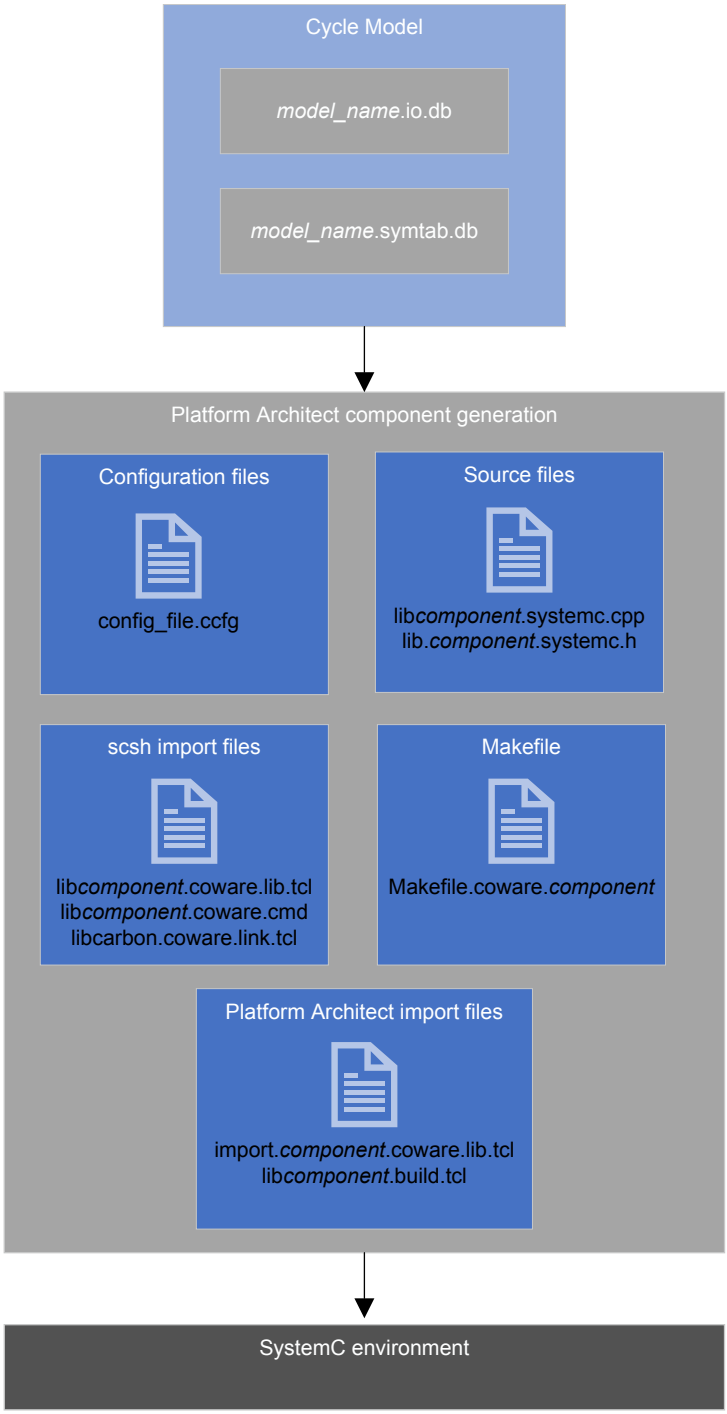

Figure 3-7 Component output files

Configuration file

The configuration file has the extension `.ccfg`. Cycle Model Studio uses this file to create the customizable source files and `Makefile`, which are used to generate the component.

If you are migrating to Cycle Model Studio from the Platform Architect Component Generator tool, use this file as input for a run of Cycle Model Studio. Then make modifications to the component.

Source files

To customize the component prior to passing it to the simulation environment, edit the source files (`*.cpp` or `*.h`).

Make all edits inside the `User Code` section of these files. This ensures that the Component Generator retains your edits if you reuse the edited source files for a future run.

Makefiles

`Makefile.coware`*`component`* creates the component for the Platform Architect environment.

3.4 Configure ports, ties, and disconnects

You can tie Cycle Model input ports (test inputs) to a constant value rather than leaving them exposed as component ports. You can also leave Cycle Model output ports unconnected instead of leaving them exposed as component ports.

Double-click a component's `.ccfg` file in the Project Explorer to display the **Component editor**. The **Component editor** has tabs for editing the component ports, ties, and disconnects. Platform Architect components have additional tabs to edit the component **Registers** and **Memories**.

The figure shows the component editor for a SystemC component:

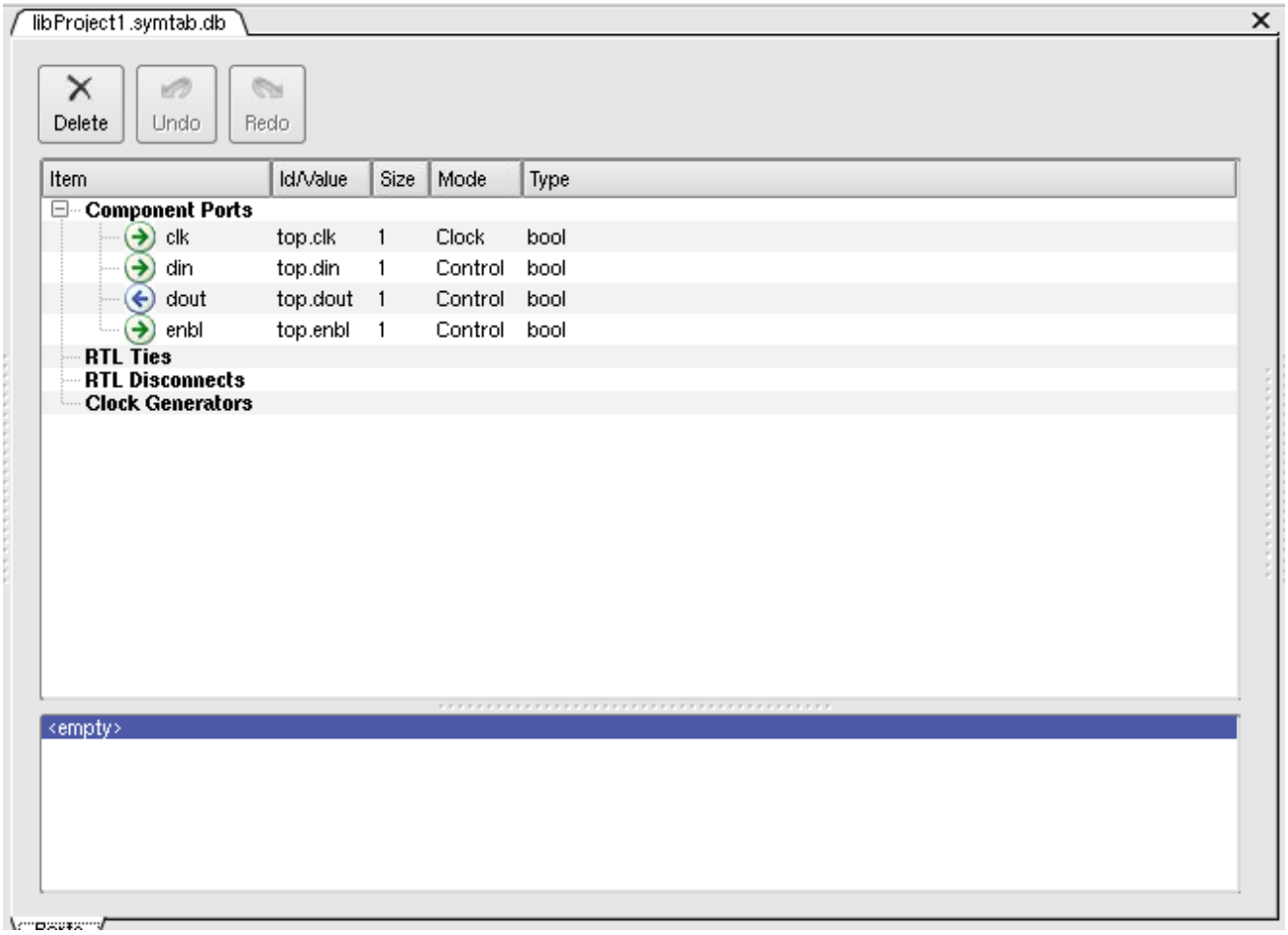

Figure 3-8 SystemC component editor

The RTL ports that are shown include all the primary I/Os of the Cycle Model, plus internal signals that were marked with the following directives when the Cycle Model was compiled:

- `observeSignal`
- `scObserveSignal`
- `depositSignal`
- `scDepositSignal`

The size of the port is determined automatically from the RTL port size.

The Clock Generators for SystemC ports use the SystemC `sc_clock` primitive to generate the clock values.

Changing port names

To change the name of a port:

1. Click on the port you want to rename. The field becomes editable.
2. Type the new name.

Changing port types

The default port **Type** is based on the size, type, and direction of the RTL port.

To change the port to a different type:

1. Click in the **Type** column for the port whose type you want to change. The **Type** field becomes active.
2. Click the currently-assigned type to display the menu options.
3. Select the new type from the menu:

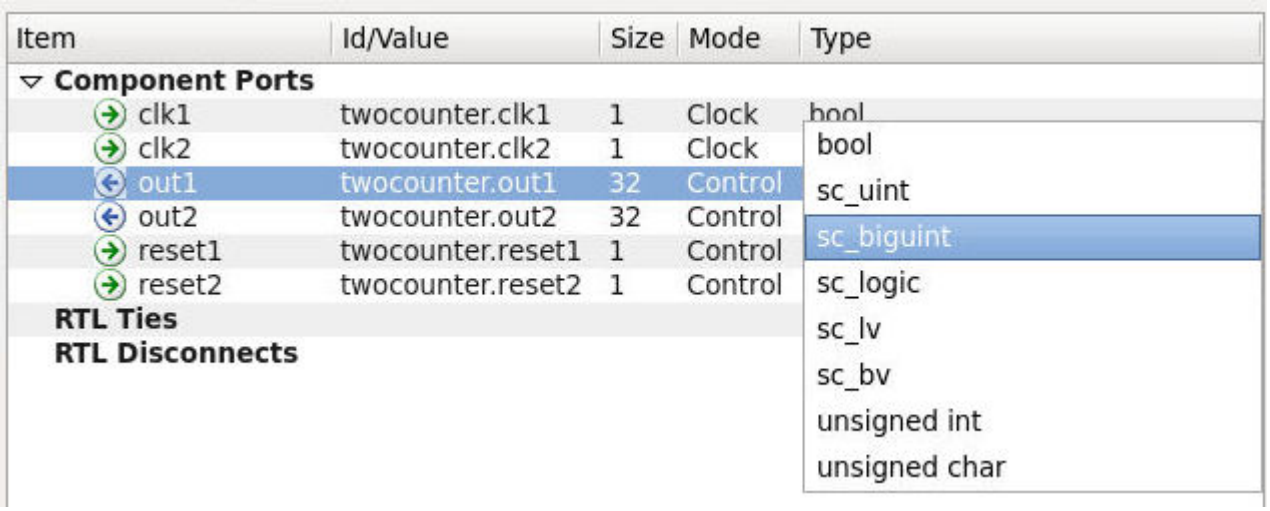

Figure 3-9 Changing port type

Tying off signals

Arm recommends tying off signals that do not need to change during validation, such as `scan-enable`. To tie off input pins:

1. Select the input pin whose signal you want to tie.
2. Right-click and select **Tie High** or **Tie Low** from the context menu.

   The port is moved to the **RTL Ties** section, and removed from the **Component Ports** list. This pin will not be used as an input port.

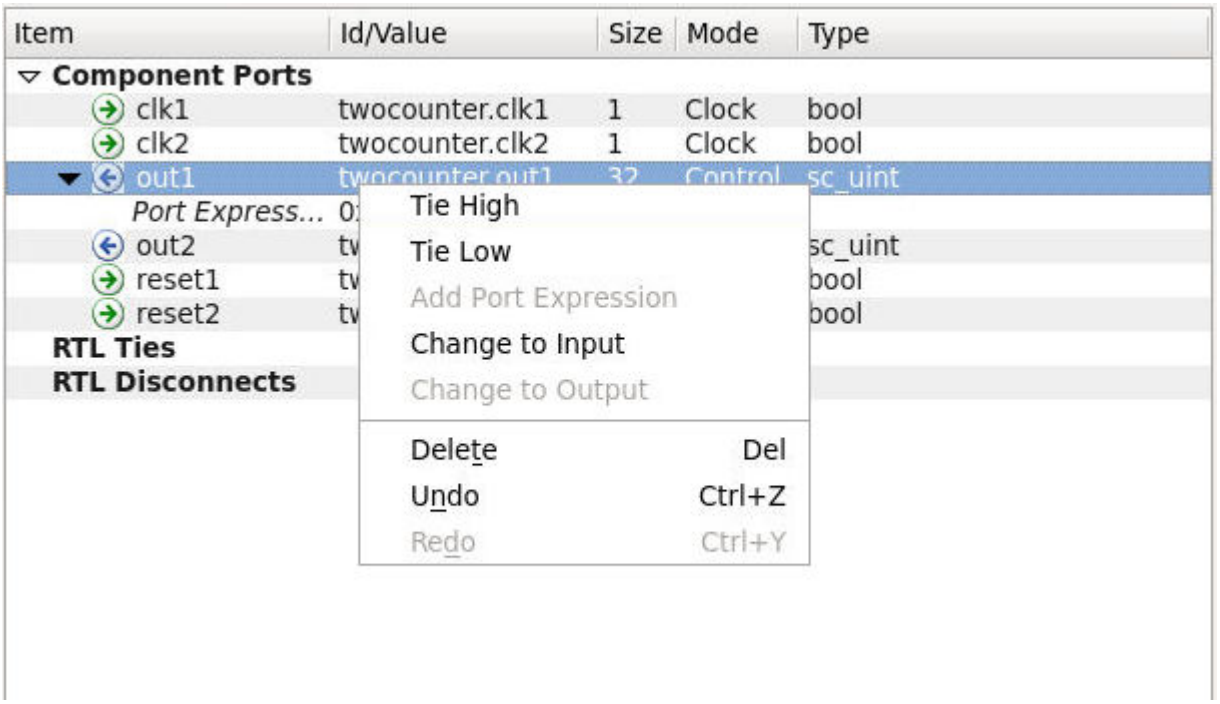

Figure 3-10 Tying ports

To re-add the pin as an input port, right-click the port in the **RTL Ties** section and select **Undo** from the context menu. The port is removed from the **RTL ties** list and appears back in the **Component Ports** list.

Disconnecting signals

To disconnect any input or output pins that are not needed:

1. In the **Component Ports** list, select the pin whose signal you want to disconnect.
2. Right-click and select **Delete** from the context menu.

   The port is moved to the **RTL Disconnects** list, and removed from the **Component Ports** list. This pin will not be used as an input or output port.

   To re-connect the port, right-click the port in the **RTL Disconnects** section and select **Undo** from the context menu. The port is removed from the **RTL Disconnects** list and appears back in the **Component Ports** list.

Specifying generated clocks (SystemC components only)

Clocks indicate when it is time for your component to execute another cycle. HDL designs typically require alternating signal values of 0 and 1. The Cycle Model Studio tool's clock generators automatically stimulate your Cycle Model with signal values on every clock pulse, or on the schedule you configure.

To create a generated clock:

1. In the **Component Ports** list, select the signal you want to designate as a clock.
2. Right-click and select **Create Clock Generator** from the context menu. The signal and default values appear in the **Clock Generators** section, and are removed from the **Component Ports** list.
3. Modify the clock characteristics as necessary in the **Clock Generators** section.

Chapter 4
Advanced features

Describes advanced features of the GUI.

It contains the following sections:

4.1 Create reusable sets of compiler options

You can create multiple sets of compiler properties with settings customized for different stages of development.

For example, you could have the Default standard set of compiler options, and an additional set of compiler options that includes the use of a directives file (`-directive`).

To create a new set of compiler properties:

1. Select **Project** > **Configuration Manager**. The **Edit Configurations** dialog launches:

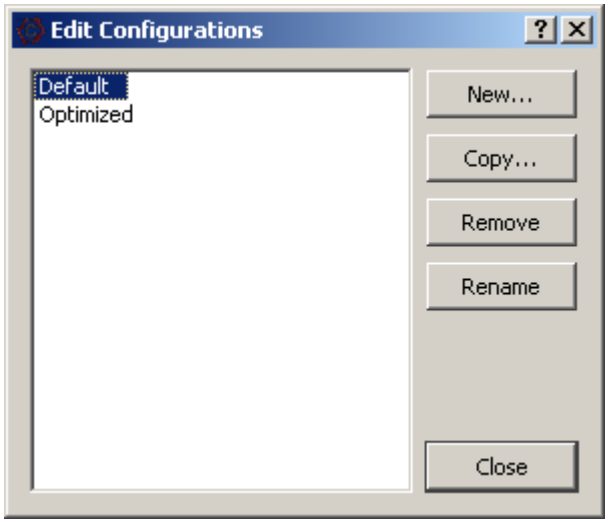

Figure 4-1 Edit Configurations dialog

2. Click **New**. The **Create Configuration** dialog launches:

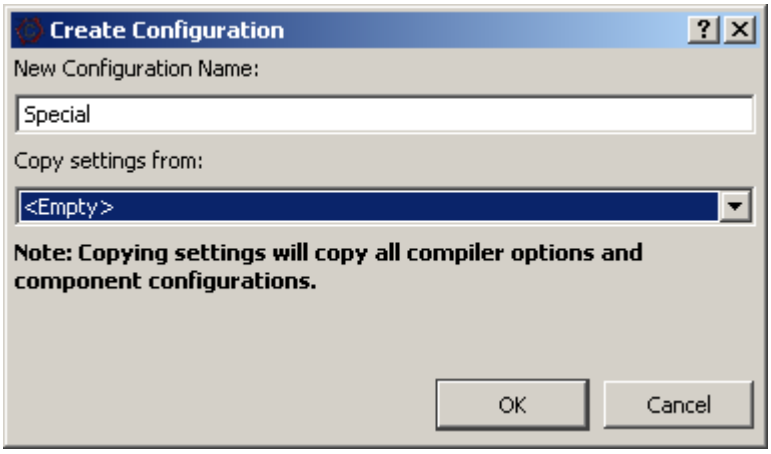

Figure 4-2 Create Configuration dialog

3. In the **New Configuration Name** field, enter a name for the new property set.
4. To copy and customize an existing property set, select it in **Copy settings from**. Click **OK**. The new configuration is now listed in the **Edit Configurations** dialog:
5. **Close** the **Edit Configurations** dialog.

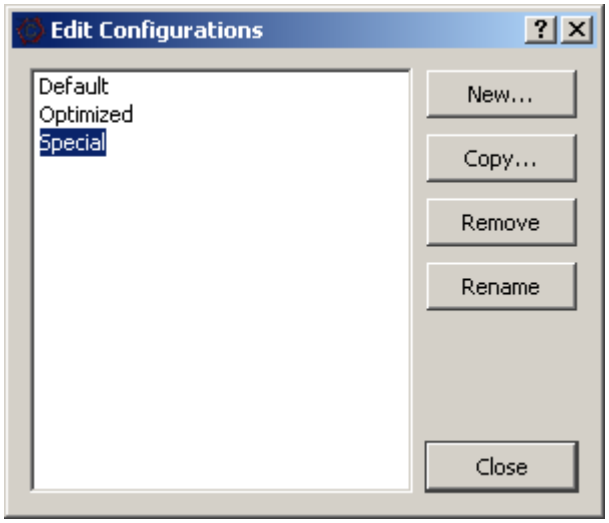

Figure 4-3 Edit Configurations dialog with new set

4.2 Import a configuration file

You can import an existing configuration (`.ccfg`) file into an existing project.

Prerequisites

- To ensure that all RTL sources and Cycle Model compile settings are available for further configuration, your project must have been created using the option **Project from existing Arm Cycle Model Studio Command (.cmd) file**. This is described in *2.2 Create a new project on page 2-17*.
- The `.ccfg` file contains the name assigned to the Cycle Model when it was created. The Cycle Model name used in Cycle Model Studio must match, or a compilation error occurs. To change the name in Cycle Model Studio, go to **Compiler Properties** and use the **-o** option.

Importing a configuration file

1. Select **Project** > **Import Arm Cycle Model Studio Wizard File**.

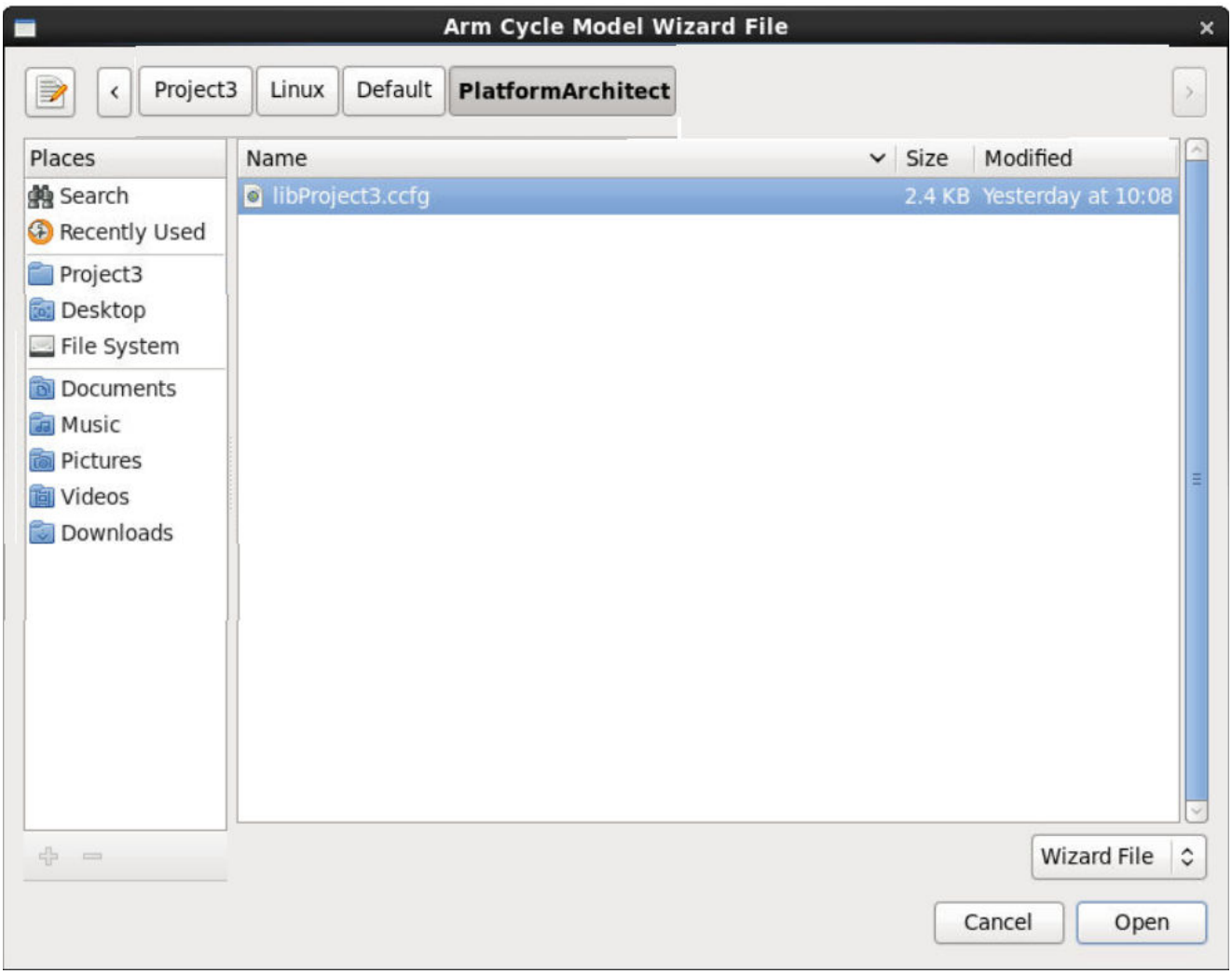

Figure 4-4 Importing a configuration file

2. In the dialog, select the file type.
3. Browse to the file location.
4. Click **OK** to import the file into the project.

4.3 Assign directives to nets

The **Design Hierarchy** view shows your design structure and allows you to manage the functions of design modules and signals.

If you plan to assign directives to the nets in your design, using the Design Hierarchy takes less time than compiling, assigning directives to nets, and then compiling a second time. Modules and nets display in the **Design Hierarchy** window without the need to perform a compile of your RTL source.

Access the **Design Hierarchy** using **Build** > **Explore Hierarchy**.

The left (**Modules**) pane of the **Design Hierarchy** displays the modules in the design tree. The right (**Nets**) pane displays the nets associated with the design modules:

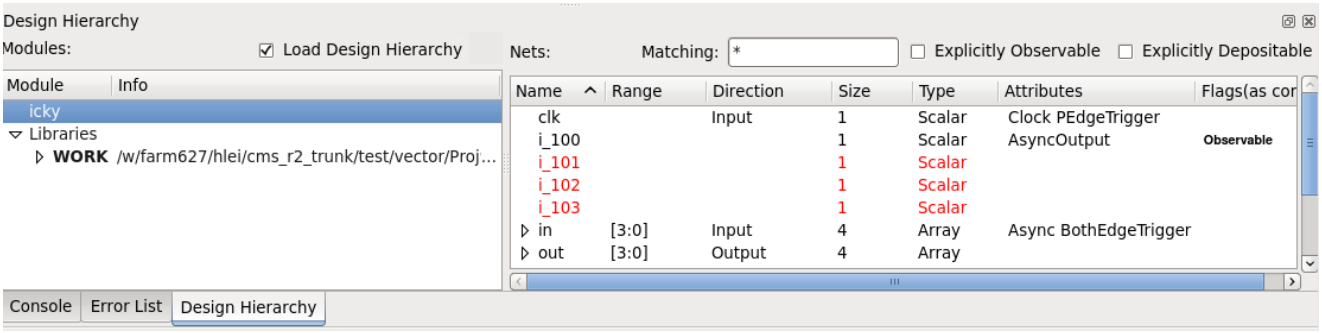

Figure 4-5 Design Hierarchy view

Clicking a module in the **Modules** pane displays the nets associated with that module in the **Nets** pane. If the module contains Verilog parameters, the **Nets** pane shows the parameters for each instance.

Nets displayed in red are not visible (not observable).To access any of these nets, explicitly set the `observeSignal` directive.

To set the `observeSignal` or `depositSignal` directive on the nets in a module, right-click a module in the **Modules** pane.

To display the RTL code for the module, or the RTL code for that instance of the module, in the **Main** view, right-click a module in the **Modules** pane.

To set the `observeSignal`, `depositSignal`, or `forceSignal` directive on a net, right-click the net.