Carbon Model Studio

Version 8.2

User Manual

Non-Confidential

ARM®

ARM DUI 0958B (ID051216)

Carbon Model Studio
User Manual

Release Information

The following changes have been made to this document.

Change History

Date	Issue	Confidentiality	Change
February 2016	A	Non-Confidential	Update for 8.1
May 2016	B	Non-Confidential	Update for 8.2

Non-Confidential Proprietary Notice

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.

Product Status

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

Web Address

http://www.arm.com

Contents

Chapter 3.
Compiling RTL into a Cycle Model

Chapter 4.
Creating a Component for Model Validation

Chapter 5.
Creating Components for Specific Platforms

Chapter 6.
Understanding Component and Platform Interaction

Appendix A.
ARM-supplied Transactors for SoC Designer Plus

Appendix B.
User-Defined Transactors with SoC Designer Plus

Appendix C.
ARM-Supplied RTL Models

Index

Preface

Audience

This user manual is intended for development engineers who create components for use with *SoC Designer Plus*, *Synopsys Platform Architect*, or Cycle Model Validation. It assumes familiarity with the following technologies:

- Hardware design verification
- C/C++ programming language
- Verilog programming language
- The Carbon compiler

About This Guide

This guide provides all the information needed to compile RTL as Cycle Models, generate platform-specific components, and manage interactions between Cycle Models and platforms. The following chapters are included:

- Chapter 1. Introduction
- Chapter 2. Understanding the Carbon Model Studio Interface
- Chapter 3. Compiling RTL into a Cycle Model
- Chapter 4. Creating a Component for Model Validation
- Chapter 5. Creating Components for Specific Platforms
- Chapter 6. Understanding Component and Platform Interaction
- Appendix A. ARM-supplied Transactors for SoC Designer Plus
- Appendix B. User-Defined Transactors with SoC Designer Plus
- Appendix C. ARM-Supplied RTL Models

Conventions

This book uses the following conventions:

Convention	Description	Example
`courier`	Commands, functions, variables, routines, and code examples that are set apart from ordinary text.	`sparseMem_t SparseMemCreate-New();`
italic	New or unusual words or phrases appearing for the first time.	*Transactors* provide the entry and exit points for data ...
bold	Action that the user performs.	Click **Close** to close the dialog.
<text>	Values that you fill in, or that the system automatically supplies.	<platform>/ represents the name of various platforms.
[text]	Square brackets [ ] indicate optional text.	`$CARBON_HOME/bin/modelstudio [ <filename> ]`
[text1 | text2]	The vertical bar | indicates "OR," meaning that you can supply text1 or text 2.	`$CARBON_HOME/bin/modelstudio [<name>.symtab.db | <name>.ccfg ]`

Also note the following references:

- References to C code implicitly apply to C++ as well.
- C side, S side, and test side refer to code or operations written in C or originating in C. Similarly, simulation side refers to code and operations residing within the simulation side of the verification environment.
- File names ending in .cc, .cpp, or .cxx indicate a C++ source file.

Additional Documentation

ARM provides extensive documentation covering the installation of Cycle Models products, the use of transactors, and performance enhancements. Documents supporting these features are listed in the following table.

Document	Contents
Carbon Model Studio Installation Guide	Provides instructions for installing the Cycle Models software products, as well as information about system requirements, environment variables, and licensing.
Carbon Transactors Overview	Overview of architecture and usage of some of the ARM transactors.
Transactor User Manuals	Many manuals describing the interfaces, registers, I/O, packet handling, and validation environments for the supported transactors.
Using Carbon Transactors with SystemC	Describes the process for integrating Cycle Models containing transactors into a SystemC environment.
Carbon SystemC User Manual	Describes the process for integrating a Cycle Model into a SystemC development environment, and examines the case of replacing an existing module in a design with a Cycle Model.
Carbon Model API Reference	Information about Cycle Model API files, data types, and functions.
Carbon System API Reference	The Carbon System API is a library of C functions that allows you to manage Cycle Model components in a simulation system.
Carbon Model Database API Reference	C library to access the Cycle Model database.
Carbon RTL Style Guide	How to write your code to optimize runtime performance.

Glossary

C-side/S-side	Software side. The environment that controls the transactors via interface modules.
Cycle Model	A software object created by the Carbon Model Studio (or *Carbon compiler*) from an RTL design. The Cycle Model contains a cycle- and register-accurate model of the hardware design.
Carbon Model Studio	Tool for the automatic generation, validation, and execution of hardware-accurate software models. It creates a Cycle Model, and it also takes a Cycle Model as input and generates a component that can be used in SoC Designer Plus, Platform Architect, or SystemC for simulation. It is a graphical tool, providing a user-friendly method to manipulate ports, registers and memories; add transactors to the Cycle Model; and set up variables for profiling.
Component	Building blocks used to create simulated systems. Components are connected together with unidirectional transaction-level or signal-level connections.
ESL	Electronic System Level. A type of design and verification methodology that models the behavior of an entire system using a high-level language such as C or C++.
HDL	Hardware Description Language. A language for formal description of electronic circuits, for example, Verilog.
io.db	The *io.db* is a subset of the full *symtab.db*. It includes only the design I/Os, clocks, and any nets that were specified in directives.
RTL	Register Transfer Level. A high-level hardware description language (HDL) for defining digital circuits.
SOC or SoC	System-on-a-Chip
SoC Designer	High-performance, cycle accurate simulation framework which is targeted at SOC hardware and software debug as well as architectural exploration.
S2T	Signal-to-transaction. The component drives the transaction into the SoC Designer or Platform Architect environment.
symtab.db	A *symtab.db* file is a database of the entire design.
T2S	Transaction-to-signal. SoC Designer or Platform Architect drives the transactions into the component.
Transactor	Transaction adaptors. You add transactors to your component to connect your component directly to transaction level interface ports for your particular platform.

Chapter 1

Introduction

The *Carbon Model Studio* is a graphical tool designed for the generation, validation, and execution of hardware-accurate software models. It is designed to simplify the task of compiling an RTL hardware model into a Cycle Model, generate platform-specific components for simulation environments (such as SoC Designer Plus, Platform Architect, and SystemC), and tune Cycle Models for optimal performance during simulations.

1.1 Virtual Prototype Methodology

Figure 1-1 shows how Carbon Model Studio shortens the design simulation flow process by running RTL and software debugging tasks concurrently.

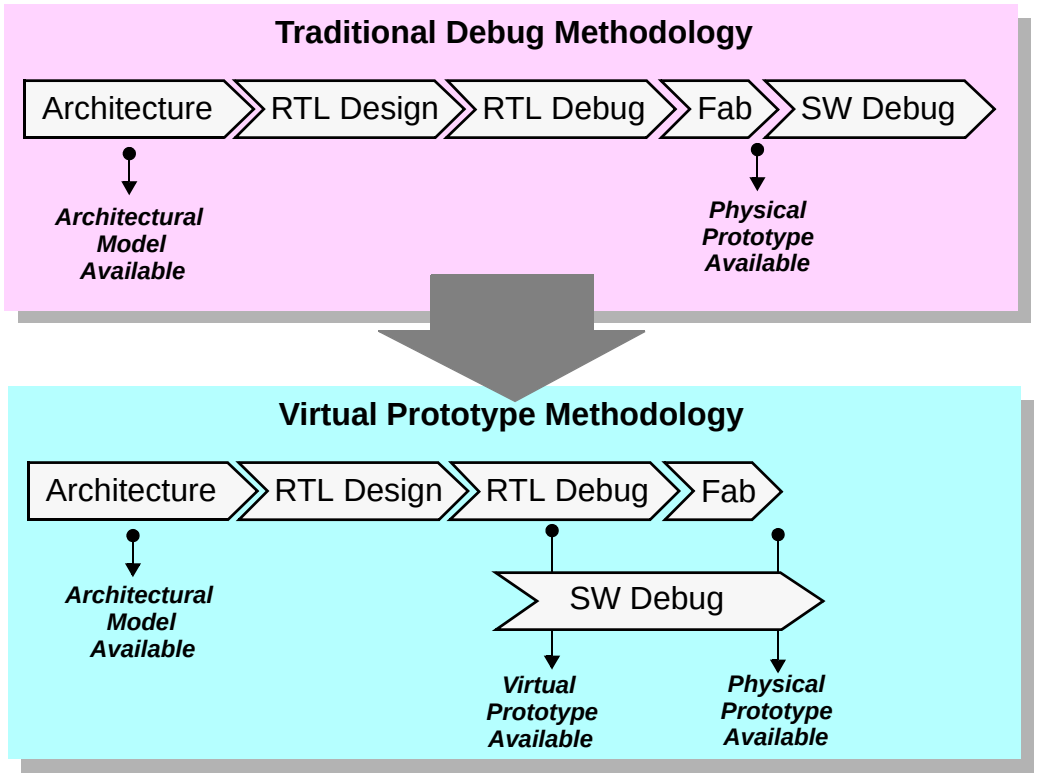


Figure 1-1 Virtual Prototype Benefits

As you can see in the figure, using the Carbon Model Studio enables you to perform software debugging earlier in the cycle so that your final product is available sooner. Also, some additional RTL debug issues may be uncovered during software debug.

The Carbon Model Studio Graphical User Interface (GUI) manages all aspects of the Cycle Model compile and runtime processes.

1.2 Overview of Carbon Model Studio Functions

Figure 1-2 is used throughout this manual. Red borders surrounding a yellow Carbon Model Studio component indicate which Carbon Model Studio function is being described in the chapter.

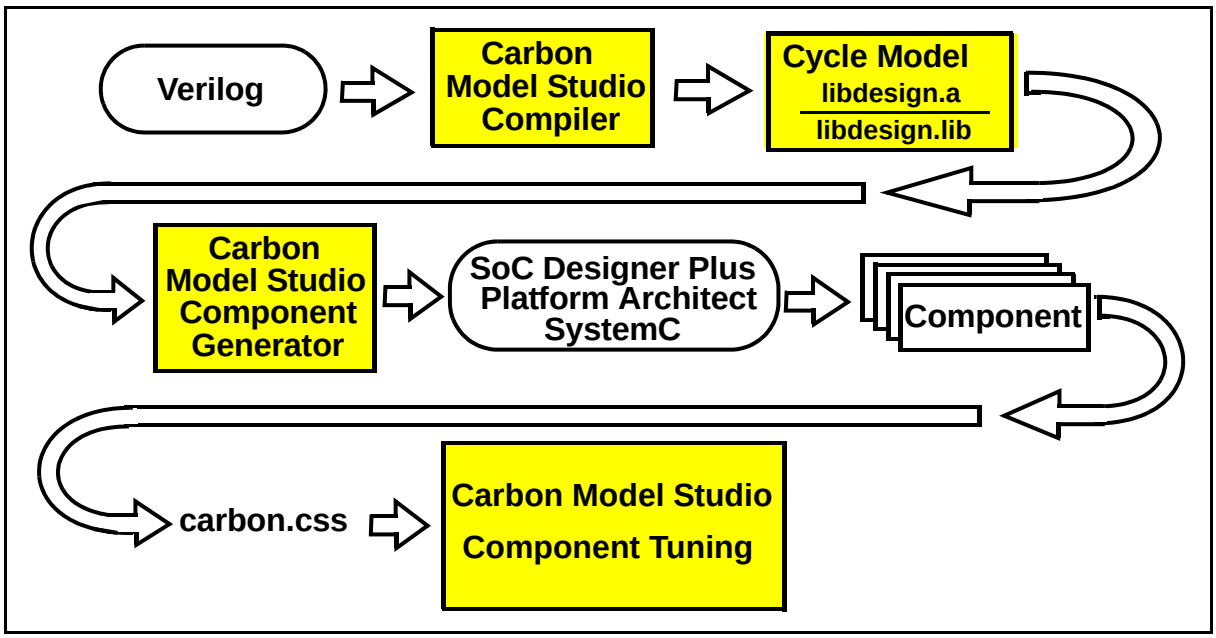


Figure 1-2 Carbon Model Studio Process Flow

The overall process is described below:

1. Create a Project in Carbon Model Studio, and then add the Verilog design and library files to the project.
2. Compile the project using the Carbon compiler. This creates the *Cycle Model*.
3. Generate a component that is compatible with your simulation environment. Carbon Model Studio supports SoC Designer Plus, Platform Architect, and SystemC. You can also generate a component that is compatible with the Carbon Model Validation (MV) tool.
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 Understanding Carbon Model Studio Configuration Options

Carbon Model Studio can be configured to run in either of the following ways:

- Running the compile functions on a Linux server and running the post-compile, runtime (validation) functions in a Microsoft Windows environment.
- Running both compile and runtime functions on a Linux server.

Refer to the *Carbon Model Studio Installation Guide* for more information regarding these options.

1.4 Starting Carbon Model Studio

You should follow the instructions in the *Carbon Model Studio Installation Guide* to make sure you have installed all components successfully, and that the appropriate environment variables have been set, before starting Carbon Model Studio.

Follow the steps below to start Carbon Model Studio:

For Linux, from your working directory, enter:

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

For Windows:

From the *Start Menu*, select **Programs ->Carbon ->Carbon Model Studio *version* ->Carbon Model Studio**

or

From Windows Explorer, go to the location where Carbon Model Studio is installed (the CARBON_HOME), go to the *\Win\bin* directory, and launch **modelstudio.exe**.

Chapter 2

Understanding the Carbon Model Studio Interface

Carbon Model Studio provides an integrated environment that places system validation in parallel with the hardware development flow. The Carbon compiler within the Carbon Model Studio takes an RTL hardware model and creates a high-performance linkable object, called a Cycle Model, that is both cycle and register accurate. The Cycle Model provides the capability for interfacing with your validation environment.

In addition, Carbon Model Studio can compile models that are compatible for use with specific design platforms, such as SoC Designer Plus, Synopsys Platform Architect, and SystemC, or create generic models used for model validation tasks.

The Carbon Model Studio supports a variety of Compile and Runtime functions. *Compile* functions include:

- Loading Projects and Files
- Configuring Compile Settings
- Assigning Directives
- Checking code
- Compiling the Cycle Model

The Carbon Model Studio *Runtime* functions include:

- Running Cycle Models

2.1 Reviewing GUI Features and Functions

The Carbon Model Studio graphical user interface (GUI), shown in Figure 2-1, includes a Menu bar, Toolbar, and a variety of Views that can be docked or undocked to suit your viewing preferences.

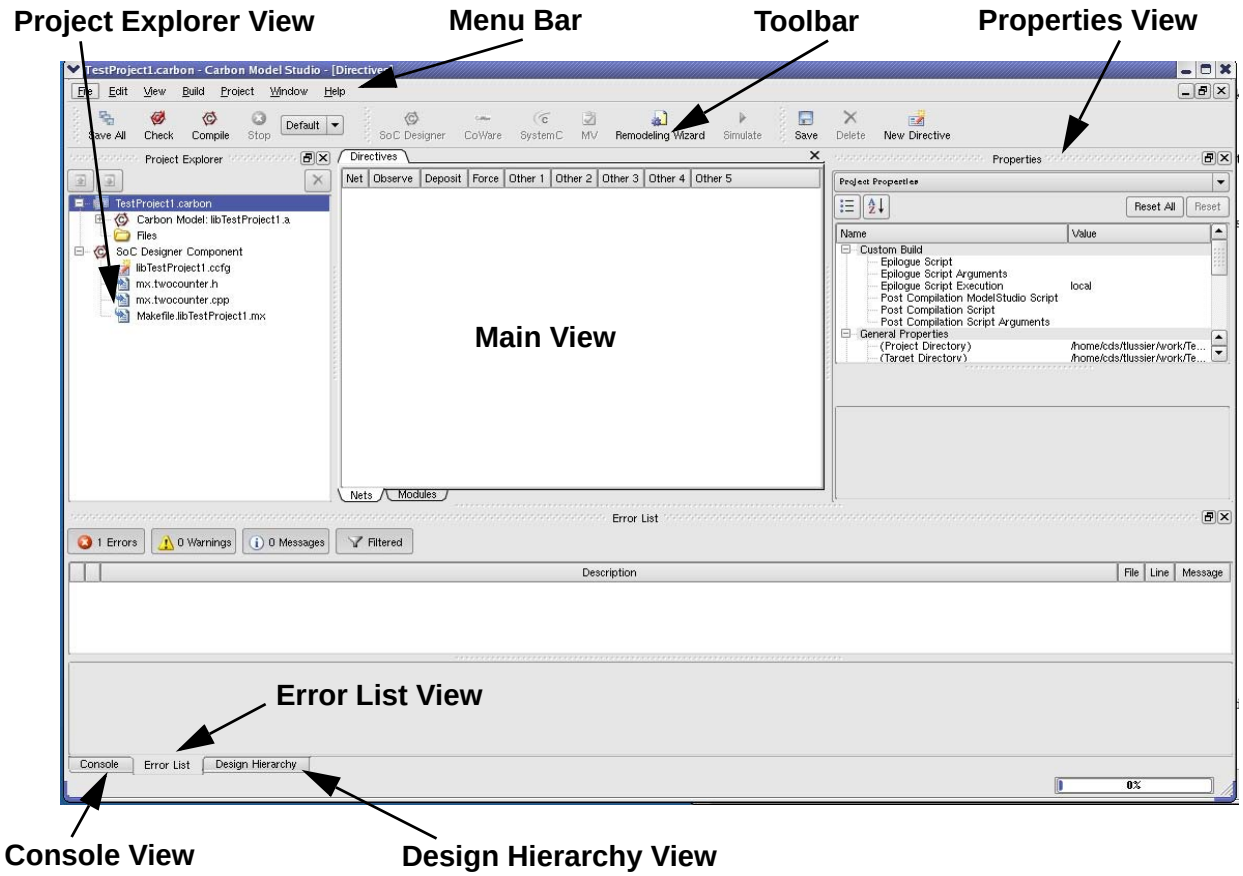


Figure 2-1 Carbon Model Studio Graphical User Interface

The Carbon Model Studio Main menu provides access to the main functional areas of the GUI.

- **File** - Open, close, or import projects and files.
- **Edit** - Perform cut, copy, paste, and editing functions. Set Model Studio preferences.
- **View** - Show or hide the different view windows.
- **Build** - Check your code and compile your project into a Cycle Model.
- **Project** - Manage project settings.
- **Window** - Move the focus to open windows.
- **Help** - Open the *Carbon Model Studio User Manual* PDF, or view version information.

2.1.1 Using the File Menu

Use the *File* menu to:

- Create new projects and files
- Open existing projects and files
- Close the current project
- Save the current project or file
- Perform advanced *find* operations to find files or strings within files
- Define drive mappings (for Windows users compiling on a Linux machine)
- Exit the Carbon Model Studio

2.1.1.1 Creating a New Project

Selecting **File** > **New** displays the New Project dialog, shown in Figure 2-2.

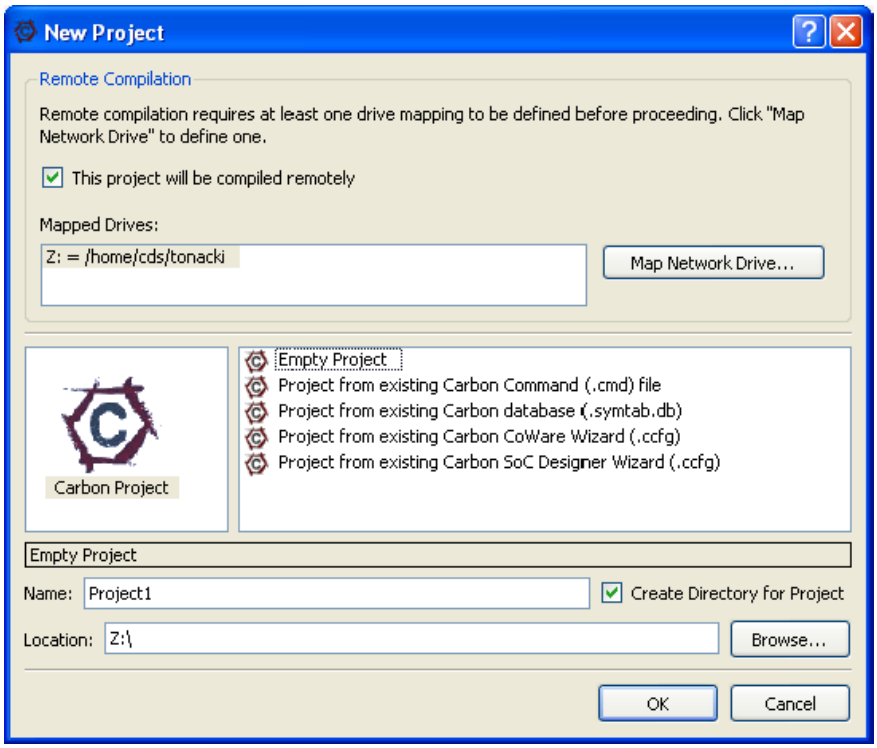


Figure 2-2 New Project Dialog Box

Note that the top part of this dialog box appears only for Windows users who are compiling their RTL project on a Linux machine. Windows users must select the Mapped Drive to be used to perform remote compilation of the Cycle Model. Click **Map Network Drive...** to map a network drive if no drives have been set up. See **Drive Mappings** on page 27 for more information.

Note: If the project you are working on does not need to have the RTL compiled using the Carbon compiler on a Linux machine, you can uncheck the "This project will be compiled remotely" box. This enables you to configure and compile components that run exclusively on Windows. The project types that require the Carbon compiler are removed from the list when you uncheck this box.

To create a new project:

1. In the *New Project* dialog box, you have the following choices:
   - *Empty Project* - Select this option if you are starting a new project from scratch, and have not previously used the Carbon compiler.
   - *Project from existing Carbon Command (.cmd) file* - Select this option if you have an existing command file containing RTL source files and specific compile settings, from a previous Carbon compile that you wish to use to start your new project.
   - *Project from existing database* (*.symtab.db*) - Select this option if you have an existing database file from a previous Cycle Model that you wish to use to initiate a new project.
   - *Project from existing Carbon CoWare Wizard (.ccfg)* - Select this option if you have an existing configuration file from the old CoWare Wizard product and you wish to continue customization of the component.
   - *Project from existing Carbon SoC Designer Plus Wizard (.ccfg)* - Select this option if you have an existing configuration file from the old SoC Designer Plus Wizard product and you wish to continue customization of the component.

Note: You can import an existing .ccfg file into a project as well, as described on page 37*. In that case you will have access to the RTL files to further configure the Carbon Model. The options above still allow customization of Component settings, but will not include RTL source.*

2. In the *Name* field, enter the name for your new project.
3. In the *Location* field, browse to the location where you will store your new project.
4. Click **OK**.

Note: Select the checkbox "Create Directory for Project" to automatically create a new directory using the name of the project.

2.1.1.2 Opening Projects and Files

Select **File** > **Open** to open an existing Carbon Model Studio project or file. Selecting *Open* displays the *Open Project* dialog, shown in Figure 2-3.

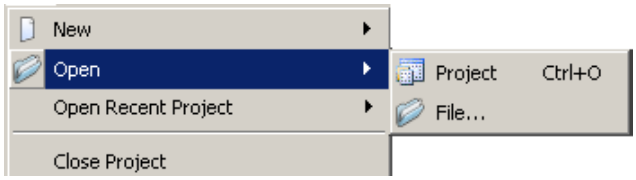


Figure 2-3 File Open Options

To open an existing Project or File:

1. Select **File** > **Open** > **Project** to display the *Open Project* dialog, or select **File** > **Open** > **File** to display the *Open File* dialog (Figure 2-4).

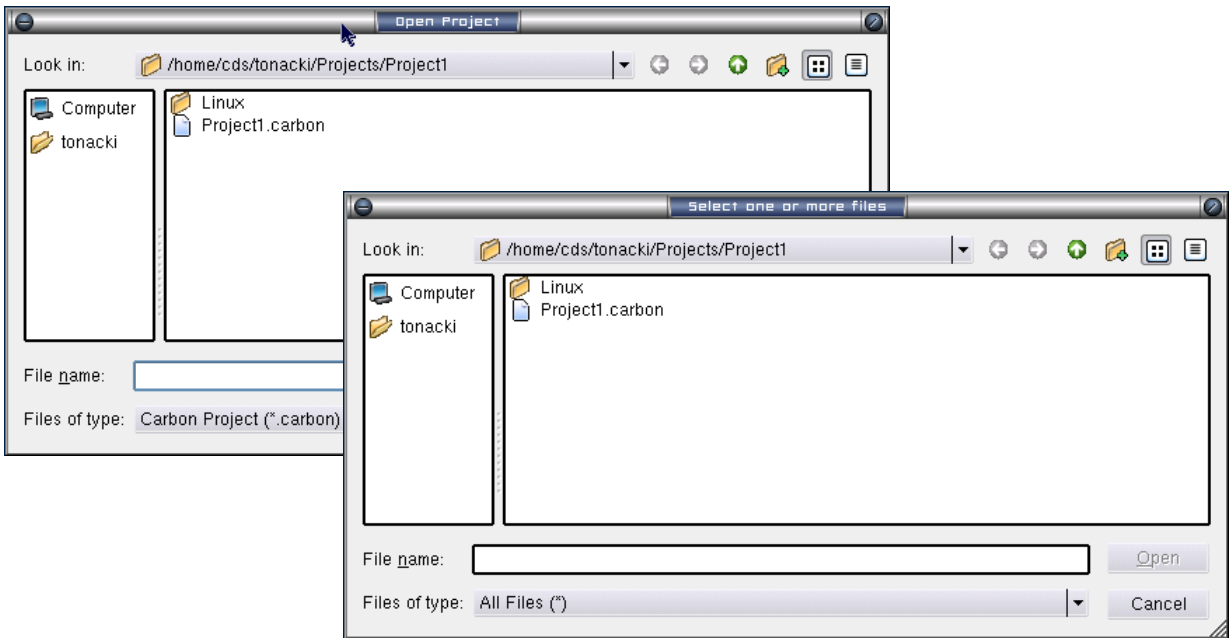


Figure 2-4 Open Project and Open File Dialog Boxes

2. If you select *Open* and *Project*, browse to an existing project and click the *Open* button to open that project. You can also use the *Open Recent Project* option to open any of the most recently opened projects.

   If you select *Open* and *File*, browse to a file of the type listed in the *Files of Type* field, and click the *Open* button to open the selected file in a text editor window.

2.1.1.3 Opening a Recent Project

File > Open Recent Project provides a list of recently-opened projects. Select a project from the list to open that project.

2.1.1.4 Closing a Project

Select **File > Close Project** to close a project while keeping Carbon Model Studio open.

2.1.1.5 Saving All Files

Select **File > Save All** -to save all files associated with the current project.

2.1.1.6 Saving the Current File

Select **File > Save** to save the current file.

2.1.1.7 Using the Find Option

Select **File > Find in Files...** to search for specific strings within the files in your project, or anywhere on your drive. When you select this option, the Search for string in Files dialog box appears (Figure 2-5).

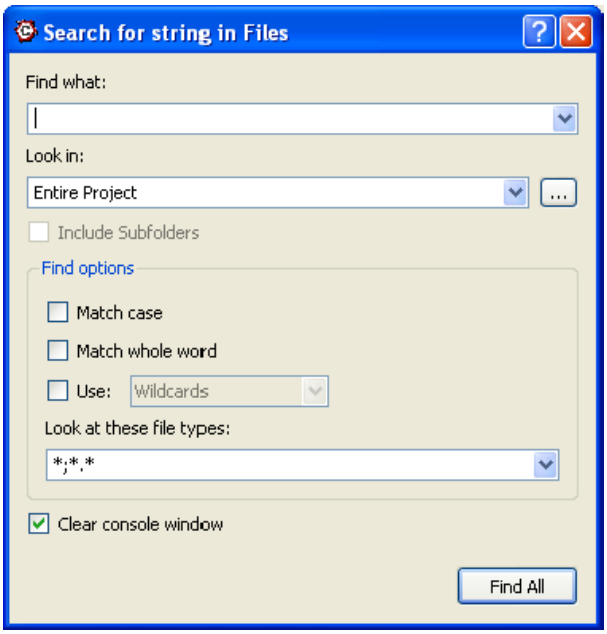


Figure 2-5 Search for Strings Dialog Box

1. Enter the string you want to find in the *Find what* field.
2. Define the scope of the area to be searched in the *Look in* field.
3. Set any additional options you want
4. Click **Find All**.

The files that contain the search criteria are displayed in the *Console* window. Click on each entry in the Console window to display the file that contains the string in the Main view area.

2.1.1.8 Using the Drive Mappings option

The **Drive Mappings...** enables you to manage the Windows Network drive to Linux directory mappings. This menu option is available only for Windows users who are compiling on a remote Linux system. At least one drive mapping must be set up before you can define a new project. When you select this option, the Network Drive to Unix Directory Mappings dialog box appears (Figure 2-6).

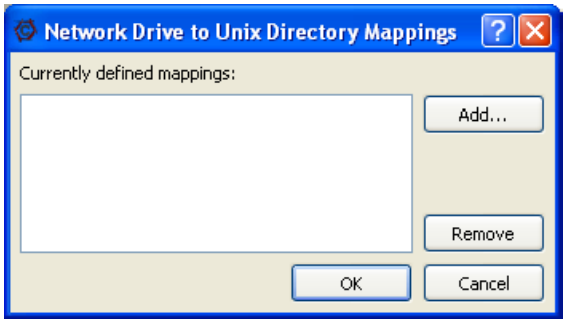


Figure 2-6 Windows Network Drive Mappings Dialog Box

You can add new drive mappings or remove existing drive mappings. Click **Add** to add a new drive mapping; the Map Network Drive to Unix Path dialog box appears (Figure 2-7).

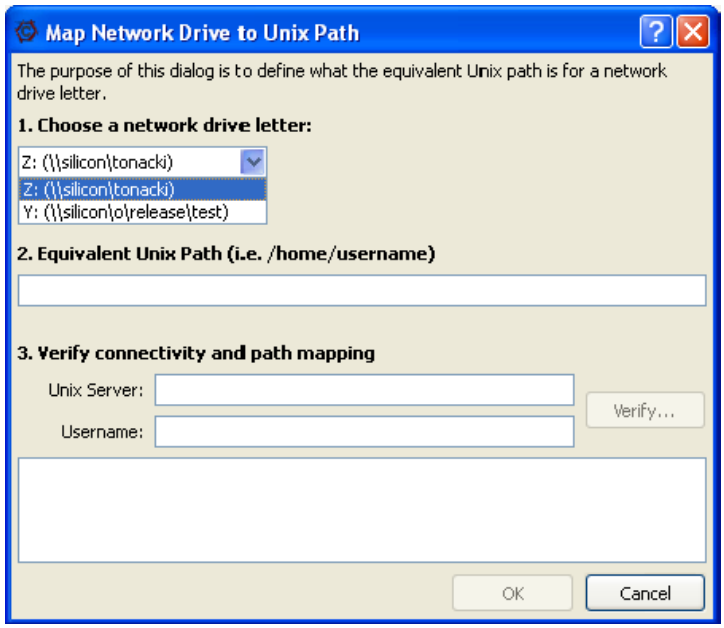


Figure 2-7 Windows Map Network Drive Dialog Box

1. Select a previously-mapped network drive that links a Windows drive letter to an area on a Linux machine.
2. Enter the Linux path on the machine where the project files will be located.
3. Enter the name of the Linux Server and the Username to be used to log in to the server.
4. After you have entered this information, click the **Verify** button to verify that the mapping is set up correctly. You must enter the password for the remote Linux server before verification can complete.

2.1.1.9 Exiting Carbon Model Studio

Selecting **File > Exit** closes all open projects and exits Carbon Model Studio. If any changes have not been saved, you are prompted whether or not you want to save the changes before closing the project.

2.1.2 Using the Edit Menu

Use the *Edit* menu to:

- Set Carbon Model Studio preferences
- Cut, copy, paste, and perform other editing functions

Note: The editing functions are available only when an item is open in the Main window, such as a source file.

2.1.2.1 Setting Preferences

Selecting **Edit > Preferences** enables you to set Carbon Model Studio preferences that apply to all created projects:

- *General Preferences* — Set project-wide preferences, such as behavior on startup, log file creation, Visual Studio version, and environment variable value and application.
- *Text Editor Preferences* — Affect how text is displayed in the Main window.

Figure 2-8 shows the General Preferences dialog box.

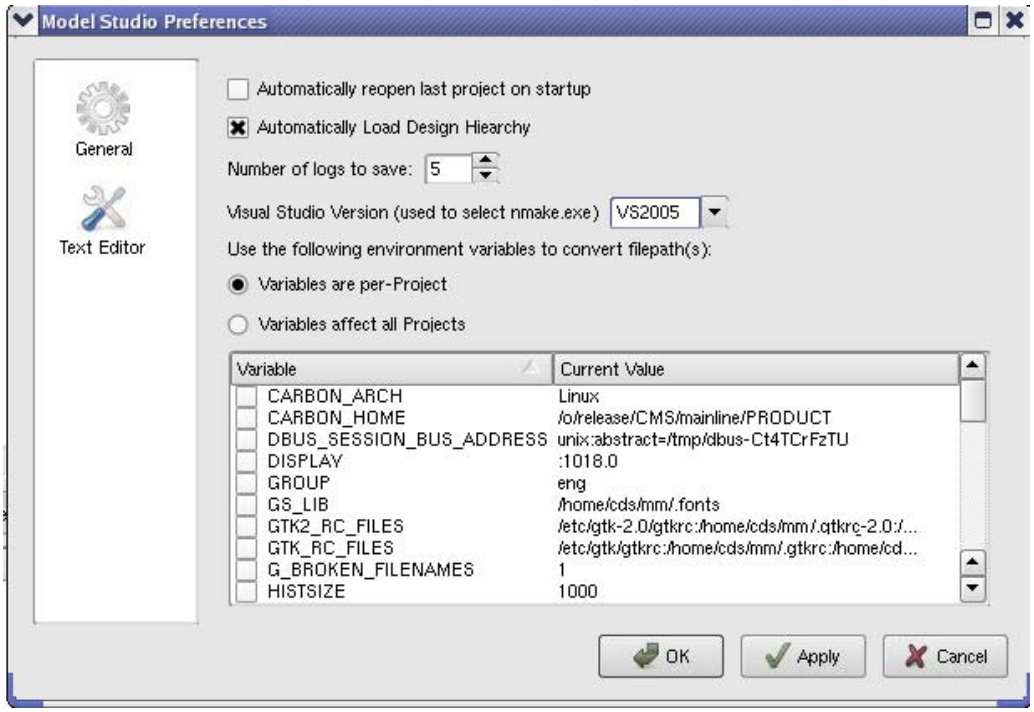


Figure 2-8 Carbon Model Studio Preferences

Automatically reopen last project on startup — Automatically opens the last active project when you start Carbon Model Studio. If you want a blank project to be opened instead, then deselect this box.

Automatically Load Design Hierarchy — Automatically loads the Design Hierarchy view, described in "Using the Design Hierarchy View" on page 56, when you start Carbon Model Studio.

Number of logs to save box — Enables you to specify the maximum number of log files to save for each project. When that number is reached, old files are deleted as new files are written.

Visual Studio Version (used to select nmake.exe) drop-down menu — Applies only to the Windows version of Carbon Model Studio. It allows you to specify which version of Microsoft Visual Studio you are using. Carbon Model Studio uses this setting to determine which version of *nmake.exe* to use when building Windows SoCD components.

Use the following environment variables to convert file path(s) — Enables you to select environment variables to be used in place of hard-coded file path values. This can be helpful when multiple users are working on a single project, and when a source control system is being used.

For example, if Verilog files are stored in a source control system under the *RTL* directory (e.g. *RTL/CPU/file1.v* or *RTL/MEM/mem2.v*), you could create the RTL_HOME environment variable for each user who needs access to the source files. As you add files to the CMS project, they are written into the Project file as *$(RTL_HOME)/CPU/file1.v* (instead of */home/cds/tonacki/Projects/RTL/CPU/file1.v*). When another user has the same environment variable set to

their local work area, for example, */home/cds/bsmith/Projects/RTL*, they are able to successfully open the Project file.

You can select whether the environment variables are used for the current project only, or for all projects.

Important: *This environment variable must be selected (checked) in the Preferences page BEFORE you begin adding source RTL files into the project.*

2.1.3 Using the View Menu

The *View* menu allows you to tailor the Carbon Model Studio user interface (UI) to your needs by adding or removing UI components. To add or remove UI components, click to select or deselect the listed views.

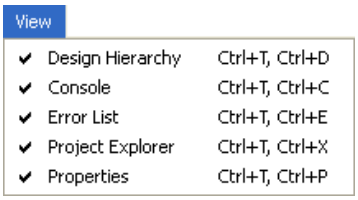


Figure 2-9 Carbon Model Studio View Menu

Note that the Main view window cannot be removed or hidden.

2.1.4 Using the Build Menu

The *Build* menu provides the functions needed to compile, check, recompile, and clean Cycle Models.

2.1.4.1 Checking for Errors

Use **Build** > **Check** to check the project components, such as environment variables and paths, for any errors prior to compiling the Cycle Model. This performs the same functions as the **Check** button.

2.1.4.2 Exploring the Modules and Nets Hierarchy

Use **Build** > **Explore Hierarchy** to display modules and nets in the *Design Hierarchy* window without having to first perform a compile of your RTL source. This is useful if you know you will be assigning directives to the nets in your design. This option takes less time than performing a Compile in the usual way, then assigning directives to nets, and then having to Compile a second time.

2.1.4.3 Compiling

Use **Build** > **Compile** to compile Verilog RTL into a Cycle Model. The compiler uses the currently-selected Configuration (and all defined settings) to perform the compilation. On Windows, you are prompted to configure remote server information — refer to "Reviewing Remote Compile Requirements" on page 60.

See “Configuration Manager” on page 35 for more information on defining configurations. This performs the same functions as the **Compile** button.

2.1.4.4 Packaging

Use **Build** > **Package** to build the package that was defined using Package Options (launched from *Project->Package Options*). The package can be moved to other computers and installed there for further work on the component. See Chapter 9, Distributing Carbon Models for more information. This performs the same functions as the **Package** button.

2.1.4.5 Stopping Compilation or Simulation

Use **Build** > **Stop Compilation** to cancel any compile or simulation that is underway. This performs the same functions as the **Stop** button.

2.1.4.6 Recompiling

Use **Build** > **Recompile** to clean out previously generated files (using the Clean functionality described below), and then compile the Cycle Model.

2.1.4.7 Using Batch Build...

Use **Build** > **Batch Build**... to build all defined configurations. When you select this option, the *Batch Build* dialog box appears (Figure 2-10).

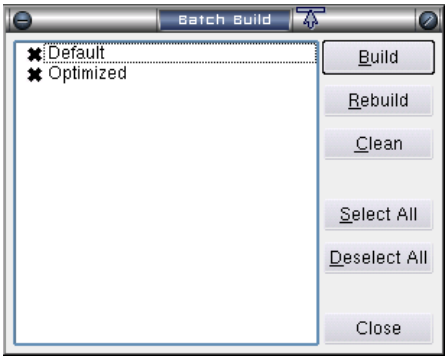


Figure 2-10 Batch Build Dialog Box

All defined configurations appear in this dialog box. Select or deselect the configuration that you want to compile and then click **Build**. See “Configuration Manager” on page 35 for more information on defining configurations.

2.1.4.8 Cleaning Files

Use **Build** > **Clean** to remove all files generated during a compile. This option removes all old files that could possibly be stale. Note that the next compile will take a much longer time than usual, in order to create all the files from scratch again.

2.1.5 Using the Project Menu

The *Project* menu provides access to the Compiler settings, Component build operations, and Configuration Manager options.

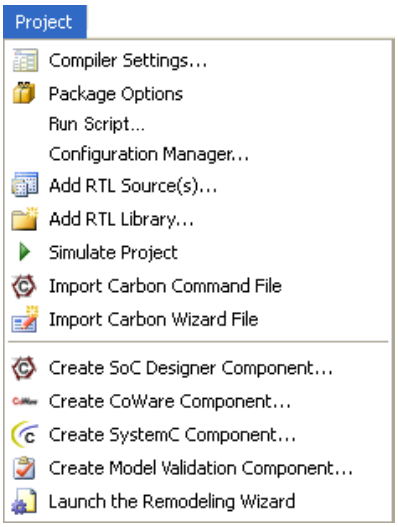


Figure 2-11 Carbon Model Studio Project Menu

Note: Some of these options may be unavailable, depending on the type of license you have purchased from ARM.

2.1.5.1 Configuring Compiler Settings

Project > Compiler Settings enables you to configure the parameters needed for compiling Cycle Models. After you select a specific Compile Property Set, all changes you make in the Properties pages are applied to that Configuration, and used the next time you compile any project using that configuration.

Note: This is equivalent to selecting the Carbon Model or RTL Sources entry in the Project Explorer view, and then editing entries in the Properties view.

To select a compile properties set, or configure your compile properties:

1. From the *Project* menu, select *Compiler Settings* to display the *Compiler Options* dialog, shown in Figure 2-12.

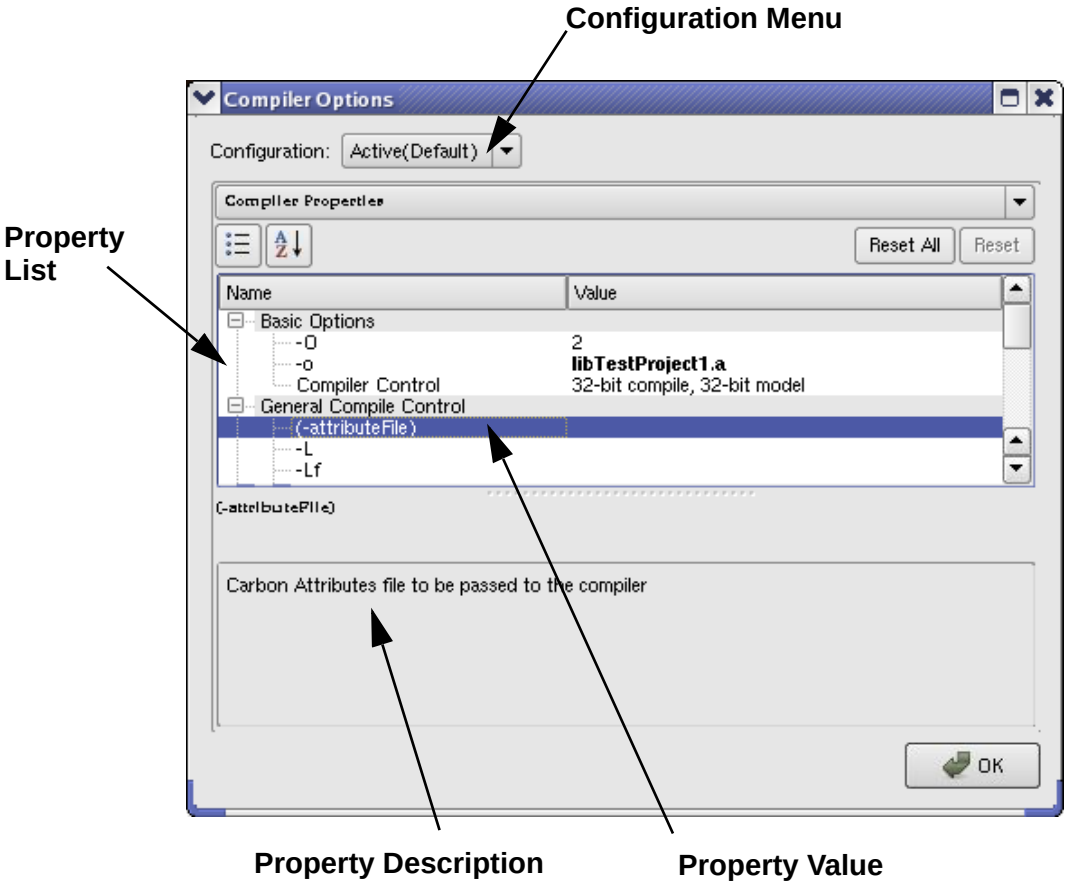


Figure 2-12 Compiler Options Dialog Box

2. At the top of the dialog, click the down-arrow in the *Configuration* menu to select the configuration for which you want to set the Compile Properties. (For instructions on creating new configurations, refer to Creating New Compile Properties Sets). You can define unique sets of properties that contain different compile settings. After you select a Configuration name, your modifications apply to that Configuration.
3. Click any property value field to edit that particular property.
4. When you have completed your configuration, click **OK** to save the settings. These new property settings are used the next time you compile any project using the selected Configuration.

2.1.5.2 Defining Package Options

Use **Project > Package Options** to define the contents of the package that you want to redistribute. For a Cycle Model, the package includes headers, libraries, makefiles, scripts, etc. For a component generated by Model Studio, such as a component for SoC Designer Plus, the package includes the component's shared library, plus any other runtime dependencies.

See "Defining the Contents of the Model Package" on page 4 for complete details.

2.1.5.3 Using the Run Script Option

Use **Project > Run Script** to execute a script. This displays the Execute Script dialog box (Figure 2-13).

Carbon Model Studio provides the Execute Script option to support specialized functionality. Consult with ARM Technical Support for more information.

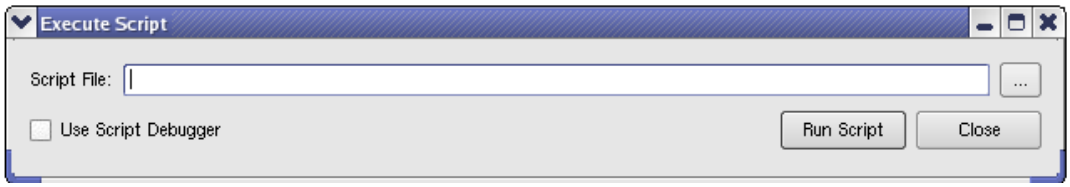


Figure 2-13 Execute Script Dialog Box

2.1.5.4 Creating New Compile Properties Sets

Use **Project > Configuration Manager** to create a new Compile properties set. This displays the *Edit Configurations dialog* (Figure 2-14):

1. Click **New** to create a new property set. The *Create Configuration* dialog box appears with the default configuration name <*New*>.

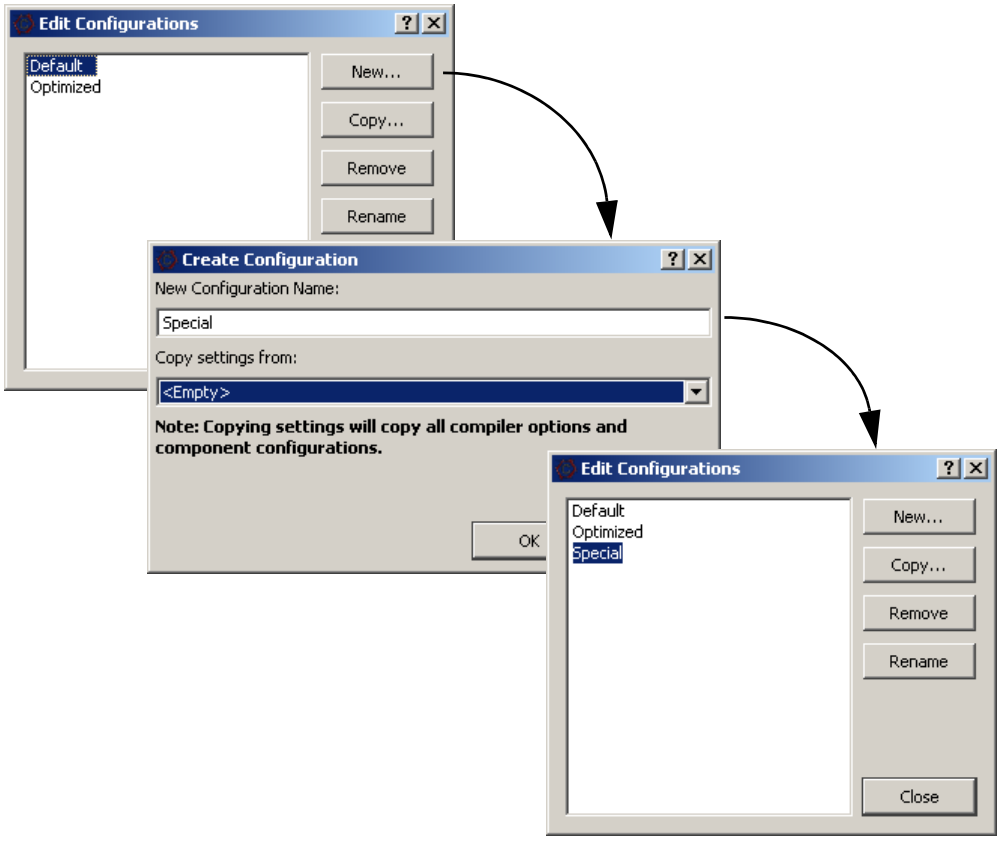


Figure 2-14 Configuration Manager Dialog Boxes

2. Type a new name to replace <*New*> for your new property set and click **OK**. You are returned to the *Edit Configurations* dialog box.
3. Click **Close** to close the dialog.
4. From the *Project* menu, select *Compiler Settings...* to display the *Compiler Options* dialog.
5. Select the new *Compile Properties Set* you have created, and edit the property parameters as needed (see Configuring Compiler Settings for more information).
6. Click the **OK** button to save your settings.

2.1.5.5 Adding RTL Sources

Select *Project > Add RTL Sources...* to add RTL source files to your project. When you select this option, the Select RTL Sources dialog box appears (Figure 2-15).

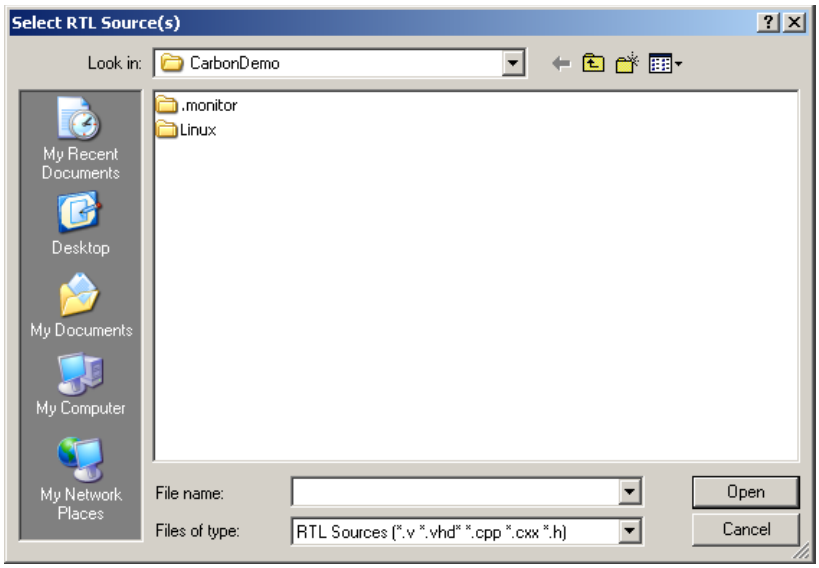


Figure 2-15 Select RTL Sources Dialog Box

Select the RTL source file you want to add and click **Open**. The source file is added to the source file list in the Project Explorer view. The new file or files are used the next time you compile your project.

2.1.5.6 Simulating a Project

Carbon Model Studio can invoke simulations of components from your simulation environment, whether that environment is SoC Designer Plus, Platform Architect, SystemC, or a custom application. **Project > Simulate Project** enables you to specify what tools to invoke, how to invoke them, and what variables or arguments to use. This option is available only after you define simulation parameters in the *Simulation Control* section of the *Project Properties*.

See "Launching a Simulation" on page 153 for complete details.

2.1.5.7 Importing a Command File

Project > Import Carbon Wizard File enables you to import an existing Command (.cmd) file into the current project. This applies all the command settings that exist in the Command file to the current project. This is useful if you have created a Command file with many custom settings that you want to apply to other Carbon Model Studio projects.

Note that you can create a new project and apply the Command file to the new project as well.

2.1.5.8 Importing a Wizard File

Project > Import Carbon Wizard File enables you to import an existing Wizard (.ccfg) file into the current project. This is useful if you had previously generated a component using the SOC-VSP Component Generator for SoC Designer Plus or SOC-VSP Component Generator for Platform Architect.

Typically, this option is used by customers who used one of the older Wizard products (listed above) and who now want to continue the customization of the Component using Carbon Model Studio. In these cases, the initial project should be created using the existing Command file (described on page 24) so that all RTL sources and Cycle Model compile settings are available for further configuration.

Warning: *The Carbon Model name in Carbon Model Studio must be the same as the Carbon Model name used when you initially created the .ccfg file using the Wizard. The .ccfg file contains the name of the original Carbon Model and will cause a compilation error if the same name is not used. You can define the name in the Compiler Properties page using the -o option.*

When you select this option, the *Import Pre-existing Carbon Wizard* dialog box appears (Figure 2-16).

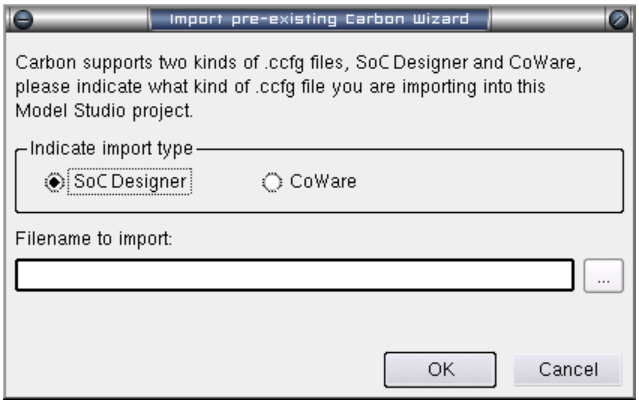


Figure 2-16 Import Carbon Component Wizard File Dialog Box

Select whether the file is a SoC Designer Plus or Platform Architect (CoWare) wizard file, browse to the file location, and click **OK** to import the file into the project.

Note: *You can create a new project based only on the .ccfg file, as described on page 24. However, in that case you will not have access to the RTL files to further configure the component.*

2.1.5.9 Creating a SoC Designer Plus Component

Use **Project > Create SoC Designer Component...** to create an SoC Designer Plus-compatible component from the Cycle Model. System engineers use the component in SoC Designer Plus to build a simulatable system and perform validation and debug operations.

See "Starting and Configuring the Project" on page 92 for complete details.

2.1.5.10 Creating a Platform Architect Component

Project > Create Platform Architect Component... creates a Platform Architect-compatible component from the Cycle Model. System engineers use the component in Platform Architect to build a simulatable system and perform validation and debug operations.

See "Platform Architect-Specific Instructions" on page 143 for complete details.

2.1.5.11 Creating a SystemC Component

Project > Create SystemC Component... creates a component that can be linked directly into a SystemC design environment.

See "Creating Components for SystemC" on page 148 for complete details.

2.1.5.12 Creating a Model Validation Component

Project > Create Model Validation Component... generates a component to be used with the Carbon Model Validation tool.

See Chapter 4, Creating a Component for Model Validation for complete details.

2.1.6 Using the Window Menu

The *Window* menu provides control for the various view windows within the Carbon Model Studio GUI.

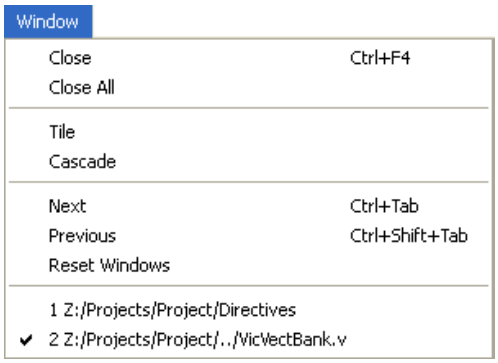


Figure 2-17 Carbon Model Studio Window Menu

Close — Closes the particular view or pane that is currently active (the one that is checked in the list).

Close All — Closes all project panes, including the active one.

Tile — Displays multiple components within a view side by side, replacing the tabbed view.

Cascade — Displays multiple components within a view in a cascading format, replacing the tabbed view.

Next — Displays the next element when multiple elements are present in a view.

Previous — Displays the previous element when multiple elements are present in a view.

Reset Windows — Resets the windows to their default locations. This is useful if you have undocked some of the view windows and want to move them all back to their default positions.

2.2 Using the Toolbar

The Carbon Model Studio Toolbar provides access to some of the frequently-used actions.

Figure 2-18 Carbon Model Studio Toolbar

Note: Some buttons may be unavailable depending on the type of license you have purchased from ARM.

The **Save**, **Delete**, and **New** buttons are context-sensitive and are available only when certain configuration settings are displayed in the Main view window.

There are tool tips for the buttons so that a brief button description appears as you float the mouse cursor over each button. The buttons include:

- **Save All** — Saves all open project files.
- **Check** — Checks the design for errors.
- **Compile** — Compiles the project.
- **Package** — Builds the package that was defined using Package Options (launched from *Project > Package Options*).
- **Stop** — Use this button to stop an in-process compile or simulation.
- **Compile Properties Set selection** — This drop-down menu enables you to select a specific set of Compile properties to be used for the next compile of the project. You can define multiple compile property sets so that you can create Cycle Models with different attributes. See “Creating New Compile Properties Sets” on page 35 for more information.
- **SoC Designer Plus** — Creates an SoC Designer Plus-compatible component from the Cycle Model. See “Starting and Configuring the Project” on page 92 for complete details.
- **CoWare** — Creates a Synopsys (formerly CoWare) Platform Architect-compatible component from the Cycle Model. See “Platform Architect-Specific Instructions” on page 143 for complete details.
- **SystemC** — Creates a component that can be linked directly into a SystemC design environment. See “Creating Components for SystemC” on page 148 for complete details.
- **MV** — Generates a component to be used with the Carbon Model Validation tool. See Chapter 4, Creating a Component for Model Validation for complete details.
- **Simulate** — Specifies what tools to invoke, how to invoke them, and what variables or arguments to use during simulation. See “Launching a Simulation” on page 153 for complete details.
- **Save** — Saves a new or changed module or net directives entry in the Main View.

- **Delete** — Deletes a module or net directives entry in the Main View.
- **New** — Creates a new module or net directives entry in the Main View.

2.3 Using the View Windows

The Carbon Model Studio provides many view windows that perform different tasks as you build and compile your projects. The possible views include:

- Project Explorer (Section 2.3.1)
- Main (Section 2.3.2)
- Properties (Section 2.3.3)
- Console (Section 2.3.4)
- Error List (Section 2.3.5)
- Design Hierarchy (Section 2.3.6)

2.3.1 Using the Project Explorer View

Use the *Project Explorer* view to manage your projects and to compile Cycle Models.

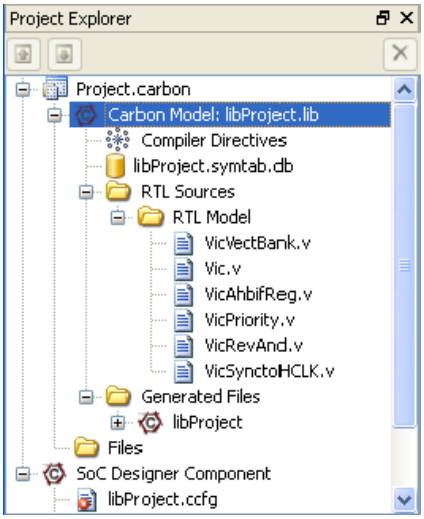


Figure 2-19 Carbon Model Studio Project Explorer Window

The Project Explorer view displays:

- Projects
- Project-related Verilog RTL source files and models
- Platform-specific (Platform Architect, SoC Designer Plus, SystemC, or Model Validation) components
- Carbon Model Studio-generated files

Use the Project Explorer view to perform a variety of tasks, such as:

- Compiling Cycle Models from RTL
- Configuring Compile properties
- Generating platform-specific models
- Configuring path and platform parameters

Note: Some items in the Project Explorer are affected by the order in which they are listed; for example, the list of RTL Source files. You can move the files up or down in the Explorer tree view using the Move Up and Move Down buttons.

To use the Project Explorer view:

- Click to select any node in *Project Explorer* view. In many cases, selecting a node displays its configuration information in the *Properties* view.
- Double-click an item to activate it. When an item is activated (for example, an RTL source file or a directives file), its contents are displayed in the *Main* view (see Section 2.3.2).
- Right-click an item to display a context-sensitive menu of available actions for that item, and select the action to execute it. This is shown in Figure 2-20.

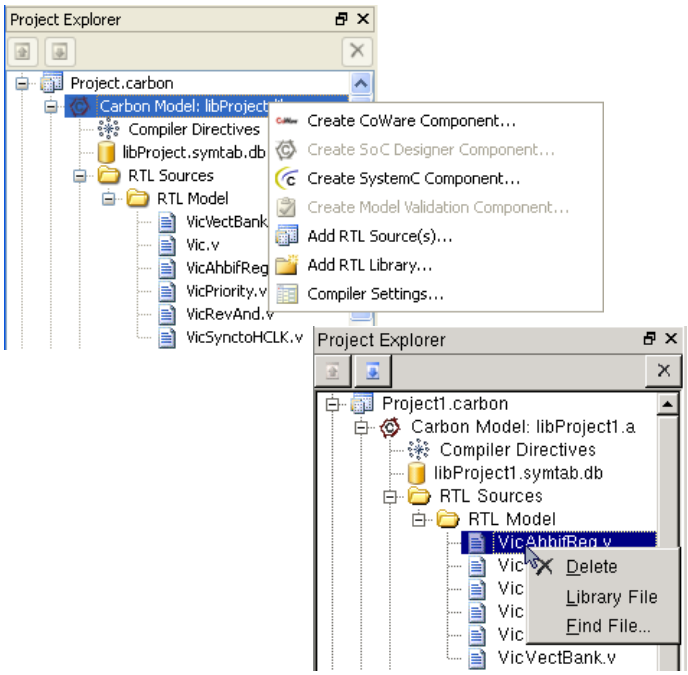


Figure 2-20 Project Explorer Context Menus

The same options are also available from the *Project* menu. See "Using the Project Menu" on page 32 for more details.

2.3.2 Using the Main View

Use the *Main* view to view and configure project settings. When you:

- Click once on a file in the Project Hierarchy — the Properties window displays properties for that file.
- Double-click a file in the Project Hierarchy — the Main view displays the file's contents. In the Main view, you can edit the file, or values in the configuration window.

As shown in Figure 2-21, the Modules and Nets directives display when you double-click the "Compiler Directives" option. The Directives file appears when you double-click the directives file (if one exists for your project). The Verilog file content displays when you double-click the file in the RTL Folder list.

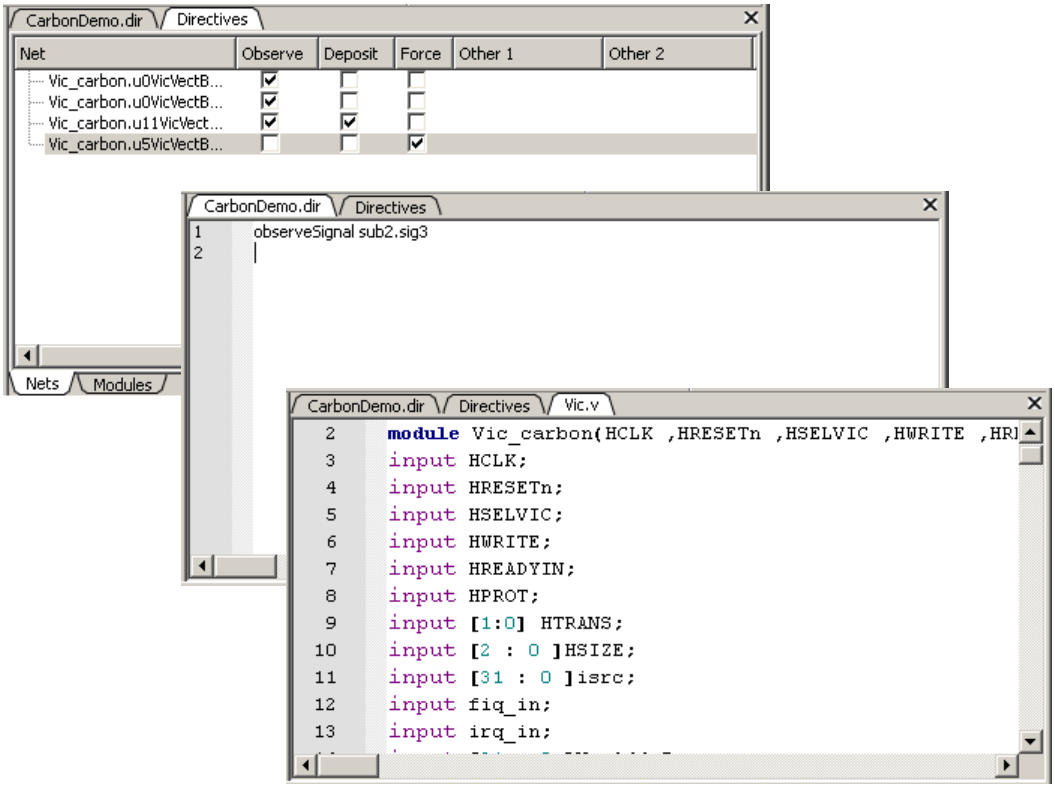


Figure 2-21 Sample Contents of Main Window

There are many other files that appear in the Main view as you define the settings for your project.

Use the close button ( ) to close the active file in the Main view when you no longer want it displayed.

To adjust the font size in these text windows, set the Text Editor *Preferences* from the *Edit* menu. To make temporary changes to the font size, use the key sequence *Ctrl+<Number Pad +>* to increase the font and *Ctrl+<Number Pad ->* to decrease the font.

2.3.3 Using the Properties View

Use the *Properties* view to display all properties associated with a selected item, and edit those properties as needed. Editable and read-only fields vary, depending on the item selected in the *Project Explorer* view. Property views (sets) are provided for:

- Projects (Section 2.3.3.1)
- Cycle Model Compiler (Section 2.3.3.2)
- RTL Folders (Section 2.3.3.3)
- RTL Files (Section 2.3.3.4)
- Compiler Directives Files (Section 2.3.3.5)

The Properties view (Figure 2-22) includes three elements:

- Property Type Field
- Property Parameters Window
- Parameter Help Window.

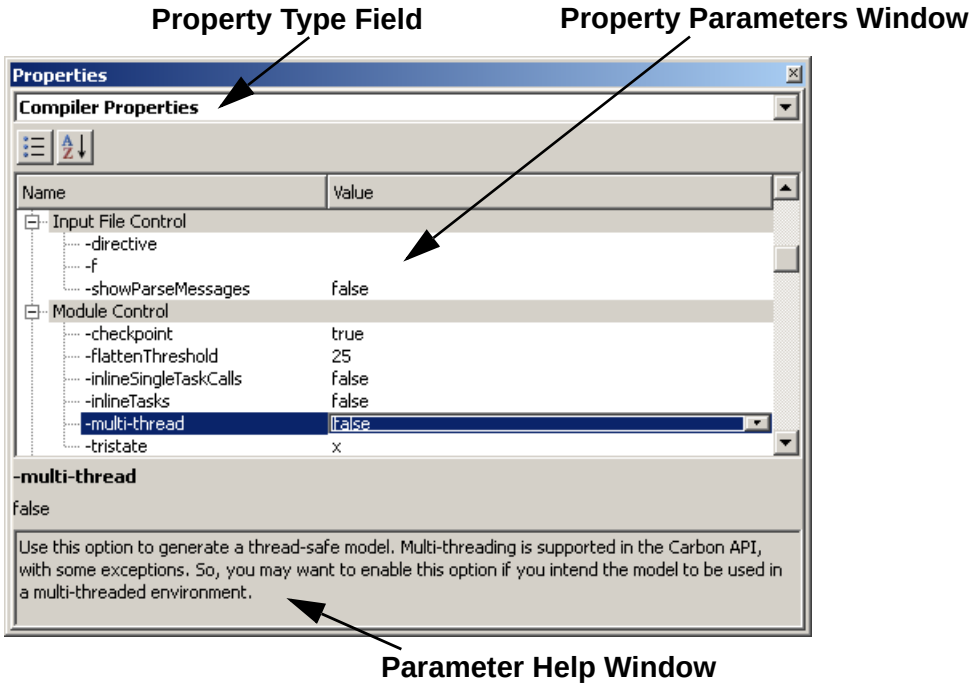


Figure 2-22 Carbon Model Studio Sample Properties Window

If the Property Type field:

- Displays one item — The parameters displayed in the Properties window apply to that item.
- Is blank — The properties displayed may apply to multiple items.

The Parameter Help Window at the bottom displays additional information about selected items.

2.3.3.1 Project Properties

The Carbon Model Studio *Project* (*.carbon* file) contains all the data required to configure the Carbon Model Studio environment and manage Cycle Models, including all the files that are generated during the compile process. The Project properties are listed in the properties view (Figure 2-23). For descriptions of individual items, select the item and view the information in the Parameter Help window.

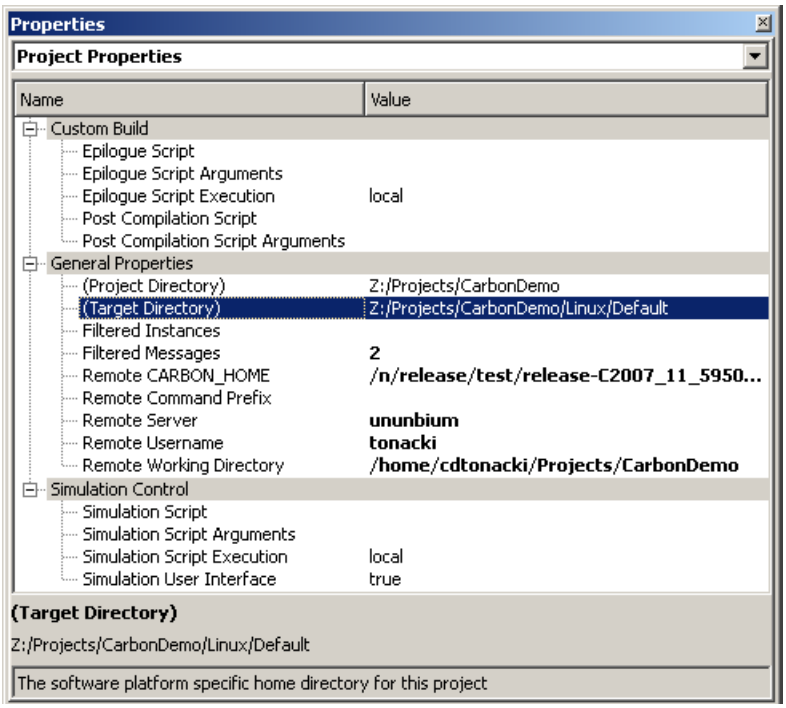


Figure 2-23 Project Properties Window

Some of the Project properties are required to be filled, some are optional, and some are read-only fields.

- Read-only fields are in parentheses; for example, (Project Directory).
- Required fields include Remote CARBON_HOME, Remote Server, Remote Username, and Remote Working Directory. For Windows users, these fields are filled automatically from the values entered in the Drive Mapping dialog box, as described in “Drive Mappings” on page 27.

Note that the following environment variables are available whenever Carbon Model Studio launches a program (for example, an Epilogue or Simulation script):

CARBON_PROJECT=<project directory>
CARBON_CONFIGURATION=<platform>/<name>
CARBON_OUTPUT=$CARBON_PROJECT/$CARBON_CONFIGURATION
CARBON_MODEL=<design.name>

For example, the following values would be available for scripts:

CARBON_PROJECT=/home/username/Carbon/Projects/Project1
CARBON_CONFIGURATION=Linux/Default

```
CARBON_OUTPUT=/home/username/Carbon/Projects/Project1/Linux/Default
CARBON_MODEL=libdesign.a
```

2.3.3.2 Compiler Properties

You use the *Compiler properties* to configure global parameters to be applied when compiling your Cycle Models. These parameters display when *Carbon Model* or *RTL Sources* is highlighted in the *Project Explorer* view.

The Compiler properties are saved in a Compiler Settings Property Set. You select the Compiler Property Set from the drop-down menu in the toolbar. After you select a specific Compile Property Set, all changes you make in the Properties pages are applied to that Configuration and used the next time you compile any project using that Compile Property Set.

You can define multiple compile property sets so that you can create Cycle Models with different attributes. See “Configuration Manager” on page 35 for more information.

The Compiler Properties are organized into the following categories:

- Basic Options
- General Compile Control
- Input File Control
- Module Control
- Net Control
- Output Control
- Verilog Options

Enter information in the parameter fields in any of the following ways:

- Clicking a drop-down menu and selecting the appropriate value.
- Clicking a **Browse** button and navigating to a file.
- Clicking the field and entering text.

Options you change from their default settings are indicated in bold text.

For descriptions of individual parameters in any of the categories, select the item and view the information in the Parameter Help window.

Basic Options Parameters

The *Basic Options* parameters are shown in Figure 2-24.

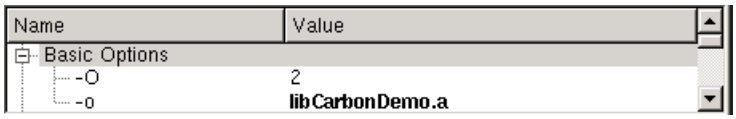


Figure 2-24 Carbon Compiler Basic Options Parameters

The *-o* option defines the default Cycle Model name. This name defaults to *lib<project_name>.lib* when running Carbon Model Studio on Windows, and to *lib<project_name>.a* when running on Linux.

General Compile Control Parameters

The *General Compile Control* parameters are shown in Figure 2-25.

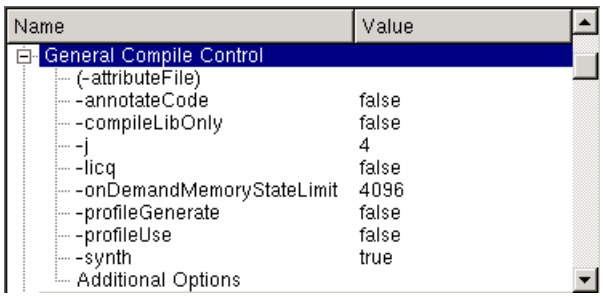


Figure 2-25 Carbon Compiler General Control Parameters

The *Additional Options* parameter enables you to add special compile parameters that are typically not required. See the *Carbon Compiler User Manual* for more information, and consult with ARM Technical Support for guidance.

Input File Control Parameters

The *Input File Control* parameters are shown in Figure 2-26.

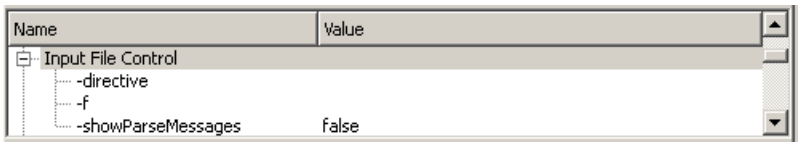


Figure 2-26 Carbon Compiler Input File Control Parameters

The *-directive* option enables you to specify a directives file to be used during compilation of the Cycle Model. Directives are compiler commands that control how the Carbon compiler interprets and builds a linkable object. See "Directives File Properties" on page 50 for more information.

Module Control Parameters

The *Module Control* parameters are shown in Figure 2-27. They control compilation settings that affect modules.

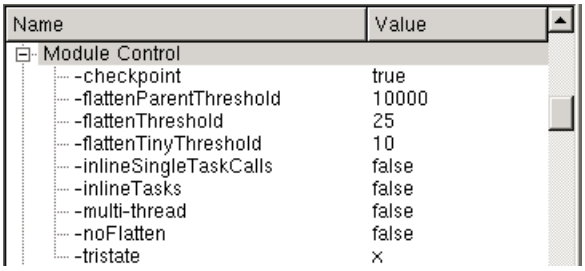


Figure 2-27 Carbon Compiler Module Control Parameters

Net Control Parameters

The *Net Control* parameters are shown in Figure 2-28. They control compilation settings that affect nets.

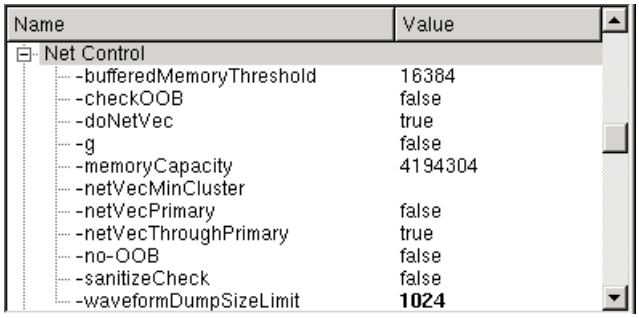


Figure 2-28 Carbon Compiler Net Control Parameters

Output Control Parameters

The *Output Control* parameters are shown in Figure 2-29. They control the data that is output from the compile process.

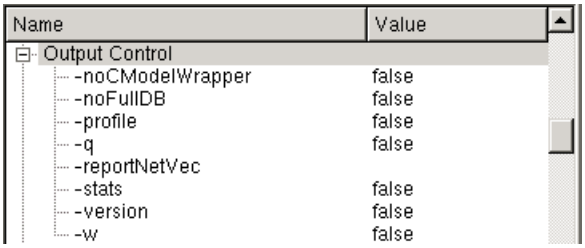


Figure 2-29 Carbon Compiler Output Control Parameters

Verilog Options Parameters

You can use the *Verilog Options* parameters (Figure 2-30) only with Verilog design files. They control how Verilog parameters are handled during the compilation process.

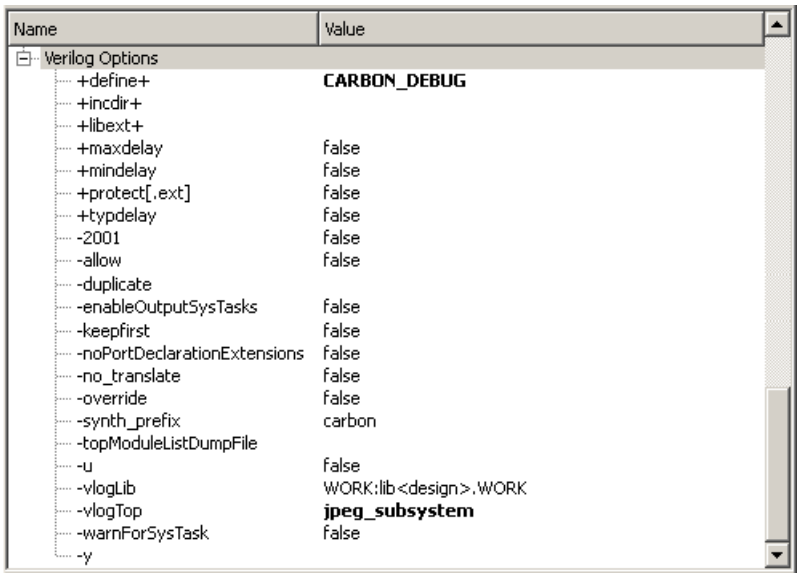


Figure 2-30 Carbon Compiler Verilog Options Parameters

2.3.3.3 RTL Folder Properties

You display the RTL Folder properties page by selecting the RTL folder in the Project Explorer view (named *RTL Model* in our sample). The RTL Folder properties, shown in Figure 2-31, allow you to define the:

- name of the RTL folder
- name of the Verilog library in which the files in this folder should be placed
- Verilog compile options to apply to the files in this folder when the project is compiled

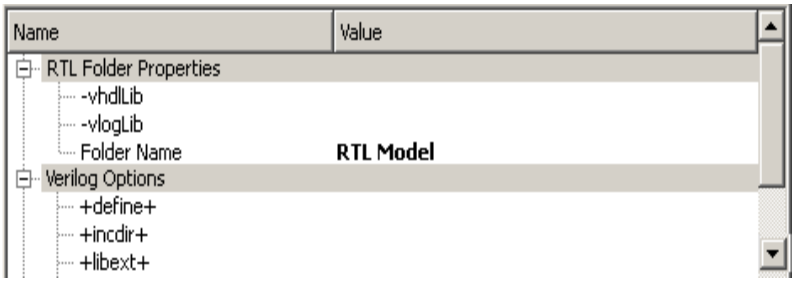


Figure 2-31 RTL Folder Properties

Enter information in the parameter fields in any of the following ways:

- clicking a drop-down menu and selecting the appropriate value
- clicking a **Browse** button and navigating to a file
- clicking the field and entering text

For descriptions of these parameters, select the item and view the information in the Parameter Help window.

Note: The compiler property settings you make here override the Cycle Model project compiler settings. These settings apply to the files in this folder only.

2.3.3.4 RTL File Properties

You display the RTL File properties page by selecting a file within the *RTL Model* in the Project Explorer view. The RTL File properties, shown in Figure 2-32, allow you to define compiler properties unique to a specific file that is part of your project.

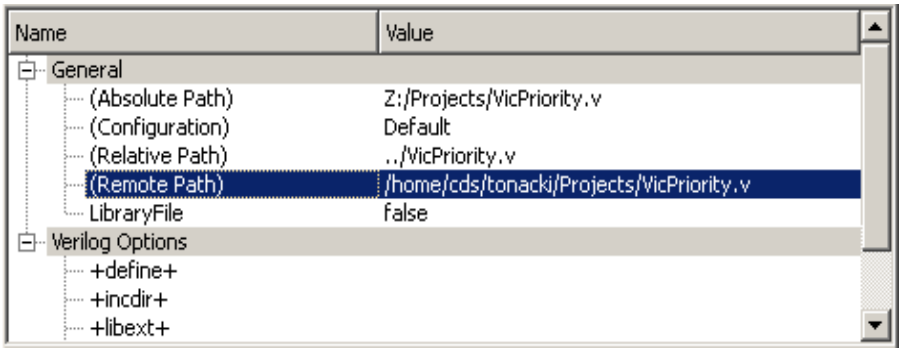


Figure 2-32 RTL File Properties

Note: The property settings you make here override the compiler settings for the project and the RTL Folder. The settings made here apply to the current file only.

For descriptions of these parameters, select the item and view the information in the Parameter Help window.

2.3.3.5 Directives File Properties

Directives control how the Carbon compiler interprets and builds a Cycle Model. There are three ways to apply directives to your project:

- Using a directives file — Discussed below.
- Manually applying directives to modules and nets — See "Applying Directives to Modules and Nets" on page 69.
- Embedding directives in Verilog source as comments — See "Embedding Directives in Verilog Source Files" on page 74.

To use a directives file, you must specify the existing file name and location using the *-directive* Compiler Property. After you define this property, the name appears in the Project

Explorer. When you click the directive file name (<*name*>.*dir*) in the Project Explorer tree, the path and file display in the Properties view, as shown in Figure 2-33.

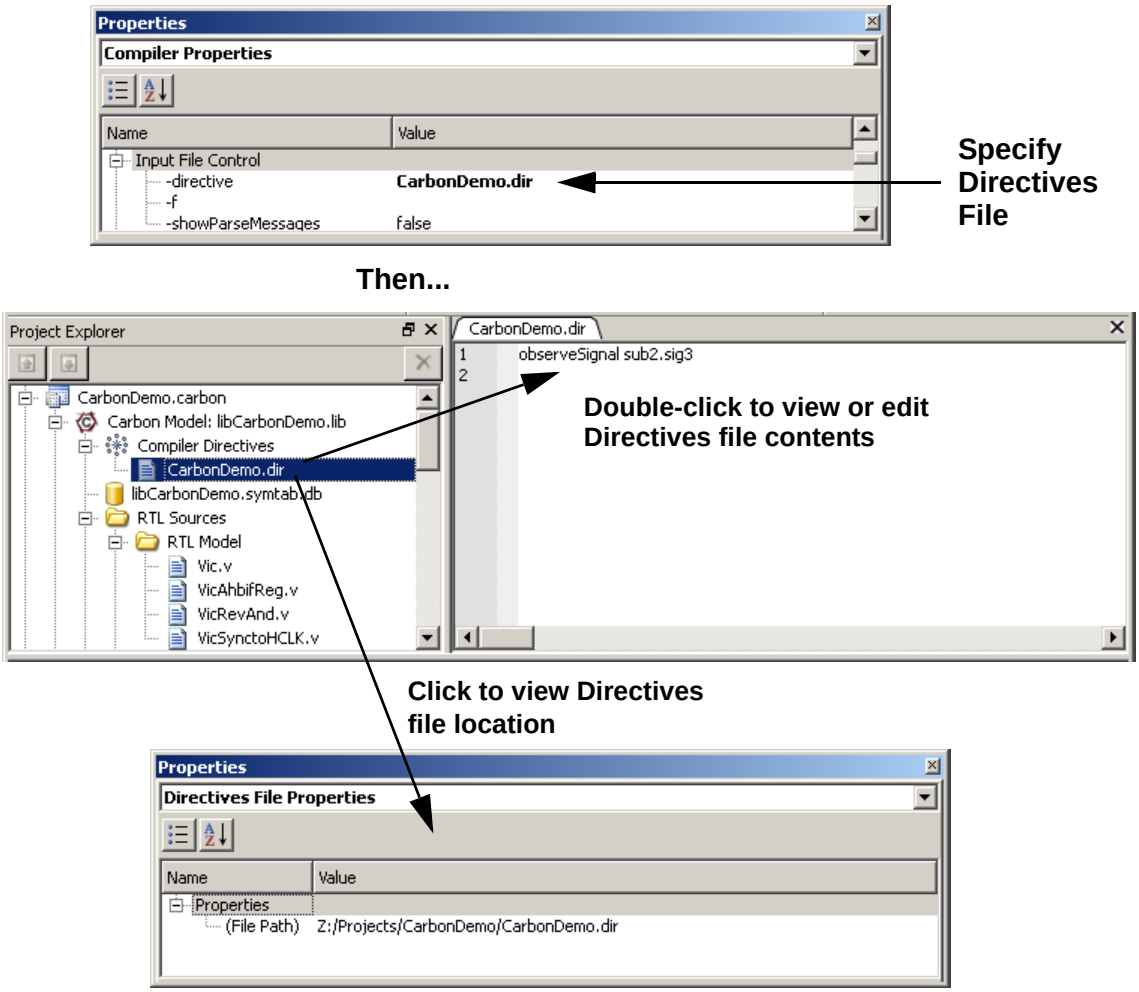


Figure 2-33 Directives File Parameters

2.3.4 Using the Console View

The *Console* view, shown in Figure 2-34, displays all process events that occur during and after compiling RTL into a Cycle Model. The Console view also displays status information when you Simulate a project.

Note: The console view is read only.

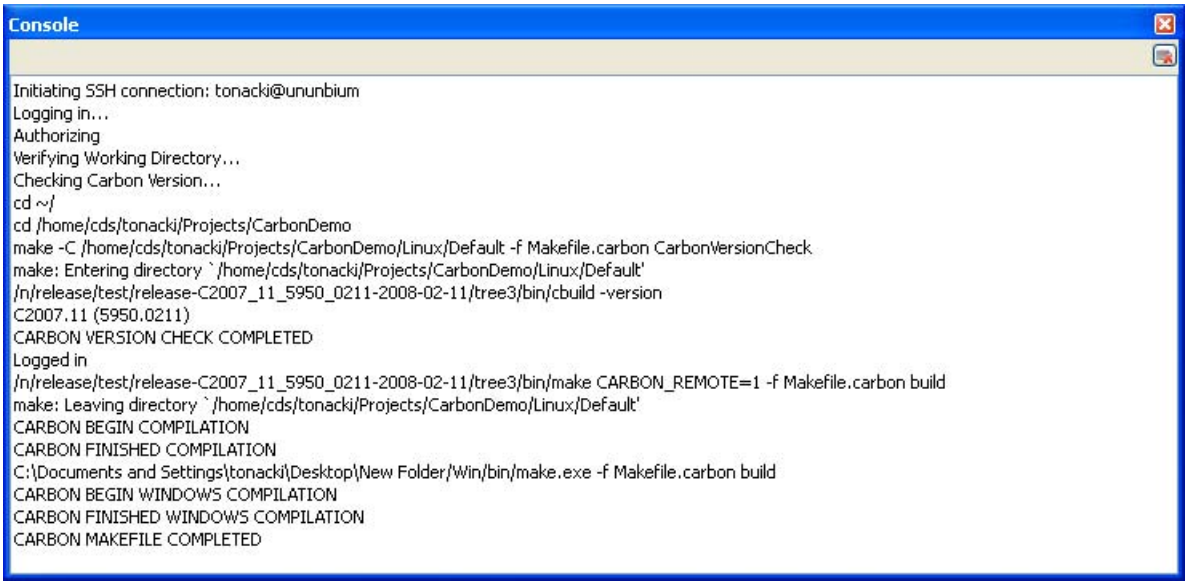


Figure 2-34 Carbon Model Studio Console Window

Colors are used in the Console view to call out important compilation information (see Figure 2-35). The colors that are used, and their meaning, include:

Red — An error discovered in the Carbon Model Studio configuration information

Dark Red — An error discovered in the Carbon compiler configuration information

Blue — A warning message about the Carbon Model Studio configuration

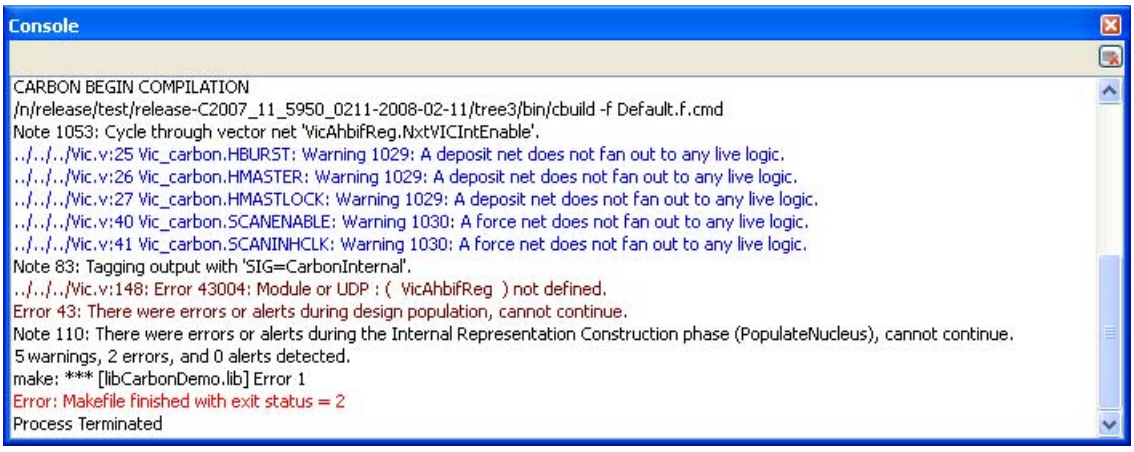


Figure 2-35 Console Window Messages

To display the location of the error in the source file in the Main view window, double-click highlighted messages.

2.3.5 Using the Error List View

The *Error List* view displays Errors, Warnings, and Messages generated during the compile process. The buttons along the top of this window display the total number of messages of each type. The buttons also enable you to filter (remove) certain messages from the list. The buttons are:

- **Errors** — Critical errors that stop the generation of the Cycle Model
- **Warnings** — Warning messages that could affect the generation of the Cycle Model
- **Messages** — Informational messages
- **Filtered** — Used to filter certain messages

Help for each error message is displayed at the bottom of the Error List window, as shown in Figure 2-36.

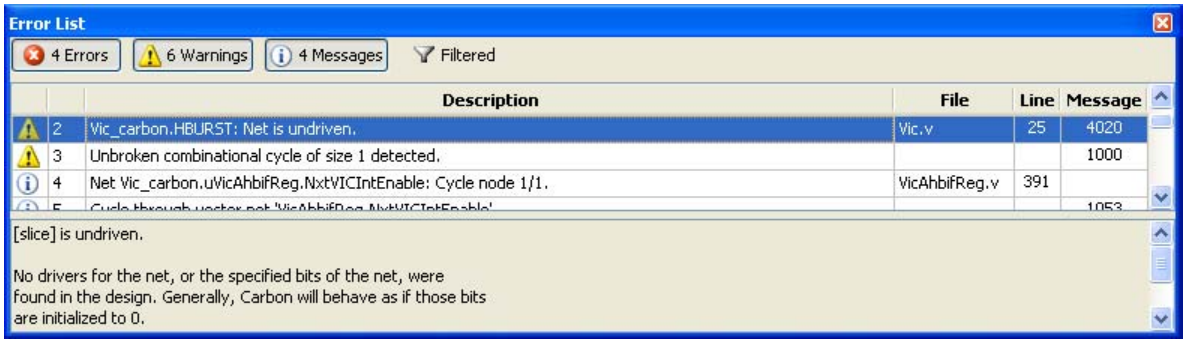


Figure 2-36 Error List Window

Error message descriptions can help you determine the cause of a problem and provide potential ways to fix it.

To use the Error List view:

1. Click any of the Select / Deselect buttons to hide or redisplay Errors, Warnings, or Informational Messages from the view (Figure 2-37).

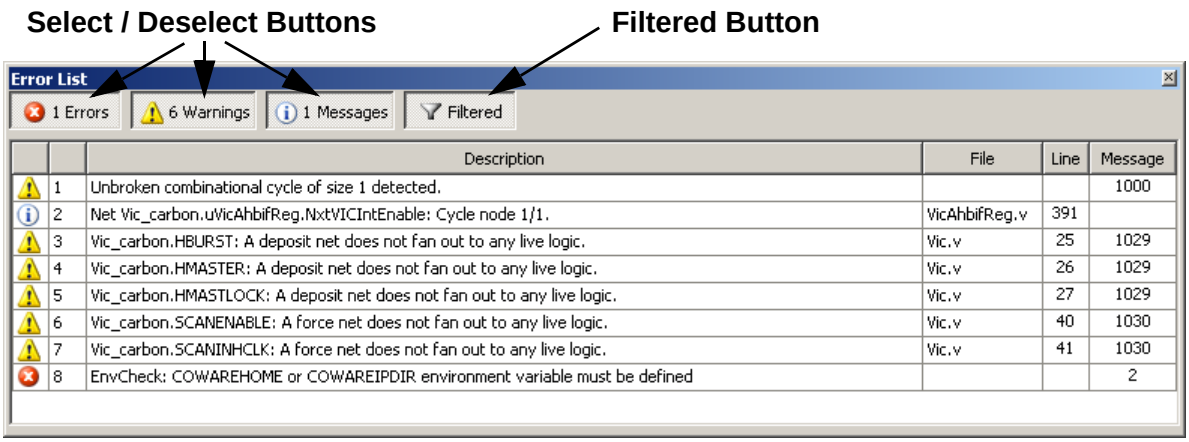


Figure 2-37 Error List Window Buttons

2. To navigate to the location of a source file error in the Main view window, double-click any error, warning, or message (Figure 2-38).

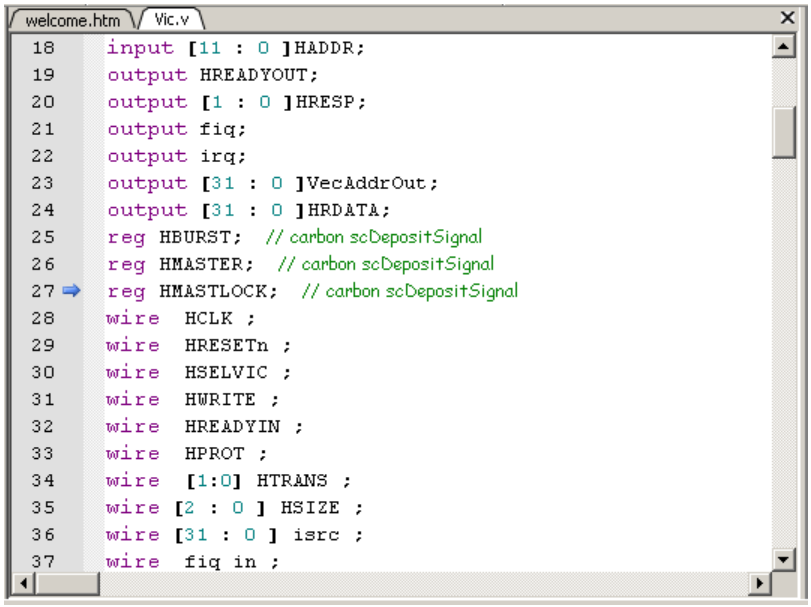


Figure 2-38 Error Location in RTL File

3. Right-click any item to display the Error List Options menu (Figure 2-39).

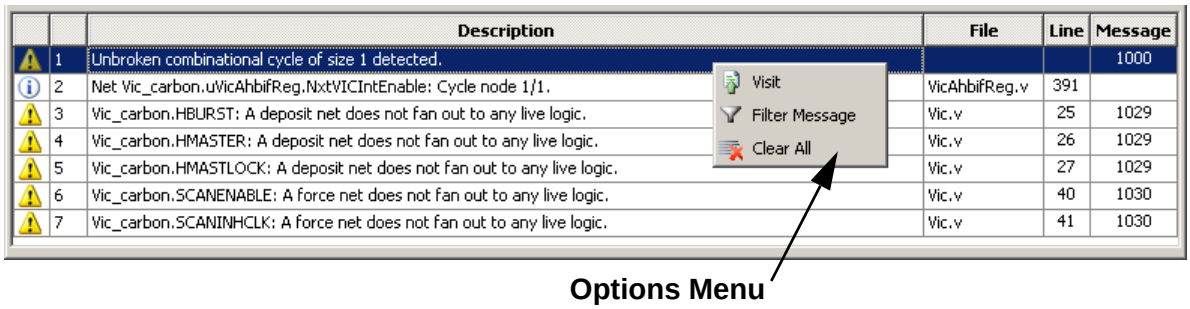


Figure 2-39 Error List Context Menu

4. Right-click any Error List view item for which a file exists. This displays the Error List Options menu.
5. Select the *Open in Design Hierarchy* option to see the hierarchical view of the item (Figure 2-40).

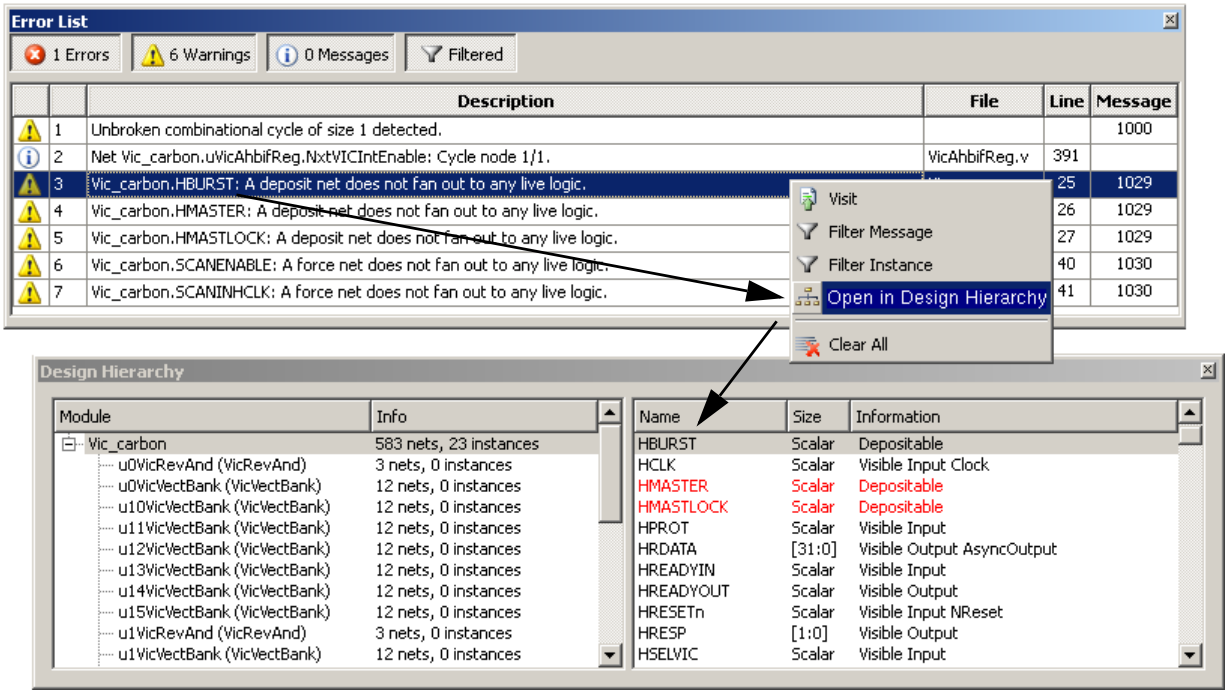


Figure 2-40 Error List Link to Design Hierarchy

To filter the errors, warnings, and messages that are displayed in the Error List view:

1. Right-click the item(s) in the error list that you wish to filter out.
2. From the *Error List Options* menu, select *Filter Message* to remove all instances of the message from the Error List view.
3. To re-display (cancel the filtering of) any messages you have previously filtered out, click the *Filtered* Button (Figure 2-41).

Figure 2-41 Error List Window Filtered Button

2.3.6 Using the Design Hierarchy View

Use the Design Hierarchy view, shown in Figure 2-42, to show design structure and manage the functions of design modules and signals. The Design Hierarchy provides two panes: the left displays the modules in the design tree, and the right pane displays the nets associated with the design modules.

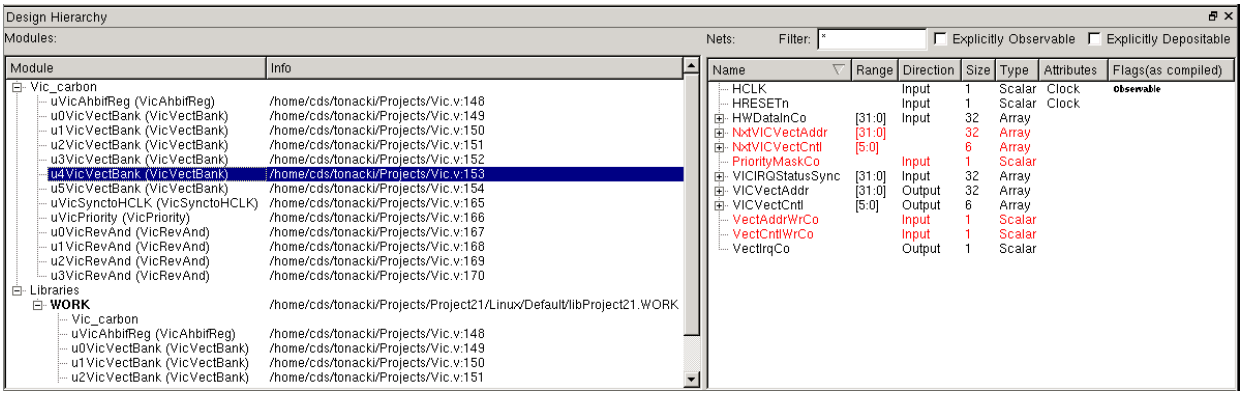


Figure 2-42 Carbon Model Studio Design Hierarchy Window

To use the Design Hierarchy:

- In the Modules pane:
  - Click any module in the left pane to display the nets associated with that module in the right pane. If the module contains Verilog parameters, the Nets pane shows the parameters for each instance.
  - Right-click a module to set the *observeSignal* directive or the *depositSignal* directive on all the nets in the module, or just for that instance of the module.
  - Right-click any module in the left pane to display the RTL code for the module, or the RTL code for just that instance of the module, in the Main view.
- In the Nets pane:
  - Right-click a net to set the *observeSignal*, *depositSignal*, or *forceSignal* directive.
  - Right-click a net to display the location in the RTL code where the net is located.

  - When nets have sub-components, click the plus sign to expand the net to view the elements.
  - When editing and tuning components, drag nets from the right pane to ports, registers, memories, and profiles (*see* Chapter 5, Creating Components for Specific Platforms).
- Use the *Filter* field and the check-boxes (*Explicitly Observable*, *Explicitly Depositable*) to restrict the view of the nets in the Nets window. This is useful when looking for specific nets within the design.

Note: Any nets displayed in red are not visible (not observable). If you require access to any of these nets, you must explicitly set the observeSignal directive.

Chapter 3

Compiling RTL into a Cycle Model

The Carbon compiler takes an RTL hardware model and creates a high-performance linkable object, the Cycle Model, that is cycle and register accurate. The Cycle Model provides an API for interfacing with your validation environment. The Carbon compiler fits into the design simulation flow as shown in Figure 3-1.

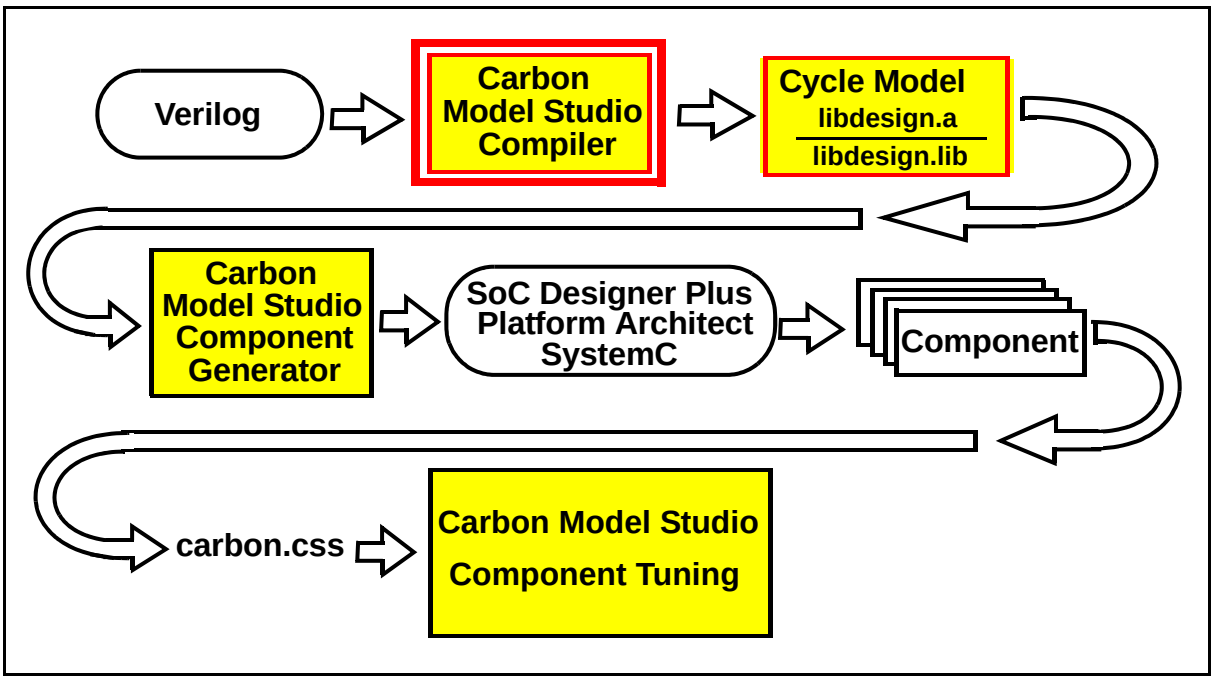


Figure 3-1 Carbon Model Studio Process Flow - Carbon Compiler

This chapter provides information regarding the Carbon compiler functions that are supported by the *Carbon Model Studio*. For comprehensive information regarding the Carbon compiler, refer to the *Carbon Compiler User Manual*.

This chapter includes the following sections:

3.1 Reviewing Remote Compile Requirements

Carbon Model Studio requires that your RTL hardware description be compiled on Linux. However, it is possible to run the Carbon Model Studio on Windows and control a compile process that is running on Linux. This section describes how to set up Carbon Model Studio on Windows to manage remote compilation on a Linux server.

If you are running on Windows, when you initially click **Compile**, the Remote Server Configuration dialog displays (Figure 3-2).

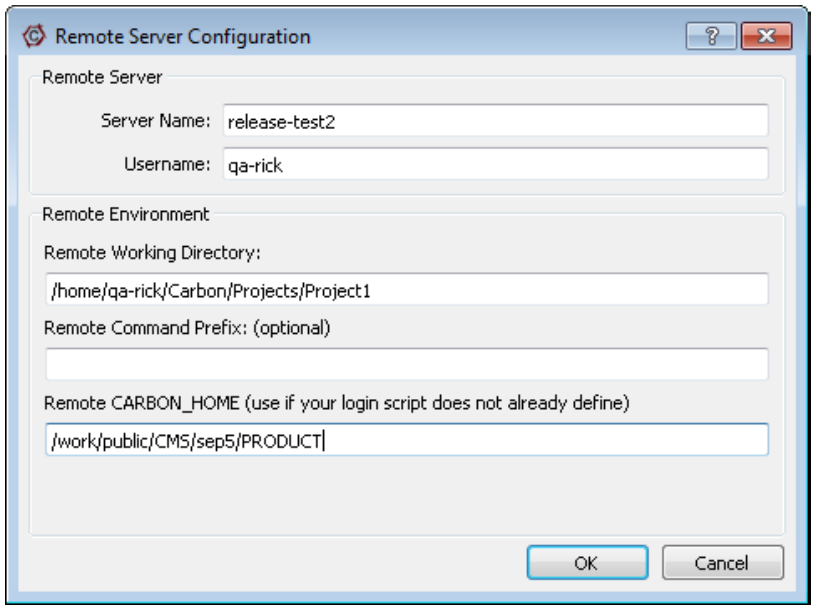


Figure 3-2 Remote Server Configuration dialog

Complete the fields as follows:

- **Server Name** —Name of the Linux machine used for compilation.
- **Username** — Username for access to the Linux machine.
- **Remote Working Directory** — The directory where compilation occurs.
- **Remote Command Prefix** (optional) — Script to run after logging in - for example, one that defines CARBON_HOME and also places `CARBON_HOME/bin` into the Linux path.
- **Remote CARBON_HOME** — Location where *CARBON_HOME* resides on the remote Linux machine: the directory where Carbon Model Studio is installed.

Note: The project must not reference any files that reside outside the scope of the defined Remote Working Directory. For example, source files located in a Linux path /library are not opened by Carbon Model Studio; however, they still compile on the remote machine. This also means that errors might not be easily attributed to those files.

Click **OK** when this dialog is complete. You are prompted to enter the password for the Linux account.

These settings are reflected in the CMS Project Properties window (Figure 3-3).

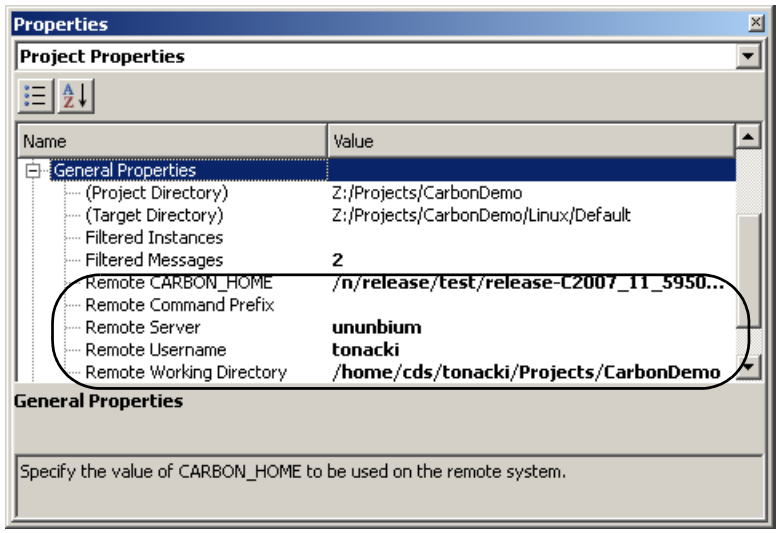


Figure 3-3 Carbon Model Studio Project Properties

Carbon Model Studio performs a check to ensure that these directories exist when executing a remote compilation. Refer to the *Carbon Model Studio Installation Guide* for more installation information.

Important note regarding SSH: Carbon Model Studio comes with re-distribution of PuTTY's *plink* program (*http://www.chiark.greenend.org.uk/~sgtatham/putty/*) and uses *plink* to open up the remote Linux shell.

3.2 Carbon Compiler Inputs

A Cycle Model can be generated only by the Carbon compiler. The Carbon compiler reads the following files, in order, and generates a Cycle Model for the design.

1. Options files – Contain command options that provide control and guidance to the Carbon compiler (sometimes these are called “switches”).
2. Directives files – Contain directives that control how the Carbon compiler interprets and builds a Cycle Model.
3. Verilog design and library files – *Golden* RTL of the hardware design.

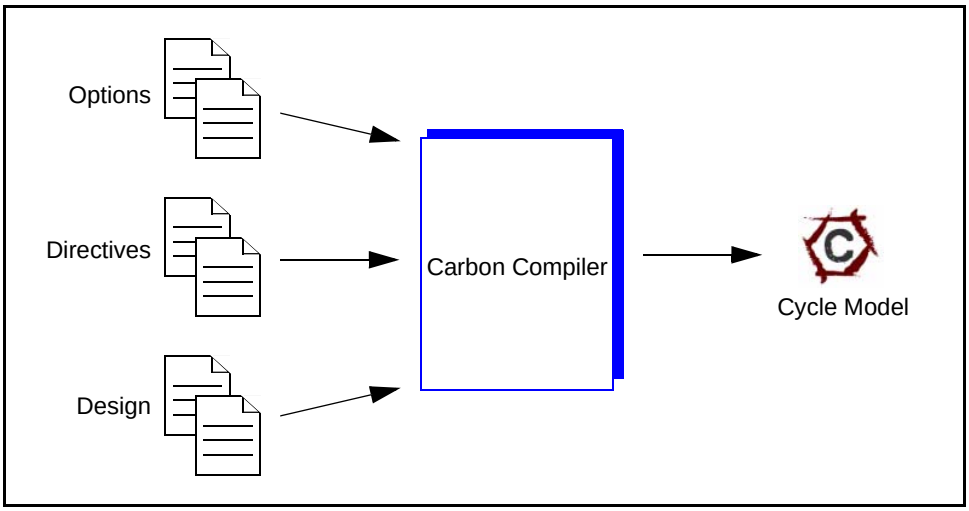


Figure 3-4 Cycle Model Compilation Overview

3.2.1 Carbon Compiler Options

You use the Cycle Model *Compiler properties* to configure parameters to be applied when compiling your Cycle Models. These parameters display when *Carbon Model* or *RTL Sources* is highlighted in the *Project Explorer* view. They are organized into a variety of categories.

3.2.2 Carbon Compiler Directives

Directives are compiler commands that can be contained in a directives file or embedded in Verilog source code. Directives control how the Carbon compiler interprets and builds a linkable Cycle Model. As previously mentioned, Carbon Model Studio can manage two types of compile directives: *Net* directives and *Module* directives. *Net* directives provide access to specified signals in the Cycle Model from the external environment. *Module* directives identify a module in the design hierarchy that you intend to be hidden in the Cycle Model. For more information, refer to the *Carbon Compiler User Manual*.

3.3 Compiling RTL with Carbon Model Studio

The Carbon Model Studio provides access to all the features needed to configure the compile process to your needs. You can:

- Define compiler properties
- Define source files
- Add directives
- Use compiler options
- Compile for Verilog

To compile RTL using the Carbon Model Studio you must follow the tasks listed below:

- Create your project
- Add RTL source files
- Define compiler properties
- Compile the project
- Define directives for modules and nets
- Recompile the project

3.3.1 Creating your Project

Follow the steps below to create a project.

1. In the *File* menu, select *New*, and select *Project*.

Figure 3-5 New Project from File Menu

2. The *New Project* dialog box appears.

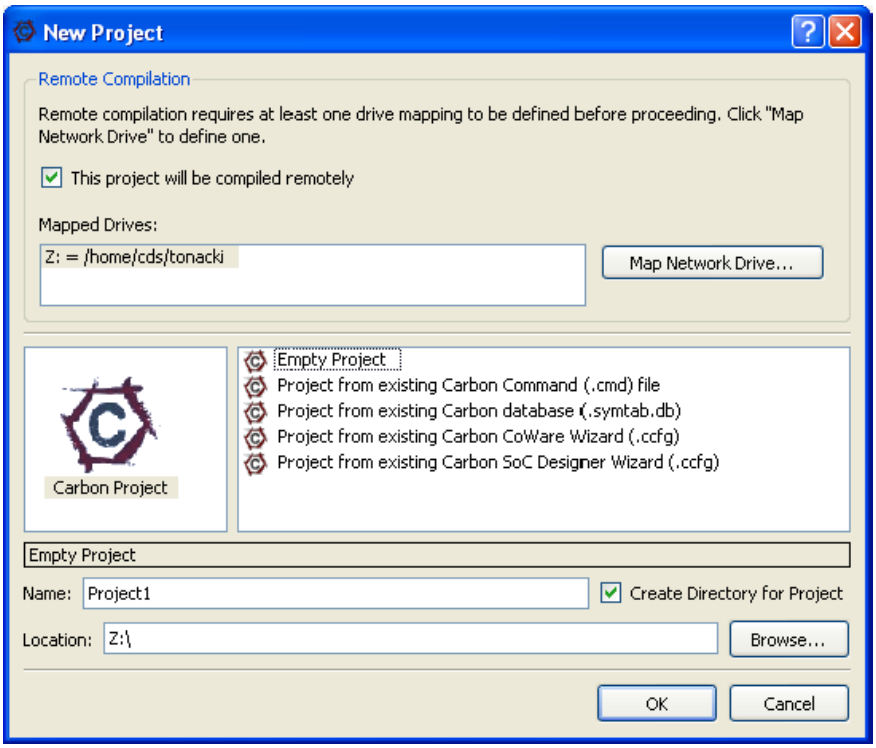


Figure 3-6 New Project Dialog Box

The top part of the New Project dialog box appears only for Windows users who are performing remote compilation on a Linux machine. At least one drive mapping must be set up before you can define a new project. For Windows users, follow the next step. For Linux users, skip to step 8 on page 65.

3. For Windows users, select the Mapped Drive to be used to perform remote compilation of the Cycle Model. Click **Map Network Drive...** to map a network drive if no drives have been set up, and the following dialog box appears.

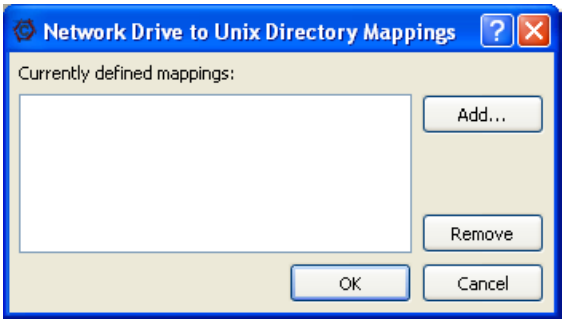


Figure 3-7 Windows Network Drive Mappings Dialog Box

This dialog enables you to manage the Windows Network drive to Linux directory mappings. You can add new drive mappings or remove existing drive mappings.

4. Click **Add** to add a new drive mapping and the following dialog box appears.

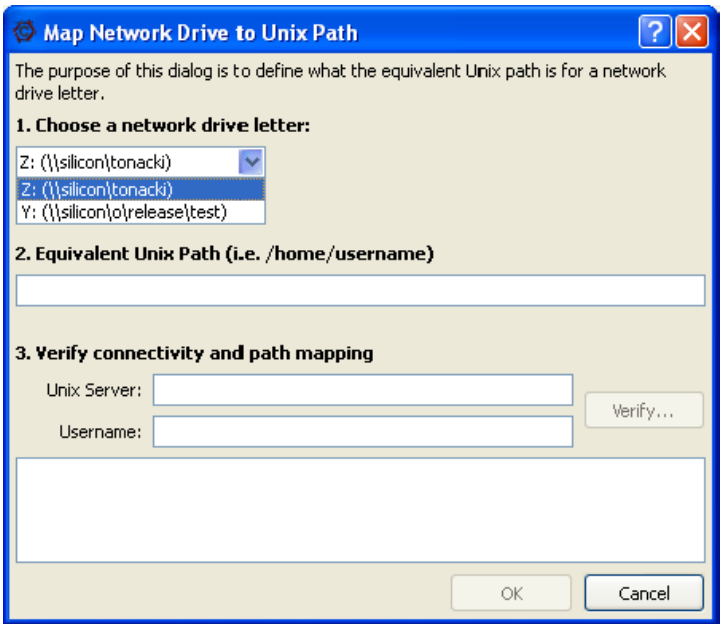


Figure 3-8 Windows Map Network Drive Dialog Box

5. Perform the following steps in this dialog box:
   - Select the network drive that has already been set up in Windows.
   - Enter the Linux path to be equated to the network drive letter.
   - Enter the name of the Linux Server and your valid Username.
   - Click the **Verify** button to verify that the mapping is set up correctly. You must enter the password for the remote Linux server.
6. Once the verification is successful, click **OK** to return to the Network Drive dialog.

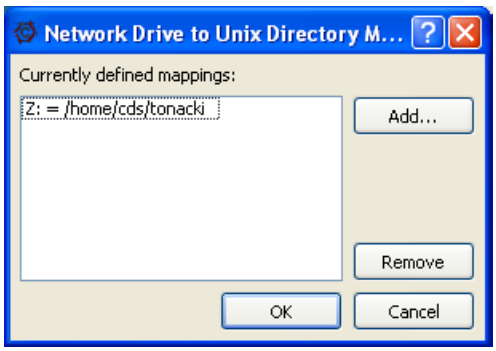


Figure 3-9 Completed Windows Network Drive Mappings Dialog Box

7. Click **OK** to return to the *New Project* dialog.
8. Select the type of new project that you want to create. See the list of project types on page 24 for a description of the different methods you can use to create a new project.
9. In the *Name* field, enter the name for your new project.

10. In the *Location* field, browse to the location where you plan to store your new project.
11. Click **OK**.

Note: Select the checkbox "Create Directory for Project" to automatically create a new directory using the name of the project. The project files are placed in that directory.

3.3.2 Adding RTL Source Files

Follow the steps below to add RTL source files to the project.

1. From the *Project* menu, select *Add RTL Source(s)...* You can also right-click the project in the Project Explorer view and select *Add RTL Source(s)...* The Select RTL Source(s) dialog box appears.

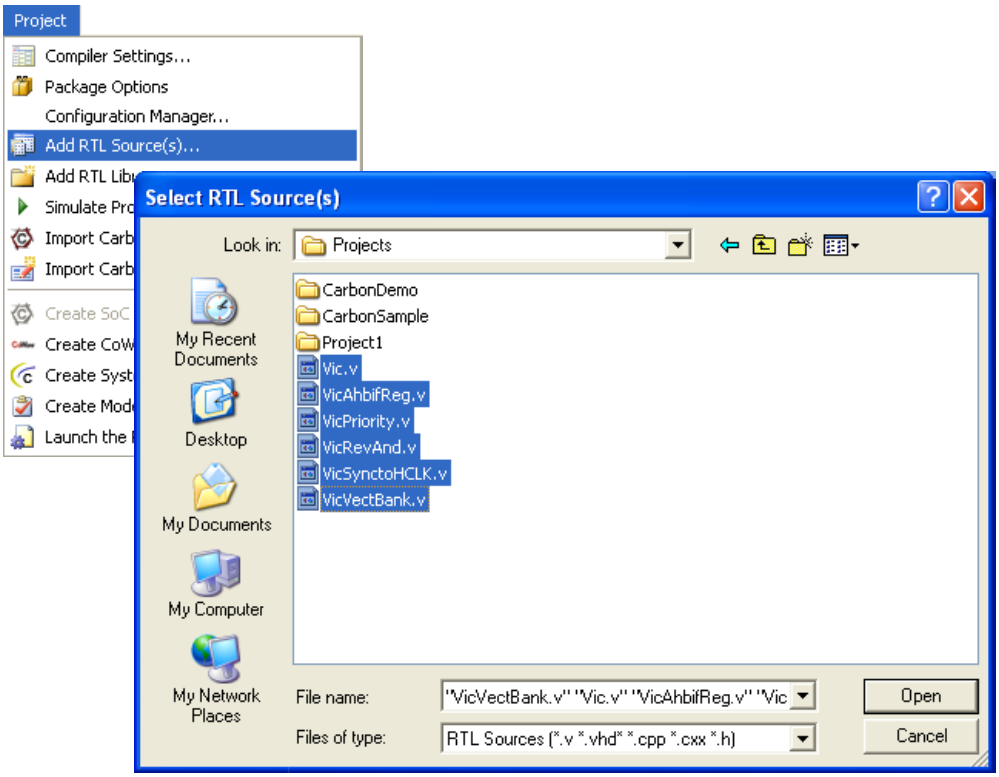


Figure 3-10 Add RTL Source Files

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 result is shown in Figure 3-11.

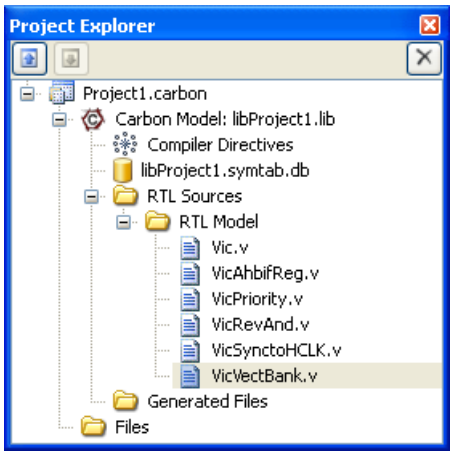


Figure 3-11 RTL Source Files in Project Explorer

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 Explorer tree view using the Move Up and Move Down buttons.

3.3.3 Defining Compiler Options and Compiling the Cycle Model

By default, the initial compiler settings allow you to compile your project immediately into a Cycle Model. The -o option (under Basic Options) defines the default Cycle Model name. This name defaults to *lib<project_name>.lib* when running Carbon Model Studio on Windows, and to *lib<project_name>.a* when running on Linux. Many other options exist; fine-tune these to create the best Cycle Model for your application.

Follow the steps below to define the Carbon compiler properties.

1. From the drop-down list in the Button bar, select the Compiler Configuration you want to use to compile the Cycle Model. The Compiler Configuration contains the unique compiler options to be used when you click the **Compile** button.

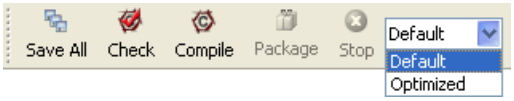


Figure 3-12 Compiler Configuration Selection

You can have many different sets of compiler properties that include different settings depending on the stage of development you are in. For example, you could have the Default standard set, a compiler set that includes the use of a directives file (-directive), and so forth. You create new sets of compiler properties using the *Configuration Manager* option under the *Project* menu.

2. In the Compiler Properties view, set the desired option values. A brief description of each option appears at the bottom of the view in the Parameter Help Window. See Chapter 3 of the *Carbon Compiler User Manual* for more detailed information.

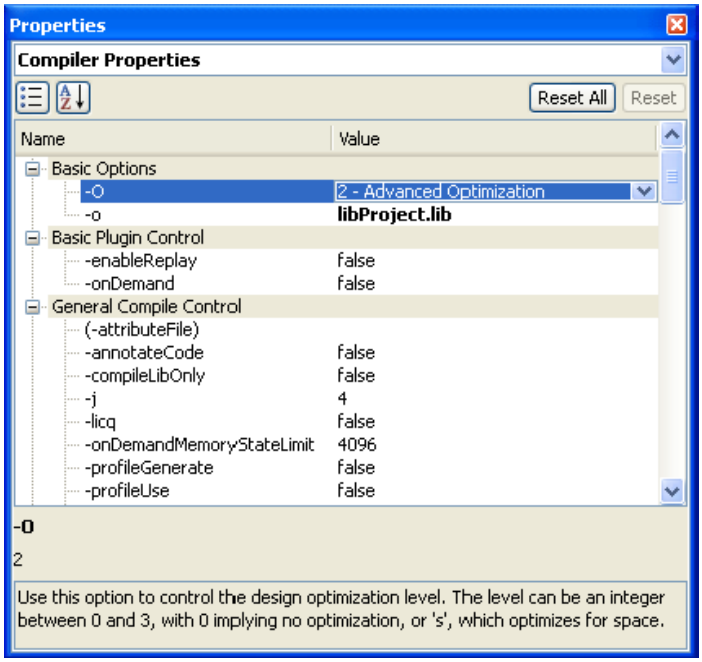


Figure 3-13 Carbon Compiler Properties Window

3. Click the **Compile** button.

This creates the initial Cycle Model.

3.3.4 Defining Directives and Recompiling the Cycle Model

Directives are compiler commands that can be contained in a directives file, or embedded in Verilog source code. Directives control how the Carbon compiler interprets and builds a Cycle Model. There are three ways to apply directives to your project:

- using a directives file
- manually applying directives to modules and nets
- embedding directives in Verilog source as comments

As mentioned in the previous section, you can use a directives file that affects the compilation result of your Cycle Model by using the `-directive` option. This is also described in "Directives File Properties" on page 50.

After you define the directives for the modules and nets in your design, you must recompile the Cycle Model.

3.3.4.1 Applying Directives to Modules and Nets

You can assign directives to modules and nets within your Cycle Model. This allows you to control the handling of modules and nets. Follow the steps below to define the module and net directives for the project.

Adding Module Directives

1. In the Project Explorer view, select *Compiler Directives*. The Directives view displays in the Main view. Note the *Nets* and *Modules* tabs at the bottom of the view.

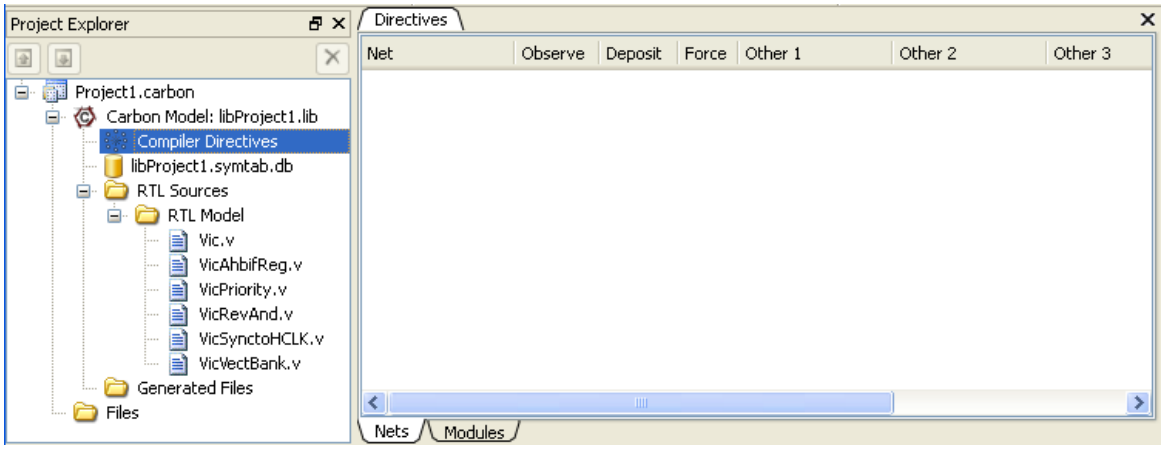


Figure 3-14 Carbon Model Studio Nets Directives Window

2. In the Design Hierarchy, click any module to display the nets.

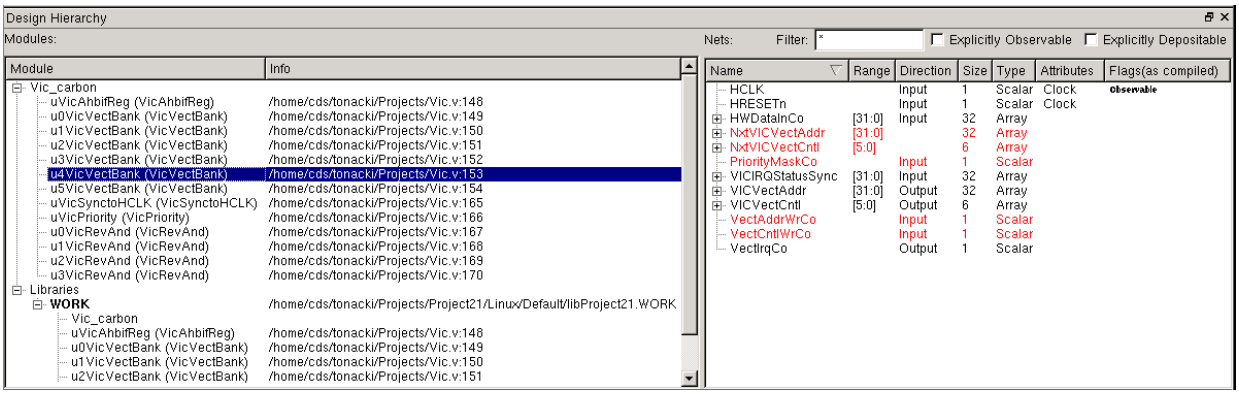


Figure 3-15 Modules and Nets in Design Hierarchy Window

3. Drag and drop modules to the *Directives* pane in the Main view, and apply directives to them (see Figure 3-16). Note that you can only place modules in the Modules tab, and nets in the Nets tab.

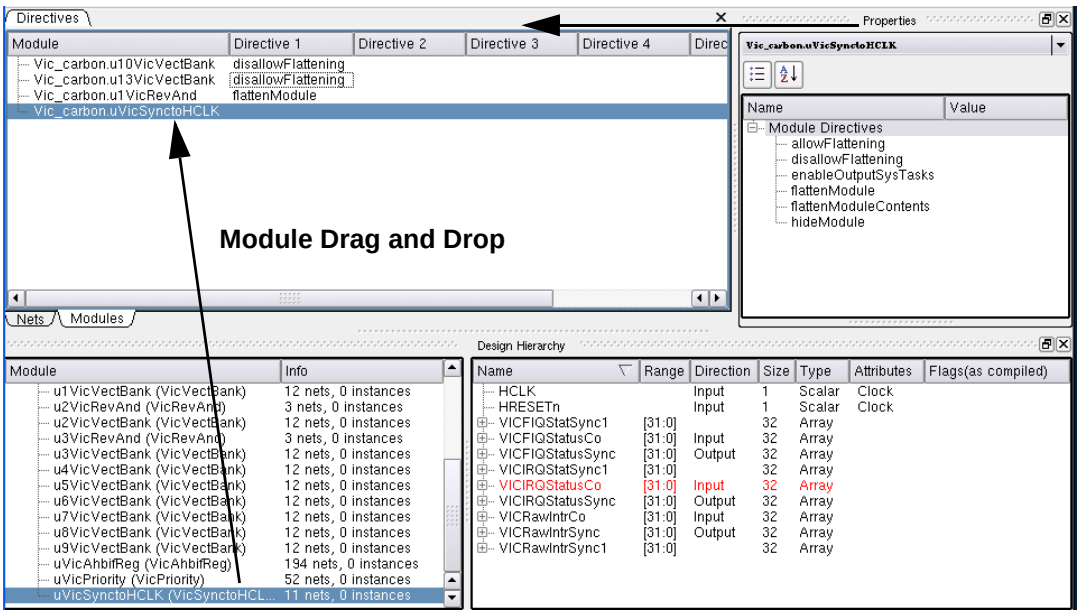


Figure 3-16 Add Module to Module Directives Window

4. Once the Module is in the Directives pane you can apply a single directive, or multiple directives, by selecting a Directives column and selecting the directive from the list. You can also select the directive from the Properties window, as shown below.

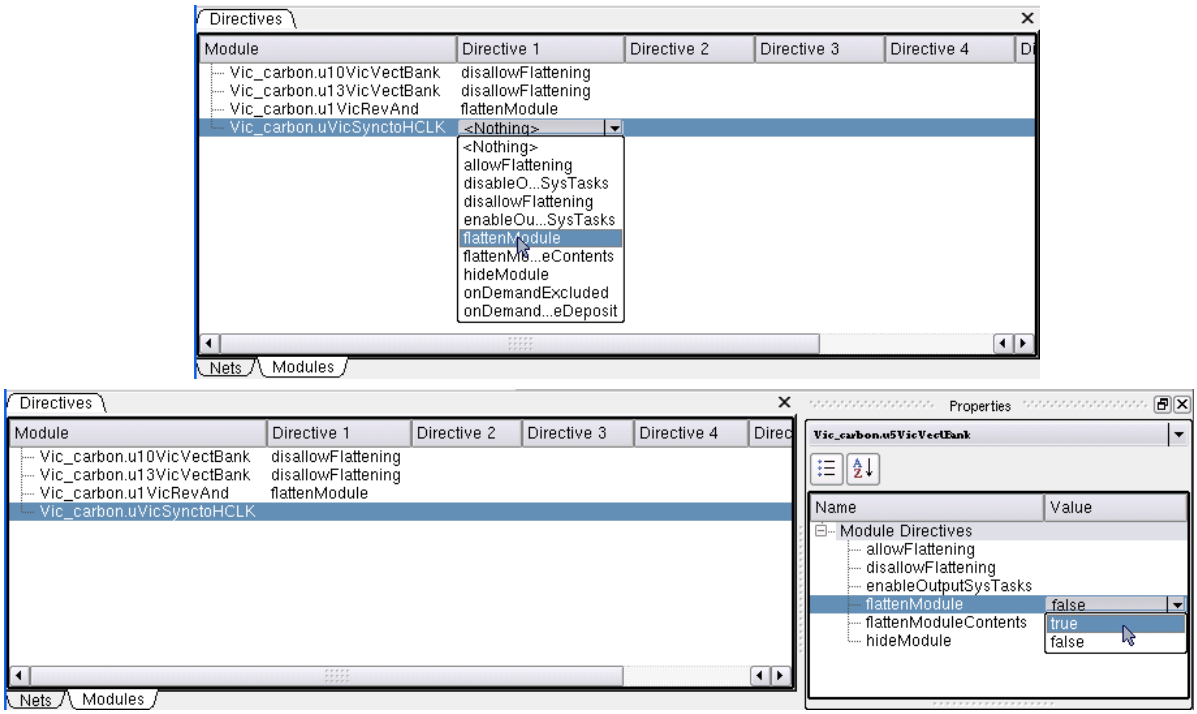


Figure 3-17 Set Module Directives in Directives Window and in Properties

One additional way to set two key directives (Observe and Deposit) to all nets within a module is to right-click the module in the Module window and select the directive.

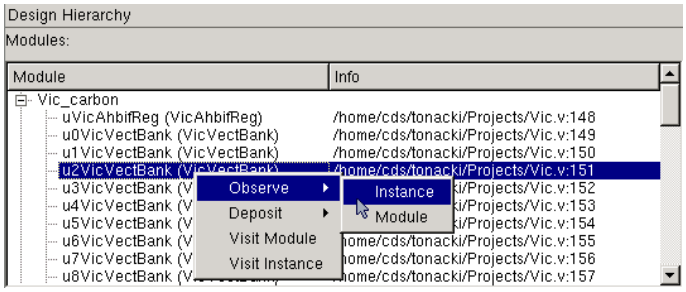


Figure 3-18 Set Module Directives in Design Hierarchy

You can set the *observeSignal* directive or the *depositSignal* directive on all the nets in the module, or just for that instance of the module. Once you click the directive option, the

module is automatically moved to the Module tab in the Directives window and the selected directive (Observable, etc.) is set for that module or instance.

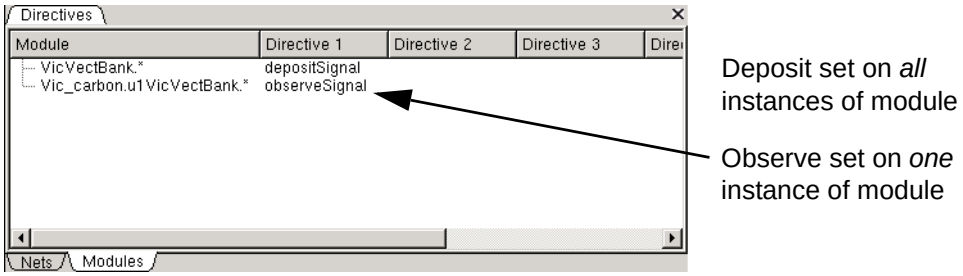


Figure 3-19 Set Module Directives Results

Adding Net Directives

1. Drag and drop nets to the Nets tab in the *Directives* pane and apply directives to them (see Figure 3-20).

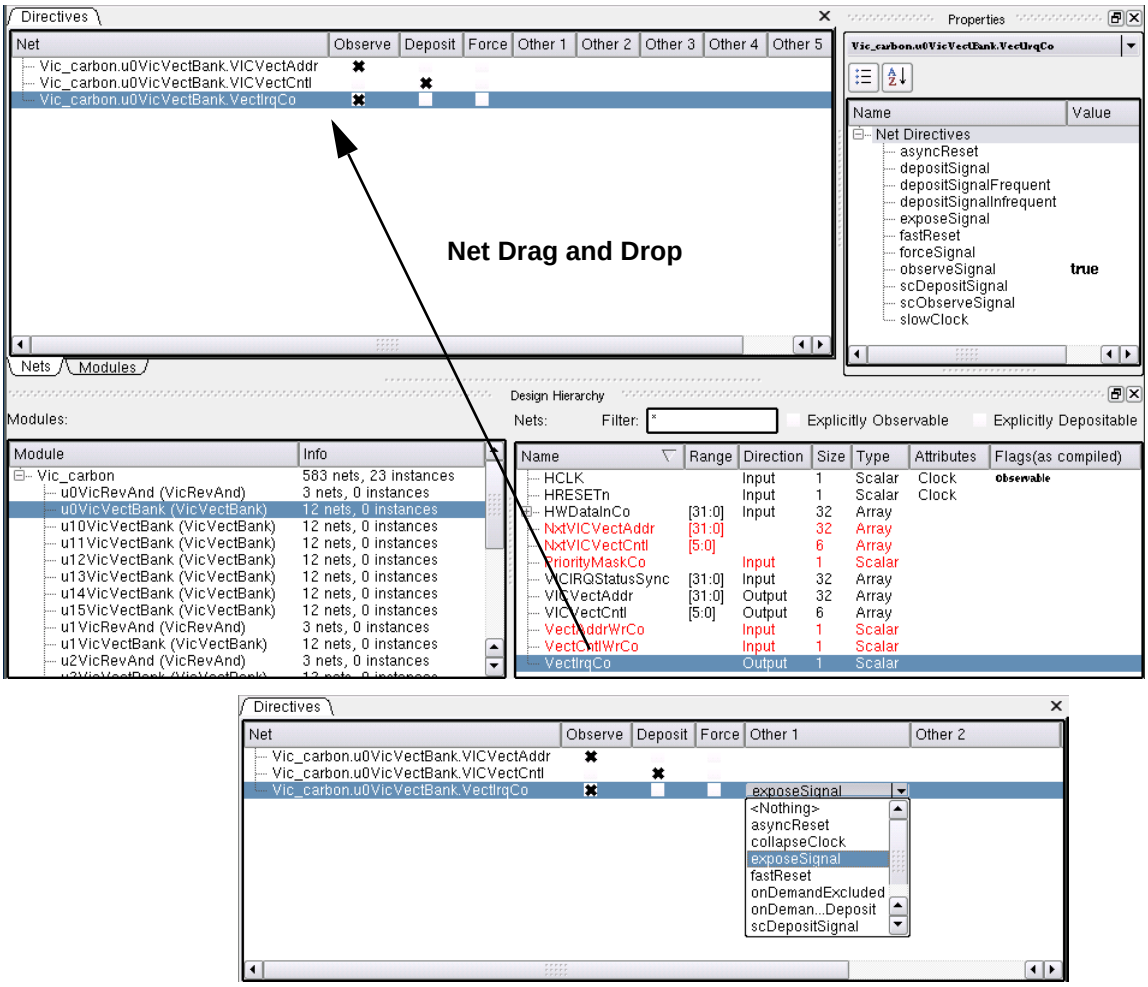


Figure 3-20 Add Net to Net Directives Window and Set Directives

2. Once the Net is in the Directives pane you can apply a single directive, or multiple directives, by selecting a Directives column and selecting the directive from the list. The three

most common net directives (Observe, Deposit, and Force) are provided as check boxes to make selection easier.

You can also select the directive from the Net Directives Properties window as shown below.

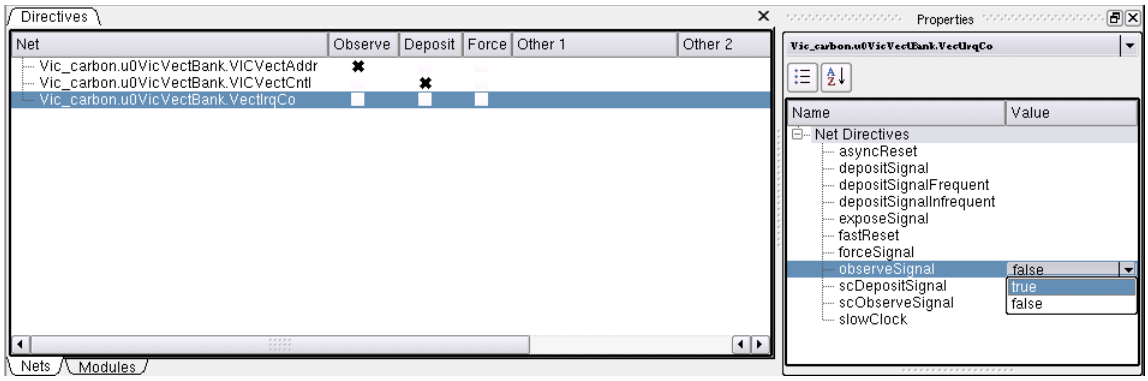


Figure 3-21 Set Net Directives in Properties Window

One additional way to set the common net directives (Observe, Deposit, and Force) is to right-click the net in the Net window and select the directive.

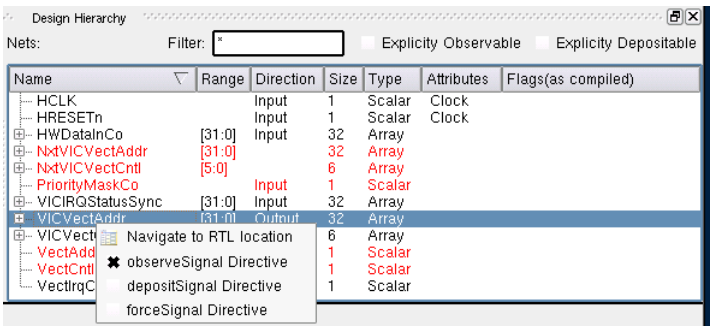


Figure 3-22 Set Net Directives in Design Hierarchy

Once you click the directive, the net is automatically moved to the Net tab in the Directives window and the selected directive (Observable, etc.) is checked.

See Chapter 4 of the *Carbon Compiler User Manual* for a complete description of all the available compiler directives.

3.3.4.2 Embedding Directives in Verilog Source Files

As an alternative to the previous section of applying directives to modules and nets, directives may be embedded in Verilog source as comments. The prefix `carbon`, shown in the examples below, is automatically recognized by the Carbon compiler.

An embedded directive is associated with a net or a module. A sample section of a Verilog file is shown below:

```
...
module top(in1, in2, clk1, clk2, out, ena);
   input in1, in2;
   input ena;   // carbon tieNet 1'b1
   output out;
   input clk1;
   input clk2;   // carbon collapseClock top.clk1

   reg a;   // carbon observeSignal
            // carbon depositSignal
   always @(posedge clk1)
      if (ena)
         a <= ~in1;

   wire out;   // carbon depositSignal
   flop u2(out, in2, clk2, ena);
endmodule
...
```

To use a prefix other than `carbon` to identify embedded directives, you must specify a compiler option.

To embed directives in source files, you can edit the files using an external editor, or you can edit the file within the Carbon Model Studio.

1. Double-click a source file in the RTL Sources list and the file is displayed in the Main view.

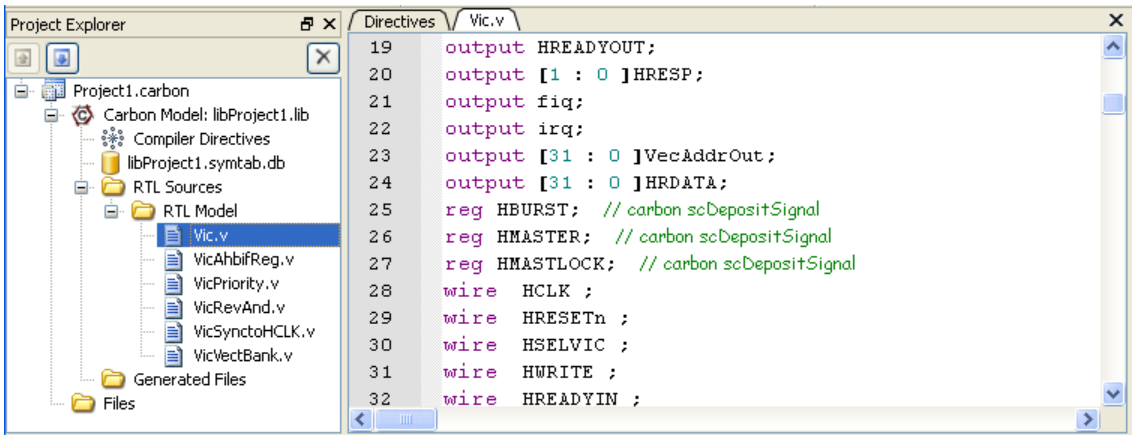


Figure 3-23 Add Directives in RTL Source File

2. Add the directives to the modules and ports. Sample directives are shown in a Verilog file in the figure above.
3. Click the **Compile** button to recompile the Cycle Model with the selected directive settings.

Chapter 4

Creating a Component for Model Validation

This chapter describes the process for creating Cycle Models for Model Validation.

The Model Validation (MV) tool facilitates running a Cycle Model in a verification environment, enabling you to use the same tools and methods to validate the Cycle Model as you use for RTL functional verification.

4.1 Overview

When an RTL model is compiled using Carbon Model Studio, the Carbon compiler generates a C library called the *Cycle Model*. The interface to the Cycle Model is no longer the HDL ports; rather it is a set of C subroutine calls. An existing HDL-based testbench can no longer call the Cycle Model directly.

Model Validation generates a Verilog wrapper to the Cycle Model so that a testbench can instantiate the Cycle Model as if it were the original HDL model. It also generates and supplies the low-level code to form connections between signals in the generated HDL and the Cycle Model.

The generated Verilog model contains only a minimal shell, or *shadow hierarchy.* It contains only the primary (top-level) ports and declarations of signals that the testbench needs to access hierarchically.

The actual simulation is still run through the Cycle Model. A combination of generated C code and libraries provide the connections between the signals in the shadow hierarchy and the Cycle Model.

See the examples in *$CARBON_HOME/examples/model_validation/<simulator_type>/* for more information about using Model Validation.

4.2 Requirements

This release supports:

- ModelSim Verilog on Linux
- Incisive (NCSim) Verilog on Linux
- VCS Verilog on Linux

See the *Carbon Model Validation User Manual* for the supported versions of these simulators, and details about command-line options.

4.3 Model Validation Features

Shareable Library

The MV tool creates a shareable library that can be loaded into the customer's target simulator. The shareable library contains all of the logic needed to update the Cycle Model at the appropriate time and keep all data synchronized between the simulator and Cycle Model domains.

Shadow Hierarchy

The MV tool creates a generated HDL wrapper that includes all primary I/Os and a *shadow hierarchy* for all Cycle Model internally accessible signals. The internally accessible signals are those that were marked either depositable or observable during the Carbon compiler run. This includes the directives `observeSignal`/`scObserveSignal` and `depositSignal`/`scDepositSignal`. The shadow hierarchy allows your testbench to access, via standard RTL code, the internal signals specified during the compile. See the documentation on Directives in the *Carbon Compiler User Manual*.

In other words, your testbench interacts with the shadow hierarchy via the design's RTL signal names, and the shadow hierarchy relays the testbench's commands to the Cycle Model. The testbench can access the shadow hierarchy's signals as if they existed within the simulator's domain because they are kept in sync with their Cycle Model counterparts. In the shadow hierarchy, depositable signals are treated as bidirects so data flows in both directions. Observable signals are treated as Cycle Model outputs.

Tracing

If the shareable library is built with the tracing feature enabled, the generated wrapper allows tracing of all interaction between the Cycle Model and simulator domains. The tracing includes messages for all I/Os between the two domains as well as Cycle Model execution calls.

Waveform Dump and Memory Control

MV defines a *carbon* command that is callable from the HDL and from the various simulator command line and scripting interfaces. This command supports accessing each Cycle Model in the simulation to perform operations such as memory loading and dumping, as well as waveform control.

The *carbon* command is implemented as a Verilog user-defined system function for Verilog designs. Therefore it is also callable from a Verilog testbench. The syntax for each simulator is different. The semantics of the commands are the same, however. See the *Carbon Model Validation User Manual* for more information about the *carbon* command.

4.3.1 Signal Types

Model Validation supports only certain data types from design files for the purpose of observing and depositing data.

4.3.1.1 Verilog Support

MV supports the following Verilog data types:

- reg: bit and vector
- wire: bit and vector
- integer

Data types that are not supported:

- two-, or more, dimensional memories

4.4 Prerequisites

Before you generate the Cycle Model component for Model Validation, make sure you have performed the following steps:

- If they have not already been defined, set the necessary environment variables:

```
MTI_HOME = <ModelSim installation directory>, or
SIM_HOME = <NCSim installation directory>, or
VCS_HOME = <VCS installation directory>

CARBON_HOME = <Carbon installation directory>
```

- Make sure that the Cycle Model output libraries are static:

  For Linux: `lib<design_name>.a`

  For Windows: `lib<design_name>.lib`

  Mark all signals that are accessed by your testbench as depositable and/or observable using the `depositSignal`/`scDepositSignal` and `observeSignal`/`scObserveSignal` directives. See "Defining Directives and Recompiling the Cycle Model" on page 69 for more information.

- Compile your Cycle Model following the steps defined in Chapter 3.

4.5 Generating the Component for Model Validation

This section describes how to generate a component to be used with the Carbon Model Validation tool.

1. From the Project Explorer, right-click the Carbon Model to display the context menu.
2. From the context menu, select *Create Model Validation Component*.

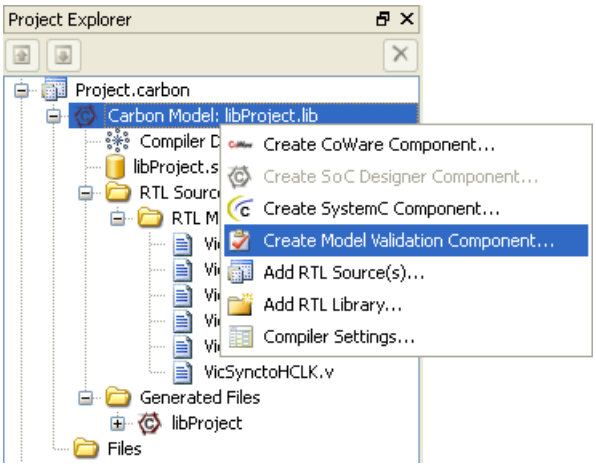


Figure 4-1 Create Model Validation Component

You can also click the **MV** button on the Toolbar.

The new component, and associated output files, are displayed in the Project Explorer as shown in Figure 4-2.

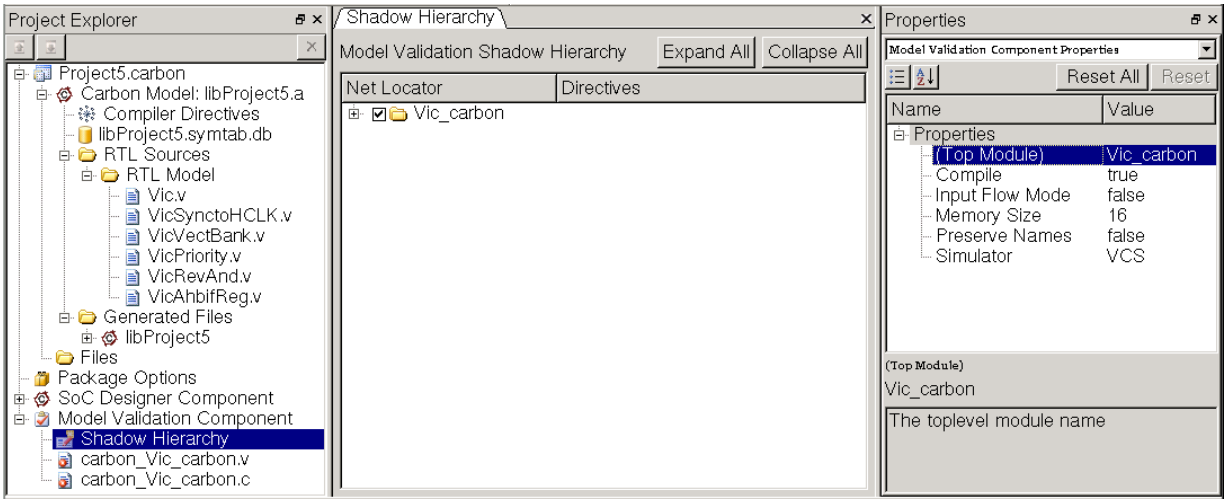


Figure 4-2 Model Validation Component Windows

Additionally, the *Model Validation Shadow Hierarchy window* appears in the Main view, and the *Model Validation Properties* appear in the Properties view.

4.6 Setting Model Validation Properties

Use the Model Validation Properties view to select the simulation environment type, enable or disable automatic makefile generation, and select other settings when compiling the MV component.

To set Model Validation properties:

1. Click on the Model Validation component in the *Project Explorer* and the Model Validation Component Properties settings appear in the *Properties* view.

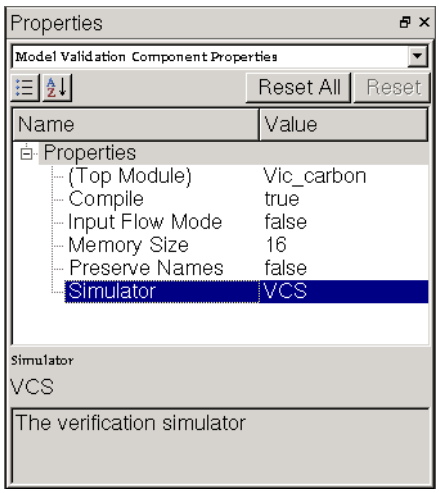


Figure 4-3 Model Validation Properties

2. Set the desired values, as described in the following table.

Property	Description
Compile	Select *true* to automatically generate a makefile during the compilation of the Model Validation component. Set to *false* to disable this feature. The default is *true*.
Input Flow Mode	This option determines whether the data or the clock wins when they both arrive at the primary ports at the same time. The default setting is *false*, where the input data is delayed; the clock wins race conditions and flops the previous data. Set this value to *true* to instruct MV to generate the interconnect such that the data is flopped into the affected registers.
Memory Size	This option determines the maximum number of elements in a memory net. If a net that has been marked observable or depositable has more elements than the limit set by this option, then MV does not use this net. The default maximum number of elements is 16. *This option is available when using ModelSim only.*

Property	Description
Preserve Names	This option determines if the original module names are preserved in the created shadow hierarchy. The default setting is *false*, where the shadow hierarchy that MV generates uses different module names. It does this to avoid name conflicts, in case you want to load in parts of your original HDL model. Set this option to *true* to keep the original module names and not create unique names.
Simulator	Select the simulation environment to be used. The choices are: • VCS • ModelSim • NCSim

3. Click **Save All** to save your MV and project settings.

4.7 Removing Nets from the Model Validation Shadow Hierarchy

In addition to the primary (top-level) ports, all nets that have been marked with the directives `observeSignal`/`scObserveSignal` or `depositSignal`/`scDepositSignal` are observable and depositable when running functional verification. These same nets are observable and depositable when running Model Validation.

By default, all observable and depositable nets are displayed in the *Model Validation Shadow Hierarchy window*, as shown below.

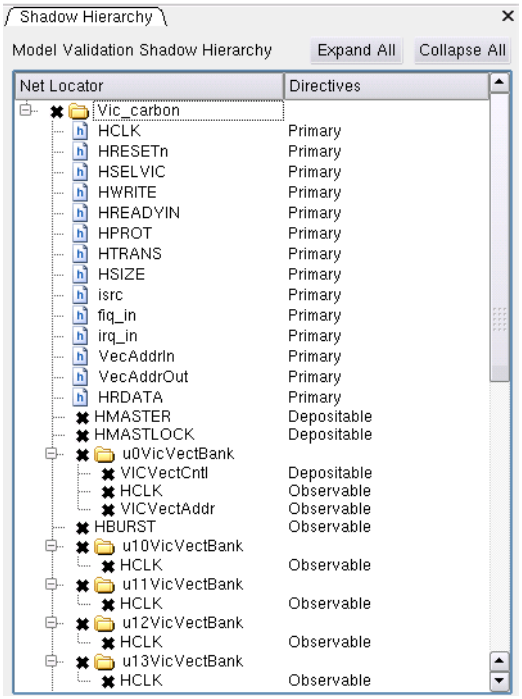


Figure 4-4 Expanded Shadow Hierarchy

The top-level module is listed at the top of this window. The Primary ports are listed next, followed by all subordinate modules and nets. The check-mark indicates that the nets are currently observable and/or depositable. The check-mark on the top module indicates that ***all*** subordinate modules and nets are checked. You can check or uncheck this top-level module to check or uncheck all subordinate modules and nets. Primary ports are unaffected and are always observable and depositable through the shadow hierarchy.

Since you want fewer nets available for Model Validation, restrict the nets that are observable and depositable from MV before you generate the shadow hierarchy.

To remove nets from the shadow hierarchy, simply uncheck the nets that you do not need for your testbench. You can click a module-level check-box to enable or disable all nets in the module.

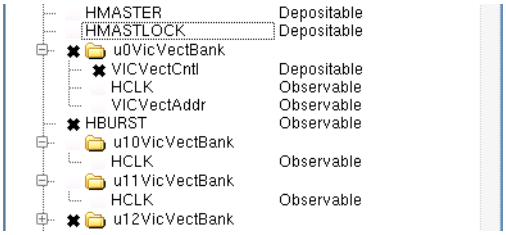


Figure 4-5 Remove Nets from Shadow Hierarchy

4.8 Recompiling the Model

After you have defined all the specific settings for your MV component, you need to recompile your project. Click the **Compile** button at this point to generate the shadow hierarchy and update other files that to be used with your component.

Note that if you change just the shadow hierarchy contents, when you click **Compile**, only a new Model Validation component is generated - no compile of the Cycle Model occurs.

However, if you make any changes to the Cycle Model, like making additional signals observable, then the Cycle Model is recompiled at the same time the new MV component is generated.

Chapter 5

Creating Components for Specific Platforms

This chapter describes the process for creating Cycle Models for specific platforms. *Carbon Model Studio* can create models for multiple platforms, as described in the following sections:

- Understanding the Process
- Starting and Configuring the Project
- SoC Designer Plus-Specific Information
- Using the Component in SoC Designer Plus
- Platform Architect-Specific Instructions
- Creating Components for SystemC

5.1 Understanding the Process

Carbon Model Studio manages the process of creating models for specific platforms so that changes made to the source RTL automatically regenerate any components derived from that RTL (see Figure 5-1).

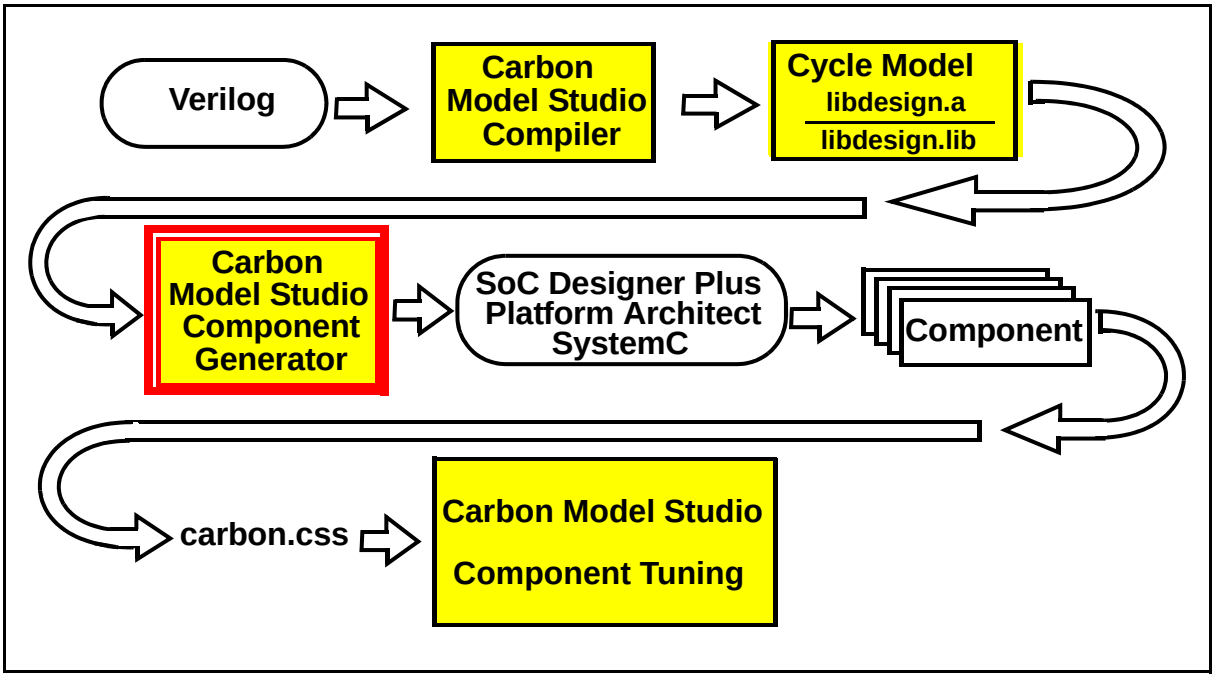


Figure 5-1 Carbon Model Studio Process Flow - Component Generation

After compiling the Cycle Model from your source RTL, the next step is to generate a component that is compatible with your simulation environment.

5.1.1 Carbon Model Studio Component Generator Overview

The Carbon Model Studio tool can package a Cycle Model in the form of an SoC Designer Plus, Platform Architect, or SystemC component. Refer to the section Starting and Configuring the Project for general instructions and information about configuration options. For important information about the different component output formats, refer also to the relevant section: SoC Designer Plus-Specific Information, Platform Architect-Specific Instructions, or Creating Components for SystemC.

The Carbon Model Studio tool has several powerful features that allow you to integrate more tightly with the SoC Designer Plus environment, for example:

Transactor Adaptors

Connect your component directly to transaction-level interface ports.

Clock and Reset Generators

Note: Clock and Reset Generators are not supported for Platform Architect components.

The clocks are an abstract construct that merely indicate when it is time for your component to execute another ‘cycle.’ HDL designs typically require alternating signal values of 0 and 1. The Carbon Model Studio tool’s clock and reset generators automatically stimulate your Cycle Model with signal values on every clock pulse, or on the schedule you set up. Refer to “Clock Inputs and Clock Generators” on page 160 for more information.

Debug Registers

RTL signals deep inside the Cycle Model may be made available as named debug registers for read and write during simulation.

Debug Memories

HDL memories may be made available for debugging by adding a memory to the memories tab (see “Memories Tab” on page 119).

Profiling

Capture Cycle Model state data in a form that can be easily analyzed using the SoC Designer Plus profiling tools.

Note: Profiling is not supported for Platform Architect components.

Ties and Disconnects

Cycle Model input ports (e.g. test inputs) may be tied to a constant value instead of exposed as component ports.

Cycle Model output ports may be left unconnected (disconnected), instead of exposed as component ports.

Port Expressions

The value of signal ports may be modified by providing a C language expression. This is often useful to bridge the gap between an abstract and RTL signal interface (for example, to convert an integer value into a bit mask).

Pseudo-Transactors

Some common Port Expressions have been packaged in the form of Transactors for your use. For example, the Interrupt_Master and Interrupt_Slave transactors handle conversion of integer interrupt numbers into bit masks, and vice-versa.

The Null_Input and Null_Output transactors may be used to add extra SoC Designer Plus ports to the component that are not connected to the Cycle Model. This is useful for matching the port interface of a behavioral component for which the component will be substituted. See Appendix A for more information.

5.1.1.1 Transactors

SoC Designer Plus supports transaction-level port interfaces, but Cycle Models typically communicate at the pin level. Transaction adapters (known as *Transactors*) are provided to convert between several common transaction level interfaces and the Cycle Model pins.

Note: If you are creating a model for Platform Architect, note that the term used in the GUI changes from "Transactor" to "Protocol."

Refer to the *SoC Designer Plus User Guide* or Platform Architect documentation for details about the difference between transaction-based and signal-based communication in the selected environment.

The transactors you add to your component in the component typically add new SoC Designer Plus transaction ports, and hide Cycle Model ports.

For example, an AHB_Slave_T2S converts AHB transactions to signal value changes (T2S means "transaction to signal"). When you add an AHB_Slave_T2S transactor to your component in the Carbon Model Studio tool, a new transaction slave port is created in the component for incoming AHB transactions. You must use the Carbon Model Studio tool to connect the transactor to the appropriate Cycle Model signal ports. Each Cycle Model signal connected to the transactor is no longer exposed as a port.

The resulting component may be connected to a transaction master port.

5.1.1.2 Component Clocking

The component generated by the Carbon Model Studio tool uses cycle-based scheduling. Refer to the *SoC Designer Plus User Guide* or Platform Architect documentation for information about cycle-based scheduling.

The component has a clock input port named 'clk-in'. This is a cycle-based clock, not an RTL signal port. The clk-in port determines how frequently the component is called to execute a cycle. For more information about clocking options, refer to "Clock Inputs and Clock Generators" on page 160.

- If you connect the clk-in port to the clock master port of another component (e.g., a CDIV clock divider), then that component determines when the component runs a cycle.
- If you do not connect the clk-in port to anything, then the component is driven by the simulator reference clock, running one cycle per time unit.

The component is a subclass of C++ class `sc_mx_module`. It implements the two methods — `update()` and `communicate()` — which are called by the clock master to which the clk-in port is connected.

In the `communicate()` method, all output ports that have new values are driven.
In the `update()` method, the Cycle Model is executed.

5.1.1.3 Clock Generation

In SoC Designer Plus cycle-based scheduling, there is no signal value (i.e., 1 or 0) associated with a clock slave port. The SoC Designer Plus clock slave port is only a mechanism for configuring when the component `communicate()` and `update()` methods are invoked.

Note: Clock Generators are not supported for Platform Architect components.

You can drive the Cycle Model clock signal ports using a Carbon Model Studio clock generator. A clock generator applies 0 and 1 signal values to the Cycle Model clock signal ports every time the component's `update()` method is called. A clock generator may be configured to run at the same speed, faster, or slower than the SoC Designer Plus reference clock.

Note: Rather than creating a clock generator, ARM recommends creating a clock input transactor and exposing the clock input port externally.

For additional information on the use of clock generators, refer to "Clock Inputs and Clock Generators" on page 160. For instructions on creating a clock generator, refer to "Specifying Generated Clocks" on page 100.

5.2 Starting and Configuring the Project

Use the Carbon Model Studio Project Explorer view to begin.

1. In the Project Explorer, right-click the Carbon Model to display the context menu.
2. From the context menu, select:
   - *Create SoC Designer Component* for SoC Designer Plus
   - *Create CoWare Component* (for Platform Architect)
   - *Create SystemC Component* (for SystemC)

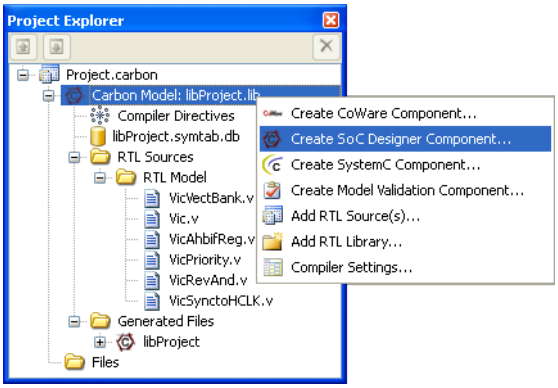


Figure 5-2 Select the Type of Component to Create

The new component, and its associated output files, are displayed in the Project Explorer as shown in Figure 5-3.

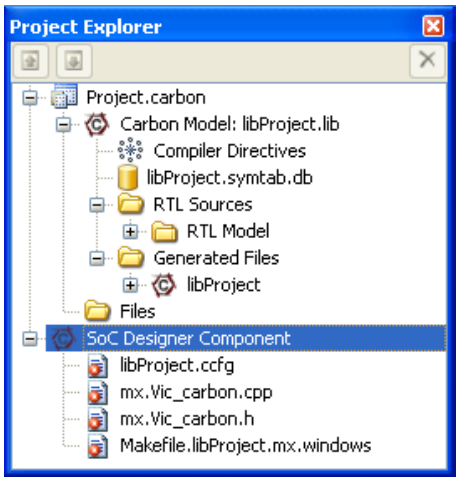


Figure 5-3 Component for SoC Designer Plus in Project Explorer

5.2.1 Editing the Component Properties

Use the Component Properties view (shown in Figure 5-4) to edit the component properties.

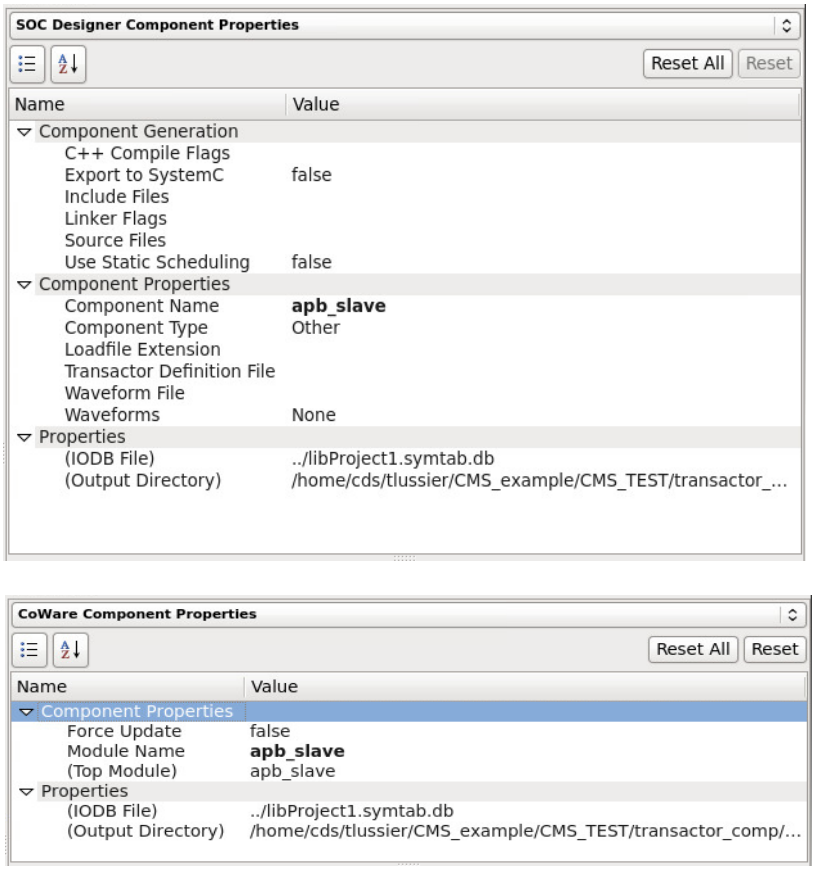


Figure 5-4 SoC Designer Plus Component Properties

The component properties differ depending on the type of component you are working with; for example, SoC Designer Plus components have “Waveform File” and “Waveforms” properties, while Platform Architect components do not display these properties. When a property is selected, information about that property is displayed in the lower-right information panel of the CMS GUI.

5.2.1.1 Naming the component

By default, the name of the top-level module is used for the component, the SoC Designer Plus C++ class name, and the `*.cpp` and `*.h` files generated by Carbon Model Studio. The Carbon Model Studio project name is not used. This name is shown in the *Component Name* or *Model Name* field.

To change the component name:

1. Click in the *Component Name* field (for Platform Architect, *Module Name*) to highlight the current component name.
2. Type the new name.

5.2.1.2 Specifying Force Update

Force Update (supported for Platform Architect only) specifies that calls to `sc_prim_channel::request_update` are forced for all input changes. If this is not specified, `request_update` is called only for clock, reset, and feed-through inputs.

5.2.1.3 Enabling Waveform Generation

For SoC Designer Plus

To use the dump to waveform feature:

1. In the *Properties* view, under the *Component Properties* section, click the **Browse** button in the *Waveform File* field to display the *Output File* dialog (Figure 5-5).

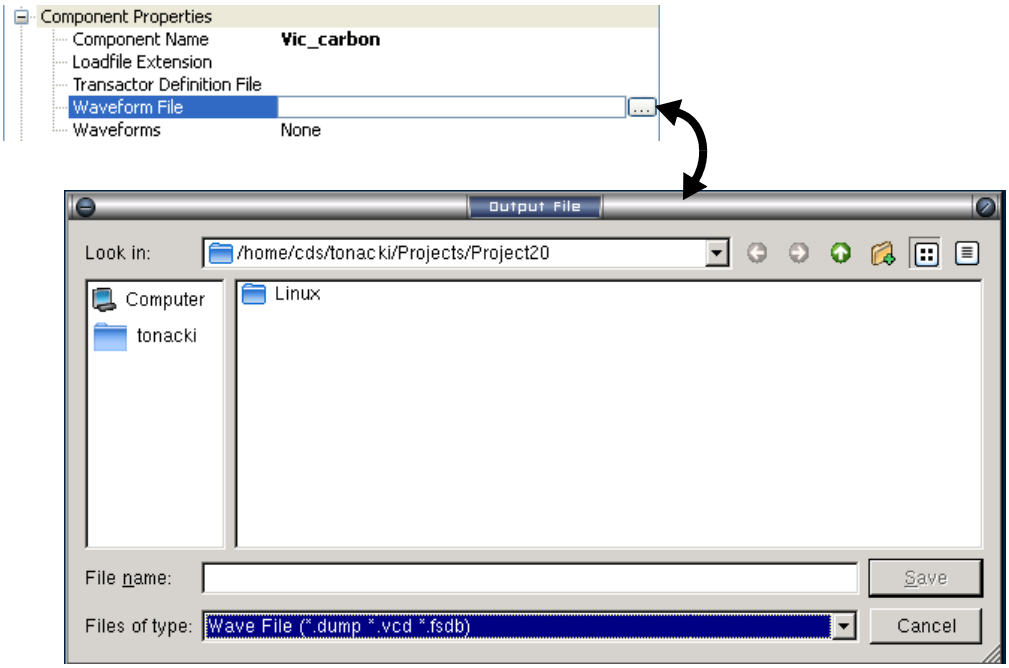


Figure 5-5 Enable Waveforms in SoC Designer Plus Component

2. In the *Output File* dialog, browse to the location of an existing wave file, or create a new file name in the *File name* field, and then click **Save**.
3. Click the drop-down menu from the *Waveforms* field and select the waveform file format you want to use, as shown in Figure 5-6.

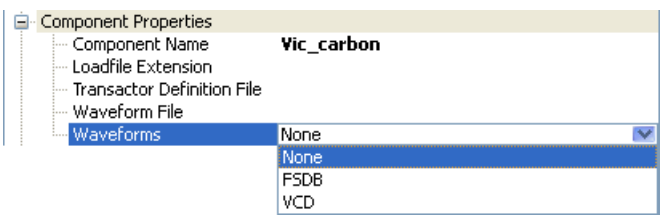


Figure 5-6 Waveform Type

Note: By default, design elements larger than 1024 bits are not dumped to waveform files. To allow elements larger than 1024 bits to be dumped, increase the maximum size using the -waveformDumpSizeLimit compile option under Net Control.

For Platform Architect

To dump data to waveforms for Platform Architect:

add

```
-DCARBON_DUMP_VCD=1
```

or

```
-D CARBON_DUMP_FSDB
```

to scsh preprocessor options.

For example:

```
::scsh:: cwr_append_simbld_opts preprocessor -DCARBON_DUMP_FSDB=1
```

Note: By default, design elements larger than 1024 bits are not dumped to waveform files. To allow elements larger than 1024 bits to be dumped, increase the maximum size using the Carbon compiler -waveformDumpSizeLimit option under Net Control.

5.2.1.4 Setting Compiler and Linker Flags

The Carbon Model Studio allows you to customize your component by adding C++ Compile flags, with paths to include files, macros, etc., and Linker flags, with paths to include libraries, source files, etc.

Note: This functionality is not supported for Platform Architect.

To use the Compiler and Linker flag feature:

1. In the *Component Properties* view, click the *Value* cell in the *C++ Compile Flags* row to enable the text field (Figure 5-7).

Figure 5-7 Compiler and Linker Files and Flags

2. Enter the required compile flags.

Note: If you are creating a model to be used as a Swap & Play memory controller, add the following C++ Compile Flag:

```
-DCARBON_MEMORY_RESTORE_PORT_<portname>
```

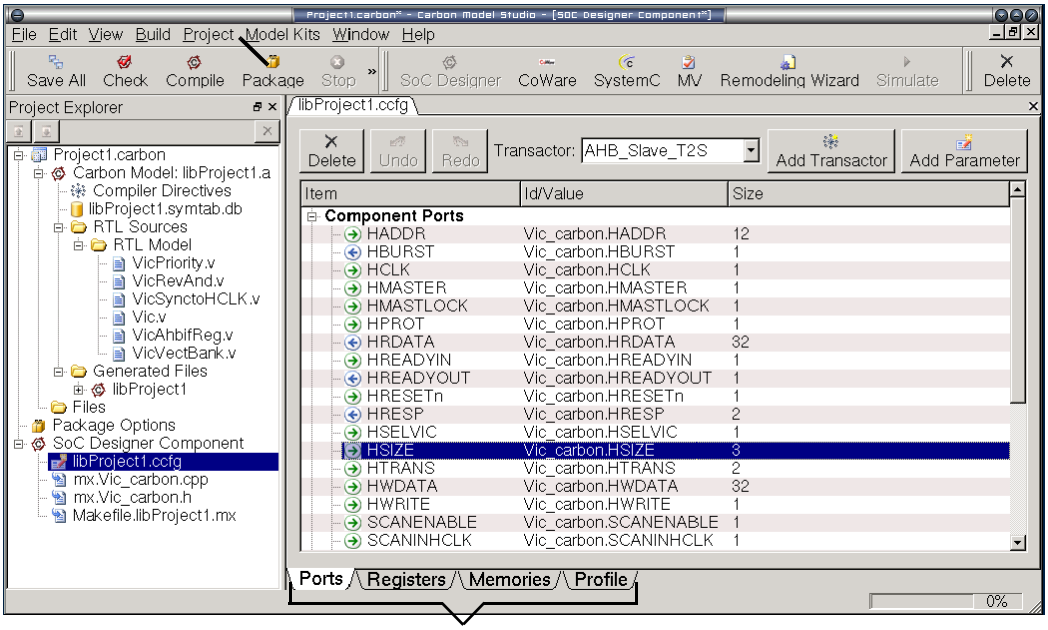
where <portname> is the port with the lowest memory address region.

3. Click the *Value* cell in the *Include Files* row to add additional include files to be passed into the generated Makefile.
4. Click the *Value* cell in the *Linker Flags* row and enter any required *Linker* flags.

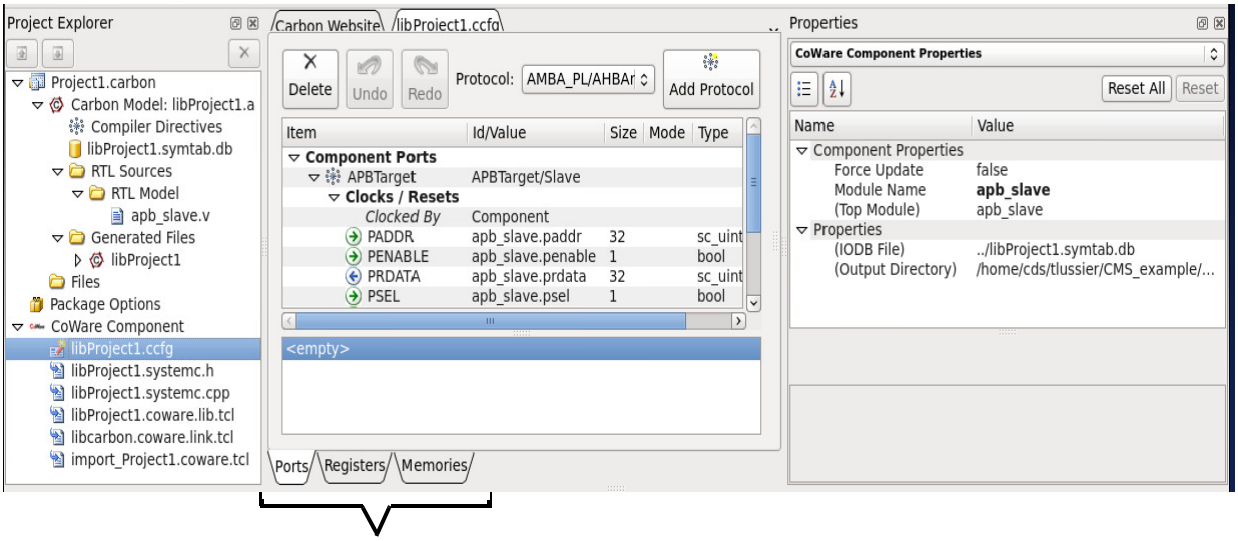
5. Click the *Value* cell in the *Source Files* row to add additional source files to be passed into the generated Makefile. For example, if you plant to use disassemblies (see "Creating Disassembly Views" on page 126), then you must list here the source files to be used.

5.2.2 Component Editing View

The *Component Editor* provides tabs to access *ports*, *registers*, *memories*, and *profile* (SoC Designer Plus only). Double-click the .ccfg file in the Project Explorer to display the Component Editor (Figure 5-8).



Component Editor Tabs for SoC Designer Plus



Component Editor Tabs for Platform Architect

Figure 5-8 CMS Component Editor

5.2.2.1 Ports Tab

Use the options on the *Ports* tab to connect the Cycle Model's inputs and outputs to the component's signals and drivers.

The *Ports* window is composed of sections in a tree-like structure: *Component Ports*, *Component Parameters*, *RTL Ties*, *RTL Disconnects*, *Reset Generators* (not available for Platform Architect), and *Clock Generators* (not available for Platform Architect). You use this window to define and configure the ports that are exposed in the completed component.

Component Ports Window

By default, the *Component Ports* section of the tree is expanded. The first column lists the ESL ports mapped to the corresponding RTL ports in the second column. Initially, there is a one-to-one correspondence of port names in the RTL and ESL Ports lists. The RTL ports that are displayed include all the primary I/Os of the Cycle Model, plus internal signals that were marked with the directive *observeSignal/scObserveSignal* or *depositSignal/scDepositSignal* when the Cycle Model was compiled with the Carbon compiler.

Each RTL port may be connected to the following sources:

- ESL Input or ESL Output
- Generated Clock
- Generated Reset
- Tie
- Disconnect
- Transactors

These connections are established by selecting one or more RTL ports (use **Ctrl-click** to select multiple ports) and then selecting an entry from the **right-click** context menu. The context menu offers different functions depending on the type of port, or node, you select in the Ports tab. This is shown in Figure 5-9.

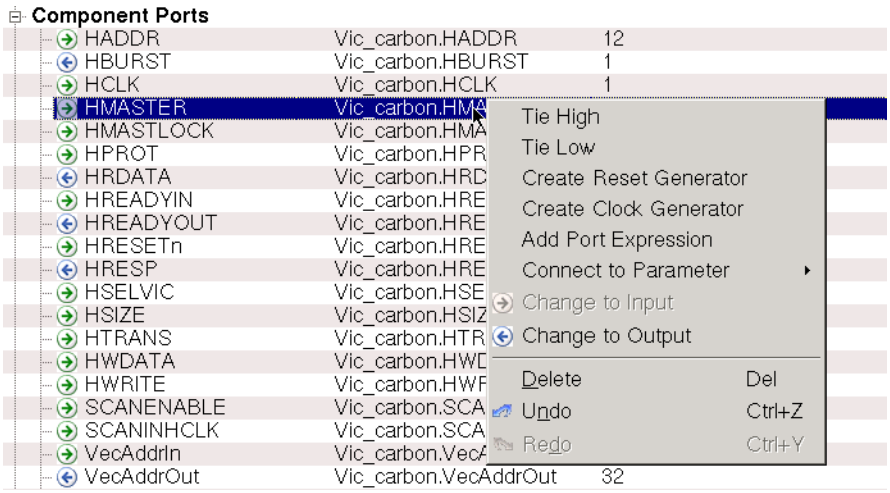


Figure 5-9 SoC Designer Plus Component RTL Port Connection Options

The ESL ports are the ports through which SoC Designer Plus or Platform Architect connect to the component. These include signal ports (connected directly to the Cycle Model) and transactor ports. As necessary, reconnect RTL ports to clock generators, reset generators, ties, and

transactor ports. You can also add additional ESL ports from the Design Hierarchy, and change the direction of ports.

Depending on the option you select, the RTL port is moved to the applicable section. For example, if you disconnect a port it is moved to the *RTL Disconnects* section. Any errors that occur during these operations appear in the Model Studio Status Bar. These selections, and their corresponding fields, are described in the following pages.

Note: If an RTL signal is connected to a Generated Clock, Generated Reset, or Tie, then it may only have one connection.

Ports Toolbar

The *Ports* tab includes the Ports Toolbar (Figure 5-10), which you can use to further define your SoC Designer Plus component.

Figure 5-10 Ports Toolbar

The buttons include:

- **Delete** — Performs different functions depending on where this operation is used. For example, in the *Component Ports* section this moves an RTL port to the *Disconnects* section. In the *Disconnects* section this operation moves the port back to the *Component Ports* section to reconnect to an ESL port.
- **Undo** — Cancels the last operation.
- **Redo** — Performs a redo of a previous undo operation.
- **Transactor:** — Lists the available transactors. For Platform Architect, this is “Protocol.”
- **Add Transactor** — Opens the *Add SoC Designer Transaction Port* dialog, where you can add a transactor to the SoC Designer Plus component and map the RTL ports to the transactor ports. For Platform Architect, this button is called “Add Protocol.”
- **Add Parameter** (not available for Platform Architect) — Adds parameters to your component that can be accessed in the SoC Designer Plus environment. The parameter can also be tied to RTL or ESL ports.

Changing ESL Port Names

The port names shown in the list are the names to be assigned to the ports of the finished component. By default, the input and output pin names used in the RTL code are also used in the ESL port list. Carbon Model Studio enables you to change the ESL port names within the *Component Ports* section (see Figure 5-11).

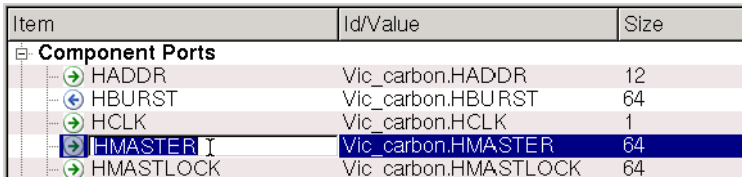


Figure 5-11 Component ESL Port Name

To change the name of an *ESL port* or *transactor port*:

1. In the *Component Ports* section, click on the port you want to rename.
2. Type the new name.

Tying Off Signals

You should disconnect, or *tie off*, any signals that do not need to change during validation, such as scan-enable. To tie off *input* pins:

1. In the *Component Ports* section, 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 along with the selected tie value, and it is removed from the *Component Ports* list, indicating that this pin will not be used as an input port. You can change the tie value later in the *RTL Ties* list (Figure 5-12) if required.

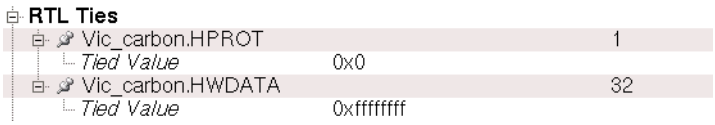


Figure 5-12 RTL Ties list

Note: Any RTL port on which the tieNet directive has been applied is automatically marked as tied in this pane, with the message "Set by Compiler".

If you decide later that a port does not need to be tied off, just right-click the port in the *RTL Ties* section and select **Delete** from the context menu. The port is removed from the tie offs list and appears back in the ESL ports list.

Disconnecting Signals

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

1. In the *Component Ports* section, 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 (Figure 5-13), and it is removed from the *Component Ports* list, indicating that this pin will not be used as an input or output port.

Figure 5-13 RTL Disconnects list

If you decide later that a port should be connected to an ESL port, just right-click the port in the *RTL Disconnects* section and select **Delete** from the context menu. The port is removed from the Disconnects list and appears back in the *Component Ports* section.

Specifying Generated Clocks

When you mark your design's input clock signals as generated clocks, the Component generator does not create ESL ports for these signals. Instead, the Component generator creates internal signals that are used to activate the component when SoC Designer Plus sends a reference clock signal to the component.

Note: Clock Generators are not supported for Platform Architect.

Note: Rather than using clock generators, ARM recommends creating clock input transactors and exposing the clock input ports externally. For more about the use of clock generators, refer to the section "Clock_generator" *on page 164.*

1. In the *Component Ports* section, select the signal that 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 it is removed from the *Component Ports* list.
3. Modify the clock controls as necessary in the *Clock Generators* section, as shown in Figure 5-14.

Figure 5-14 SoC Designer Plus Component Clock Settings

For details about parameter settings, refer to "Clock Inputs and Clock Generators" on page 160.

Specifying Generated Resets

In general, you should mark your design's reset signals as generated resets. This prevents the Component generator from creating unnecessary ESL ports for these signals. Instead, the Component generator creates internal signals that activate the component when SoC Designer Plus sends a user-activated reset signal to the component.

Note: Reset Generators are not supported for Platform Architect.

1. In the *Component Ports* section, select the signal that you want to designate as a reset.
2. Right-click and select **Create Reset Generator** from the context menu.

   The signal and default values appear in the *Reset Generators* section, and it is removed from the *Component Ports* list.
3. Modify the reset controls as needed in the *Reset Generators* section, as shown in Figure 5-15.

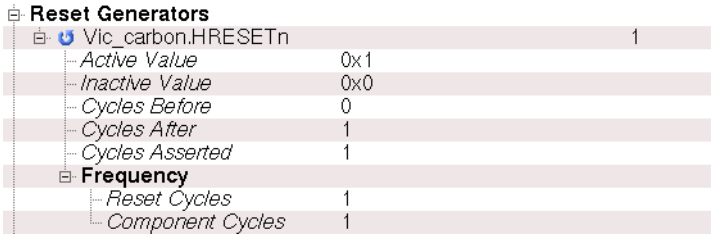


Figure 5-15 SoC Designer Plus Component Reset Generator Settings

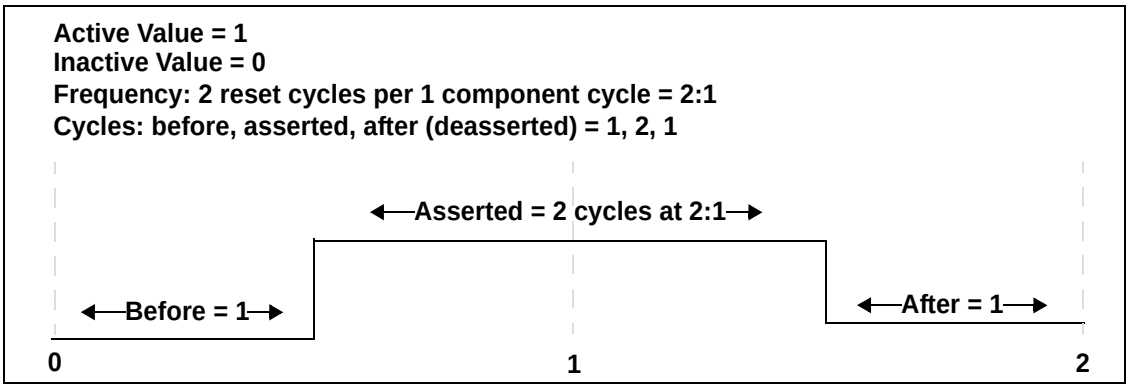
Resets are specified as follows:

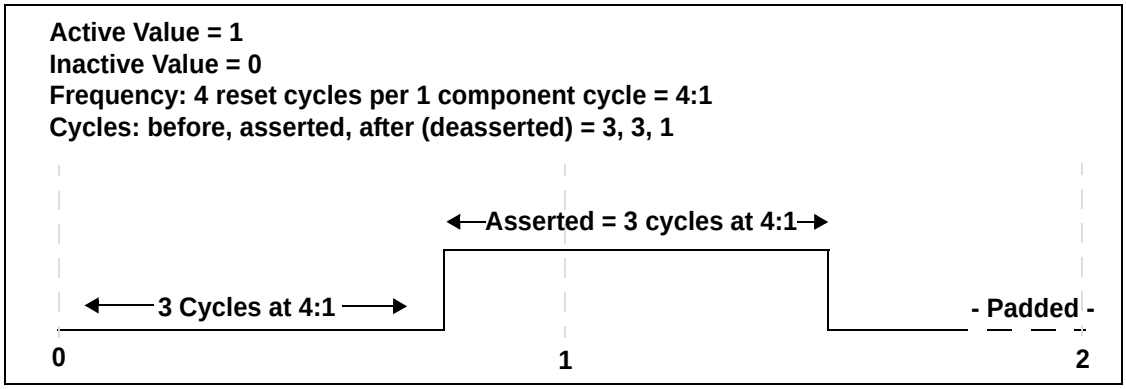
– *Active Value* and *Inactive Value*: Specify a hex or decimal value to be used when reset is active (asserted) and when it is inactive (deasserted). The default is 1 for active and 0 for inactive. The values can be used so the reset generator can be applied to a bus of reset signals.

– *Cycles Before*, *Cycles Asserted*, *Cycles After*:
Cycles Before is the number of cycles to wait during the reset period before asserting the signal.
Cycles Asserted is the number of cycles the signal remains asserted.
Cycles After is the minimum number of cycles the signal remains deasserted before the reset period ends.

– *Frequency*, as a reset cycle-to-component cycle ratio:
1:1 means one reset cycle per one component cycle; this is the default setting
2:1 means 2 reset cycles to one component cycle
1:2 means 1 reset cycle extends across two component cycles

The length of the reset period is determined by the longest reset waveform duration. All reset waveforms that complete before the end of the period are extended by padding the deasserted time. See the examples below.





Adding ESL Ports

All ports are automatically added and displayed in the Ports view. You can add additional ESL Ports that you want to be accessible through the component. Only ports that have been marked observable or depositable can be added to the component.

Right-click on the *Component Ports* node, as shown in Figure 5-16, and select “Add transaction port” from the context menu.

Figure 5-16 Add Port to Component

Click on the new port and use the context menu to make further changes.

By default, the ESL Port name is the same as the RTL net name. See “Changing ESL Port Names” on page 99 if you want to change the port name that is exposed in the component.

Adding Transactors and Other Interface Entities

This section applies to transactors, protocols (for Platform Architect), and other entities, referred to as pseudo-transactors, which act as go-betweens from ESL ports to Cycle Model ports. The pseudo-transactors handle interrupt signals, input clocks driven at variable speeds, generated output clocks, and null ports (ESL ports that are not connected to RTL ports). In the steps below, the terms *transactor* and *protocol* include pseudo-transactor entities. See Appendix A for more information.

To use transactors/protocols in your component, you need to specify the type of transactor or protocol, and then map the desired RTL ports from your design to the transactor’s or protocol’s ports. SoC Designer Plus creates a connection to the transactor/protocol, but the connections between the transactor/protocol and the Cycle Model are invisible to SoC Designer Plus.

To add a transactor or protocol:

1. In the *Ports Toolbar*, click the **Add Transactor** button (for Platform Architect, the button is **Add Protocol**). The Add Port dialog appears (Figure 5-17).

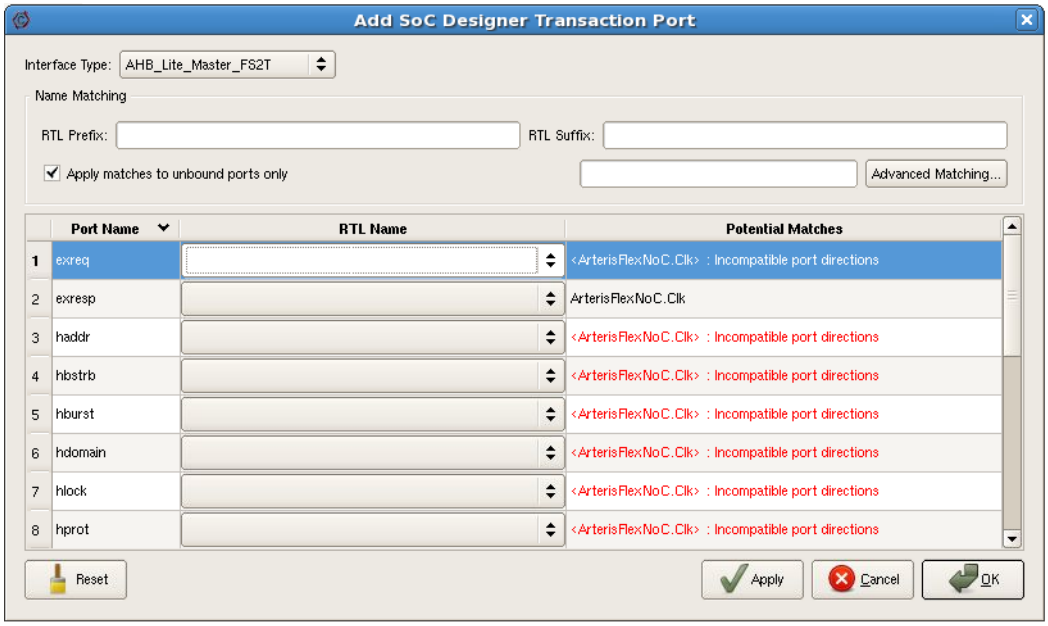


Figure 5-17 Add Port dialog

2. Click the *Interface Type* pulldown menu to display the available transactors. Select the transactor or protocol to use for your design; the example (Figure 5-18) selects an *AXI_Flowthrough_Master* transactor.

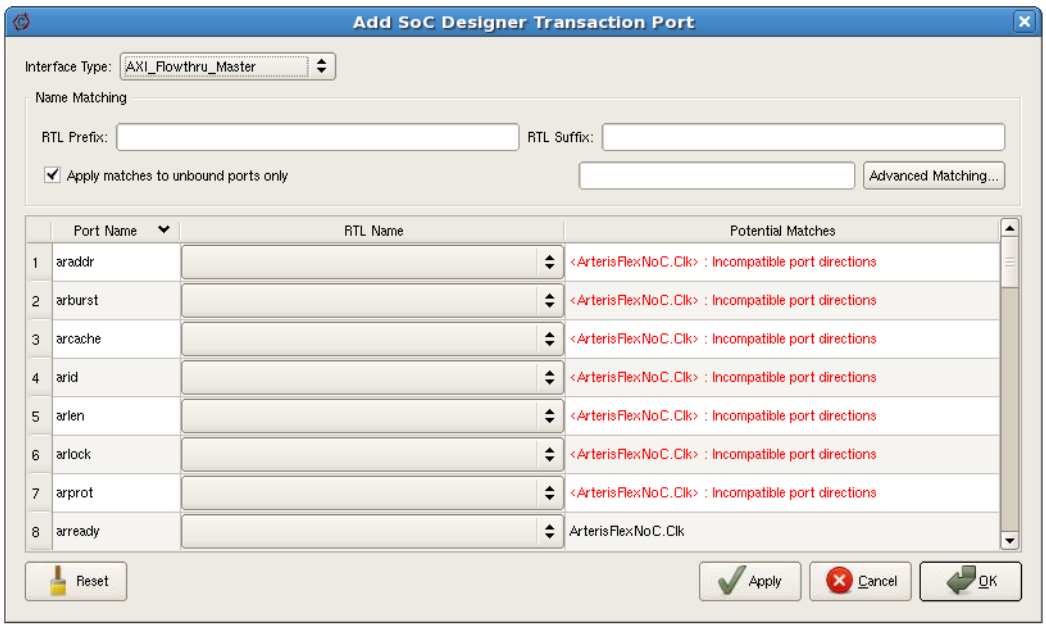


Figure 5-18 Selected Transactor in SoC Designer Transaction Port dialog

By default, Model Studio selects the most likely RTL ports to be mapped to the transactor or protocol ports in the *Potential Matches* column. It does this by matching the name of the port to the RTL port. Refer to the transactor's or protocol's user manual for pin names.

3. Use the following options to complete the mapping of RTL ports to transactor ports:
   - If your RTL pin names match the transactor/protocol pin names, and the ports are mapped correctly for your design, click **Apply**. The ports in the *Potential Matches* column move to the *RTL Name* column.
   - Use regular expressions to more efficiently match port names to RTL signals. Refer to "Using Regular Expression Name Matching" on page 106 for more details.
   - If the RTL ports adhere to the transactor or protocol naming convention, but have a common prefix or suffix appended to the name, use the *RTL Prefix* and *RTL Suffix* fields to identify the common prefix or suffix to use in the name matching for the port bindings.
   - Use the pulldown menu selections in the *RTL Name* column to select an RTL port for each transactor or protocol port.

*Note: Clicking the **Reset** button clears the list of RTL ports if you want to restart the port mapping.*

4. Click **OK**. The transactor or protocol is added to the *Component Ports* section of the tree (Figure 5-19).

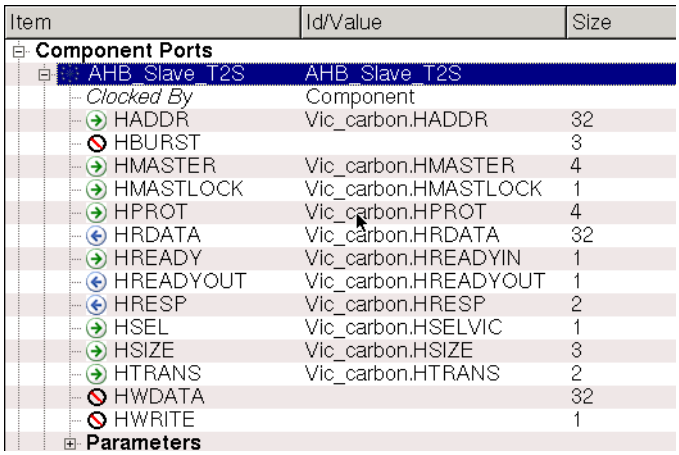


Figure 5-19 Transactor added to Component Ports section

The transactor or protocol added to the *Component Ports* section includes its name, how it is clocked, the list of ports and their mapping to RTL ports, and the available transactor or protocol parameters (with default settings).

5. To change the name of the transactor or protocol, click on its name and type a new name.
6. If you need to change the transactor or protocol port mapping, double-click the top-level name and the *Edit Connections* dialog appears. From there you can change any port mappings.

Note: If you need to create and use your own transactor, see Appendix B.

Using Regular Expression Name Matching

You can use regular expressions to more efficiently map transactor or protocol ports to RTL signals:

1. Click **Add Transactor** (**Add Protocol** on Platform Architect).
2. On the Add Port dialog, click **Advanced Matching**. The Regular Expression Name Matching dialog appears (Figure 5-20).

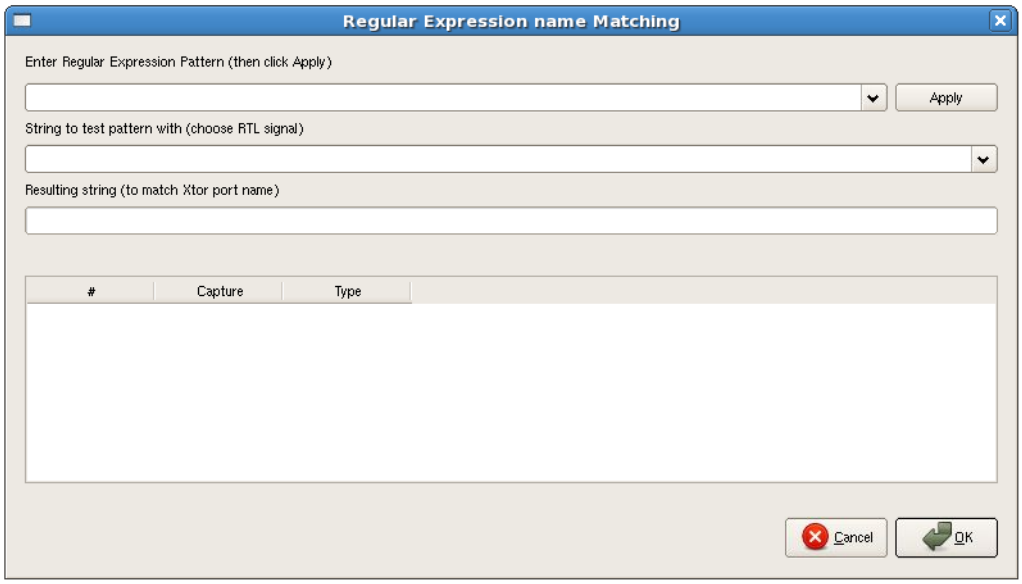


Figure 5-20 Regular Expression Name Matching dialog

3. Click the *String to test pattern with* pulldown to select one of the RTL signals you want to match (Figure 5-21). This is used to test and verify your regular expression output.
4. In the *Enter Regular Expression Pattern* field, specify the regular expression to use (Figure 5-21). For details about using regular expressions, refer to the QT® Developer Network QRegExp Class Reference.

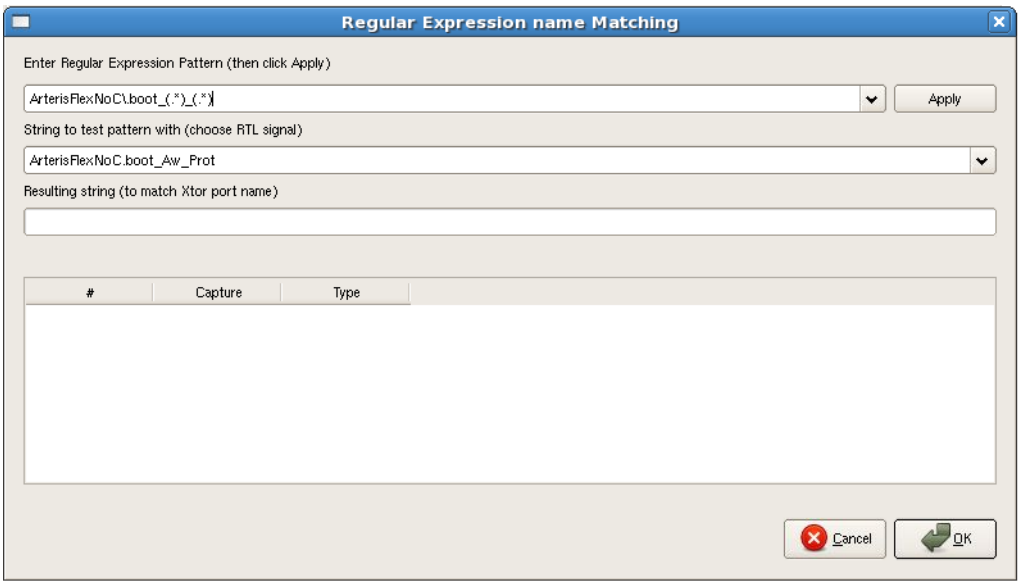


Figure 5-21 RTL signal and regular expression specified

5. Click **Apply**. SoC Designer Plus applies the regular expression pattern you specified to the test string, and displays the result in the *Resulting String* field (Figure 5-22). Modify the regular expression as needed and click **Apply** until the required string results.

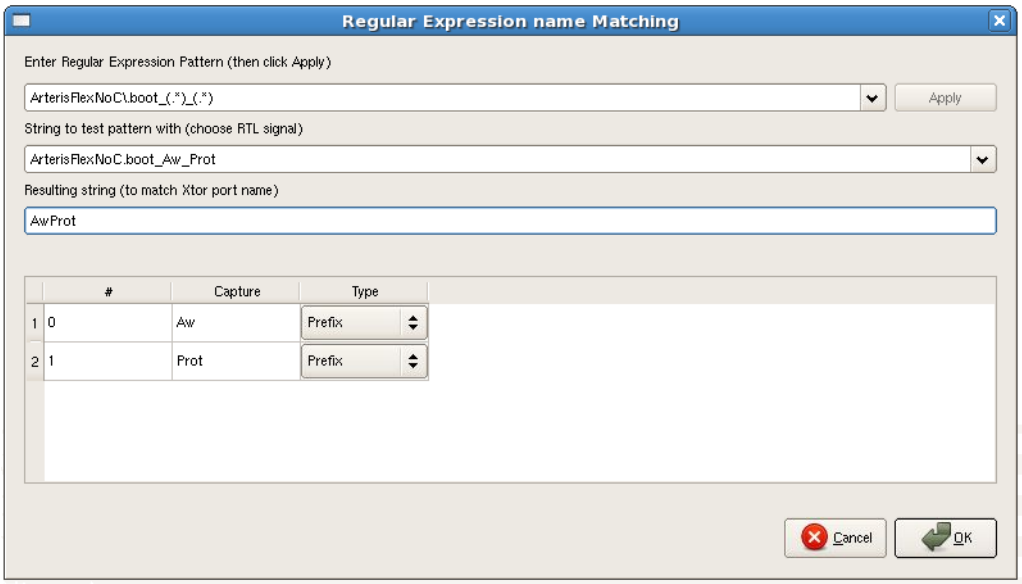
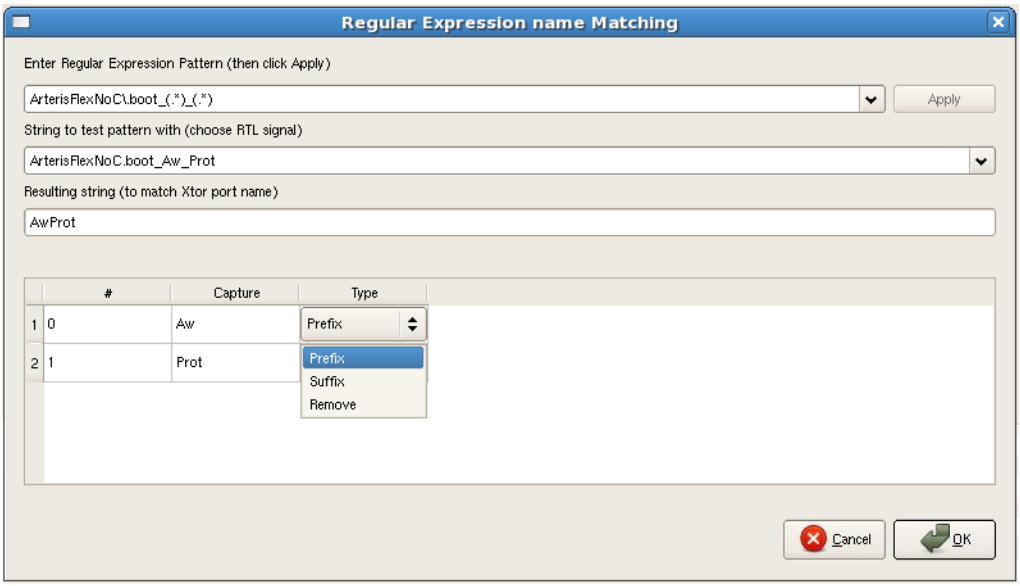


Figure 5-22 Resulting string

6. If necessary, assign Prefix, Suffix, or Remove to the captured name elements.



7. Click **OK**.

   The Regular Expression Name Matching dialog closes. SoC Designer Plus converts each RTL signal name based on your regular expression. The converted strings are used to find

potential matches for each transactor or protocol port, and displayed in the Add Port dialog (Figure 5-23).

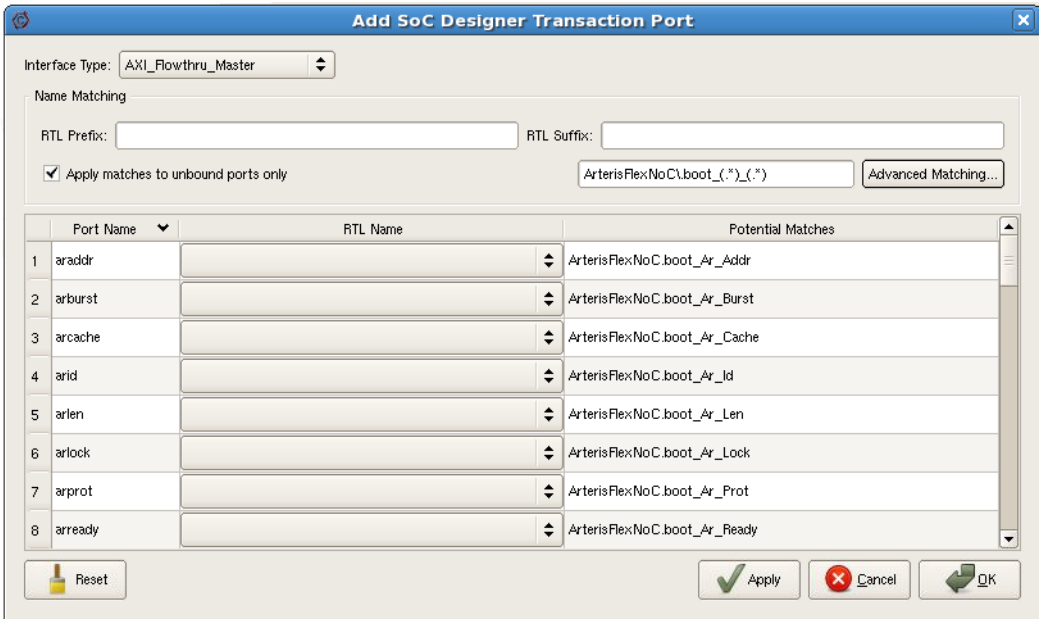


Figure 5-23 Matches resulting from regular expression

8. Click **Apply** to make the connections shown. The ports in the Potential Matches column move to the RTL Name column.

If a transactor's signals use multiple naming conventions, click **Advanced Matching** again to repeat the process. Ensure that *Apply matches to unbound ports only* is enabled.

Editing Transactor Parameters

Note: This funcitonality is not available for Platform Architect.

Most transactors have parameters associated with them. These parameters appears as component parameters in the SoC Designer Plus environment. To edit transactor parameters:

1. Under the transactor, click the plus sign next to *Parameters* to display the full list of parameters. Then click the plus sign next to the selected parameter to display the Value field (Figure 5-24).

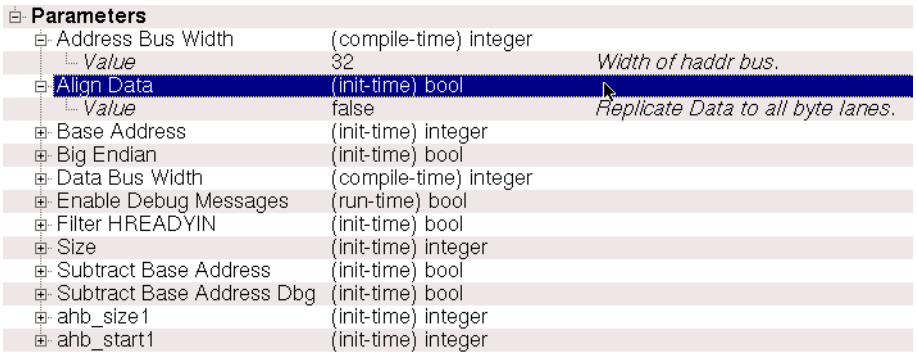


Figure 5-24 SoC Designer Plus Component Transactor Parameters

2. In the *Value* field, enter the new parameter value, or select the value from a drop-down menu (Figure 5-25).

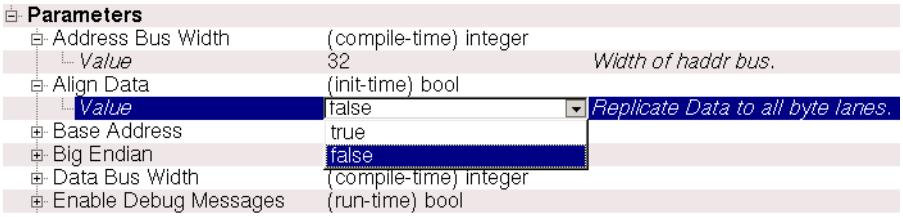


Figure 5-25 SoC Designer Plus Component Transactor Parameter Values

The default value for a parameter is shown in regular text. Any value which has been overridden is shown in **bold** text.

3. You can right-click a parameter and select **Connect to RTL** if you want to tie an RTL input port to a transactor parameter. This removes the corresponding ESL port from the component and moves the RTL port under the transactor parameter. When in SoC Designer Plus, setting the parameter value changes the value assigned to the RTL port.

Adding Unused (Null) Ports

Note: This funcitonality is not available for Platform Architect.

In the event SoC Designer Plus requires more ports than are used by your design, you can add unused ports. Unused ports comprise *Null Inputs* and *Null Outputs* and show up as ports in SoC Designer Plus, but they do not affect the operation of the generated component.

To create *Null ports*:

1. In the *Ports Toolbar*, click the **Add Transactor** button. The *Add SoC Designer Transaction Port* dialog appears.
2. Select *Null_Input* or *Null_Output* from the *Transactor Type* pull-down menu.
3. Click **OK** and the transaction port is added to the *Component Ports* list.

Transactor Clocking

Note: This funcitonality is not available for Platform Architect.

You can run a transactor at a different speed than the ESL reference clock by associating it with a Clock_Input or Clock_Generator pseudo-transactor. Follow the steps below:

1. Create and configure a *Clock_Input* or *Clock_Generator* transactor (for specific instructions see “Specifying Generated Clocks” on page 100).
2. Select the transactor in the ESL Ports list; for example, the AHB_Slave_T2S.

3. In the *Clocked By* field, select the *clock* pseudo-transactor that has been defined with the different clock settings (Figure 5-26).

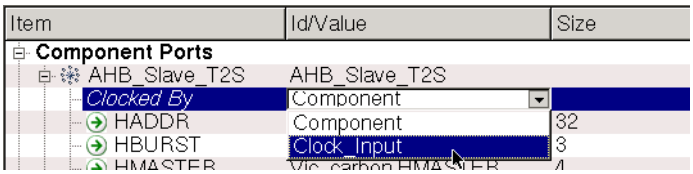


Figure 5-26 Selecting the Clock to Use for a Transactor

Adding Parameters to your Component

Note: This funcitonality is not available for Platform Architect.

You can add parameters to your component that appear as object parameters in SoC Designer Plus. You can create init-time parameters and run-time parameters. *Init-Time* parameters can be changed in the SoC Designer Canvas only. *Run-Time* parameters can be changed in the SoC Designer Canvas and at runtime during simulation.

To add a parameter in the *Component Parameters* list, follow the steps below:

1. Right-click the *Component Parameters* heading and select the **Add Parameter** option from the context menu. You can also click the **Add Parameter** toolbar button. The *Add New Parameter* dialog box appears (Figure 5-27).

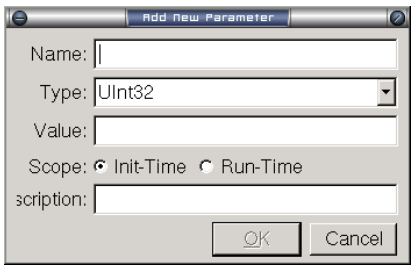


Figure 5-27 Add Parameter Dialog Box

2. Define the new parameter as described below:
   - Enter a *Name* for the parameter.
   - Select the *Type* for the parameter: *UInt32* or *Bool*
   - Enter the default *Value* for the parameter.
   - Select the *Scope* of the parameter: *Init-Time* or *Run-Time*.
   - Enter a *Description* for the parameter.
3. Click **OK** and the new parameter is added to the *Component Parameters* list (Figure 5-28).

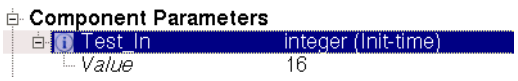


Figure 5-28 New Parameter

Note: The new parameters you add can also be used in User Code. See "Example of adding Debug Bypass Code to a source file" *on page 133 for more information.*

Connecting Parameters to Ports

Note: This funcitonality is not available for Platform Architect.

Component parameters can be tied to an RTL input port or to an ESL port. When tied to an:

- RTL port — The ESL port to which it was connected is removed from the component visibility, and the value can be changed only in Component Parameters in SoC Designer Plus.
- ESL port — The value can be changed in Component Parameters in SoC Designer Plus, and it can be changed by connecting the ESL port. In this case, the parameter value is used whenever the ESL port is not connected. *If the port is connected, the connection value takes precedence over the parameter value.*

Note: See "Editing Transactor Parameters" on page 108 for information about connecting transactor parameters to RTL input ports.

To connect a parameter to an RTL input port:

1. Right-click the parameter name and select **Connect to RTL** from the context menu (Figure 5-29).

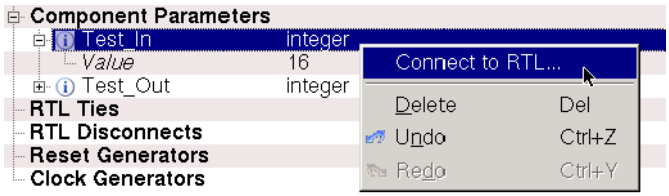


Figure 5-29 SoC Designer Plus Component Parameters

This displays a list of the available RTL input ports to which the parameter can be connected in the *Choose RTL Signal* dialog (Figure 5-30).

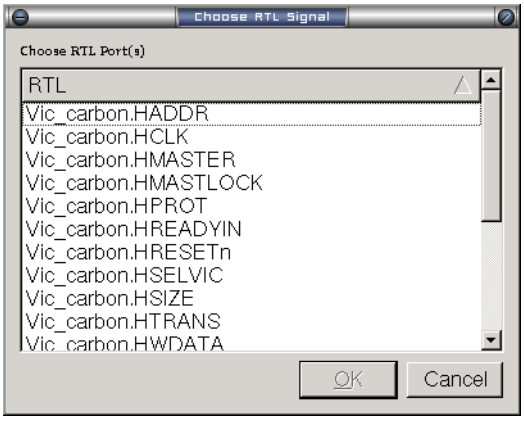


Figure 5-30 Choose RTL Signal Dialog Box

2. Select the signal from the list and click **OK**, and the RTL port is moved from the *Component Ports* section to the *Component Parameters* section (Figure 5-31).

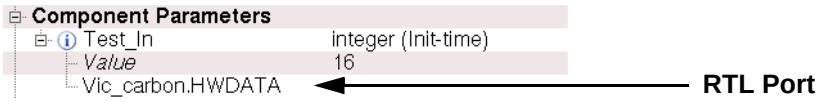


Figure 5-31 SoC Designer Plus Component Parameter Mapped to RTL Port

To connect a parameter to an ESL port:

1. Right-click the ESL port name and select **Connect to Parameter** from the context menu, and then select the existing parameter (Figure 5-32).

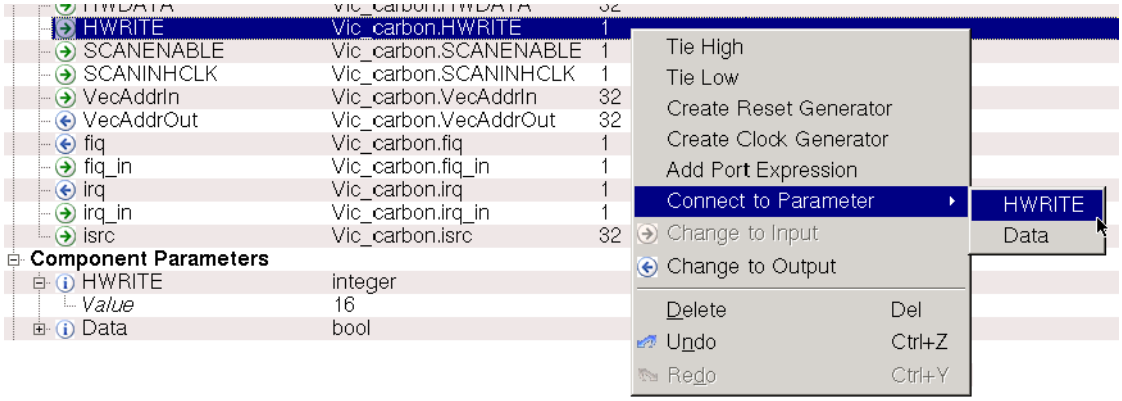


Figure 5-32 Connecting ESL Port to SoC Designer Plus Component Parameter

2. The parameter appears under the name of the ESL port (Figure 5-33).

Figure 5-33 SoC Designer Plus Component Parameter Mapped to ESL Port

Note: The parameter name does not need to be the same as the port name, but it can be useful for the user in SoC Designer Plus to know that both methods can be used to affect the port value.

Using C-Expressions to Modify Port Connections

By default, the signal applied to the ESL port or transactor port in SoC Designer Plus is transmitted unchanged to the Cycle Model's RTL port. You can use C-expressions to create a more complex connection between the ESL and RTL ports as follows:

1. Highlight an ESL input or output, or a transactor input or output, in the *Component Ports* list.
2. Right-click and select **Add Port Expression** from the context menu. The *Port Expression* field appears below the port (Figure 5-34).

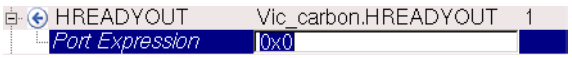


Figure 5-34 Adding a Port Expression

3. Type any valid C-expression that references the port by its ESL name in the *Port Expression* text input field.

If the connection is an RTL output and ESL output, the port expression modifies the RTL output data before writing it to the ESL port. For the other direction, the port expression modifies the ESL input value before writing it to the RTL input port.

Examples of Port Expressions

Port expressions provide a mechanism to transform a signal port value between the ESL environment and the Cycle Model. The transform is expressed as a C-expression based on the ESL port name.

Note: Currently, port expressions can only be used on ports that are 64-bits wide or smaller.

- To invert the sense of a signal named *interrupt*, type in a port expression:

  ```
  ! interrupt
  ```

- To right-shift an address by two, type:

  ```
  address >> 2
  ```

- To shift the data by a constant amount; e.g. to shift the data right by two bits before transferring it:

  ```
  my_esl_port >> 2
  ```

- To tie off the transactor or port:

  ```
  1
  ```

- To change encoding from packed to one-hot:

  ```
  1 << my_esl_port
  ```

- To change encoding from one-hot to packed, call a function in the C-expression:

  ```
  log2(my_esl_port)
  ```

The name used in the port-expression text must match the ESL port name.

Apply port expressions to signal inputs or outputs. When used on an:

- Input — The transformation is made after reading the signal value from the ESL environment and depositing it in the Cycle Model.
- Output — The transformation is made after reading the signal value from the Cycle Model and driving it into the ESL environment.

Using Functions in Port Expressions

To use a function in a C-expression, add the code for the function to the *User Code* section of the component integration output files. An example user code section for the port expression `log2(my_esl_port)` is shown here:

```
// CARBON USER CODE [POST INCLUDE] BEGIN
MxU32 log2(MxU32 value)
{
    MxU32 result = 0;
    while (value != 0)
    {
        ++result;
        value >>= 1;
    }
    return result;
}
// CARBON USER CODE END
```

Error Checking

The component generator does limited error checking on a C-expression. If any invalid C syntax or invalid identifiers are detected, these are labeled as “invalid signals” in the ESL port list. However, signals labeled as invalid can still be saved.

It is legal to call an external function in the C-expression; however, the Carbon Model Studio labels the function as an “invalid signal.” As long as a valid path to the function is included in the C-expression, the expression is valid and you can ignore the warning.

5.2.2.2 Registers Tab

The *Registers* tab, shown in Figure 5-35, manages a programmer's view of logical registers inside the RTL. These registers are accessible from the component through the CADI debug interface of SoC Designer or Platform Architect.

For Platform Architect, the available columns are: Offset, Name, Width, Endian, Description, and Fields.

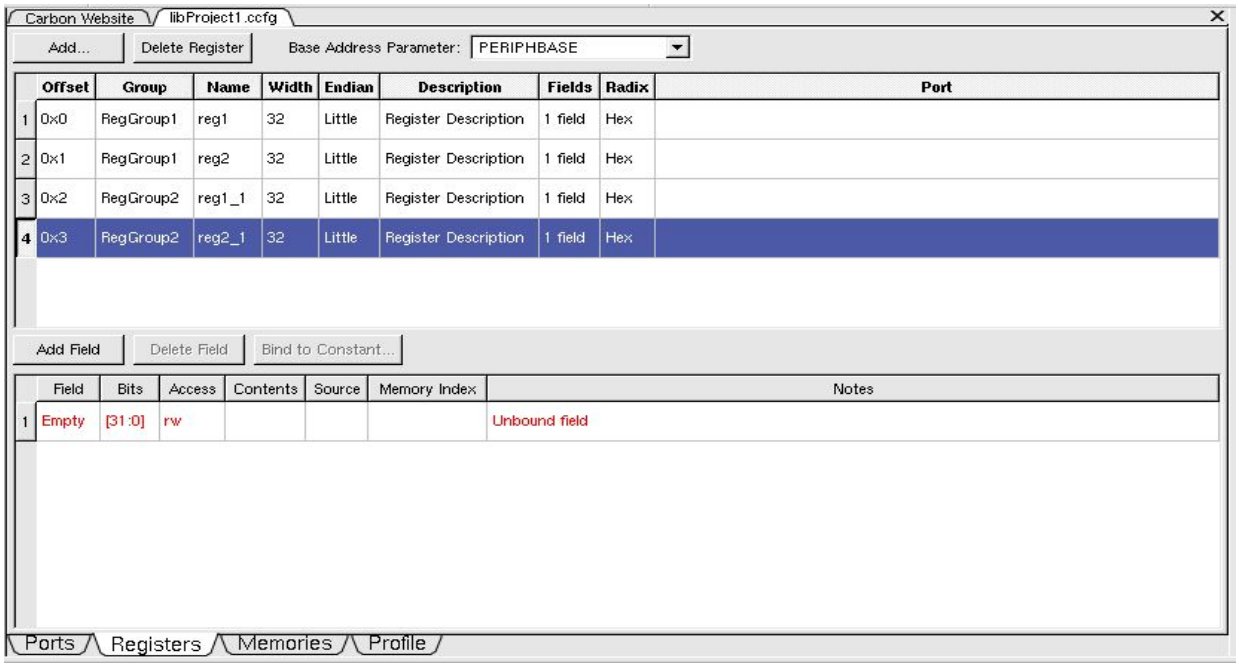


Figure 5-35 SoC Designer Plus Component Registers Tab

The mapping of logical registers to RTL is controlled by Carbon Model Studio. Each logical register is divided into one or more fields spanning a subrange of the register's width. Each of these fields can be mapped to one of the following:

- A vector or scalar register in the design hierarchy, or a bitselect/partselect of that register
- A constant index into a 2-D array of registers in the design hierarchy, or a bitselect/partselect of that indexed element
- A constant literal

This section discusses:

- Registers Tab Composition
- Adding a Register
- Setting the Base Address Parameter
- Configuring Register Fields
- Binding a Field to a Constant Value
- Binding an RTL Signal to a Register

Registers Tab Composition

The Registers tab comprises three panes:

- **Register Pane** — Displays a list of the logical registers and their attributes. You can create any number of registers.
- **Field Pane** — Displays a list of the current register fields and their attributes. Any number of fields can be added, provided their ranges are non-overlapping.
- **Design Hierarchy Pane** — The lower left subpane displays the module hierarchy of the current design. When a module instance is selected, the right subpane displays all the signals within that instance. The size of each signal is displayed, along with additional properties indicating whether the net is visible and/or depositable.

  A net needs to be visible to be used in a component register. Nets displayed in red are not visible. Nets displayed in blue and labeled Visible (sampled) are visible, but are read in an inefficient manner. To fix each of these issues, use the *observeSignal* directive when compiling the design with Carbon compiler. Similarly, only signals that are labeled Depositable can be written with new values through the component register interface. The *depositSignal* compiler directive can be used to ensure this.

Adding a Register

To add a new logical register:

1. On the Register tab, click **Add Register**. A new row is added to the Register pane.
2. Double-click the appropriate cell of the table to edit the register's attributes. For SoC Designer Plus, this includes the Register *Name* and *Group*. These fields are used by SoC Designer Plus for grouping and displaying the register.
3. Additionally, you can access registers defined for the component through the debug ports of a protocol slave transactor. To gain access to the register, specify an Offset and a Port (SoC Designer Plus only) to use to access the register definition.

Setting the Base Address Parameter

Note: This funcitonality is not available for Platform Architect.

The base address parameter is used by SoC Designer Plus to determine the base address when loading virtual registers in the Fast Model dialog. This needs to be specified if you are planning to use Swap & Play with a custom interconnect component.

By default, SoC Designer Plus uses the address specified by the PERIPHBASE parameter. If you plan to use an address other than PERIPHBASE for the register's base address, specify the address in the Base Address Parameter field.

To set the base address parameter, select the appropriate parameter from the Base Address Parameter menu. This menu is populated with component parameters that have been defined as integers (see "Adding Parameters to your Component" on page 110).

Alternatively, you may enter the base address as a hexidecimal value.

Configuring Register Fields

To configure register fields:

1. Click to select a register. The *Field* pane displays the current list of fields. Initially, each register has a single field spanning its entire width.
2. In the *Bits* column, click to edit the range of a field within the logical register.
3. In the *Access* column, click to edit the field's read/write access.
4. Click the **Add Field** button to add a new field (provided there are bits in the register not covered by an existing field).

Binding a Field to a Constant Value

To bind a field to a constant value:

1. Select the field and click **Bind to Constant**.

   The Bind Constant dialog appears (Figure 5-36).

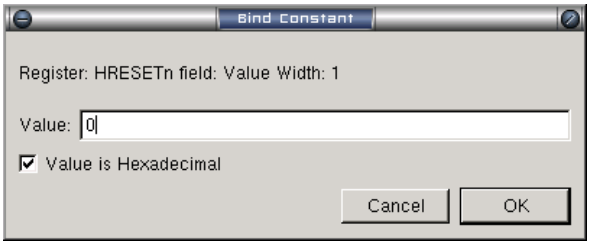


Figure 5-36 Bind to Constant Dialog Box

2. In the *Value* field, enter the hex value and click the **OK** button.

Binding an RTL Signal to a Register

You can bind signals to registers and edit their parameters as needed.

To bind a register field to an RTL signal:

1. Drag a signal from the *Design Hierarchy* view to the Register pane (Figure 5-37).

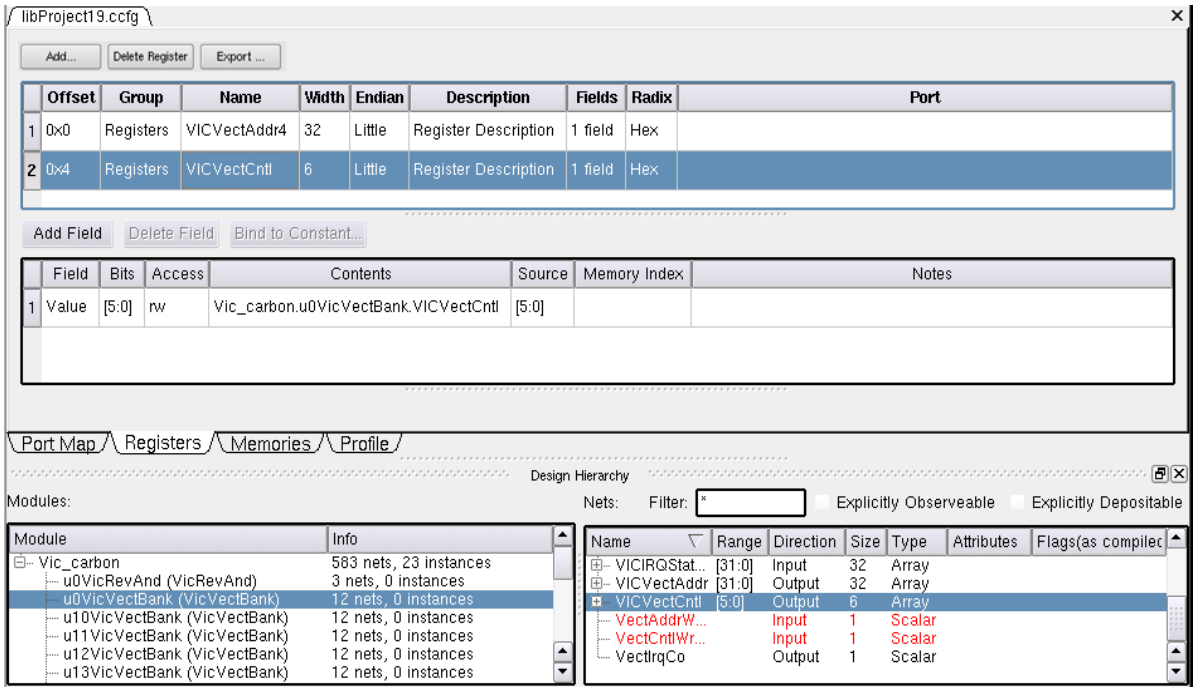


Figure 5-37 Drag Signal to Add a Register

   The signal is added to the top of the Register pane, and the field within the signal is added to the bottom of the Register pane.

2. Click any cell to edit that parameter.
3. If the signal is a 2-D array, specify the Memory Index to use for the field's value.
4. For both arrays and vectors, edit the *Source* column to specify a *bitselect/partselect* of the signal's value, for example [31:0]. The *Notes* column indicates an unbound field or conflicting width value between the bound RTL entity and the field.

5.2.2.3 Memories Tab

The *Memories* tab, shown in Figure 5-38, allows you to define debug visibility into RTL memories during system simulation. Use this tab to make memories accessible during validation or to specify a file to load into the memory during validation.

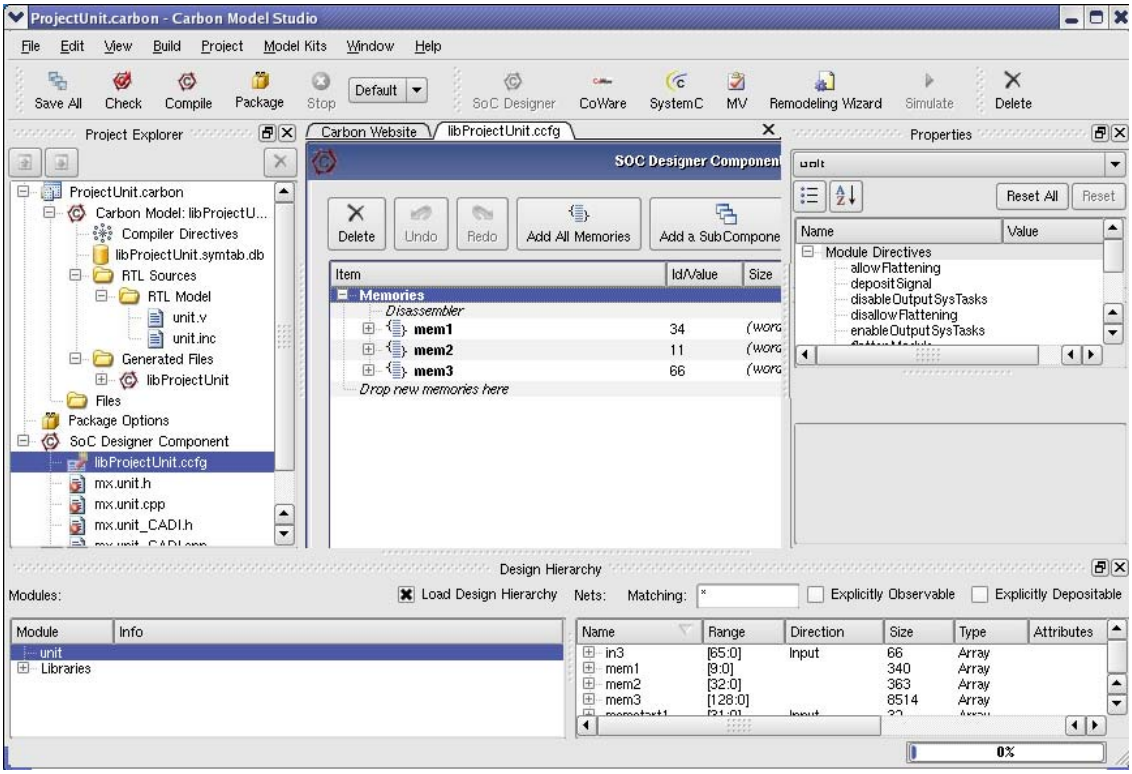


Figure 5-38 SoC Designer Plus Component Memories Tab

The following sections discuss these tasks:

- Disabling Automatic Memory Checking
- Adding a Memory
- Working with Non-Byte-Aligned Memories
- Deleting a Memory
- Modifying Memory Parameters
- Changing the Memory's Max Address
- Specifying System Address Mapping for Slave Transactors
- Creating Custom Code Sections
- Creating Blocks within Memories
- Specifying the Disassembler
- Creating Disassembly Views
- Creating Subcomponent Views

Disabling Automatic Memory Checking

By default, when you add memories or make changes using the Memories tab, Carbon Model Studio performs validation and checks on your choices. Normally you do not notice any performance lag unless you are working with many memories or the memories are large.

If you notice a performance lag or if you are making many changes and prefer to wait for checking to occur until after you have made the changes, you can temporarily disable the checking by deselecting the Automatically Check Memories option at the top of the Memory tab shown in Figure 5-39.

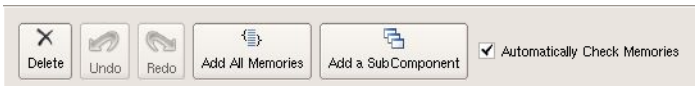


Figure 5-39 Automatically Check Memories Option

After you have completed your changes, re-enable automatic memory checking to make sure that no problems have been introduced by your changes.

Adding a Memory

To add a memory:

1. In the *Design Hierarchy* pane, select a memory instance. (If the *Design Hierarchy* pane is not visible, choose View > Design Hierarchy from the main menu. Make sure that the top module is selected in the *Module* pane.)
2. Drag the instance to the *Memories* window, as shown in Figure 5-40.

   To specify the memories to be observed during simulation individually, or all at once:

   – **Individually.** Highlight individual memories in the *Nets* list of the *Design Hierarchy* pane. Highlight one memory at a time and drag it to the *Memories* box in the main *Memories* tab window, dropping it where you want it to appear in the list (see Figure 5-40).

   — or —

   – **All at once.** Click the **Add All Memories** button. This button searches through the RTL design and adds all the memory instances to the *Memories* pane.

Note: The memory must be **Visible** *to be added to the list. It cannot be shown in red color, which means 'not visible'.*

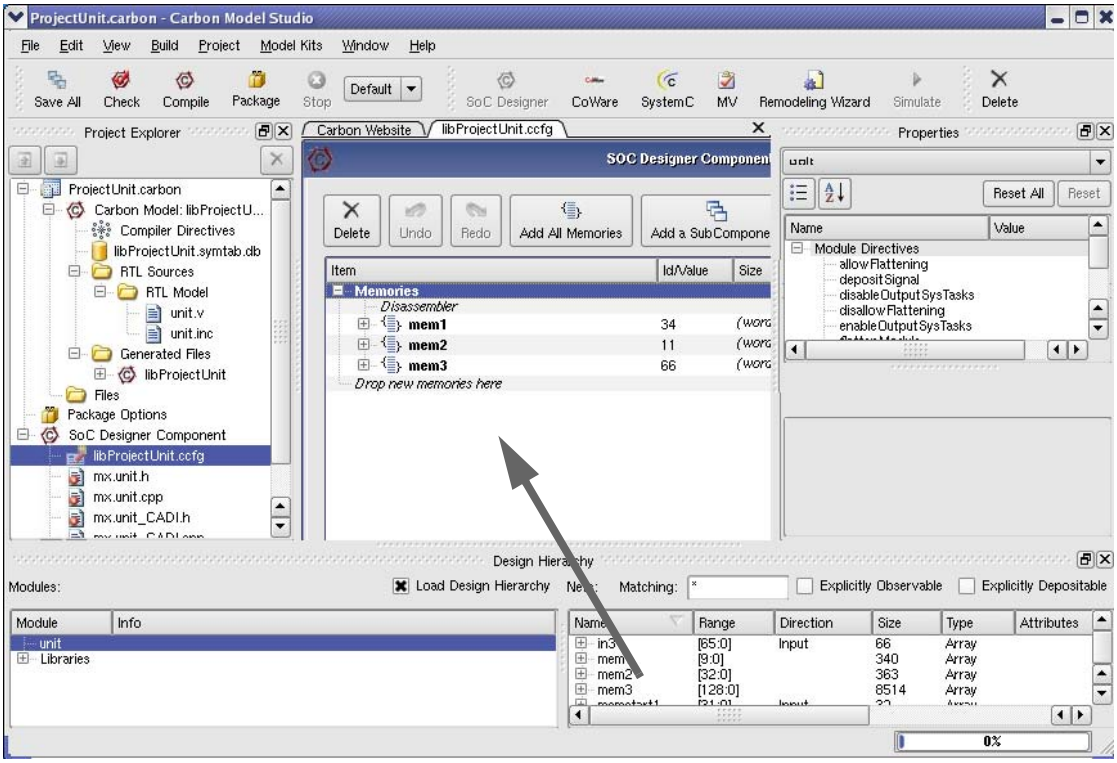


Figure 5-40 Drag Memory to Memories Page

Working with Non-Byte-Aligned Memories

Note: This section applies to SoC Designer Plus components only.

SoC Designer Plus component memories must be represented in byte-aligned Minimum Addressable Unit (MAU) format. MAU is the minimum size of data (in bits) that can be accessed.

This section describes how to display a Cycle model that has a non-byte-aligned memory width (for example, 11 bits) as SoC Designer Plus component memory.

To support non-byte-aligned Cycle model memories as SoC Designer Plus component memories, the SoC Designer Plus Wrapper must pad the memory word size to a byte-aligned size.

To pad the SoC Designer Plus component view of the memory:

1. In Carbon Model Studio, access the Memories tab for the SoC Designer Plus wrapper.
2. Increase the "word size in bits" value to a byte-aligned value by clicking on the value in the Id/Value column and entering the new value (see Figure 5-41).

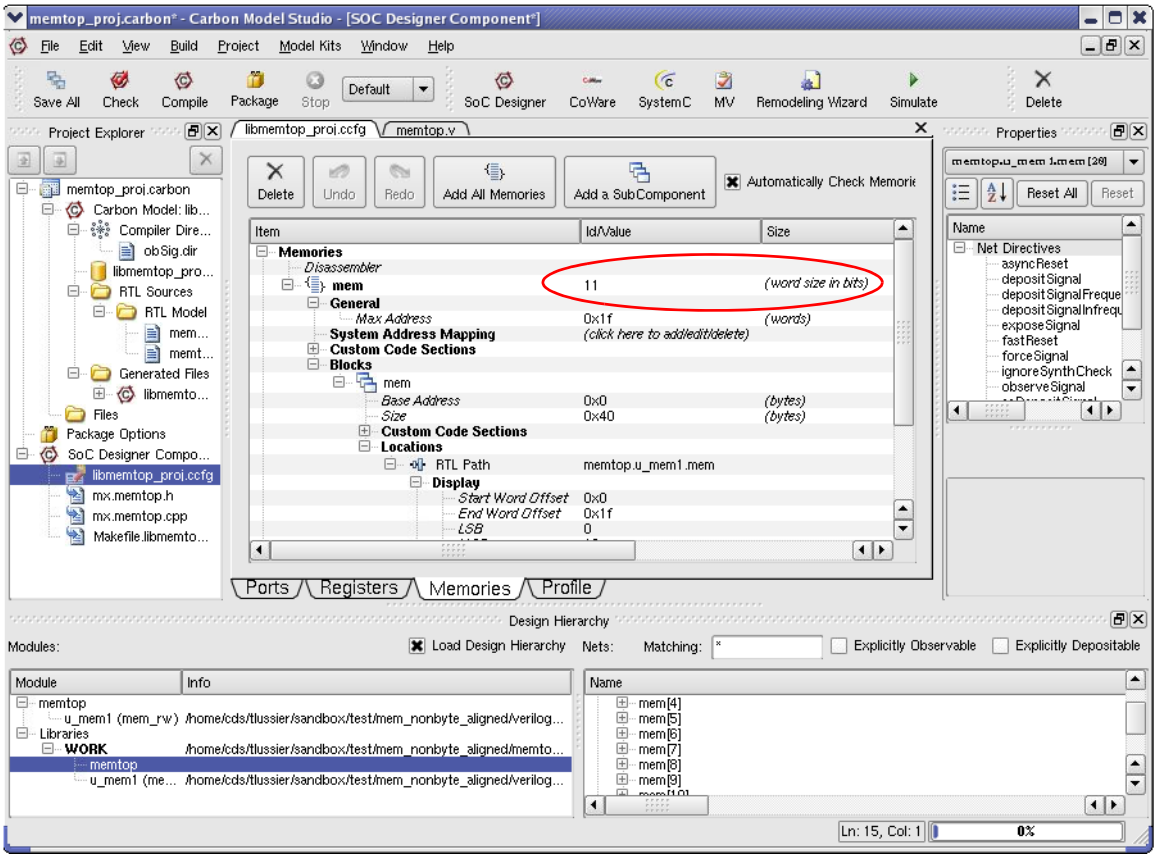


Figure 5-41 Changing the Word Size in Bits value

For example, if your memory has an 11-bit data width, increase the "word size in bits" from 11 to 16.

Note: Padding the SoC Designer Plus component view of the memory does not impact the Cycle model's memory functionality or actual width. It impacts only the SoC Designer Plus component memory view, including SoC Designer Plus read and write debug accesses to the memory. That is, when you view the component's memory in SoC Designer Plus, the padded values are shown.

If you load the memory using the MxScript `CADIMemLoadFromFile()` function or the sdsim GUI option Write memory image, you must use byte-aligned padded values in the file being loaded to the memory.

If the SoC Designer Plus wrapper does not properly pad a Cycle model memory being displayed as SoC Designer Plus component memory, sdsim displays a warning similar to the following when viewing the memory:

```
WARNING :: sdsim does not support non-byte aligned MAU (sys_memtop.mem-
top:mem, bits per MAU=11)
```

Deleting a Memory

To remove a memory from the list, highlight the memory and click the **Delete** button.

If you have clicked the **Delete** button by mistake, click **Undo** to add the memory back.

Modifying Memory Parameters

You can change a memory's parameter by selecting it (click underneath the *id/Value* column) and changing the value there. Figure 5-42 shows the parameters that are currently available.

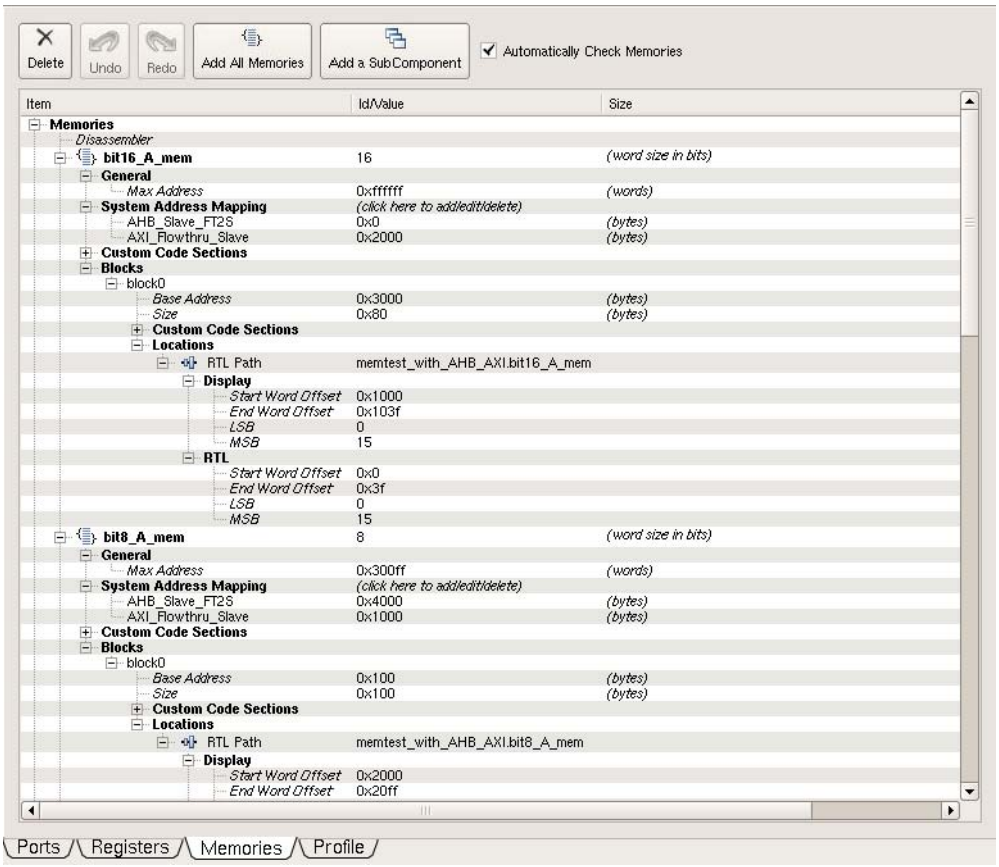


Figure 5-42 Memories Page

If a parameter appears dimmed (grey), it indicates that the parameter is read-only. Whether a parameter is editable or not depends on the memory; a parameter that is editable in one memory might not be editable in another.

Changing the Memory's Max Address

Each memory is assigned a maximum address by default. The **Max Address** is the base address of the component and the address offset into the component. If you have more than one memory to preload in a component, the base addresses of the memories cannot overlap. In other words, the base address and the max address of the memory cannot overlap with another memory for the selected slave transactor.

Currently, only 32-bit wide memories are supported. The memories are assumed to be byte-addressable. Therefore, when loading the component, the address passed into the load function

is shifted by 2 bits to calculate the address for the memory to be loaded. The 2 bits are used to select the byte in the word of the memory.

To change or view the **Max Address**, expand the **General** heading.

Specifying System Address Mapping for Slave Transactors

Note: This functionality is not supported for Platform Architect.

You can instruct the program that is loaded into the SoC Explorer environment to initialize the memory through one or more slave transactors. In previous versions of Carbon Model Studio, you were allowed to specify a single slave T2S (Transactor-to-signal) transactor. You are no longer restricted to either a single slave transactor nor the T2S type, but the transactor must be a slave transactor.

To specify **System Address Mapping**, expand the **System Address Mapping** heading.

Creating Custom Code Sections

When you compile the component for SoC Designer Plus, Carbon Model Studio generates a C++ header file and source file that contain the generated code that represents the implementation. You can add your own C++ code using the *Class*, *Constructor*, *Destructor*, *Read*, and *Write* sections, and include pre-condition and post-condition processing. For example, you might add custom code that specifies how to initialize a memory component during initialization.

To add custom code, expand the Custom Code Sections heading, as shown in Figure 5-43.

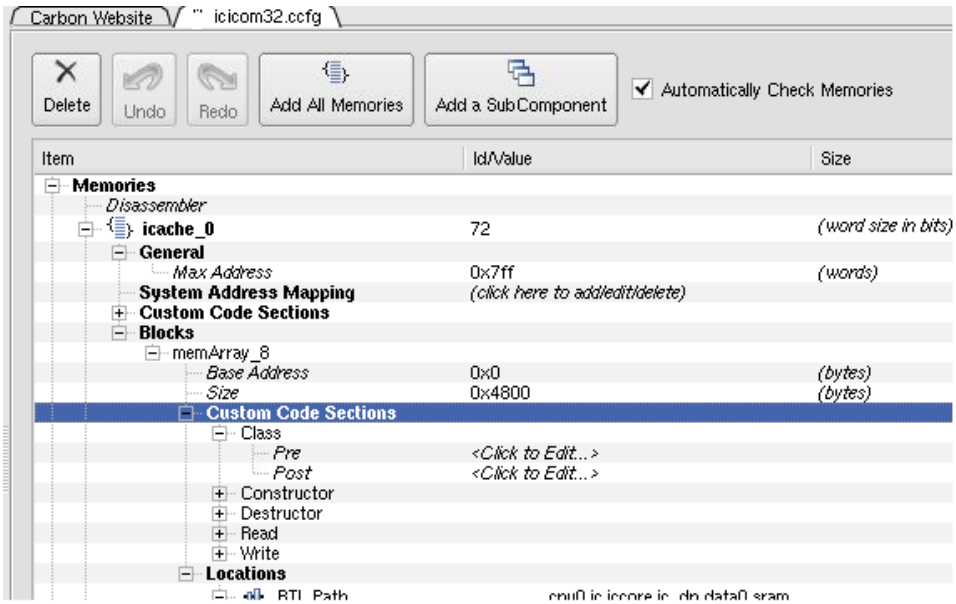


Figure 5-43 Adding Custom Code

Creating Blocks within Memories

Blocks allow you to group memory units together.

To create a new block:

1. Right-click **Blocks** and choose **Add Block**.
2. Edit the following fields as required.
   - **Base Address** — This allows you to specify the base address of the component and the address offset into the component. If you have more than one memory to preload in a component, the base addresses of the memories cannot overlap. In other words, the base address and the max address of the memory cannot overlap with another memory for the selected slave transactor.
   - **Size** — Currently, only 32-bit wide memories are supported. The memories are assumed to be byte-addressable. Therefore, when loading the component, the address passed into the load function is shifted by 2 bits to calculate the address for the memory to be loaded. The 2 bits are used to select the byte in the word of the memory.
   - **Custom Code Sections** (see "Creating Custom Code Sections" on page 124 for more information).

The **Locations** section displays the following information under **RTL Path (**the path to the RTL code from the top-level module):

- **Display** — *Start Word Offset*, *End Word Offset*, Least Significant Bit (*LSB*), and Most Significant Bit (*MSB*).
- **RTL** — *Start Word Offset*, *End Word Offset*, Least Significant Bit (*LSB*), and Most Significant Bit (*MSB*).

Specifying the Disassembler

Note: This functionality is not supported for Platform Architect.

To specify the name of the disassembler to use to display the disassembly view in SoC Designer Plus:

1. Click *Disassembler.*

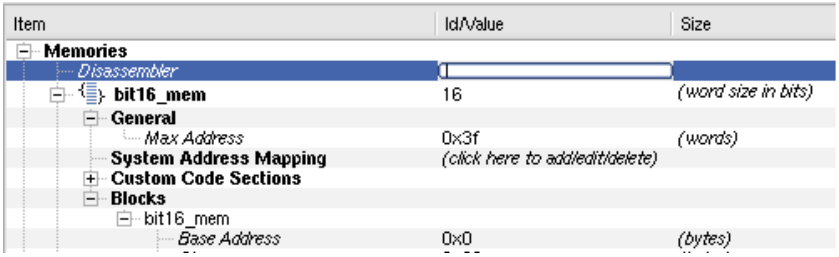


Figure 5-44 Specifying the Disassembler

When a box appears in the *Id/Value* column (Figure 5-44), enter the name of the class to be used to disassemble the memory. The source code that contains this class must be specified as part of the project in the *SoC Designer Component Properties* page (see "Setting Compiler and Linker Flags" on page 95).

Note: All the memories in a subcomponent must use the same disassembler. (See "Creating Subcomponent Views" on page 126 for information on subcomponents.)

Creating Disassembly Views

Previous versions of Model Studio referred to creating “disassembly views.” This terminology has changed. The view is now called a *subcomponent view,* as described in the following section (Creating Subcomponent Views).

Creating Subcomponent Views

Note: This functionality is not supported for Platform Architect.

If you have multiple memories within your Cycle Model, you can group them into subcomponents. You can then use the subcomponent as a single debug interface for the group of memories.

To create a subcomponent view:

1. Click the **Add a Subcomponent** button. A subcomponent row is added to the *Memories* list as shown in Figure 5-45.

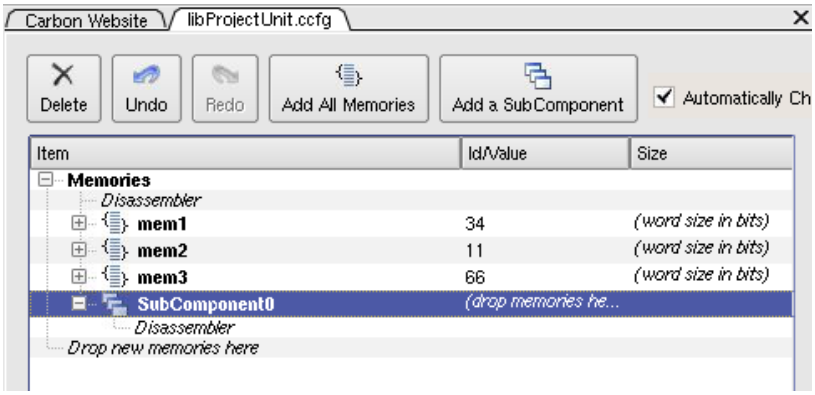


Figure 5-45 Subcomponent Views in Memories Page

2. Drag memories into the area labeled **drop new memories here**.
3. Define a disassembler for the subcomponent. (See “Specifying the Disassembler” on page 125 for instructions.)

Note: All of the memories in the subcomponent must use the same disassembler.

5.2.2.4 Profile Tab

Note: This tab is available only for SoC Designer Plus components.

Use the *Profile* tab, shown in Figure 5-46, to facilitate the measuring of events internal to a Cycle Model, and display them graphically with SoC Designer Plus. The profiling mechanism is organized into *streams*, *channels*, and *buckets*. You control when and how the output displays during SoC Designer Plus profiling by defining *control statements* for the streams' triggers, channels, and buckets.

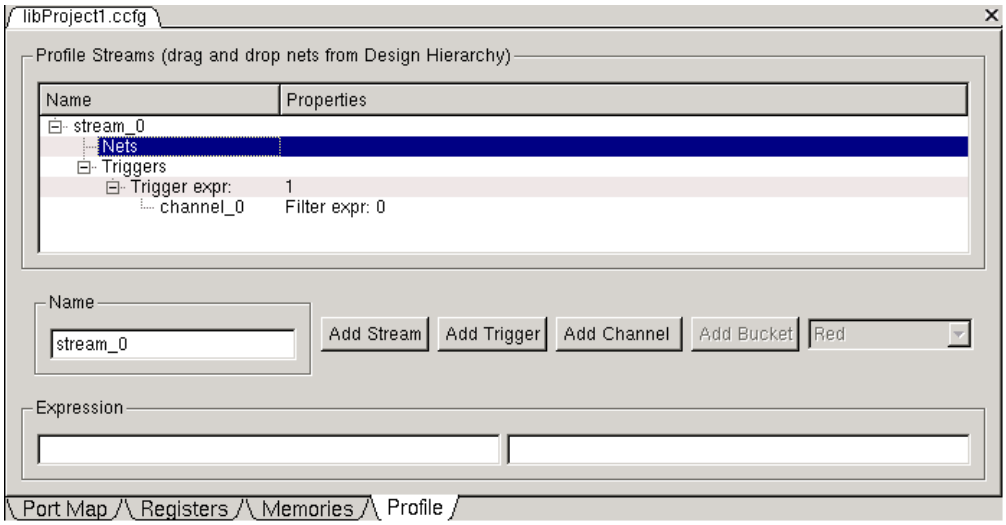


Figure 5-46 SoC Designer Plus Component Profile Tab

Streams, Triggers, Channels, and Buckets

A *stream* is a collection of nets that form a logical entity whose values are measured and displayed during simulation. Streams can have multiple triggers that control when their channels are activated at simulation time in the SoC Designer Plus environment. Streams consist of one or more channels and can view one or more RTL nets.

A *trigger* is a boolean C expression that determines when SoC Designer Plus records data values for each of the stream's channels.

A *channel* represents a single simulation event value that you wish to measure. When its trigger is activated in SoC Designer Plus, the channel's data is displayed graphically. To display the data as:

- Integers — Set up a channel controlled by a filter (C) expression. The filter expressions generally return the value of a specified net, but can also return the value of a computation involving more than one net.
- A set of discrete symbolic values in a color-coded bar graph — Set up a channel with *buckets*. Each bucket has a boolean C expression to determine which of the channel's buckets are active at any given time. SoC Designer Plus checks buckets in sequence; the first bucket expression to return a non-zero value results in a profiling event.

Expressions control the output of a stream, channel, or bucket:

- Trigger expressions are boolean C expressions that trigger a stream to record data values for its channel; for example: `posedge (clk)`.
- Integer channel filter expressions can be C expressions that return an integer; for example: `HSIZE`.
- Bucket channel expressions are boolean C expressions; for example:
  `VectAddr > 8` (green)
  `VectAddr > 16` (blue)

Creating Streams

To create a stream:

1. Click **Add Stream** to add a stream to the profile list.
2. Edit the stream's name in the *Name* field.
3. Define the nets belonging to a stream:
   – Open the Design Hierarchy view and select a module from the *Modules list*.
   – Select nets, one at a time, from the *Nets* list, and keeping the left mouse button depressed, drag-and-drop the nets from the Design Hierarchy to the *Nets* section of the desired stream (see Figure 5-47).

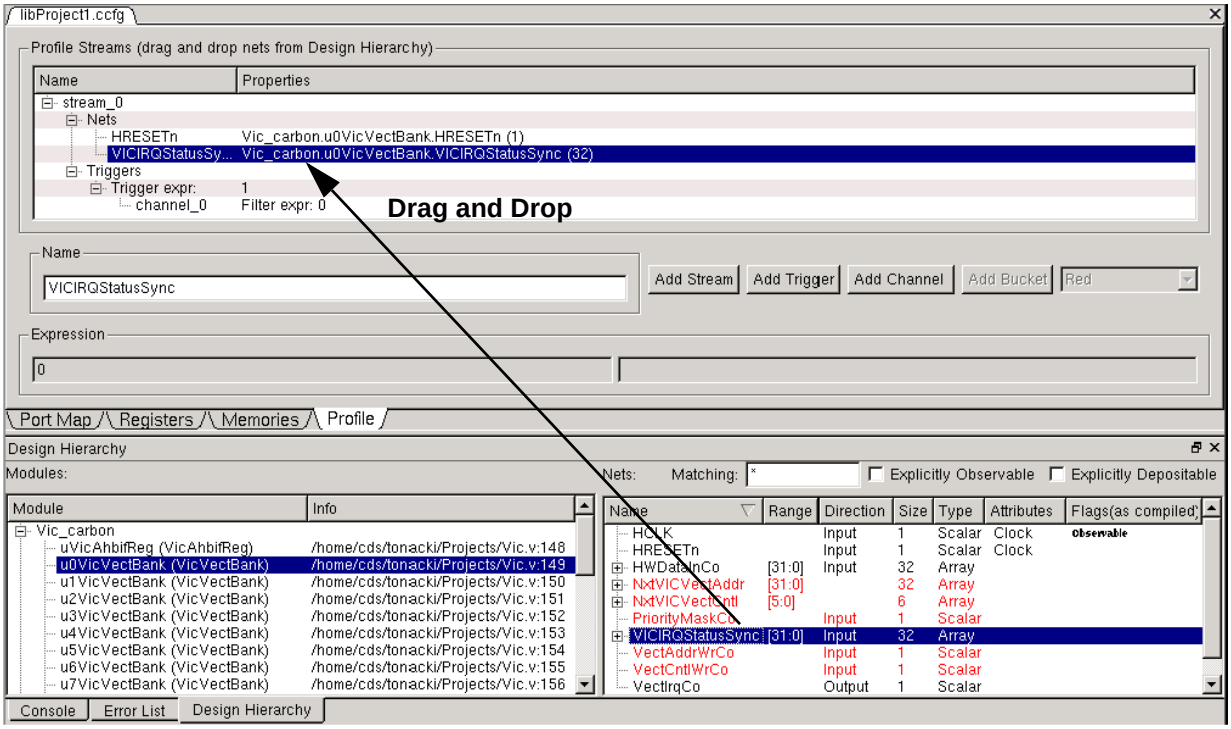


Figure 5-47 Drag Nets to Stream in Profile Tab

Defining the Triggers

To define a trigger control expression for a stream:

1. Highlight the desired *Trigger expr* line in the Profile Streams list.
2. In the *Expression* text entry field, enter a boolean expression that involves at least one of the nets in the stream.
3. To define multiple triggers for a stream, click the **Add Trigger** button.
4. Specify a different triggering expression for each added Trigger in the *Expression* field.

Note: In the control expressions, use the net name only, do not use the full path name.

Setting Up Channels and Buckets

Figure 5-48 illustrates setting up channels and buckets in the Profile tab.

Note: SoC Designer Plus requires that every stream contain a bucket channel, so you must add at least one bucket channel to each stream.

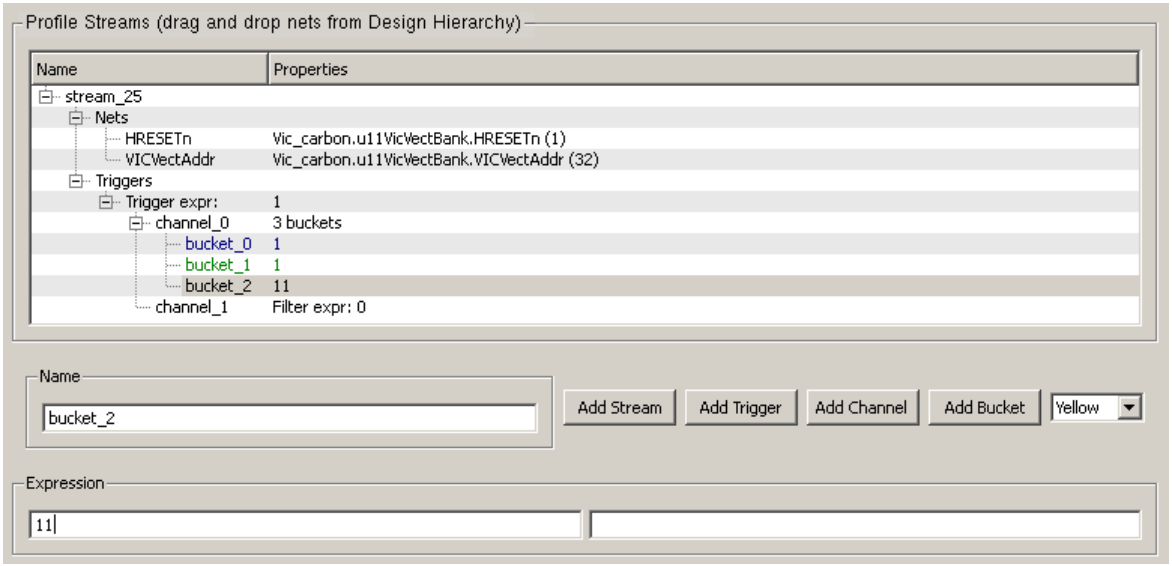


Figure 5-48 Set Up Channels in Profile Tab

To set up channels:

1. Highlight any trigger in the stream.

   By default, each trigger contains one integer, or filter expression, channel. You can keep this as an integer channel (and add other channels if you wish) or change this to a bucket channel by highlighting it and clicking the **Add Bucket** button.
2. To add more channels to the stream, click the **Add Channel** button to add additional channels to that stream.

Note: Each new channel is added under every trigger in the stream; in other words, every trigger activates the same set of channels.

3. If desired, change the channel's name in the *Name* text entry field.

4. For integer channels, type a net name or a computation involving more than one net in the *Expression* text entry field.
5. For channels containing a set of discrete buckets:
   – Click **Add Bucket**. You can add multiple buckets to the channel.
   – Type a boolean control expression for that bucket in the *Expression* text entry field. This expression defines when that bucket is active.
   – Change the display color of the bucket as desired in the right-hand pull-down menu. Note that while the Carbon Model Studio automatically assigns a different color to each bucket, you can override these settings as desired. If none of the bucket conditions are met, an event on a default OutOfRange bucket is generated; these events appear in red.

5.2.2.5 Control Expressions

Control expressions for triggers, channels, and buckets are based on nets contained in their stream. Control expressions are C-expressions and can include C operators and the functions `posedge<net>`, `negedge<net>`, and `changed<net>`. The type of C-expression used depends on the entity being controlled:

- Trigger expressions are boolean.
- Integer channel expressions are integers.
- Bucket channel expressions are boolean.

5.2.3 Recompiling the Model

After you have defined all the specific settings for your component using the *Component Editor*, you need to recompile your project. Click the **Compile** button at this point to generate the updated files to be used with your component.

5.2.4 Generated Output Files for the SoC Designer Plus Component

This section describes the output files associated with a Carbon Model Studio component—the configuration file, source files, make files, and component file (see Figure 5-49).

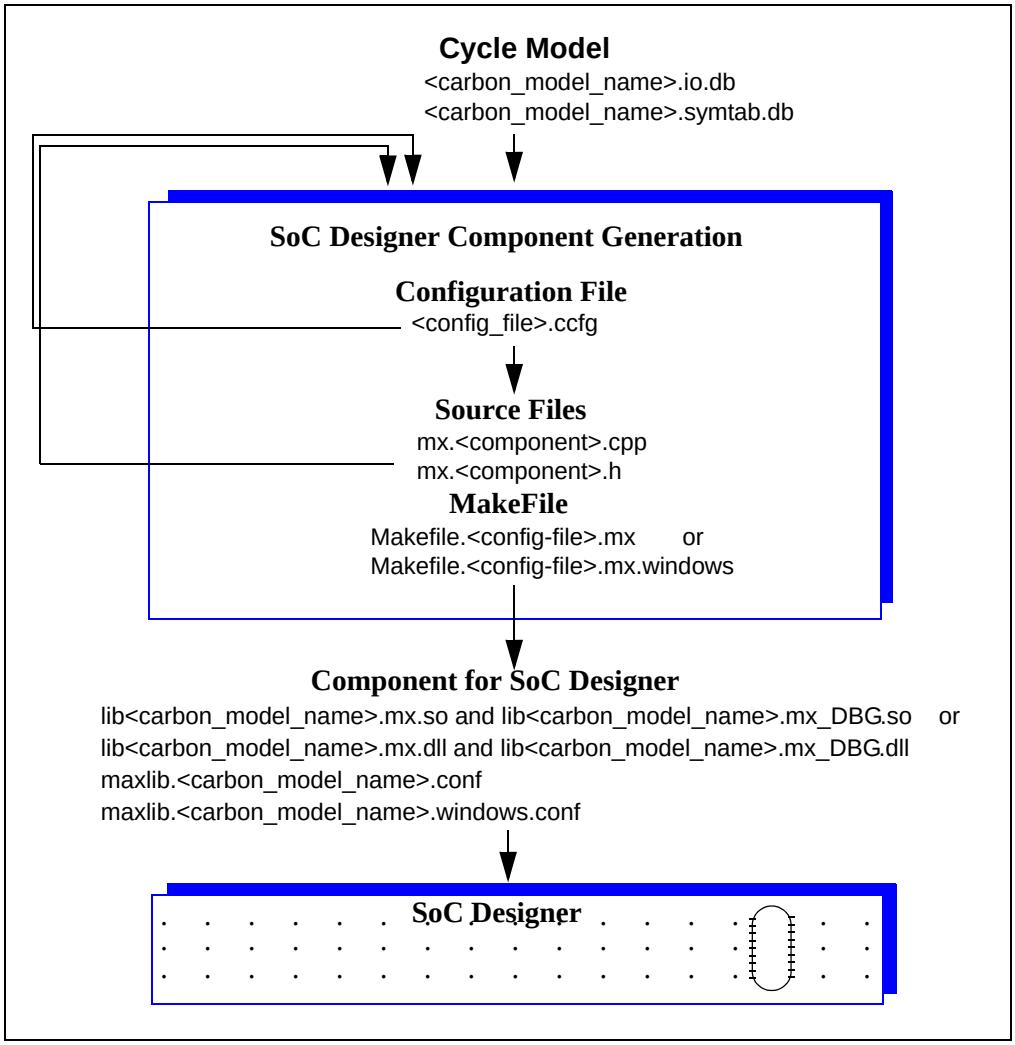


Figure 5-49 SoC Designer Plus Component Output Files

5.2.4.1 Configuration Files

A configuration file, **.ccfg*, is the first of several intermediate files generated by Carbon Model Studio. This file can be used:

- By users who are migrating to Carbon Model Studio from the SOC-VSP tool. Use this file as input for a run of the Carbon Model Studio, allowing you to make additional modifications to the component.
- By the Carbon Model Studio to create the customizable source files and Makefile, which are used to generate the component.

5.2.4.2 Makefiles

The Carbon Model Studio generates a Makefile for the following platforms:

- Linux: `Makefile.<config-file>.mx`
- Windows: `Makefile.<config-file>.mx.windows`

These Makefiles use the source `*.cpp` and `*.h` files to create the component.

5.2.4.3 Customizing Component Source Files

You may want to customize a component prior to passing it to SoC Designer Plus. For example, you might add an SoC Designer Plus API call to activate *readDbg* or *writeDbg* for a slave port, or include debug bypass code for a component (see the example below).

To customize the component, edit the component source files (**.cpp* or **.h*). By default, the source files take the name of the top-level module in your design. However, if you changed the *Component Name* field on the Carbon Model Studio's *Properties Tab*, these files use the new name.

Review the following important rules before making any changes to component source files:

- You must make all edits inside an *existing Carbon User Code* section of these files. This ensures that Carbon Model Studio retains your edits when you re-run the Carbon Model Studio using the configuration file. User-created *Carbon User Code* sections are not retained.
- Do not move the source files from the output directory.
- Empty *Carbon User Code* sections or sections containing white space are ignored. Any *Carbon User Code* sections that are unused during component generation are appended to the end of the generated source in an `#if 0` block and a warning is generated.
- If compile or link flags are needed, enter the relevant paths in the Model Studio *General Properties* tab (see "Setting Compiler and Linker Flags" on page 95).
- Editing these files requires familiarity with the SoC Designer Plus API. Refer to the *ESL API Developer's Guide* for information on how to customize these files.

To customize the component:

1. Edit the `mx.<component>.cpp` and/or `mx.<component>.h` files
2. Re-run the Make process at the command prompt:

For Linux platforms — `make -f Makefile.<config_file>.mx #`
For Windows platforms —
`nmake /F Makefile.<config_file>.mx.windows #`

Example of adding Debug Bypass Code to a source file

The following code is an example of adding customized code to the *Carbon User Code* section, inside the method `CompName::debugAccess`. The example shows a 4x4 matrix that forwards transactions based on the address map defined:

```
// CARBON USER CODE [PRE compName::debugAccess] BEGIN

if ((addr >= 0x00000000) &&  (addr <= 0x0fffffff))
 {
   return (AXI_Flowthru_Master_CarbonAxiFS2T->debugAccess(dir, addr,
numBytes, buf, numBytesAccepted, ctrl));
 }
   if ((addr >= 0x10000000) &&  (addr <= 0x1fffffff))
 {
   return (AXI_Flowthru_Master_1_CarbonAxiFS2T->debugAccess(dir, addr,
numBytes, buf, numBytesAccepted, ctrl));
 }
   if ((addr >= 0x20000000) &&  (addr <= 0x2fffffff))
 {
   return (AXI_Flowthru_Master_2_CarbonAxiFS2T->debugAccess(dir, addr,
numBytes, buf, numBytesAccepted, ctrl));
 }
   if ((addr >= 0x30000000) &&  (addr <= 0x3fffffff))
 {
   return (AXI_Flowthru_Master_3_CarbonAxiFS2T->debugAccess(dir, addr,
numBytes, buf, numBytesAccepted, ctrl));
 }

// CARBON USER CODE END
```

Note: Implementation note for Debug Bypass Code: If overlapping memory regions exist for two ports, debug bypass code is necessary on only one of the ports.

5.2.4.4 Adding a Component to the SoC Designer Plus Model Library

Carbon Model Studio creates a configuration file that can be used to add the generated component into the model library of SoC Designer Plus. To add the component to the model library from SoC Designer Plus:

1. Select **File** > **Preferences**.
2. Click on **Component Library** in the list on the left.
3. Under the *Additional Component Configuration Files* window, click **Add**.
4. Select either `maxlib.lib<carbon_model_name>.conf` (Linux) or `maxlib.lib<carbon_model_name>.windows.conf` (Windows) depending on your platform.
5. Click **OK**.
6. If you want to save the preferences permanently click **OK & Save**. To save them for the current session only, click **OK**.

The component is now available from the SoC Designer Plus *Component Window.*

5.2.4.5 SoC Designer Plus Component Files

The component file is the final output file from the Carbon Model Studio and is the input to SoC Designer Plus. The Carbon Model Studio generates two versions of the component; an optimized release version for normal operation and a debug version.

On Linux the *debug* version of the component is compiled without optimizations and includes debug symbols for use with gdb. The *release* version is compiled without debug information and is optimized for performance.

On Windows the *debug* version of the component is compiled referencing the debug runtime libraries, so it can be linked with the debug version of SoC Designer Plus. The *release* version is compiled referencing the release runtime library. Both release and debug versions generate debug symbols for use with the Visual C++ debugger on Windows.

Linux and Windows debug versions all define the macro CARBON_DEBUG when compiled. The CARBON_DEBUG macro can be used to filter debug messages or other code that should only be run when executing the debug version of the component.

The name of the component libraries are:

- *lib<carbon_model_name>.mx.so* and *lib<carbon_model_name>.mx_DBG.so* for Linux platforms
- *lib<carbon_model_name>.mx.dll* and *lib<carbon_model_name>.mx_DBG.dll* for Windows platforms

By default, the name of the component file is the same as the name of the Project.

See the previous section for information regarding how to register your new component with SoC Designer Plus.

5.3 SoC Designer Plus-Specific Information

You use Carbon Model Studio to create a SoC Designer Plus-compatible component from the Cycle Model, which includes the simulation and debugging interfaces.

Using this process, system engineers use the component in SoC Designer Plus to build a simulatable system and perform the following tasks:

- Architectural exploration
- System and software validation and debug
- Embedded software development

Figure 5-50 shows the process flow from source RTL to component for SoC Designer Plus.

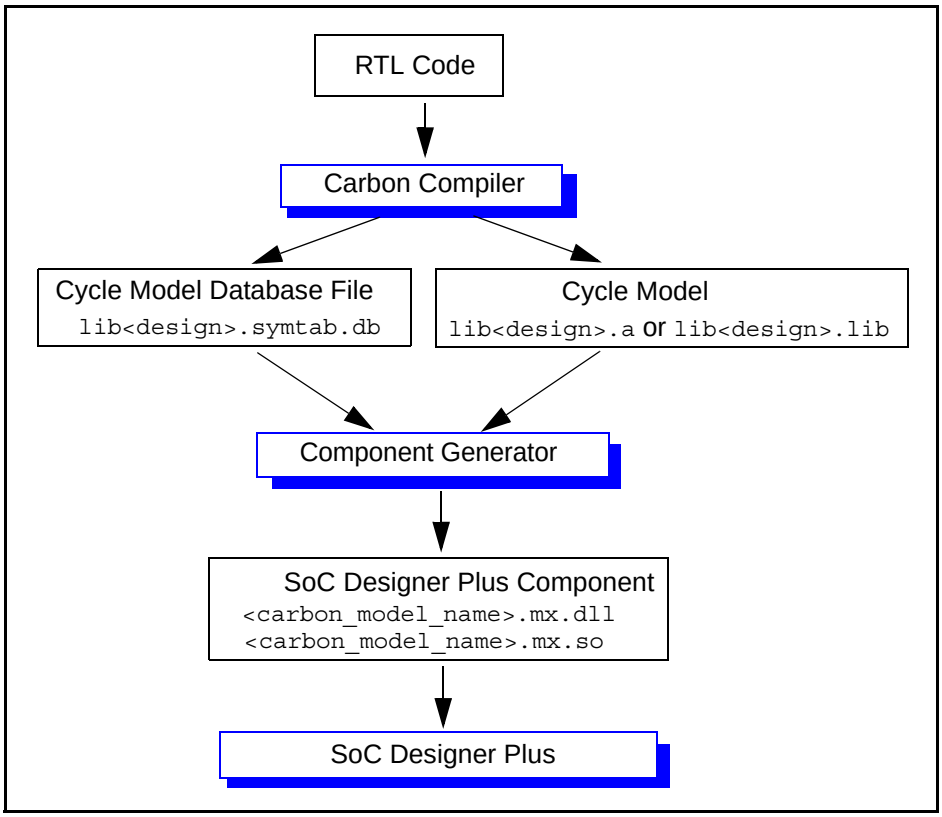


Figure 5-50 Generating a Component for SoC Designer Plus

The Component Generator enables you to:

- Specify the connection interface between your Cycle Model and SoC Designer Plus. For example you can:
  - Choose which of your Cycle Model ports are visible to SoC Designer Plus.
  - Insert and connect transactors, which shield many of your Cycle Model ports from SoC Designer Plus.
  - Create unused input and output ports (null), if required by SoC Designer Plus.
- Set the timing for the SoC Designer Plus reference clock so that your Cycle Model's clocks can fully exercise your design.
- Specify the internal registers and memories that you wish to observe during validation.

After you use the Carbon Model Studio to specify the settings above, Carbon Model Studio generates a component, which includes the simulation and debugging interfaces (Figure 5-51).

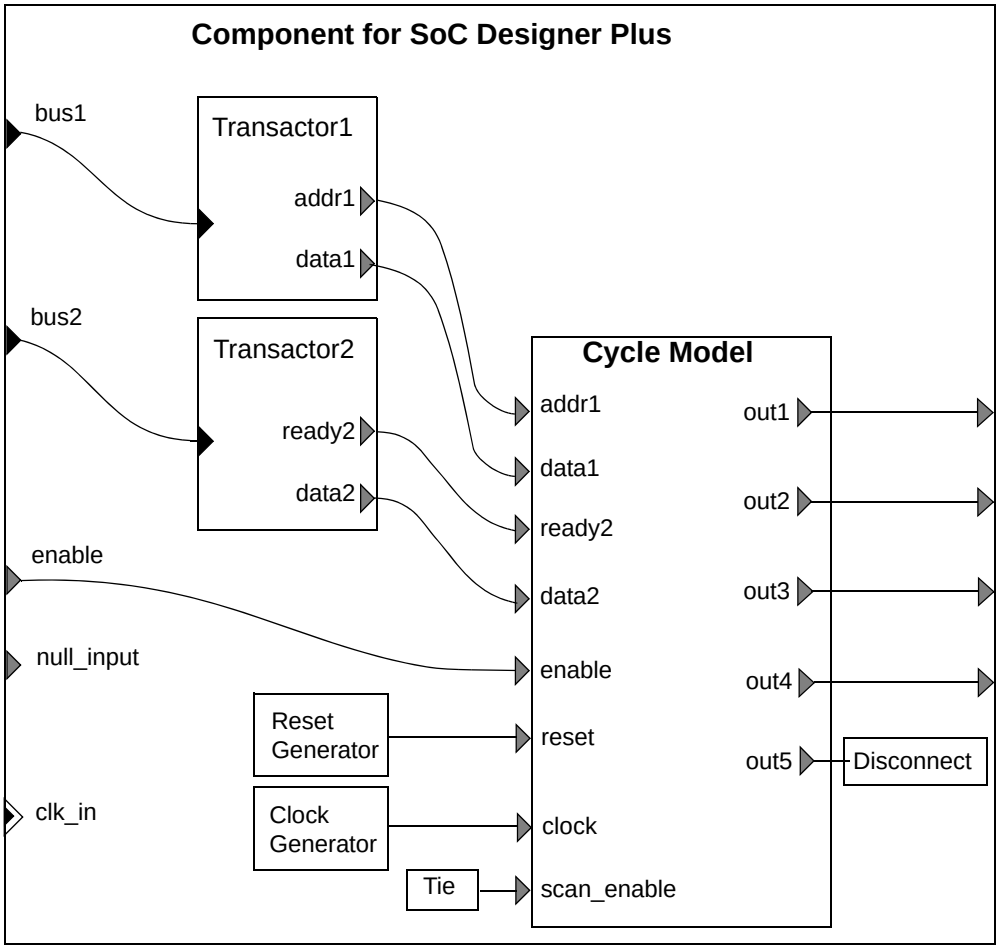


Figure 5-51 Component for SoC Designer Plus

When the component is registered with SoC Designer Plus, the validation engineer builds a system by connecting the component with other components via the visible input and output ports, as shown in Figure 5-52. The engineer then simulates the system, observing the exposed internal registers and accumulated profiling data.

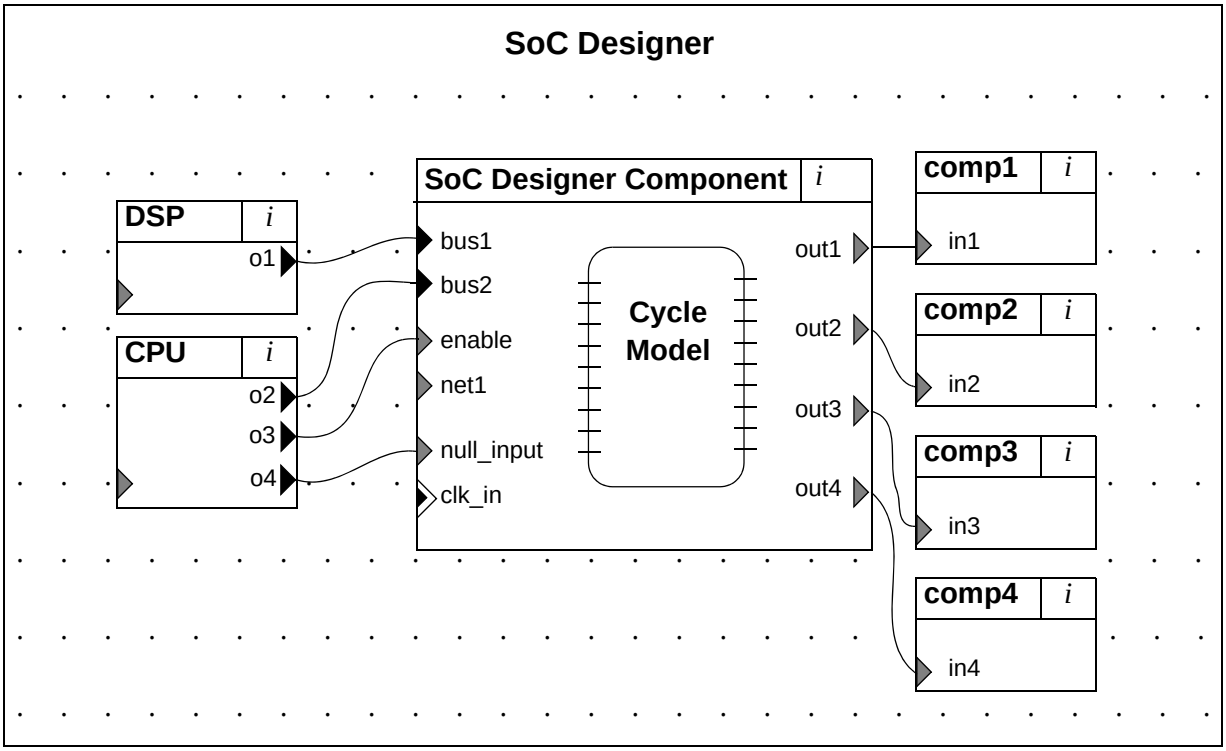


Figure 5-52 Component in SoC Designer Plus

5.3.1 Prerequisites for SoC Designer Plus

Before you generate the component for SoC Designer Plus, make sure you have performed the following steps:

- If they have not already been defined, set the necessary environment variables:

```
MAXSIM_HOME = <SoC Designer Plus installation directory>
MAXSIM_PROTOCOLS = <SoC Designer Plus installation directory>/Protocols
CARBON_HOME = <Carbon installation directory>
```

- Make sure that the Cycle Model output libraries are static:

  For Linux: `lib<design_name>.a`

  For Windows: `lib<design_name>.lib`

- 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 and/or depositable.

  Mark any internal registers intended to be I/O ports in the SoC Designer Plus component (that is, those to be connected to a transactor or to an SoC Designer Plus port) with `scObserveSignal` or `scDepositSignal`. Be aware that this can slow down performance. See "Defining Directives and Recompiling the Cycle Model" on page 69 for more information.

- Compile your Cycle Model following the steps defined in Chapter 3. Note that the `lib<design>.symtab.db` file must be present during the Carbon Model Studio run to create the component, but it is not required at SoC Designer Plus runtime.

5.3.1.1 Transactors

SoC Designer Plus supports transaction-level port interfaces, but Cycle Models typically communicate at the pin level. Transaction adapters (known as *Transactors*) are provided to convert between several common transaction level interfaces and the Cycle Model pins.

Note: If you are creating a model for Platform Architect, note that the term used in the GUI changes from "Transactor" to "Protocol."

Refer to the *SoC Designer Plus User Guide* for details about the difference between transaction-based and signal-based communication in the SoC Designer Plus environment.

The transactors you add to your component in the Cyclecomponent typically add new SoC Designer Plus transaction ports, and hide Cycle Model ports.

For example, an AHB_Slave_T2S converts AHB transactions to signal value changes (T2S means "transaction to signal"). When you add an AHB_Slave_T2S transactor to your component in the Carbon Model Studio tool, a new SoC Designer Plus transaction slave port is created in the component for incoming AHB transactions. You must use the Carbon Model Studio tool to connect the transactor to the appropriate Cycle Model signal ports. Each Cycle Model signal connected to the transactor is no longer exposed as an SoC Designer Plus port.

The resulting component may be connected to a transaction master port in SoC Designer Plus.

5.3.1.2 Component Clocking

The component generated by the Carbon Model Studio tool uses SoC Designer Plus cycle-based scheduling. Refer to the *SoC Designer Plus User Guide* for information about cycle-based scheduling.

The component has a clock input port named ‘clk-in’. This is an SoC Designer Plus cycle-based clock, not an RTL signal port. The clk-in port determines how frequently the component is called to execute a cycle. For more information about clocking options, refer to “Clock Inputs and Clock Generators” on page 160.

- If you connect the clk-in port to the clock master port of another component (e.g., a CDIV clock divider), then that component determines when the component runs a cycle.
- If you do not connect the clk-in port to anything, then the component is driven by the SoC Designer Plus reference clock, running one cycle per time unit.

The component, like all SoC Designer Plus components, is a subclass of C++ class sc_mx_module. It implements the two methods `update()` and `communicate()`, which are called by the clock master to which the clk-in port is connected.

In the `communicate()` method, all output ports that have new values are driven.
In the `update()` method, the Cycle Model is executed.

5.3.1.3 Clock Generation

Clock generators are supported for SoC Designer Plus components only (not for Platform Architect components).

In SoC Designer Plus cycle-based scheduling, there is no signal value (i.e., 1 or 0) associated with a clock slave port. The SoC Designer Plus clock slave port is only a mechanism for configuring when the component `communicate()` and `update()` methods are invoked.

You can drive the Cycle Model clock signal ports using a Carbon Model Studio clock generator. A clock generator applies 0 and 1 signal values to the Cycle Model clock signal ports every time the component’s `update()` method is called. A clock generator may be configured to run at the same speed, faster, or slower than the SoC Designer Plus reference clock.

Note: Rather than creating a clock generator, ARM recommends creating a clock input transactor and exposing the clock input port externally.

For additional information on the use of clock generators, refer to “Clock Inputs and Clock Generators” on page 160. For instructions on creating a clock generator, refer to “Specifying Generated Clocks” on page 100.

5.4 Using the Component in SoC Designer Plus

This section describes how to set up the component in SoC Designer Plus.

For your SoC Designer Plus run, you need only the component. No other files are necessary.

5.4.1 Setting Component Parameters in SoC Designer Plus

You can change the settings of all parameters in SoC Designer Canvas and of some parameters in SoC Designer Simulator. To modify the component's parameters:

1. In the Canvas, right-click on the component and select **Component Information**. The *Edit Parameters* dialog appears (Figure 5-53).

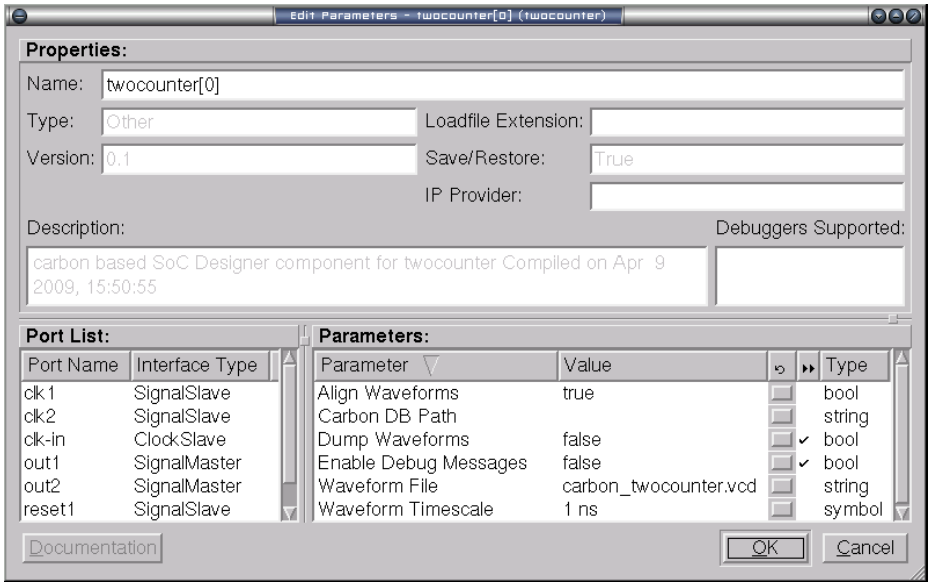


Figure 5-53 Edit Parameters dialog

2. Double-click the **Value** field of the parameter that you would like to modify.
3. If a text field appears, type a new value in the **Value** field. If a menu choice is offered, select the desired option. The parameters of relevance to the component are:
   - Dump Waveforms
   - Waveform File
   - Align Waveforms
   - Waveform Timescale
   - Carbon DB Path

These parameter options are explained below.

5.4.1.1 Dump Waveforms

SoC Designer Plus does not dump waveforms automatically. You must turn on waveform dumping by setting the *Dump Waveforms* parameter to "true."

5.4.1.2 Waveform File

To change the name of your waveform file, type a new name for the *Waveform File* parameter. If waveform dumping has been turned on, SoC Designer Plus automatically writes the accumulated waveforms to the waveform file in the following situations:

- when the waveform buffer fills
- when validation is paused and when validation finishes
- at the end of each validation run

If the initial waveform created was *carbon.vcd*, the new files created after subsequent simulation resets are named *carbon_1.vcd*, *carbon_2.vcd*, etc. In your waveform viewer, you can load the waveform file and explore the behavior of the signals in the design.

The type of waveform file generated, `*.vcd` is determined by which file type you selected on the General Properties tab (see "Enabling Waveform Generation" on page 94).

5.4.1.3 Align Waveforms

When *Align Waveforms* is set to "true" (the default setting), it causes the waveforms dumped from the component to be aligned with the SoC Designer Plus simulation time. Note that the reset sequence is not included in the Cycle Model VCD/FSDB data when the waveforms are aligned because it occurs in zero SoC Designer Plus simulation time.

If you want to capture the reset sequence for debugging, you can set this parameter to "false". This shows the reset sequence in the created Cycle Model waveform data. However, it also means that the time shown in the Cycle Model waveforms does not match the time shown in the SoC Designer Plus waveforms.

5.4.1.4 Waveform Timescale

Select the timescale to be used in the Cycle Model waveform file using the *Waveform Timescale* parameter. The default is *1 ns*, but you can click the *Value* field to display a drop-down list of available timescales values.

5.4.1.5 Carbon DB Path

SoC Designer Plus only runs if the path to your database file is correct. This parameter is usually not necessary because the `lib<model_name>.symtab.db` database file is embedded by default in the Cycle Model file.

However, it is possible to generate the Cycle Model without embedding the database file (using the `-noFullDB` compiler option). In this case you would need to define the name and location of this file. To specify the path to the database directory, set the `Carbon DB Path` parameter to the directory that contains your `lib<model_name>.*.db` file.

The database path parameter is passed to `carbonSetFilePath` in the component's `init()` method. For more information, see `carbonSetFilePath` in the *Carbon Model API Reference*.

5.4.2 Special Cases

5.4.2.1 Components with Clock_Input Pseudo-Transactors

If your component has a `Clock_Input` pseudo-transactor, then any connection to the SoC Designer Plus default `clk_in` port is ignored. To avoid confusion, keep `clk_in` unconnected in SoC Designer Plus.

5.4.2.2 Using $display in Windows Environment with SoC Designer Plus

If you use `$display` in a Windows environment while running SoC Designer Plus interactively, you get the following error message:

```
<twocounter[0]> Unable to write to file stdout||stderr.
Error message is: Bad file descriptor
```

To fix this, redirect the output to a file. You can do this by invoking SoC Designer Plus from a command prompt as follows:

```
"C:\Program Files\Carbon\SoCDesigner7.2\Bin\Release\SDSim.lnk" > out.txt
```

5.4.2.3 Using Save and Restore in SoC Designer Plus

Unlike SoC Designer Plus, the component is not endian-neutral. Therefore, after doing a save and restore of the component in SoC Designer Plus, the resulting MXR file cannot be moved to an operating system that uses a different byte order than your current platform.

5.4.2.4 Using Profiling in SoC Designer Plus

When gathering profiling data on a Cycle Model that has been compiled with the `-profile` option, you should run SoC Designer Plus in batch mode (see the *SoC Designer Plus User Guide*). Profile data collected when running the SoC Designer Plus Graphical User Interface includes time spent waiting for user responses and thus gives skewed profiling results.

5.4.2.5 Flow-through Path Support for Signal Ports

Flow-through paths are supported automatically by executing the logic defined in the flow-through paths until they settle. The Carbon compiler automatically detects when a flow-through path exists in a model compiled from RTL. It adds code to the SoC Designer Plus component such that any inputs that are in a flow-through path:

1. Drive the signal into the RTL model.
2. Execute the RTL logic defined in the flow-through path.
3. Detect changes on any RTL outputs and drive the output to any connected components.

In general, it is recommended that any models execute such flow-through paths during the communicate phase. All SoC Designer Plus components created using Carbon Model Studio from RTL initiate flow-through paths only during the communicate phase.

5.5 Platform Architect-Specific Instructions

Use Carbon Model Studio to create a Synopsys (formerly CoWare) component from the Cycle Model that is compatible with Platform Architect, and includes the simulation and debugging interfaces. Using this process, system engineers use the component in Platform Architect to build a simulatable system and perform the following tasks:

- Architectural exploration
- System and software validation and debug
- Embedded software development

Use the Carbon Model Studio Component Generator to:

- Load a Cycle Model database.
- Define communication interfaces to drive the design.
- Map data types and define which internal information inside the Cycle Model to export to the Platform Architect environment.
- Compile the Component and generate interfaces.

5.5.1 SystemC Modeling Language (SCML) Interface Overview

SystemC Modeling Language (SCML) is used to export internal register information and aggregate register information so that Platform Architect analysis tools can utilize internal Cycle Model information. Registers can be 32 or 64 bits and are used to export internal signals, registers, and memories for analysis, including Platform Architect's debuggers and analysis tools.

SCML assigns a shadow register that maps to the appropriate data inside the Cycle Model. It also provides the Platform Architect environment with the ability to read internal Cycle Model data and deposit to internal Cycle Model data.

SCML information is assembled into a register/memory bank that you define. Offsets are used to specify the registers and memories that reside within the bank. A debug access mechanism is provided to update or retrieve data from these SCML objects and perform the appropriate calls to the Carbon API to access the Cycle Model data.

For more information on SCML, refer to the Synopsys *SystemC Modeling Manual*.

5.5.2 Prerequisites

Before you generate the component for Platform Architect, make sure you have performed the following steps:

- If they have not already been defined, set the necessary environment variables:

  `COWAREHOME = <Platform Architect installation directory>`
  `CARBON_HOME = <Carbon Model Studio installation directory>`
  `COWAREIPDIR = <Platform Architect IP directory>`
  (*Optional. If not specified, then Carbon Model Studio uses $COWAREHOME/IP to identify available Platform Architect transactors.*)
  `COWAREIPLIB = <Platform Architect installation directory>`
  (*For important, version-specific instructions refer to Section 5.5.2.1,* COWAREIPLIB Implementation Notes *on* page 145.)

- Make sure that the Cycle Model output libraries are static:

  For Linux: `lib<design_name>.a`

  For Windows: `lib<design_name>.lib`

- 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 and/or depositable.

  Mark any internal registers intended to be I/O ports in the Platform Architect component (that is, those to be connected to a transactor or to a Platform Architect port) with `observeSignal` or `depositSignal`. Be aware that this can slow down performance. See "Defining Directives and Recompiling the Cycle Model" on page 69 for more information.

- Compile your Cycle Model following the steps defined in Chapter 3.

5.5.2.1 COWAREIPLIB Implementation Notes

If you are running:

- **Synopsys Platform Architect version K-2015.06**, you must do the following to enable Carbon Model Studio to emit tcl files:

  1. Create a file named `cwr.lib`. with the following content:

     ```
     INCLUDE $COWAREHOME/dmtools/data/cwr.lib
     DEFINE GenericIPtemplates GenericIPtemplates
     ```

  2. Point the COWAREIPLIB environment variable to the new file.

  The following is an example script that shows the commands needed to properly set up the environment:

  ```
  setenv COWAREVERSION K-2015.06
  setenv COWAREHOME /tools/linux/CoWare/$COWAREVERSION/SLS/linux
  source $COWAREHOME/setup.csh -pa
  setenv PATH $COWAREHOME/common/bin:$PATH
  ```

  After completing these steps, invoke Carbon Model Studio, create the component, and compile.

5.5.3 Recompiling the Model

After you have defined all the specific settings for your component using the *Component Editor*, you need to recompile your project. Click the **Compile** button at this point to generate the updated files to be used with your component.

5.5.4 Understanding the Platform Architect Component Output Files

The Carbon Model Studio Component Generator outputs several file types, including configuration, source, make, and component files. This section describes:

- Configuration Files
- Source Files
- Makefiles
- For Platform Architect
- Known Issue

Refer to Figure 5-54 to review the process flow from Cycle Model to component for Platform Architect.

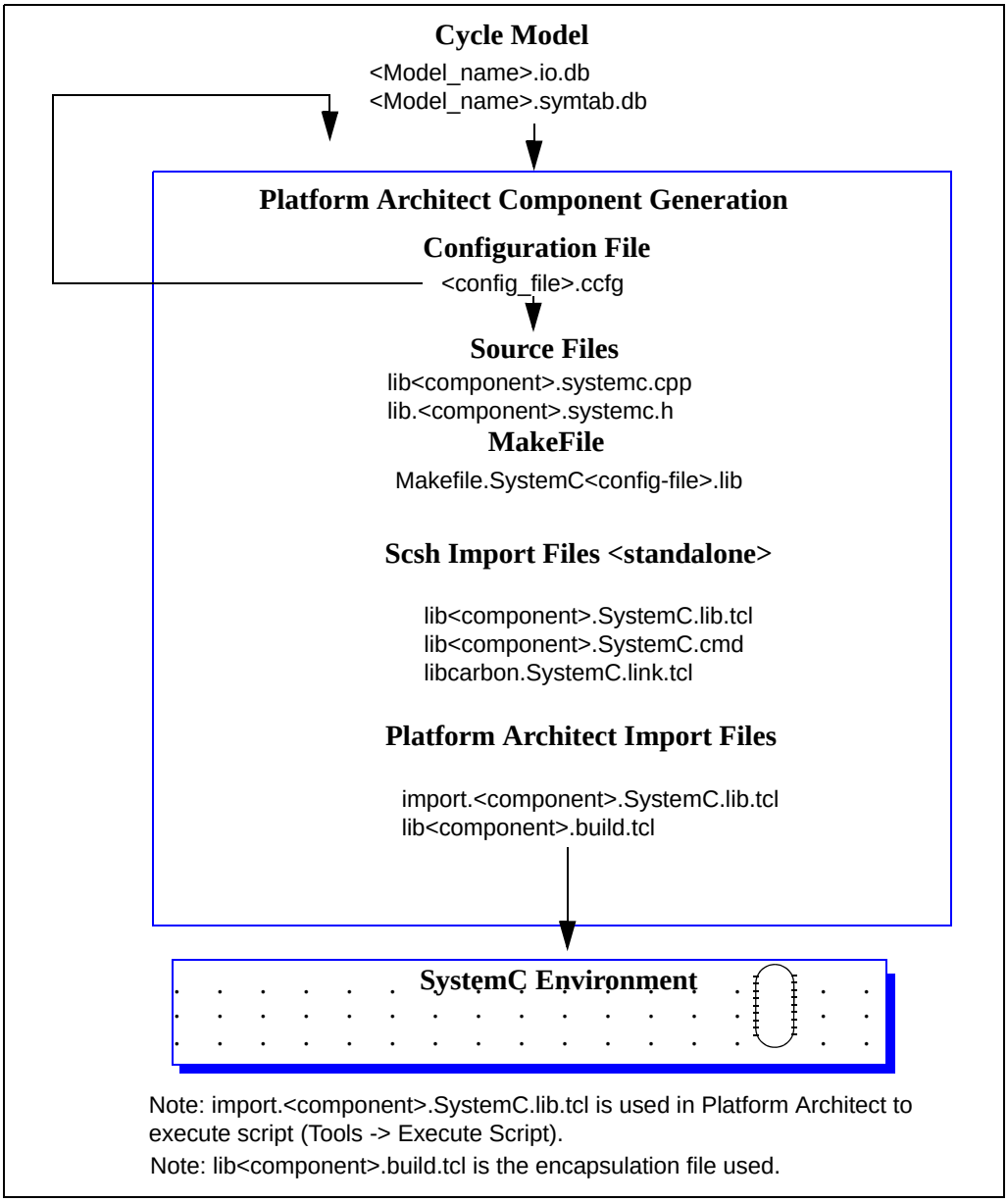


Figure 5-54 Platform Architect Component Output Files

5.5.4.1 Configuration Files

A configuration file, `*.ccfg`, is the first of several intermediate files generated by Carbon Model Studio. This file can be used:

- By users who are migrating to Carbon Model Studio from the Platform Architect Component Generator tool. Use this file as input for a run of the Carbon Model Studio, allowing you to make additional modifications to the component.
- By the Carbon Model Studio to create the customizable source files and Makefile, which are used to generate the component.

5.5.4.2 Source Files

If you want to customize the component prior to passing it to the Platform Architect environment, you can edit the source files (`*.cpp` or `*.h`).

1. Edit the `lib<component>systemc.cpp` and/or `lib<component>systemc.h` files.

   To ensure that Component Generator retains your edits if you reuse the edited source files for a future run, make all edits inside the designated *Carbon User Code* section of these files (see "Example of adding Debug Bypass Code to a source file" on page 133).

2. Re-run the Make process at the command prompt:

   ```
   make -f Makefile.coware<config_file>.lib #
   ```

5.5.4.3 Makefiles

Component Generator generates a Makefile for Linux:

- `Makefile.coware<config_file>.lib`

This Makefile uses the source `*.cpp` and `*.h` files to create the component for the Platform Architect environment.

5.5.4.4 Known Issue

When adding a transactor, the import script does not set the port direction. For example, in the script *import_Twocounter.coware.tcl* the port direction is specified as "None", which causes a problem. The workaround is to edit the port direction in the script.

The import script of the generated component includes a line similar to the following:

```
::pct::add_block_port $block port None
```

The solution is to change the line to have the correct port type, for example:

```
::pct::add_block_port $block port inout
```

5.6 Creating Components for SystemC

ARM delivers products that allow designers to compile their synthesizable RTL into an ultra-high performance Cycle Model that can then be linked directly into a SystemC design environment. This integration provides the speed necessary to perform software validation and performance modeling while maintaining the investment already made in the SystemC environment.

A traditional approach to integrate implementation-level RTL into SystemC typically involves integrating an RTL simulator via the PLI. While this approach correctly models the implementation, it does so at a relatively slow speed. In addition, PLI introduces a substantial amount of interconnect overhead.

The process of integrating a Cycle Model into a SystemC development environment includes:

- Building the Cycle Model using the Carbon compiler.
- Generating a component that integrates the Cycle Model into a SystemC environment.
- Compiling the design into an executable.

Refer to the *CMS Installation Guide* for supported SystemC versions.

5.6.1 Prerequisites

Before you generate the component for SystemC, make sure you have performed the following steps:

- Set the environment variable SYSTEMCHOME to the base directory of your SystemC installation.
- Make sure that the Cycle Model output libraries are static:

  For Linux: `lib<design_name>.a`

  For Windows: `lib<design_name>.lib`

- 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 and/or depositable.

  Mark any 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 "Defining Directives and Recompiling the Cycle Model" on page 69 for more information.

- Compile your Cycle Model following the steps defined in Chapter 3.

5.6.2 Generating the Component for SystemC

1. From the Project Explorer, right-click the Cycle Model to display the context menu.
2. From the context menu, select *Create SystemC Component* (Figure 5-55).

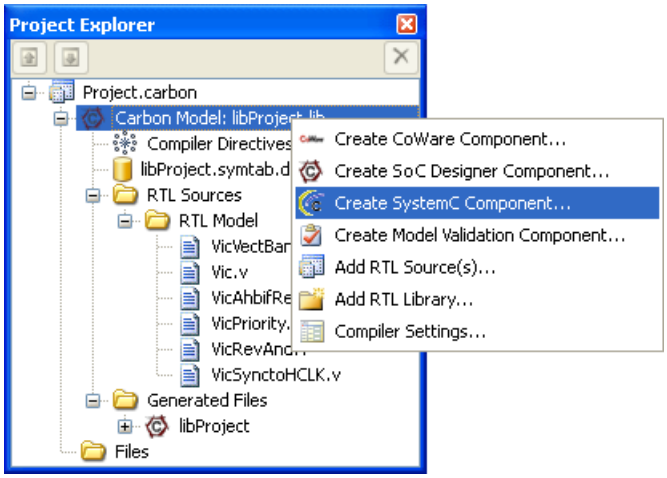


Figure 5-55 Create Component for SystemC

You can also click the **SystemC** button on the Button bar.

The new component and its associated output files are displayed in the Project Explorer (Figure 5-56).

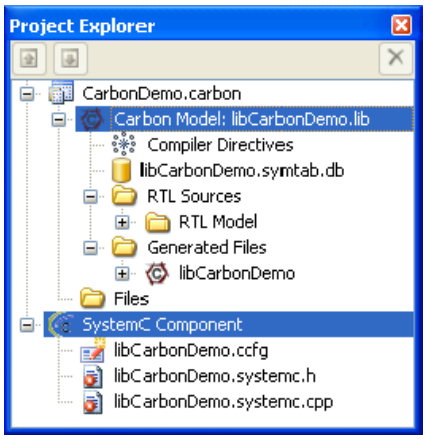


Figure 5-56 Component for SystemC in Project Explorer

5.6.2.1 Using the Component Properties View

Use the Component Properties view (Figure 5-57) to view the SystemC component properties, including:

- Component top module name.
- Force Update — Specifies that calls to `sc_prim_channel::request_update` are forced for all input changes. If this is not specified, `request_update` is called only for clock, reset, and feed-through inputs.
- Module Name — By default the SystemC module name is the same as the top module name. You can enter a unique name to be used for the SystemC module that is generated.
- Database (*.db) file location.
- Output directory.

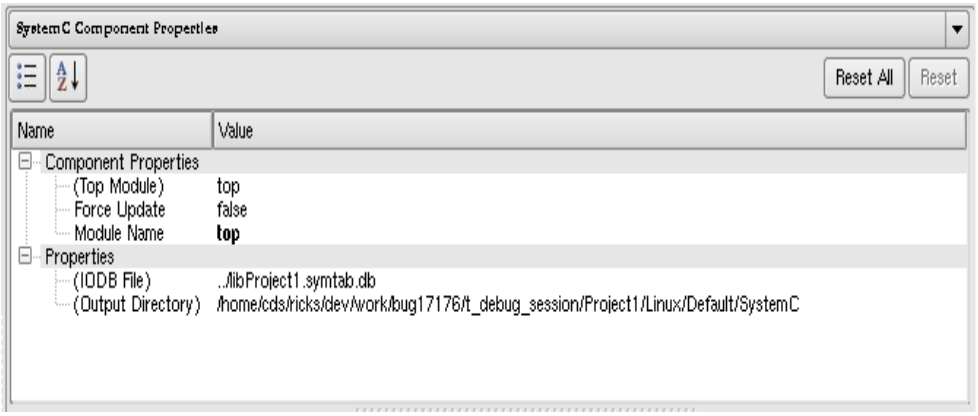


Figure 5-57 SystemC Component Properties

5.6.2.2 Using the Ports Window for SystemC

The Ports window for SystemC provides tabs to access ports. Double-click the .ccfg file in the Project Explorer to display the Ports Editor (Figure 5-58).

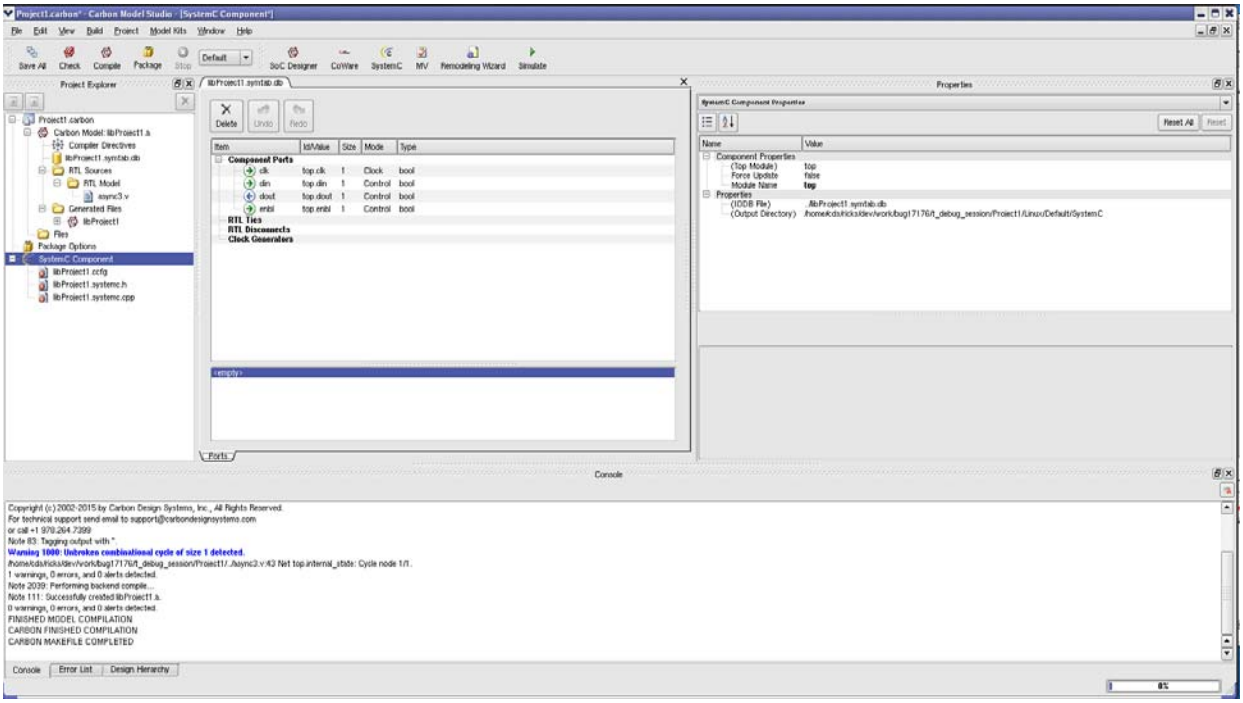


Figure 5-58 Ports Window for SystemC

This window is very similar to the one used for creating components for SoC Designer except for the following differences:

- The Ports window for SystemC *does not* contain the following:
  - Transactor, Add Transactor, Add Parameter feature
  - Registers or Memories tabs
  - Profile tab
  - Generated resets (Reset Generators section)
  - “Connect to Parameter” option when right-clicking an RTL port
  - Ability to add parameters to your component
  - Ability to add unused (null) ports
- The Ports window for SystemC contains two columns that the SoC Designer Plus view does not:
  - Mode — Not used in this release.
  - Type — Specifies the SystemC type to be used. The default SystemC type is based on the size, type, and direction of the RTL port. (The size element of the SystemC type is determined automatically from the RTL port size.) You can select a different type from the pulldown menu.

- The Clock Generators for the SystemC Ports window use the SystemC sc_clock primitive to generate the clock values. (See the SystemC reference manual for details about the sc_clock parameters.) The parameters are:
  - Initial Value (Low or High)
  - Start Time (double)
  - Start Time Units (SC_FS, SC_PS, SC_NS, SC_US, SC_MS, SC_SEC)
  - Duty Cycle (double)
  - Period (double)
  - Period Time Units (SC_FS, SC_PS, SC_NS, SC_US, SC_MS, SC_SEC)

For more information on using the Ports window for SystemC, see "Ports Tab" on page 97. That section refers to SoC Designer Plus, but for SystemC, the tool is the SystemC simulator.

5.6.3 Recompiling the Model

After you have defined all the specific settings for your component using the Ports Editor, you need to recompile your project. Click the **Compile** button at this point to generate the updated files to be used with your component.

Chapter 6

Understanding Component and Platform Interaction

Carbon Model Studio enables you to pass (and run) simulation scripts and variables for your simulation environment, and to access components in your design.

6.1 Launching a Simulation

Carbon Model Studio is designed to be flexible enough to support and invoke simulations of components from your simulation environment, whether that environment is SoC Designer, Platform Architect, SystemC, or a custom application. Carbon Model Studio allows you to specify what tools to invoke, how to invoke them, and what variables or arguments to use.

The settings for invoking a simulation script are contained in the *Simulation Control* area of the *Project Properties* view for a selected Project (see Figure 6-1).

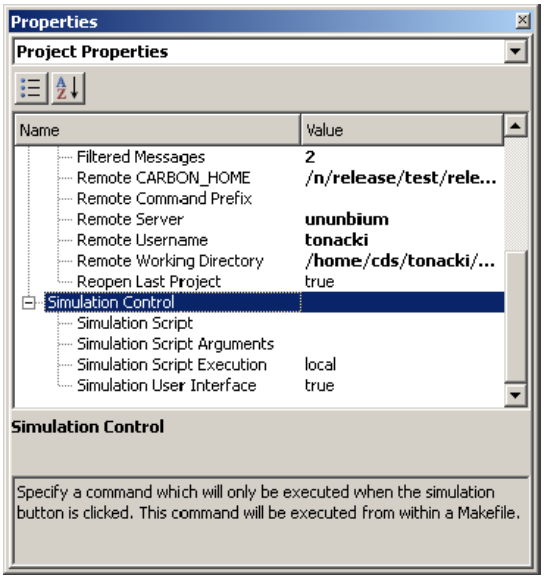


Figure 6-1 Project Properties for Simulation Control

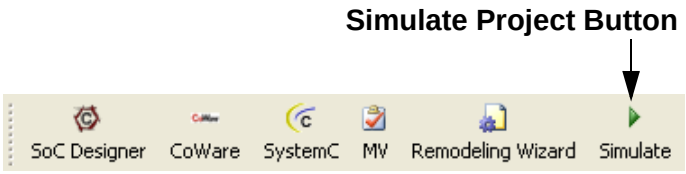
Editable parameters for simulation control include:

- **Simulation Script** —The *Simulation Script* parameter specifies a script or executable that will set up an environment and launch the simulation tool. This script is executed when the **Simulate** button is selected.
- **Simulation Script Arguments** — This parameter is a list of arguments that would be passed to the script on the script's command line. For example, you can pass the SoC Designer simulator (`sdsim`) the name of a file with the extension `.mxp` that contains information about the system that will be simulated.

  Note that the following environment variables are available when Carbon Model Studio launches a Simulation script.

  CARBON_PROJECT=<project directory>
  CARBON_CONFIGURATION=<platform>/<name>
  CARBON_OUTPUT=$CARBON_PROJECT/$CARBON_CONFIGURATION
  CARBON_MODEL=<design.name>
- **Simulation Script Execution** — This parameter is the location where the Simulation script is to be executed. The default setting is *local*, meaning the script will run on the current Linux machine. When using Windows and a remote compilation configuration, set this field to *remote*.
- **Simulation User Interface** — This parameter enables the Carbon Model Studio for controlling simulation options. The default is *true*.

Once the simulation parameters are defined, invoke the simulation by clicking the **Simulate** button, shown in the next figure.



You can also select **Simulate Project** from the *Project* menu.

6.1.1 Examples for Each Environment

The following examples illustrate sample simulation parameters used to invoke SoC Designer, Platform Architect, and SystemC simulations.

6.1.1.1 Setting Simulation Parameters for SoC Designer Plus

In Figure 6-2, the *-i* argument causes the commands contained in the file *script.mxscr* to be applied when *sdsim* is invoked.

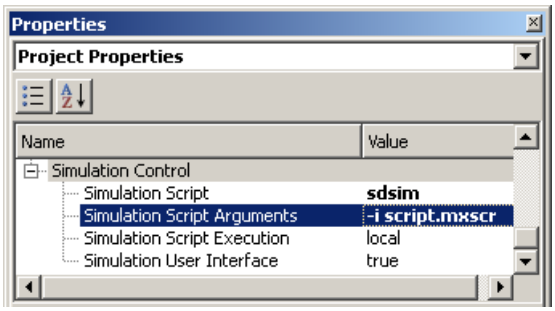


Figure 6-2 Simulation Properties for SoC Designer

6.1.1.2 Setting Simulation Parameters for Platform Architect

In Figure 6-3, Platform Architect instructions contained in the file `GenComponents.tcl` are read and executed when `pcsh` is invoked for the simulation.

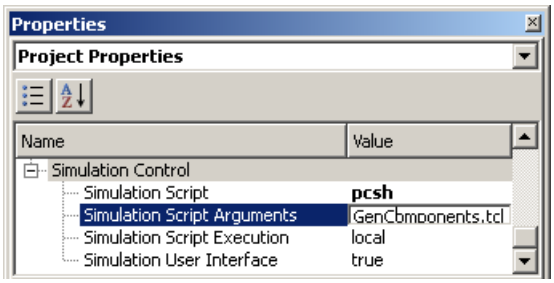


Figure 6-3 Simulation Properties for Platform Architect

6.1.1.3 Setting Simulation Parameters for SystemC

SystemC parameters, shown in Figure 6-4, can include a broad set of possible functions and values. In this example, the SystemC executable *test.exe* is set to run in verbose mode.

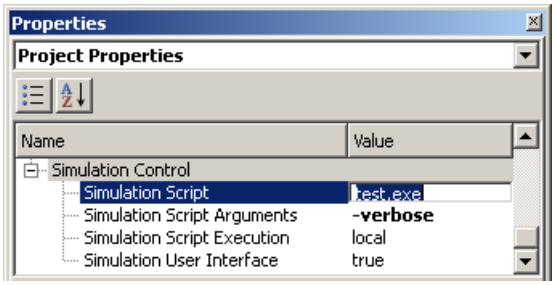


Figure 6-4 Simulation Properties for SystemC

Appendix A

ARM-supplied Transactors for SoC Designer Plus

This appendix describes the transactors and other entities that are listed in the *Transactor Type* pulldown list in the SoC Designer *Ports* tab. These entities fall into the following categories:

- **Pseudo-transactors**. These are not transactors, but they create port connections on the component. These include null ports, interrupt translators, reset inputs, and clock_inputs and outputs (see “Pseudo-Transactors” on page 158).
- **Transactors that are external to the Cycle Model**. This includes the AMBA transactors. Use *Carbon Model Studio* to add these transactors and their transaction ports to the component and connect their signals to the RTL signals (see “Transactors that are External to the Cycle Model” on page 171).
- **Transactors that are internal to the Cycle Model**. Use Carbon Model Studio to add the necessary transaction ports to the component (see “Transactors that are Internal to the Cycle Model” on page 224).

In addition, you can write your own transactor and use Carbon Model Studio to connect it to a Cycle Model. See Appendix B, User-Defined Transactors with SoC Designer Plus for more information.

A.1 Pseudo-Transactors

For convenience, the Carbon Model Studio includes a few entities in its *Add Xtor* pulldown list that are not transactors. These items include:

- Null Ports (Null_Input and Null_Output)
- Interrupt Translators (Interrupt_Master and Interrupt_Slave)
- Clock Inputs and Clock Generators
- Reset Inputs

A.1.1 Null Ports

Null_Input

This creates a component input port that is not connected to an RTL signal. This is useful when the component being generated must have the same ports as another component, even if some of the input ports are not used by the Cycle Model.

Null_Output

This creates a component output port that is not connected to an RTL signal. This is useful when the component being generated must have the same ports as another component, even if some of the output ports are not used by the Cycle Model.

A.1.2 Interrupt Translators

ARM supplies Interrupt_Slave and Interrupt_Master pseudo-transactors to translate SoC Designer interrupt signals to RTL interrupt signals. These entities are not true transactors as they lack transaction ports.

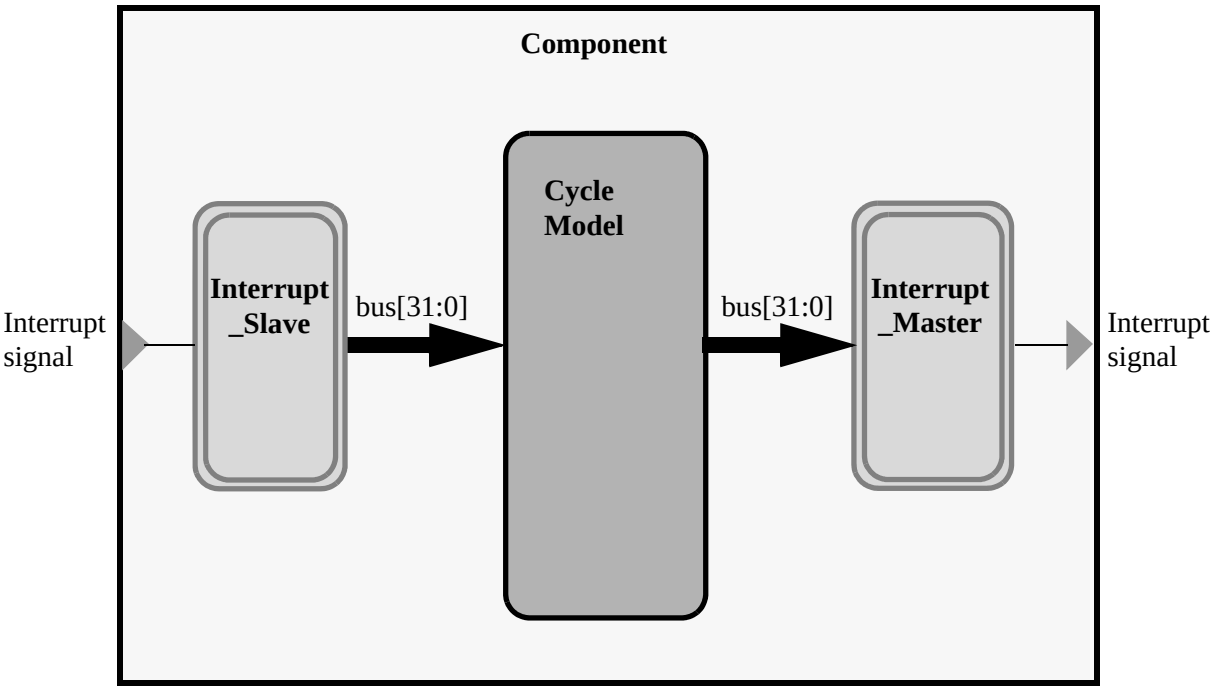


Figure A-1 Interrupt_Slave and Interrupt_Master Translators

Interrupt_Master

This is a master port that tells which interrupt signal was triggered. It receives up to 32 RTL interrupt signals and transmits that information out through its master signal transaction port. The Interrupt_Master uses the following convention:

- Each RTL signal connected to the Interrupt_Master has an ‘id’ parameter giving the `signalNumber` associated with the RTL signal.
- When the RTL signal is driven, the Interrupt_Master port’s `driveSignal` method is called with this convention:

  ```
  driveSignal(signalNumber, &rtlSignalValue);
  ```

  `signalNumber` is from the ‘id’ parameter. This can be set to -1 to disable the interrupt for the specified signal.

  `rtlSignalValue` is the value coming from the Cycle Model.

Interrupt_Slave

This is a signal slave port that drives RTL interrupt signals to the Cycle Model. The number of interrupt signals is automatically set based on the number of pins detected.

Each RTL interrupt signal maintains its state until driven by a new call to the Interrupt_Slave’s `driveSignal` method. The RTL interrupt signal is specified by the first parameter in the `driveSignal` method. The value to which the signal is set is specified by the second parameter, as follows:

```
driveSignal(signalNumber, &setOrClear);
```

`signalNumber` is the bit number of the RTL signal to modify.

`setOrClear` is 1 or 0.

For example:

```
int value = 0;
port->driveSignal(3, &value);  // clear interrupt 3
value = 1;
port->driveSignal(4, &value);  // set interrupt 4
```

The above code clears bit 3 and sets bit 4 of the RTL signal value.

A.1.3 Clock Inputs and Clock Generators

This section describes the two available clocking mechanisms — clock_inputs and clock_generators — and when to use them.

- Clock_input
- Clock_generator
- Clock-related Parameters
- Clock_output

A.1.3.1 Clock_input

A *Clock_Input* creates a component input port that you can use to clock a component data port. It is used to drive an RTL clock input of the Cycle Model, and/or control the update frequency of a transactor. Clock_inputs create a 1:1 pulse-to-cycle ratio. To specify a slower frequency, use a *CDIV* (clock divider).

For instructions on creating a:

- Clock_input — Refer to “Adding Transactors and Other Interface Entities” on page 103.
- Clock divider — Refer to the *SoC Designer Plus User Guide*.

Consider the example shown in Figure A-5, below.

Two clock dividers (CDIV1 and CDIV2) are used in combination with clock_inputs AHBclk1 and AHBclk2. CDIV1 specifies a pulse -to-cycle ratio of 1:2, and CDIV2 specifies a pulse-to-cycle ratio of 1:3. The output clocks of each clock_input initiate a 1:1 pulse frequency to the AHBclk1 and AHBclk2 ports on the Cycle Model.

In effect, this design runs the AHBclocks at 1/2 and 1/3 the frequency of the pulse, respectively.

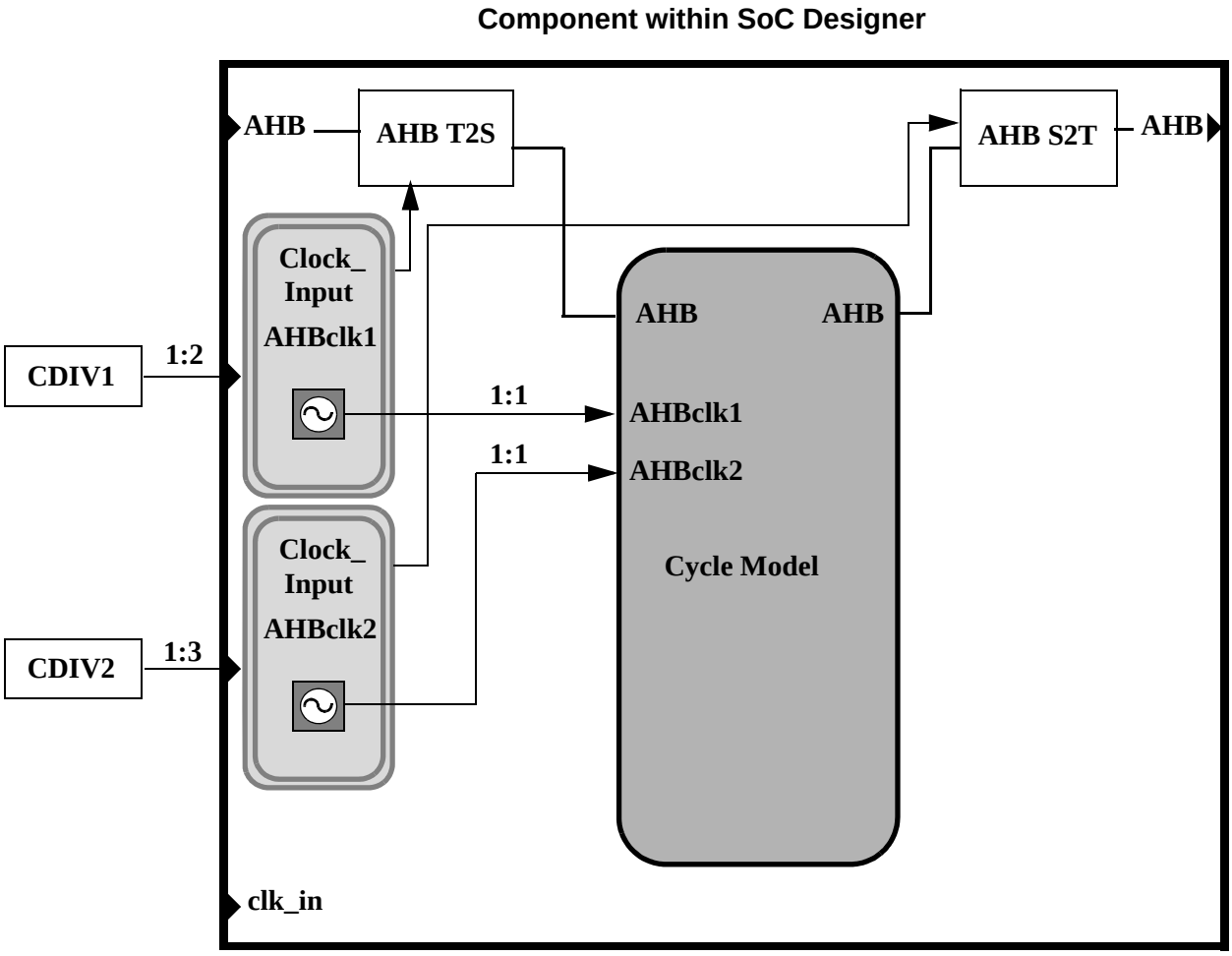


Figure A-2 Clock_inputs

Figure A-3, below, shows the waveforms generated when the CDI_S2T are used in combination with clock_inputs to run AXI clocks at the frequencies shown in Figure A-2. Note that the waveforms are generated based on the CDIV pulses, not the master SoC Designer clock pulses.

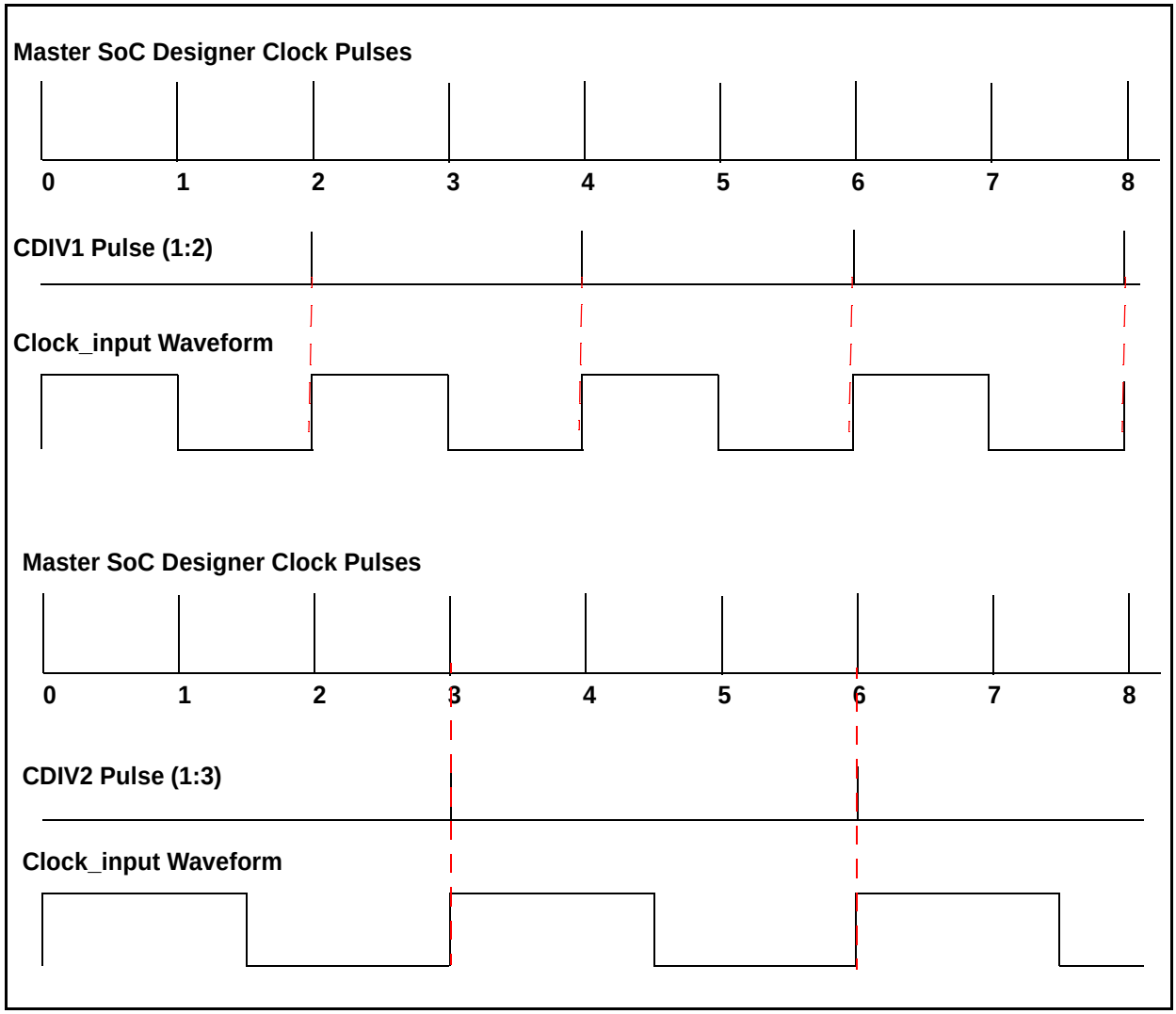


Figure A-3 CDIVs and Clock_Inputs

Using the Clock_input Enable pin

Clock_input allows RTL and transactors to be clocked synchronously; the Enable pin allows you to further refine your clocking conditions. By specifying a high or low Enable signal, in combination with code that defines your requirements, you determine the circumstances in which the RTL executes.

For example, you may want to power off/power on a device during a reset sequence. Connecting the Enable pin to an MxStub component with code that sets Enable to *low* at time Zero disables clocking during the reset sequence. When the reset sequence ends, the MxStub code sets Enable to *high*, re-enabling clocking.

To use the Enable pin:

1. Create a top-level component signal slave port to act as the Enable for the Clock_Input (in Figure A-4, **user_enable** is the signal slave port).
2. Connect the new signal slave port to a scalar enable input signal to the RTL. This requires the RTL to use the Enable signal to qualify execution of the RTL code. In Figure A-4, the RTL signal is named **rtl_enable**.
3. Drive the top-level Enable signal (**user_enable** in Figure A-4) from the SoCD Canvas.

   If the top-level Enable signal is left undriven, it defaults to Zero, so the associated transactors within the component (and the qualified RTL code) never execute.

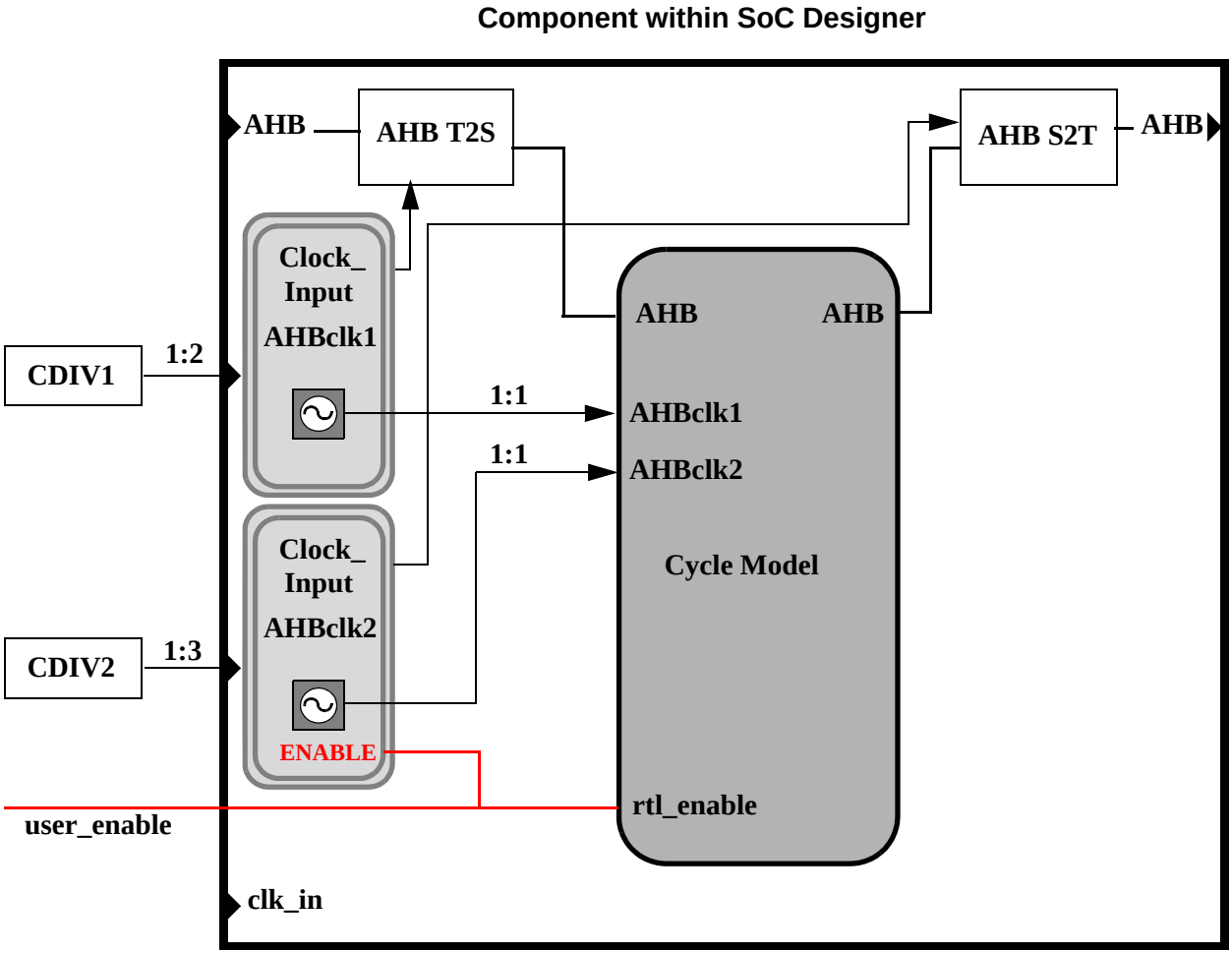


Figure A-4 Use of the Enable pin

Note: When creating a SoC Designer component wrapper, if you leave the Enable pin to the Clock_input transactor unconnected, the Clock_input transaction is always enabled, and no Enable port exists on the SoCD component.

A.1.3.2 Clock_generator

A *clock_generator* creates an internal clock, allowing you to specify how you want to map component cycles to clock cycles. Clock_generators support the creation of various pulse intervals, so that you can create multiple frequencies for internal logic.

Note: When possible, ARM recommends creating a clock input transactor and exposing the clock input port externally, rather than creating a clock generator. The clock input approach tends to simplify your design.

However, the clock generator approach may be required for certain configurations. For example:

- *Memory controller with PHY and DDR — Large ratio values are required for a CDIV-driven clock_input, due to the odd ratio clock of the PHY. This results in inefficient simulation time.*
- *Non-transactor I/Os — These ports are read and driven via the* `update()` *and* `communicate()` *methods registered to the component's own default clock input. The RTL logic directly connected to these ports must be driven by the same clock.*

In Figure A-5 below, the component's input clock (clk_in) is connected to the master clock, so that the component receives periodic clock ticks from the SoC Designer master clock. You can connect multiple clock_generators to clk_in; in this example, two clock_generators produce 1:1 and 1:2 pulse-to-cycle ratios to the Cycle Model.

For instructions on creating a clock_generator, refer to "Specifying Generated Clocks" on page 100.

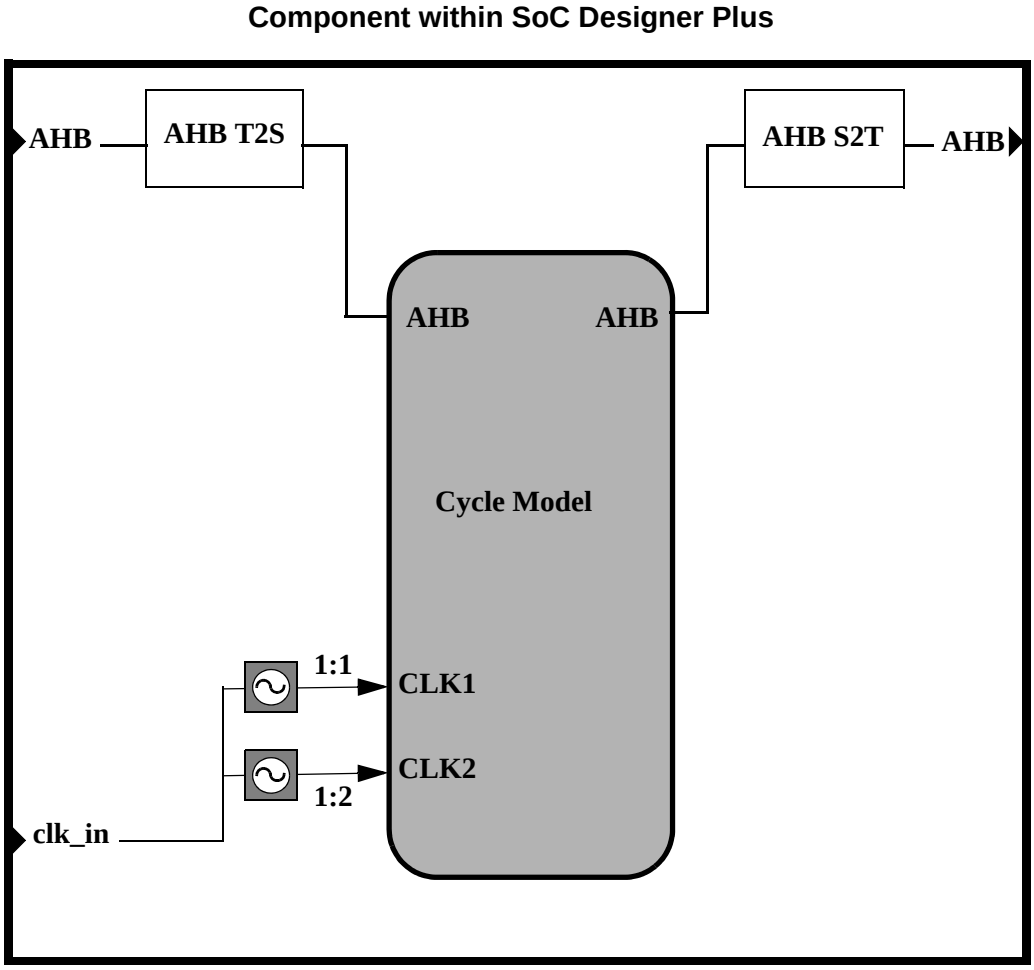


Figure A-5 Clock_generators

Figure A-6, below, shows the waveforms generated when clock_generators specify clock cycles that either match the component cycle (1:1 ratio) or differ from the component cycle (1:2), as shown in Figure A-5.

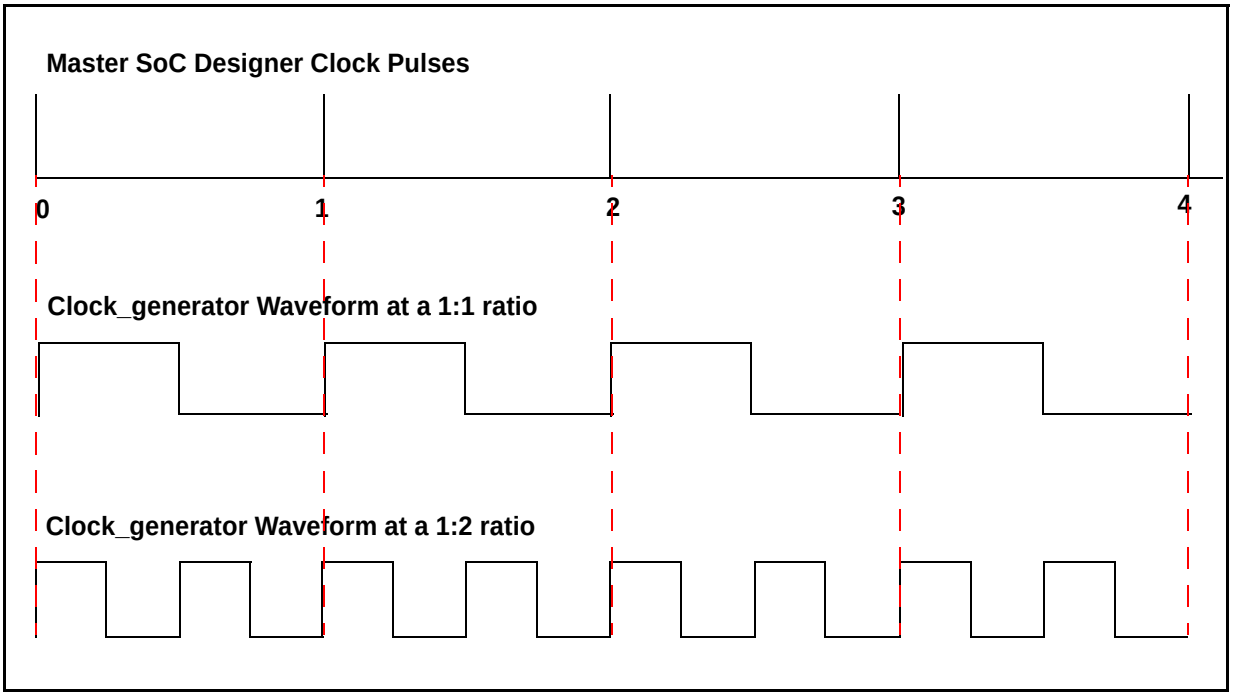


Figure A-6 Clock_generator Waveforms at Different Ratios

A.1.3.3 Clock-related Parameters

During reset, clocks are generated according to the parameters in Table A.1.

Note: The Enable element (ENABLE) that appears at the bottom of the Clock_Input parameter list is in fact a pin for the transactor port, not a parameter. Refer to "Using the Clock_input Enable pin" *on page 162* *for information about its use.*

Table A.1: Parameters for Clock_inputs and Clock_generators

Parameter	Description	Clock_input	Clock_generator
Clock Cycles[a]	The frequency of the clock cycle.	Fixed at a 1:1 ratio.	Any ratio.
Clock Port Visible	Determines whether the component displays the clock slave port.	True or False.	Not supported.
Comp Cycles[a]	The frequency of the component cycle.	Fixed at a 1:1 ratio.	Any ratio.
Delay	The percentage of a component cycle to delay before generating a clock. See Figure A-8 below.	Delay is applied at every activation.	Not supported.
Delay Enable	Determines whether changes to an enable during a cycle are applied to the next component cycle.	True or False.	Not supported.
Duty Cycle	The percentage of the clock cycle for which the clock value is high. See Figure A-9 below.	0 — 99	0 — 99
Initial Value	Clock initial value.	Low (0) or High (1)	Low (0) or High (1)
Initial Delay	The percentage of a component cycle to delay before generating a clock. See Figure A-7 below.	Not supported.	Initial Delay is applied on initial edge only.

a. Clock Frequency during reset is *Clock Cycles per Component Cycles*.

Figure A-7 shows a waveform with a 2:1 ratio and an Initial Delay set to Fifty Percent.

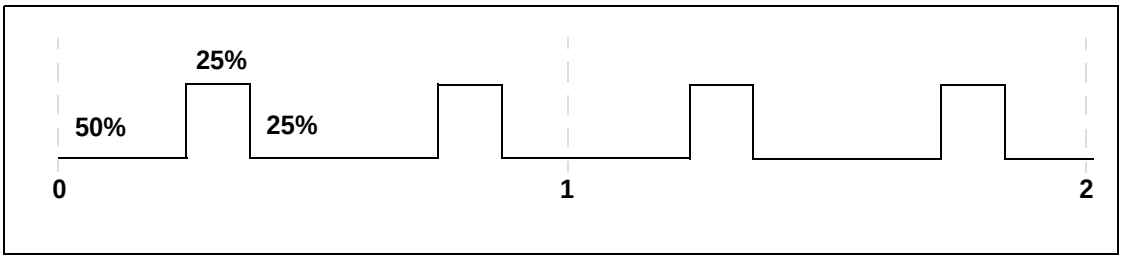


Figure A-7 Wave with an Initial Delay of 50%

Figure A-8 illustrates a waveform with a 1:1 ratio, with Delay set to Zero percent and Fifty percent.

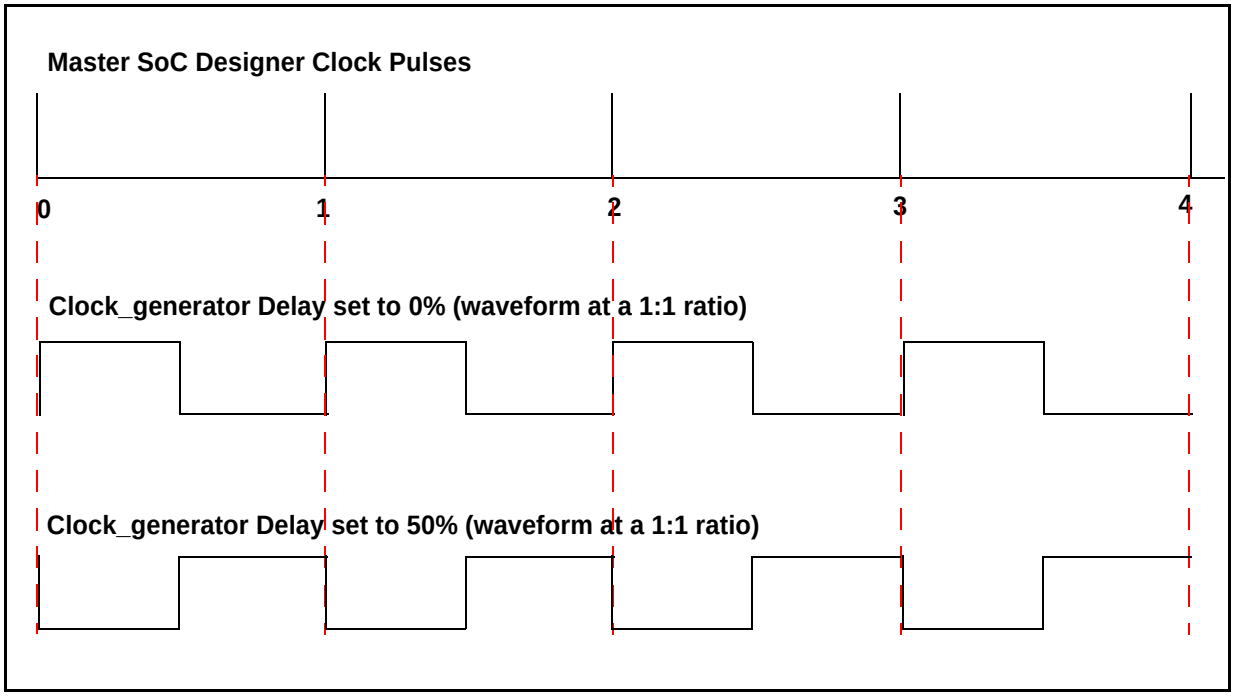


Figure A-8 Wave with Delay set to 0% percent and 50%

Figure A-9 shows a waveform with a 2:1 ratio and Duty Cycle set to Twenty-Five percent.

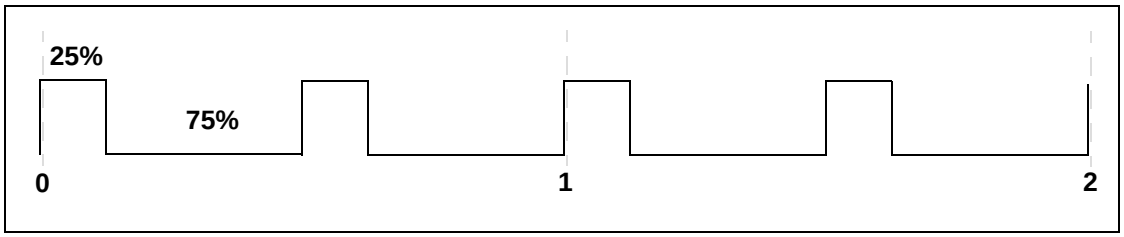


Figure A-9 Wave with a Duty Cycle of 25%

A.1.3.4 Clock_output

Clock_output creates a component output port that you can connect to a clock output of the Cycle Model.

If your Cycle Model generates an output clock, you must connect a Clock_Output to that clock.

In SoC Designer Plus, all clock slaves that are connected to the `Clock_Output` port are updated on every cycle on which a posedge value change is observed on the Cycle Model clock output.

For instructions on creating a clock_output, refer to "Adding Transactors and Other Interface Entities" on page 103.

A.1.4 Reset Inputs

Reset Inputs create a reset input port that you can connect in an RTL input of the Cycle Model. This transactor can be used when an automatic reset pulse is desired during the reset sequence, but also allows the ability to manually drive the reset during simulation.

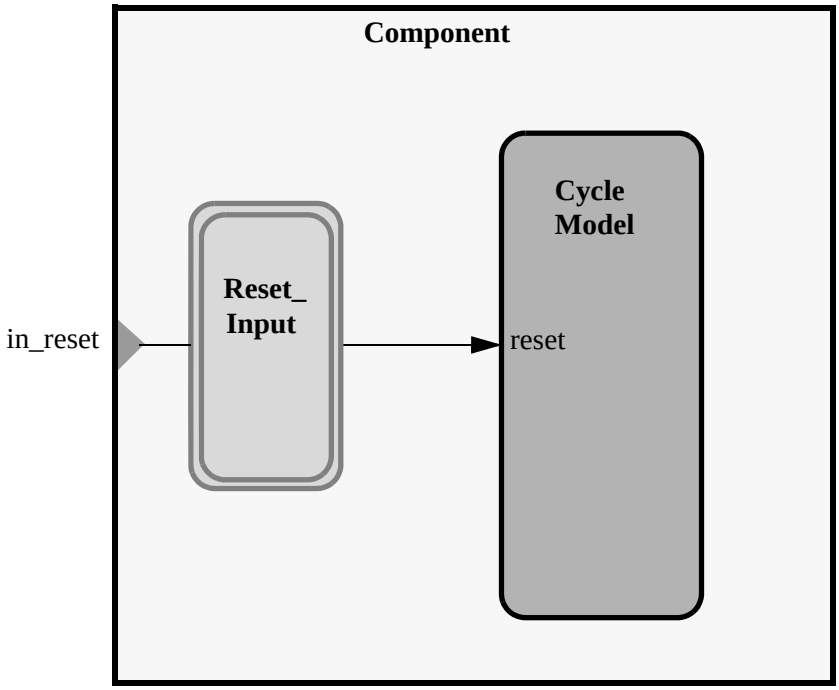


Figure A-10 Reset_Inputs

Table A.2 describes the Reset_Input parameters.

Table A.2: Parameters for Reset_Inputs

Parameter	Description
Active High	Determines whether the reset is Active High (set to true) or Active Low (set to false).
Active Value[a]	Specifies a decimal value to be used when reset is active (asserted). The default is 1.
Clock Cycles	The frequency during the reset sequence is determined by the ratio *Clock Cycles:Comp (Component) Cycles*. By default this is a 1:1 ratio.
Comp Cycles	
Cycles After	The minimum number of cycles the signal remains deasserted before the reset period ends.
Cycles Asserted	The number of cycles during which the signal remains asserted.
Cycles Before	The number of cycles to wait during the reset period before asserting the signal.
Inactive Value[a]	Specifies a decimal value to be used when reset is inactive (deasserted). The default is 0.

Table A.2: Parameters for Reset_Inputs (continued)

Parameter	Description
Use Active Value	Determines whether to use the Active / Inactive Value (set to true) or the Active High parameter (set to false).

a. The Active Value and Inactive Value parameters are used to drive a vector of resets. The vector can be a scalar value. Vectors are sometimes used by models with multiple cores that require individual resets.

   The Active Value parameter is the value used when asserting reset, and the Inactive Value parameter is used when reset is not asserted. Typically, the active and inactive values are the same across all cores of the model. So for example, a model with four subcores using Active Low resets would have an Active Value of 0 and an Inactive Value of 15.

A.2 Transactors that are External to the Cycle Model

The transactors described in this section exist inside the component, but outside of the Cycle Model. This includes the ARM AMBA transactors. Use Carbon Model Studio to add these transactors and connect their ports to the appropriate RTL ports in the Cycle Model.

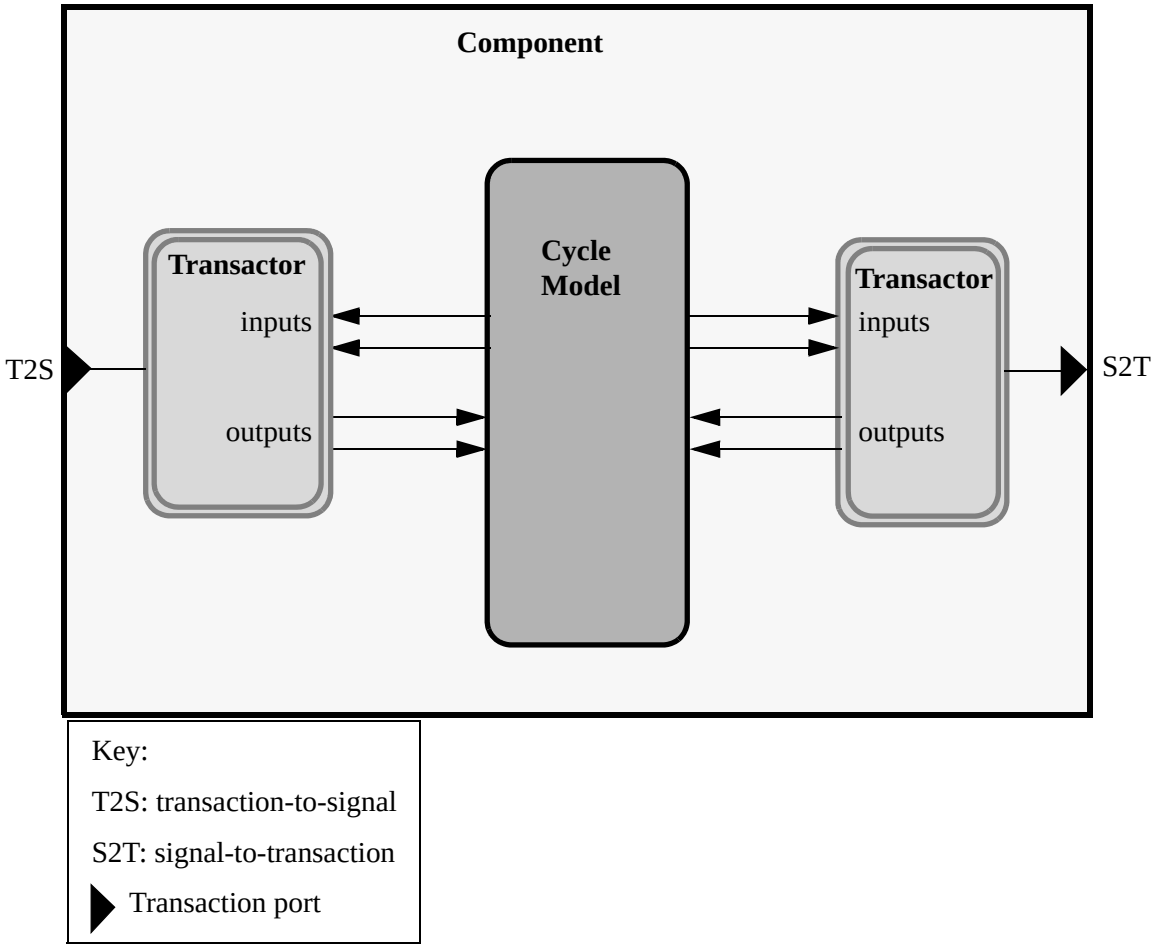


Figure A-11 Generic Transactor External to Cycle Model

Definitions

S2T stands for signal-to-transaction. The Cyclecomponent drives the transaction into SoC Designer Plus. The Cyclecomponent sends the transaction through the transaction master port. SoC Designer Plus usually *receives* the transaction through signal slave ports.

T2S stands for transaction-to-signal. SoC Designer Plus drives the transaction into the Cyclecomponent. The Cyclecomponent receives the transaction through the transaction slave port. SoC Designer Plus usually *sends* the transaction through signal master ports.

Arbiter: The arbiter regulates the bus masters so that only one bus master at a time can initiate data transfer.

AHB_*Master*_S2T|T2S, AHB_*Slave*_S2T|T2S: The term *Master* or *Slave* that appears in the transactor name refers to the master or slave side of the AHB interface. In the next four diagrams, the AHB_*Master*_S2T|T2S transactors appear to the left of the arbiter (the AHB master

interface has the request/grant signals). The AHB_*Slave*_S2T|T2S transactors appear to the right of the arbiter (the AHB slave interface does not have the request/grant signals).

Scope: The scope of each transactor parameter is one of the following values:

- **Compile-Time**. The parameter has to be set when instantiating the adaptor in Carbon Model Studio. The parameter cannot be set in the SoC Designer Canvas.
- **Init-Time**. The parameter can be changed in the SoC Designer Canvas.
- **Run-Time**. The parameter can be changed at runtime during simulation.

The following transactors are discussed in this section:

- "AHB Transactors" on page 173
- "APB Transactors" on page 193
- "APB3 Transactors" on page 196
- "AXI Transactors" on page 204
- "ARM926_DTCM Transactors" on page 213
- "CHI Transactors" on page 215

Note: The AHB and AXI transactors are available in two different versions: the original version (v1), and the newer version (v2) that supports flow through. ARM recommends that you use the v2 transactors for new projects, if possible, because the non-flow-through transactors will be deprecated in a future version of Carbon Model Studio.

A.2.1 AHB Transactors

There are two variants of the AHB transactors:

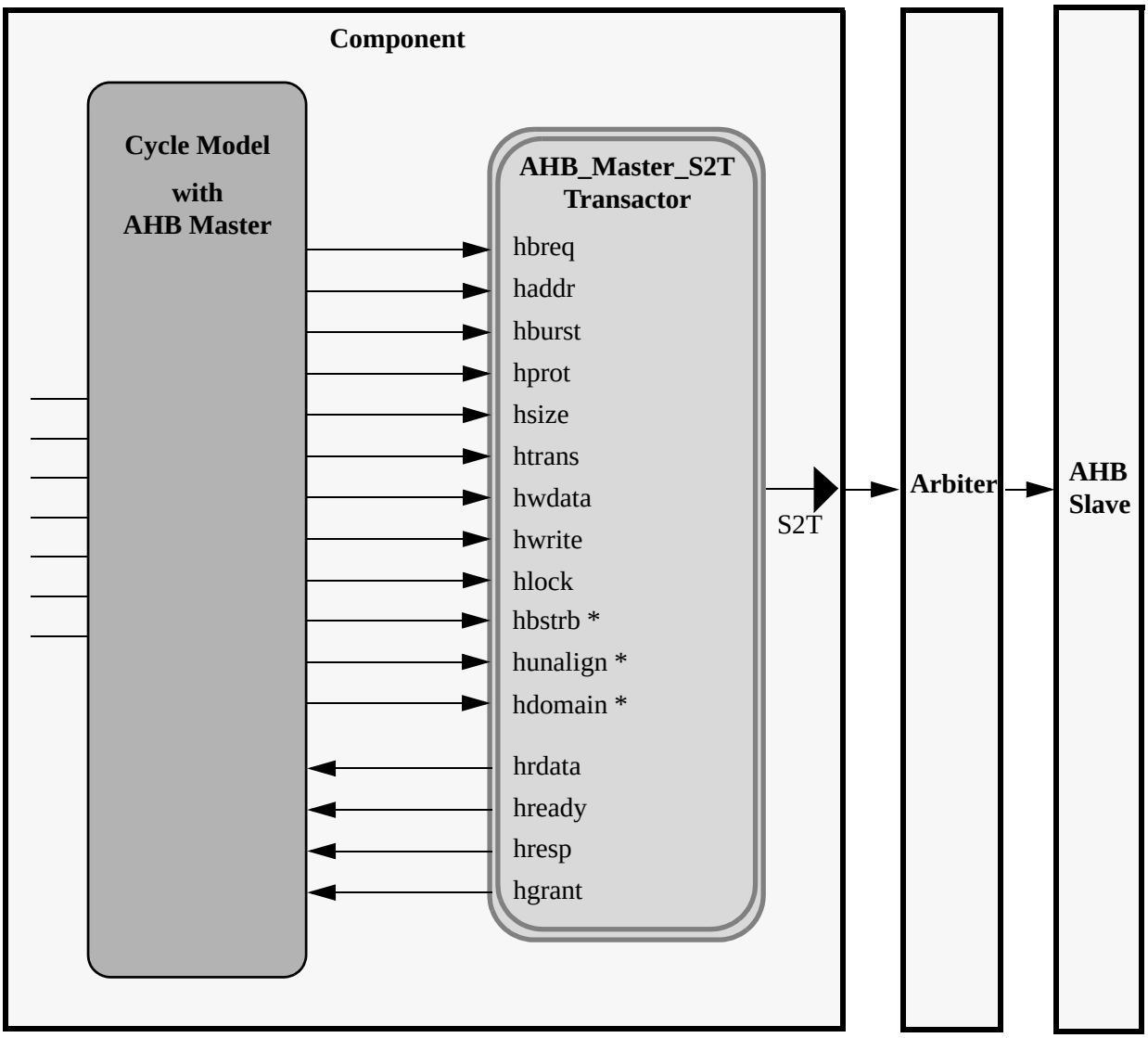
- *AHB_Master_xxx* and *AHB_Slave_xxx*: These are the original AHB transactors that have been available through Carbon Model Studio for many releases.
- *AHB_Master_Fxxx/AHB_Lite_Master_Fxxx* and *AHB_Slave_Fxxx/AHB_Lite_Slave_Fxxx*: These flow-through versions of the AHB transactors (v2) should be used if the Cycle Model has flow through (asynchronous) paths in the design.

This section describes:

- AHB_Master_S2T and AHB_Master_FS2T
- AHB_Slave_T2S and AHB_Slave_FT2S
- AHB_Master_T2S and AHB_Master_FT2S
- AHB_Slave_S2T and AHB_Slave_FS2T
- AHB_Lite_Master_S2T and AHB_Lite_Master_FS2T
- AHB_Lite_Master_FT2S
- AHB_Lite_Slave_T2S and AHB_Lite_Slave_FT2S
- AHB_Lite_Slave_FS2T

A.2.1.1 AHB_Master_S2T and AHB_Master_FS2T

A transactor that converts AHB Master interface signals from the Cycle Model into transactions that are communicated through the transaction master port. S2T stands for signal-to-transaction. These transactors can communicate with SoC Designer Plus components that implement the CASI AHB transaction interface.



* Optional signals for ARM11 extension

Figure A-12 AHB_Master_S2T Transactor

AHB_Master_S2T and AHB_Master_FS2T parameters

Parameter: `Enable Debug Messages`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: When set to *true*, the adaptor outputs debug messages while running.

Parameter: `Big Endian`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Init-time

Description: Affects the alignment of the data on the data bus. When *true*, the data is read as “big endian” on the data bus.

Parameter: `Align Data`

Parameter Type: boolean

Legal Values: true, false Default = *false*.

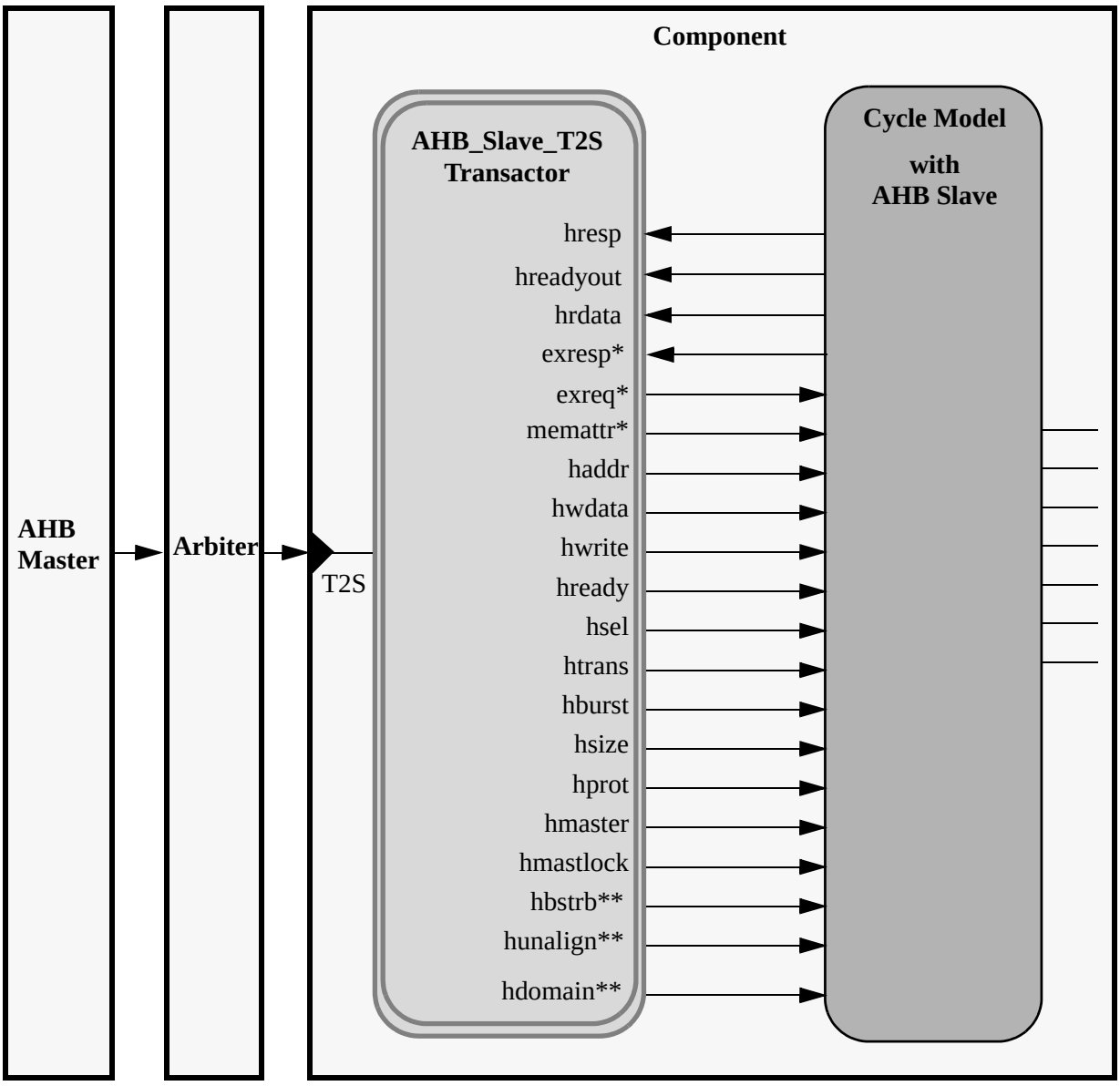
Scope: Init-time

Description: For narrow transfers on the AHB bus, SoC Designer Plus outputs data aligned to the low bytes, even though the protocol states that data for an unaligned address should be output on the correct byte lane. Setting this parameter to *true* enables aligning of the data to the correct byte lane.

Note: This field is no longer used, but is maintained for backwards compatibility.

A.2.1.2 AHB_Slave_T2S and AHB_Slave_FT2S

A transactor that converts AHB transactions from a transaction slave port into AHB Slave interface signals in the Cycle Model. T2S stands for transaction-to-signal. These transactors can communicate with SoC Designer Plus components that implement the CASI AHB transaction interface.



* AHBv2 sideband signals for the Cortex-M3. For details see the *AHBv2 Protocol Bundle User Guide* or the *ARM Cortex-M3 r2P0 TRM*.

** Optional signals for ARM11 extension

Figure A-13 AHB_Slave_T2S Transactor

AHB_Slave_T2S and AHB_Slave_FT2S parameters

Parameter: `Enable Debug Messages`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: When set to *true*, the adaptor outputs debug messages while running.

Parameter: `Base Address` (for AHB_Slave_T2S)

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = 0.

Scope: Init-time

Description: Base address of the AHB Slave memory region 0.

Parameter: `ahb_start1` (for AHB_Slave_T2S)

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = 0.

Scope: Init-time

Description: Base address of the AHB Slave memory region 1.

Parameter: `region start [0-5]` (for AHB_Slave_FT2S)

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = 0x0.

Scope: Init-time

Description: Base address of the AHB Slave memory regions 0 through 5.

Parameter: `Size` (for AHB_Slave_T2S)

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = null.

Scope: Init-time

Description: Size of memory region 0.

Parameter: `ahb_size1` (for AHB_Slave_T2S)

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = 32.

Scope: Init-time

Description: Size of memory region 1.

Parameter: `region size [0-5]` (for AHB_Slave_FT2S)

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = 0x100000000 for region 0
0x0 for regions 1-5

Scope: Init-time

Description: Size of memory regions 0 through 5.

Parameter: `Subtract Base Address` (for AHB_Slave_T2S)

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Init-time

Description: When set to *true*, the `Base Address` is subtracted from the transaction address, and the offset is sent to the RTL port. When set to *false* (the default value), the transaction address is sent to the RTL port unchanged.

Parameter: `Subtract Base Address Dbg` (for AHB_Slave_T2S)

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Init-time

Description: Set this parameter when using the readDbg and writeDbg members of the transactor ports. When set to *true*, the `Base Address` is subtracted from the transaction address, and the offset is sent to the RTL port. When set to *false* (the default value), the transaction address is sent to the RTL port unchanged.

Parameter: `Big Endian`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Init-time

Description: Affects the alignment of the data on the data bus. When *true*, the data is output “big endian” on the data bus.

Parameter: `Align Data`

Parameter Type: boolean

Legal Values: true, false Default = *false*.

Scope: Init-time

Description: For narrow transfers on the AHB bus, SoC Designer Plus outputs data aligned to the low bytes, even though the protocol states that data for an unaligned address should be output on the correct byte lane. Setting this parameter to *true* enables aligning of the data to the correct byte lane.

Note: This field is no longer used, but is maintained for backwards compatibility.

Parameter: `Filter HREADYIN`

Parameter Type: boolean

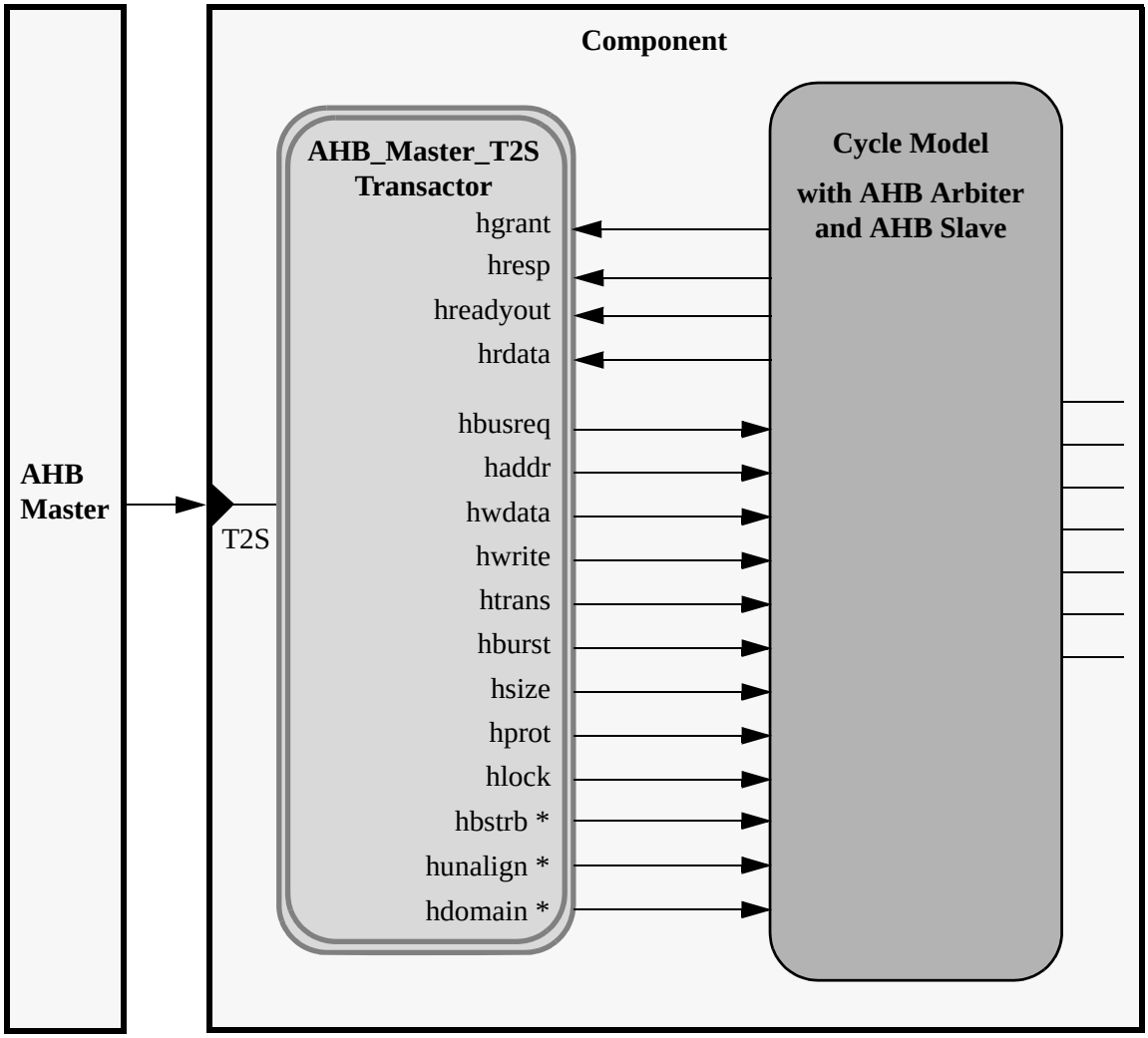
Legal Values: true, false. Default = *false*.

Scope: Init-time

Description: AHB slaves are required to have an HREADY signal as both an input and an output. The output is the slave’s ready status, while the input is the ready status of all the slaves on the bus (presumably calculated by the bus arbiter). Setting this parameter to *true* filters out the input HREADY “HREADIN” to prevent it from reaching the Cycle Model. The parameter should be set only when it is not required by the slave for correct operation.

A.2.1.3 AHB_Master_T2S and AHB_Master_FT2S

A transactor that converts AHB transactions from a transaction slave port into AHB Master interface signals in the Cycle Model. These transactors can communicate with SoC Designer Plus components that implement the CASI AHB transaction interface.



* Optional signals for ARM11 extension

Figure A-14 AHB_Master_T2S Transactor

AHB_Master_T2S and AHB_Master_FT2S parameters

Parameter: `Enable Debug Messages`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: When set to *true*, the adaptor outputs debug messages while running.

Parameter: `Base Address` (for AHB_Master_T2S)

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = 0.

Scope: Init-time

Description: Base address of the AHB Slave memory region.

Parameter: `region start [0-5]` (for AHB_Master_FT2S)

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = 0x0.

Scope: Init-time

Description: Base address of the AHB Slave memory regions 0 through 5.

Parameter: `Size` (for AHB_Master_T2S)

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = null.

Scope: Init-time

Description: Size of memory region.

Parameter: `region size [0-5]` (for AHB_Master_FT2S)

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = 0x100000000 for region 0
0x0 for regions 1-5

Scope: Init-time

Description: Size of memory regions 0 through 5.

Parameter: `ahbm port ID`

Parameter Type: value

Legal Values: 0 - Max ID of the connected arbiter. Default = 0.

Scope: Init-time

Description: Port ID used when sending a bus request to the arbiter.

Parameter: `Big Endian`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Init-time

Description: Affects the alignment of the data on the data bus. When *true*, the data is output “big endian” on the data bus.

Parameter: `Align Data`

Parameter Type: boolean

Legal Values: true, false Default = *false*.

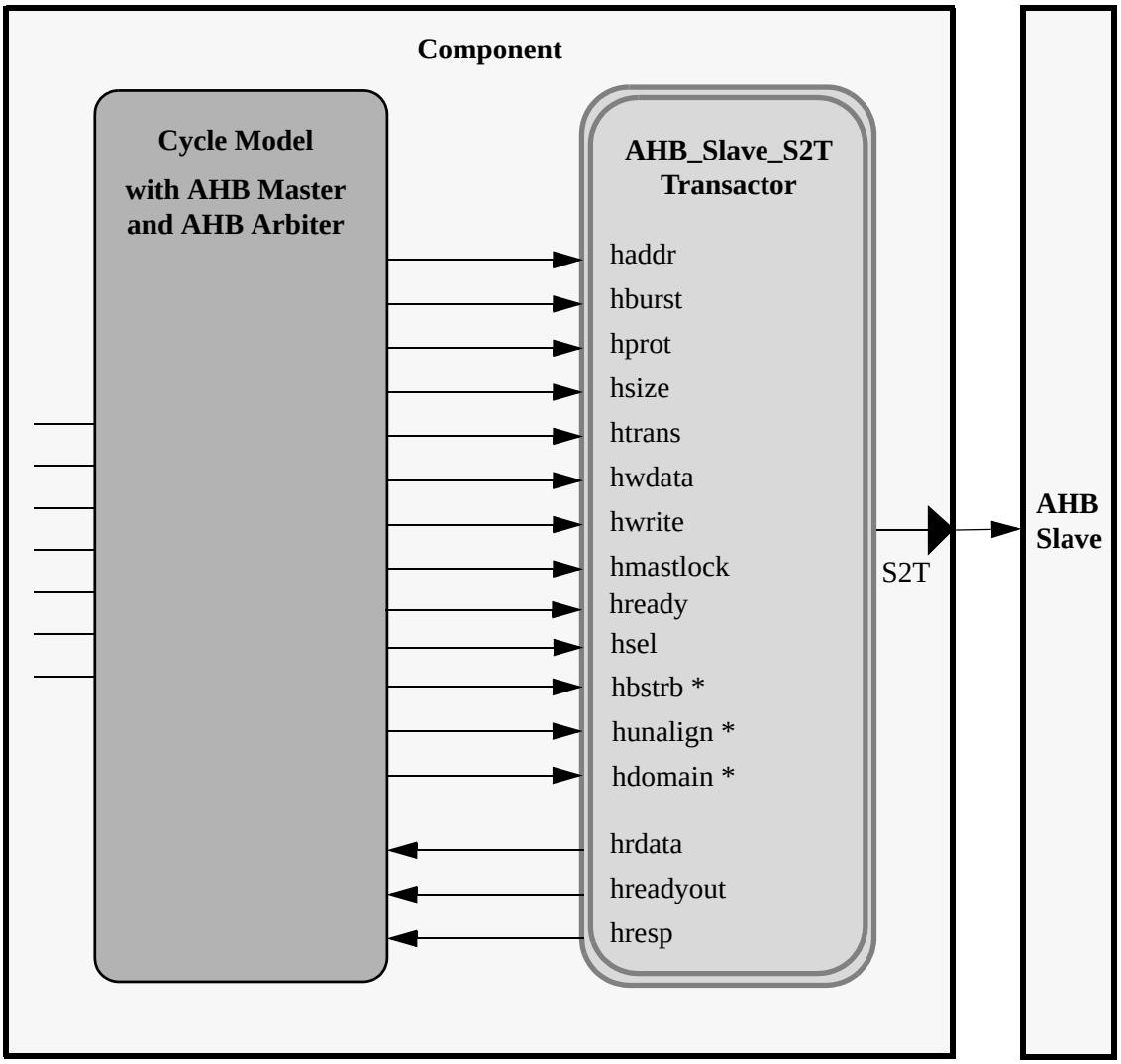
Scope: Init-time

Description: For narrow transfers on the AHB bus, SoC Designer Plus outputs data aligned to the low bytes, even though the protocol states that data for an unaligned address should be output on the correct byte lane. Setting this parameter to *true* enables aligning of the data to the correct byte lane.

Note: This field is no longer used, but is maintained for backwards compatibility.

A.2.1.4 AHB_Slave_S2T and AHB_Slave_FS2T

A transactor that converts AHB Slave interface signals from the Cycle Model into transactions that are communicated through the transaction master port. These transactors can communicate with SoC Designer Plus components that implement the CASI AHB transaction interface.



* Optional signals for ARM11 extension

Figure A-15 AHB_Slave_S2T Transactor

AHB_Slave_S2T and AHB_Slave_FS2T parameters

Parameter: `Enable Debug Messages`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: When set to *true*, the adaptor outputs debug messages while running.

Parameter: `Big Endian`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Init-time

Description: Affects the alignment of the data on the data bus. When *true*, the data is read as “big endian” on the data bus.

Parameter: `Align Data`

Parameter Type: boolean

Legal Values: true, false Default = *false*.

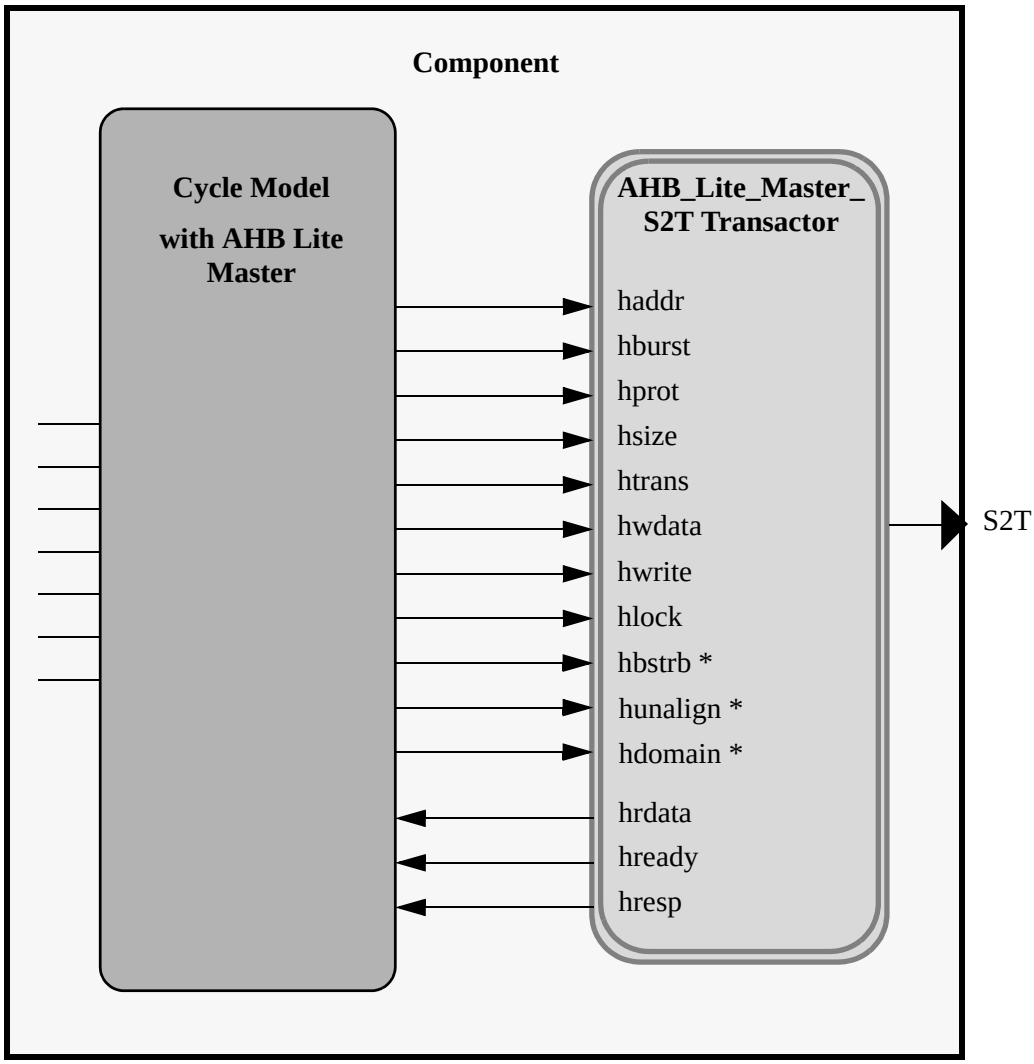
Scope: Init-time

Description: For narrow transfers on the AHB bus, SoC Designer Plus outputs data aligned to the low bytes, even though the protocol states that data for an unaligned address should be output on the correct byte lane. Setting this parameter to *true* enables aligning of the data to the correct byte lane.

Note: This field is no longer used, but is maintained for backwards compatibility.

A.2.1.5 AHB_Lite_Master_S2T and AHB_Lite_Master_FS2T

A transactor that converts AHB-Lite Master interface signals from the Cycle Model into transactions that are communicated through the transaction master port. These transactors can communicate with SoC Designer Plus components that implement the CASI AHB transaction interface.



* Optional signals for ARM11 extension

Figure A-16 AHB_Lite_Master_S2T Transactor

AHB_Lite_Master_S2T and AHB_Lite_Master_FS2T parameters

Parameter: `Enable Debug Messages`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: When set to *true*, the adaptor outputs debug messages while running.

Parameter: `Big Endian`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Init-time

Description: Affects the alignment of the data on the data bus. When *true*, the data is read as “big endian” on the data bus.

Parameter: `Align Data`

Parameter Type: boolean

Legal Values: true, false Default = *false*.

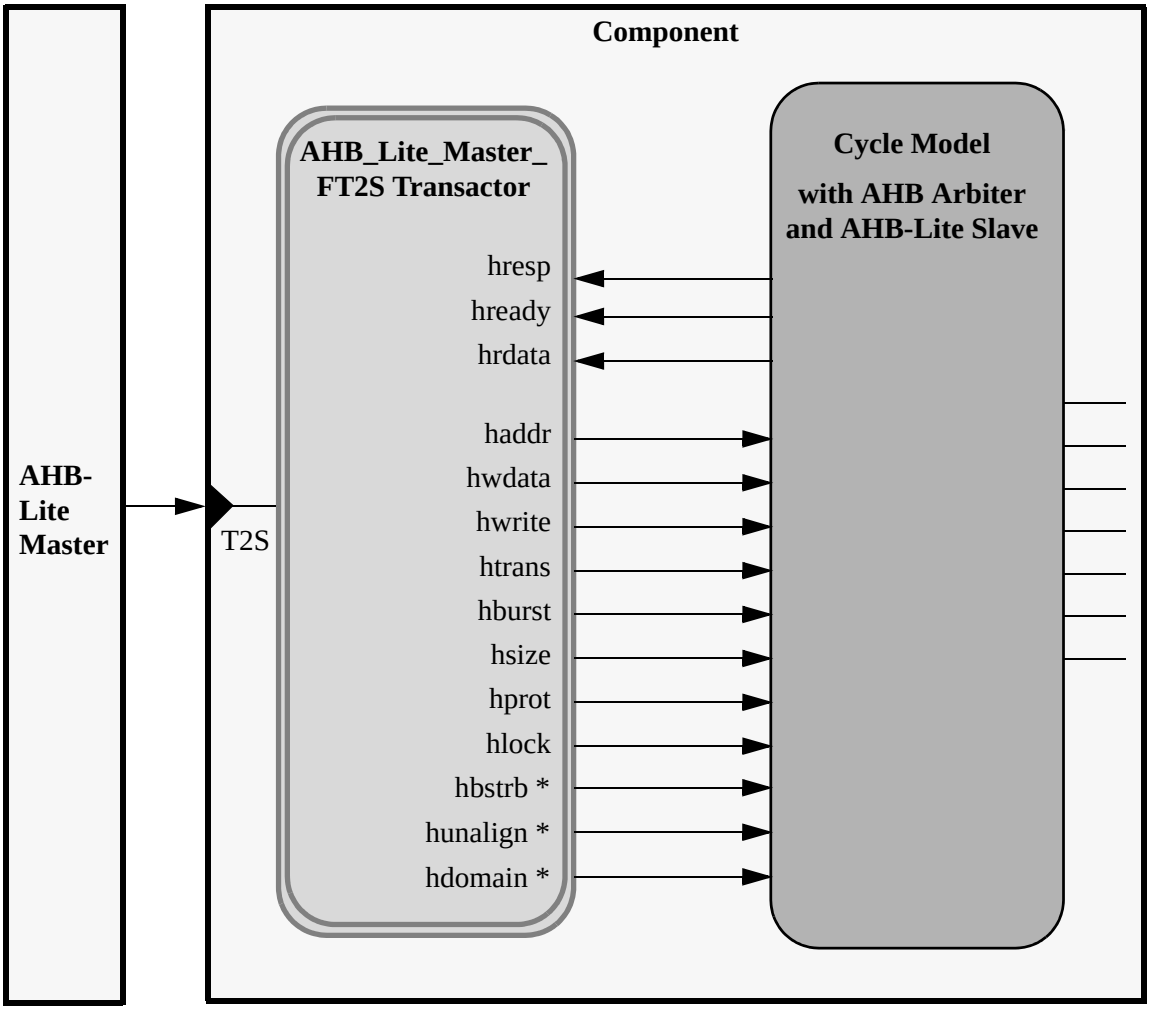
Scope: Init-time

Description: For narrow transfers on the AHB bus, SoC Designer Plus outputs data aligned to the low bytes, even though the protocol states that data for an unaligned address should be output on the correct byte lane. Setting this parameter to *true* enables aligning of the data to the correct byte lane.

Note: This field is no longer used, but is maintained for backwards compatibility.

A.2.1.6 AHB_Lite_Master_FT2S

A transactor that converts AHB-Lite Master transactions from a transaction slave port into AHB Master interface signals in the Cycle Model. These transactors can communicate with SoC Designer Plus components that implement the CASI AHB transaction interface.



* Optional signals for ARM11 extension

Figure A-17 AHB_Lite_Master_FT2S Transactor

AHB_Lite_Master_FT2S parameters

Parameter: `Enable Debug Messages`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: When set to *true*, the adaptor outputs debug messages while running.

Parameter: `region start [0-5]`

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = 0x0.

Scope: Init-time

Description: Base address of the AHB memory regions 0 through 5.

Parameter: `region size [0-5]`

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = 0x100000000 for region 0
0x0 for regions 1-5

Scope: Init-time

Description: Size of memory regions 0 through 5.

Parameter: `Big Endian`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Init-time

Description: Affects the alignment of the data on the data bus. When *true*, the data is output “big endian” on the data bus.

Parameter: `ahbm port ID`

Parameter Type: value

Legal Values: 0 - Max ID of the connected arbiter. Default = 0.

Scope: Init-time

Description: Port ID used when sending a bus request to the arbiter

Parameter: `Align Data`

Parameter Type: boolean

Legal Values: true, false Default = *false*.

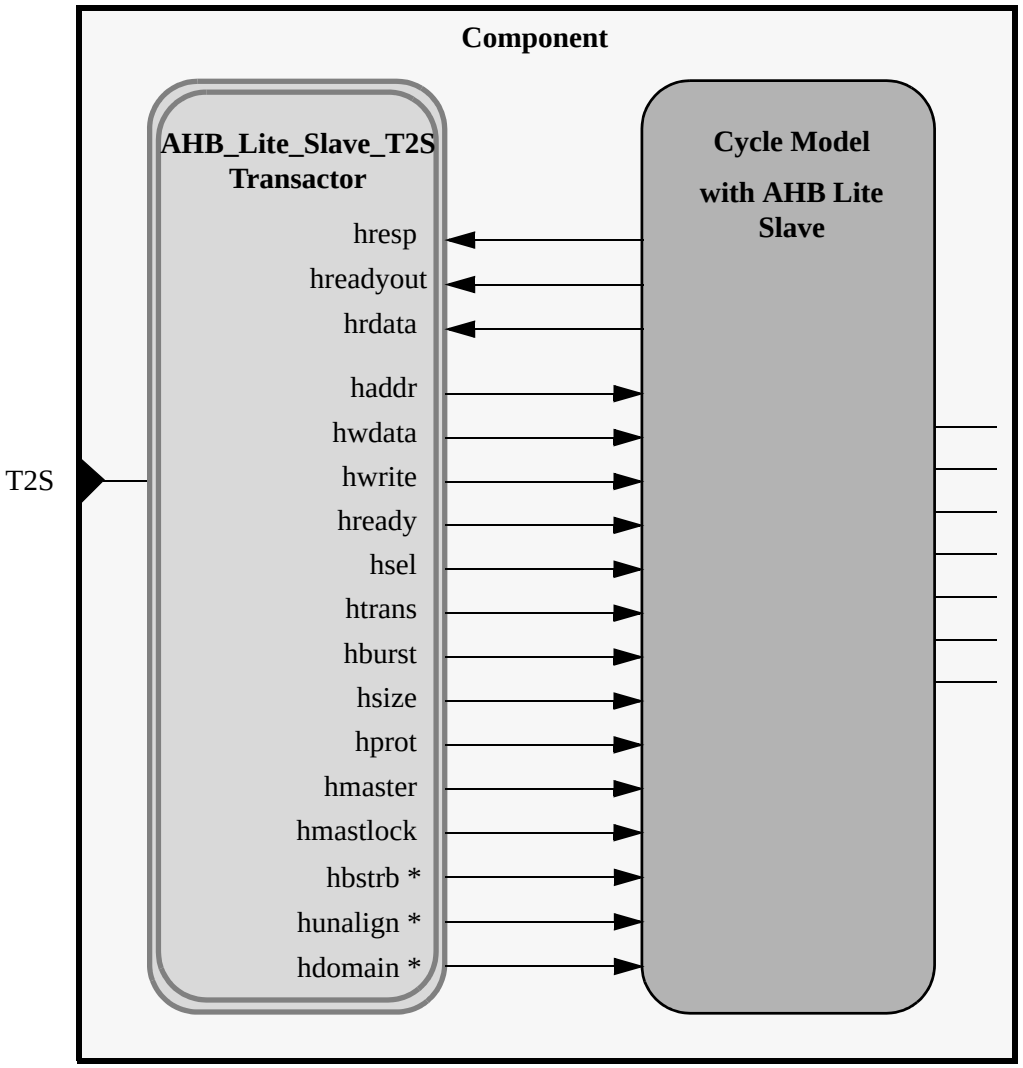
Scope: Init-time

Description: For narrow transfers on the AHB bus, SoC Designer Plus outputs data aligned to the low bytes, even though the protocol states that data for an unaligned address should be output on the correct byte lane. Setting this parameter to *true* enables aligning of the data to the correct byte lane.

Note: This field is no longer used, but is maintained for backwards compatibility.

A.2.1.7 AHB_Lite_Slave_T2S and AHB_Lite_Slave_FT2S

A transactor that converts AHB transactions from a transaction slave port into AHB-Lite Slave interface signals in the Cycle Model. These transactors can communicate with SoC Designer Plus components that implement the CASI AHB transaction interface.



* Optional signals for ARM11 extension

Figure A-18 AHB_Lite_Slave_T2S Transactor

AHB_Lite_Slave_T2S and AHB_Lite_Slave_FT2S parameters

Parameter: `Enable Debug Messages`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: When set to *true*, the adaptor outputs debug messages while running.

Parameter: `Base Address` (for AHB_Lite_Slave_T2S)

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = 0.

Scope: Init-time

Description: Base address of the AHB Slave memory region 0.

Parameter: `ahb_start1` (for AHB_Lite_Slave_T2S)

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = 0.

Scope: Init-time

Description: Base address of the AHB Slave memory region 1.

Parameter: `region start [0-5]` (for AHB_Lite_Slave_FT2S)

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = 0x0.

Scope: Init-time

Description: Base address of the AHB Slave memory regions 0 through 5.

Parameter: `Size` (for AHB_Lite_Slave_T2S)

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = null.

Scope: Init-time

Description: Size of memory region 0.

Parameter: `ahb_size1` (for AHB_Lite_Slave_T2S)

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = 32.

Scope: Init-time

Description: Size of memory region 1.

Parameter: `region size [0-5]` (for AHB_Lite_Slave_FT2S)

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = 0x100000000 for region 0
0x0 for regions 1-5

Scope: Init-time

Description: Size of memory regions 0 through 5.

Parameter: `Subtract Base Address` (for AHB_Lite_Slave_T2S)

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Init-time

Description: When set to *true*, the `Base Address` is subtracted from the transaction address, and the offset is sent to the RTL port. When set to *false* (the default value), the transaction address is sent to the RTL port unchanged.

Parameter: `Subtract Base Address Dbg` (for AHB_Lite_Slave_T2S)

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Init-time

Description: Set this parameter when using the readDbg and writeDbg members of the transactor ports. When set to *true*, the `Base Address` is subtracted from the transaction address, and the offset is sent to the RTL port. When set to *false* (the default value), the transaction address is sent to the RTL port unchanged.

Parameter: `Big Endian`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Init-time

Description: Affects the alignment of the data on the data bus. When *true*, the data is output “big endian” on the data bus.

Parameter: `Align Data`

Parameter Type: boolean

Legal Values: true, false Default = *false*.

Scope: Init-time

Description: For narrow transfers on the AHB bus, SoC Designer Plus outputs data aligned to the low bytes, even though the protocol states that data for an unaligned address should be output on the correct byte lane. Setting this parameter to *true* enables aligning of the data to the correct byte lane.

Note: This field is no longer used, but is maintained for backwards compatibility.

Parameter: `Filter HREADYIN`

Parameter Type: boolean

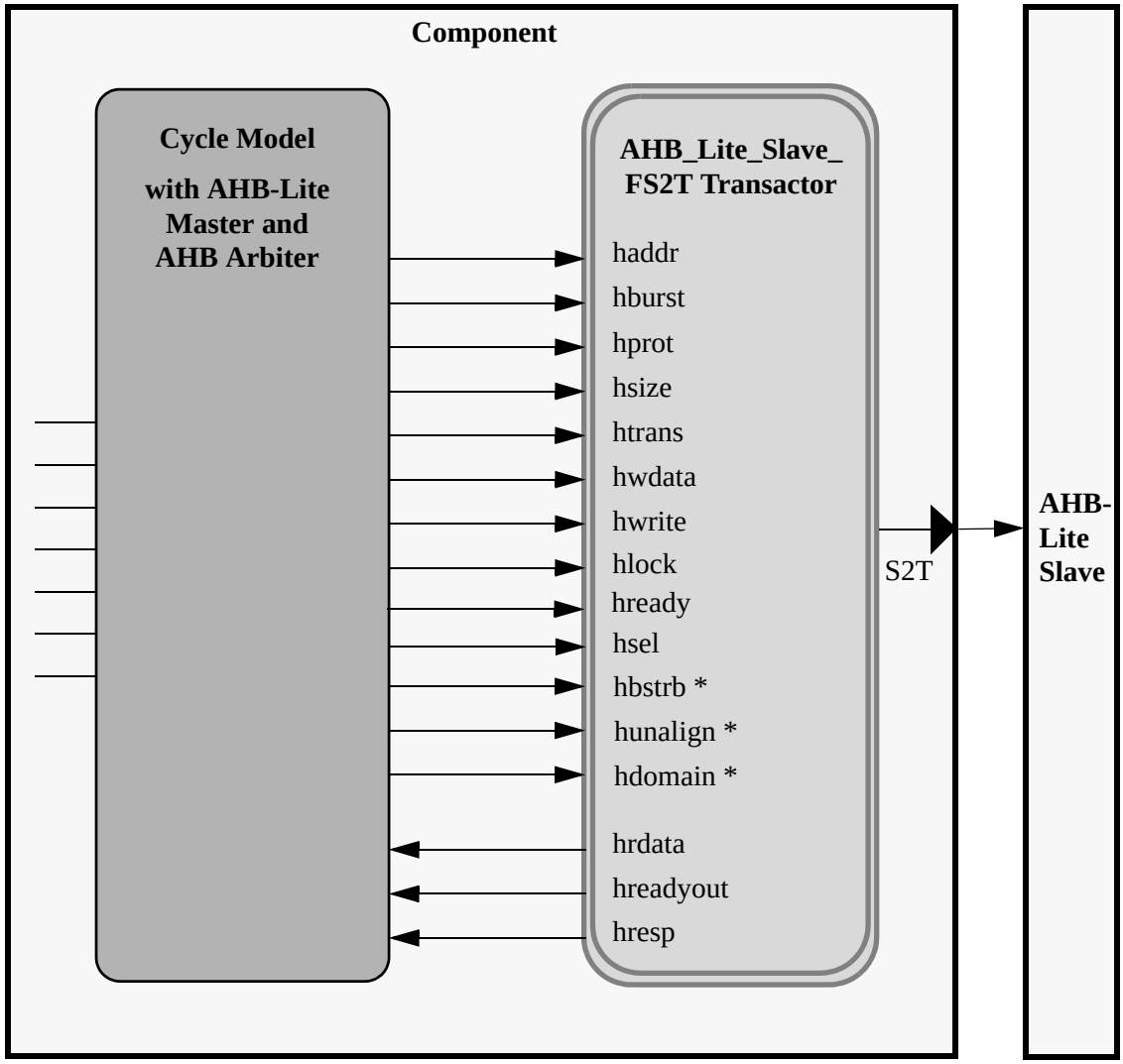
Legal Values: true, false. Default = *false*.

Scope: Init-time

Description: AHB slaves are required to have an HREADY signal as both an input and an output. The output is the slave’s ready status, while the input is the ready status of all the slaves on the bus (presumably calculated by the bus arbiter). Setting this parameter to *true* filters out the input HREADY “HREADIN” to prevent it from reaching the Cycle Model. The parameter should be set only when it is not required by the slave for correct operation.

A.2.1.8 AHB_Lite_Slave_FS2T

A transactor that converts AHB-Lite Slave interface signals from the Cycle Model into transactions that are communicated through the transaction master port. These transactors can communicate with SoC Designer Plus components that implement the CASI AHB transaction interface.



* Optional signals for ARM11 extension

Figure A-19 AHB_Lite_Slave_FS2T Transactor

AHB_Lite_Slave_FS2T parameters

Parameter: `Enable Debug Messages`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: When set to *true*, the adaptor outputs debug messages while running.

Parameter: `Big Endian`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Init-time

Description: Affects the alignment of the data on the data bus. When *true*, the data is read as “big endian” on the data bus.

Parameter: `Align Data`

Parameter Type: boolean

Legal Values: true, false Default = *false*.

Scope: Init-time

Description: For narrow transfers on the AHB bus, SoC Designer Plus outputs data aligned to the low bytes, even though the protocol states that data for an unaligned address should be output on the correct byte lane. Setting this parameter to *true* enables aligning of the data to the correct byte lane.

Note: This field is no longer used, but is maintained for backwards compatibility.

A.2.2 APB Transactors

This section describes:

- APB_Master
- APB_Slave

A.2.2.1 APB_Master

A transactor that converts APB Master interface signals from the Cycle Model into transactions that are communicated through the transaction master port. These transactors can communicate with SoC Designer Plus components that implement the CASI APB transaction interface.

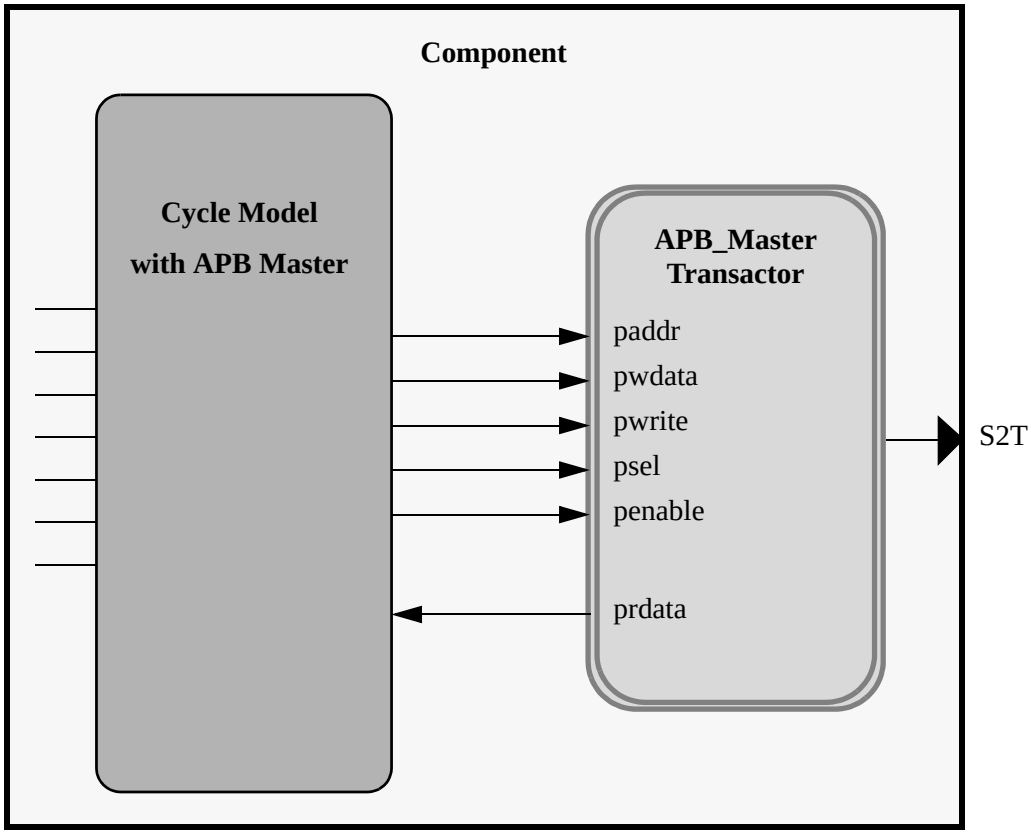


Figure A-20 APB_Master Transactor

APB_Master parameters

Parameter: `Enable Debug Messages`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: When set to *true*, the adaptor outputs debug messages while running.

Parameter: `Protocol Variant`

Legal value: APB2

Scope: Fixed (not changeable)

Description: Specifies APB2 protocol

A.2.2.2 APB_Slave

A transactor that converts APB transactions from a transaction slave port to APB Slave interface signals in the Cycle Model. These transactors can communicate with SoC Designer Plus components that implement the CASI APB transaction interface.

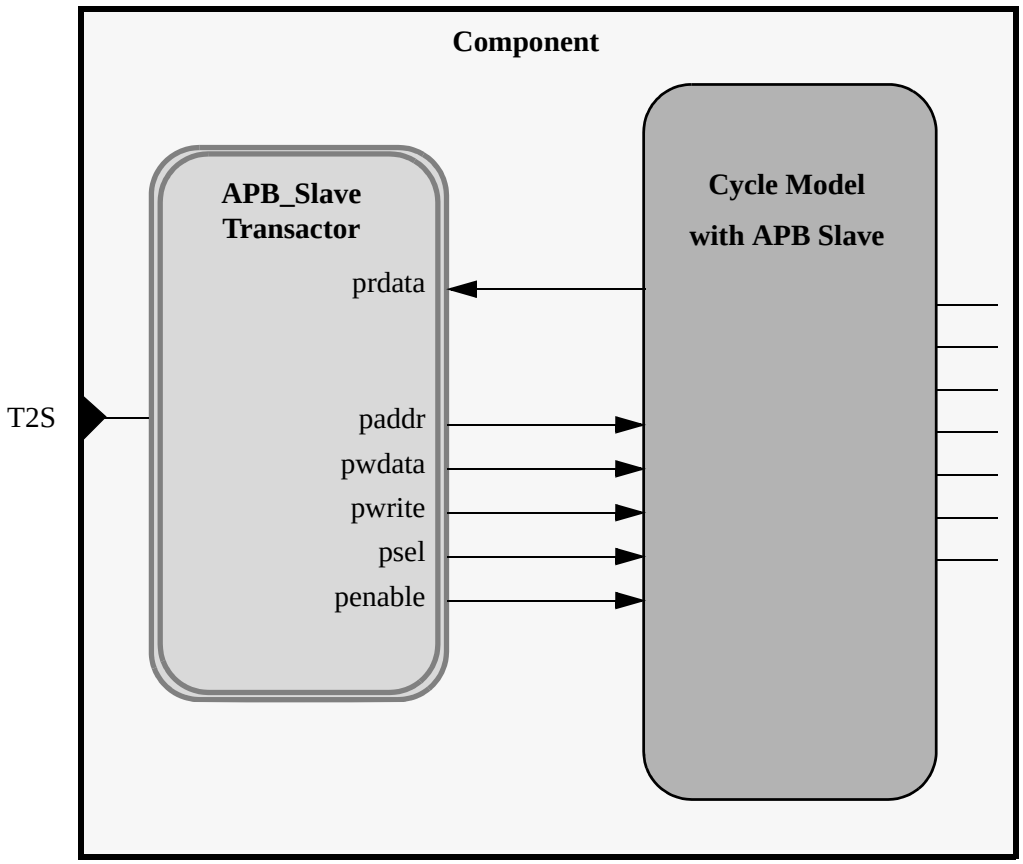


Figure A-21 APB_Slave Transactor

APB_Slave parameters

Parameter: `Enable Debug Messages`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: When set to *true*, the adaptor outputs debug messages while running.

Parameter: `Base Address`

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = 0.

Scope: Init-time

Description: Base address of the APB Slave memory region.

Parameter: `Size`

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = null.

Scope: Init-time

Description: Size of memory region.

Parameter: `Protocol Variant`

Legal value: APB2

Scope: Fixed (not changeable)

Description: Specifies APB2 protocol

A.2.3 APB3 Transactors

This section describes:

- APB3_Master
- APB3_Slave

A.2.3.1 APB3_Master

A transactor that converts APB3 Master interface signals from the Cycle Model into transactions that are communicated through the transaction master port. These transactors can communicate with SoC Designer Plus components that implement the CASI APB3 transaction interface.

The difference between the APB Master and the APB3 Master is that the APB3 adds two signals: PREADY and PSLVERR. PREADY can be used by an APB slave component to insert wait states to slow down the transfer. PSLVERR can be used by a slave component to signal an error during a transaction.

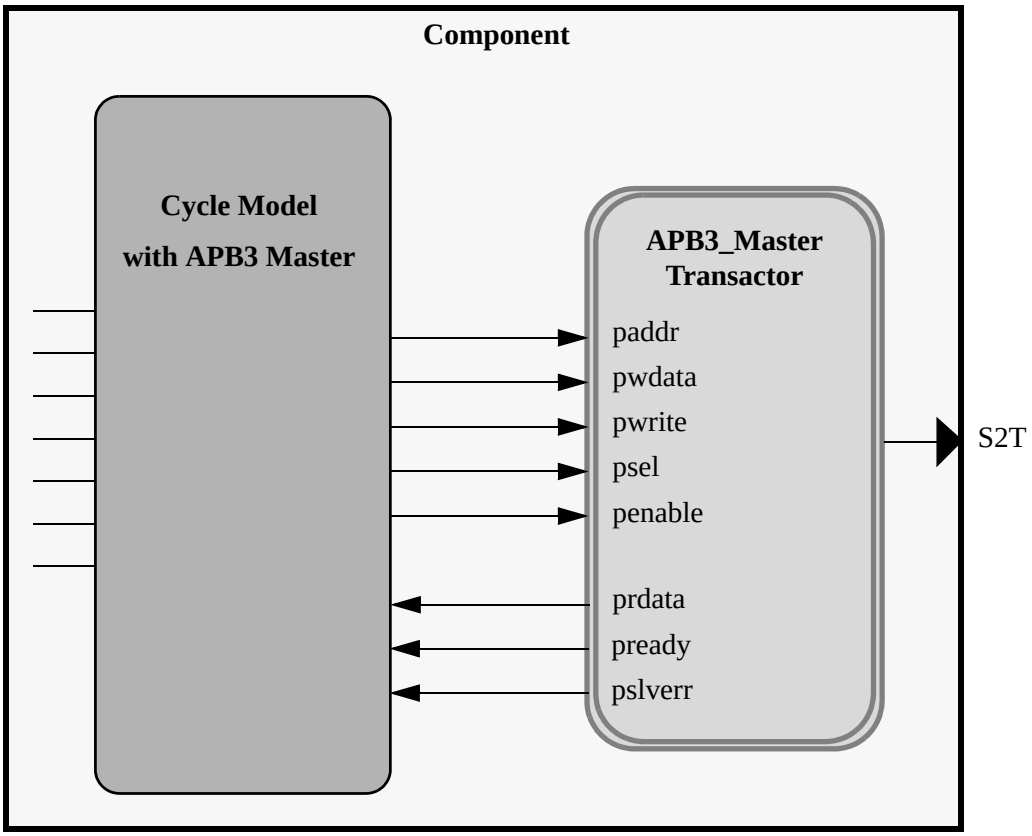


Figure A-22 APB3_Master Transactor

APB3_Master parameters

Parameter: `Enable Debug Messages`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: When set to *true*, the adaptor outputs debug messages while running.

Parameter: `Base Address`

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = 0.

Scope: Init-time

Description: Base address of the APB memory region.

Parameter: `Size`

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = null.

Scope: Init-time

Description: Size of memory region.

Parameter: `PReady Default High`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: This parameter affects the value of the PReady signal while not in a transaction. When *true*, PReady is High between transactions. When *false*, PReady is Low.

Parameter: `Protocol Variant`

Legal value: APB3

Scope: Fixed (not changeable)

Description: Specifies APB3 protocol

A.2.3.2 APB3_Slave

A transactor that converts APB3 transactions from a transaction slave port to APB3 Slave interface signals in the Cycle Model. These transactors can communicate with SoC Designer Plus components that implement the CASI APB3 transaction interface.

The difference between the APB Slave and the APB3 Slave is that the APB3 adds two signals: PREADY and PSLVERR. PREADY can be used by an APB slave component to insert wait states to slow down the transfer. PSLVERR can be used by a slave component to signal an error during a transaction.

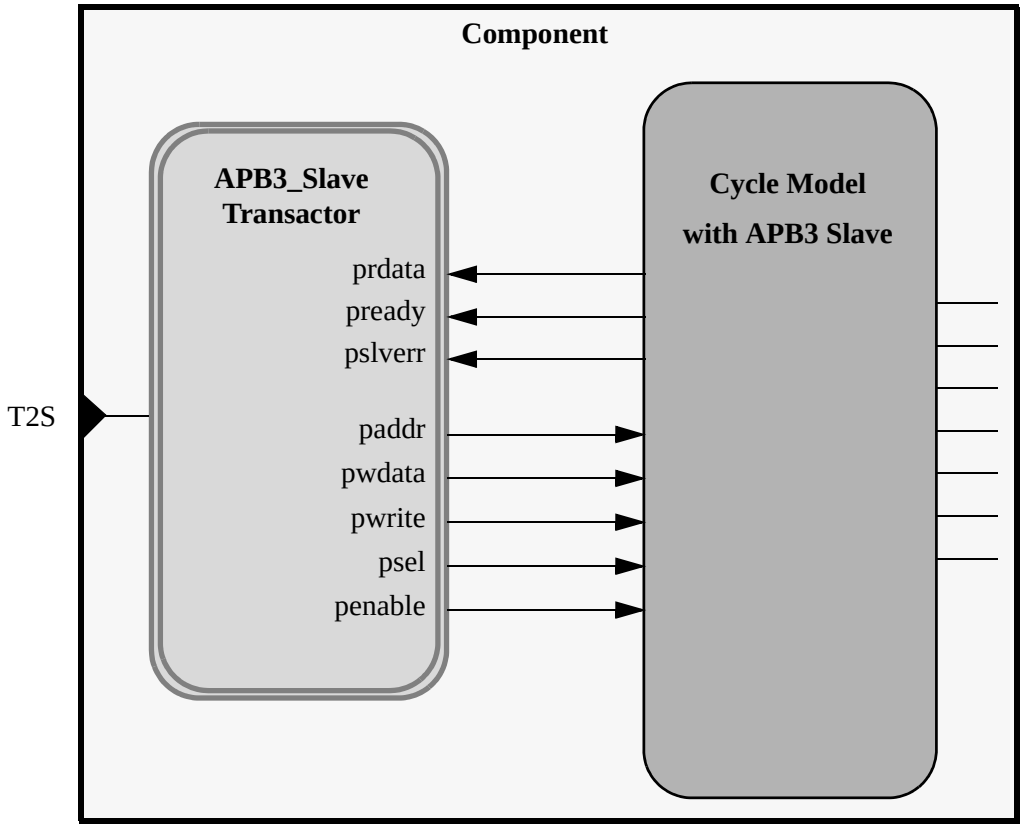


Figure A-23 APB3_Slave Transactor

APB3_Slave parameters

Parameter: `Enable Debug Messages`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: When set to *true*, the adaptor outputs debug messages while running.

Parameter: `Base Address`

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = 0.

Scope: Init-time

Description: Base address of the APB Slave memory region.

Parameter: `Size`

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = null.

Scope: Init-time

Description: Size of memory region.

Parameter: `Protocol Variant`

Legal value: APB3

Scope: Fixed (not changeable)

Description: Specifies APB3 protocol

A.2.4 APB4 Transactors

This section describes:

- APB4_Master
- APB4_Slave

A.2.4.1 APB4_Master

A transactor that converts APB4 Master interface signals from the Cycle Model into transactions that are communicated through the transaction master port. These transactors can communicate with SoC Designer Plus components that implement the CASI APB4 transaction interface.

APB4 adds two signals: PSTRB, a byte strobe for write commands, and PPROT, which selects the protection type.

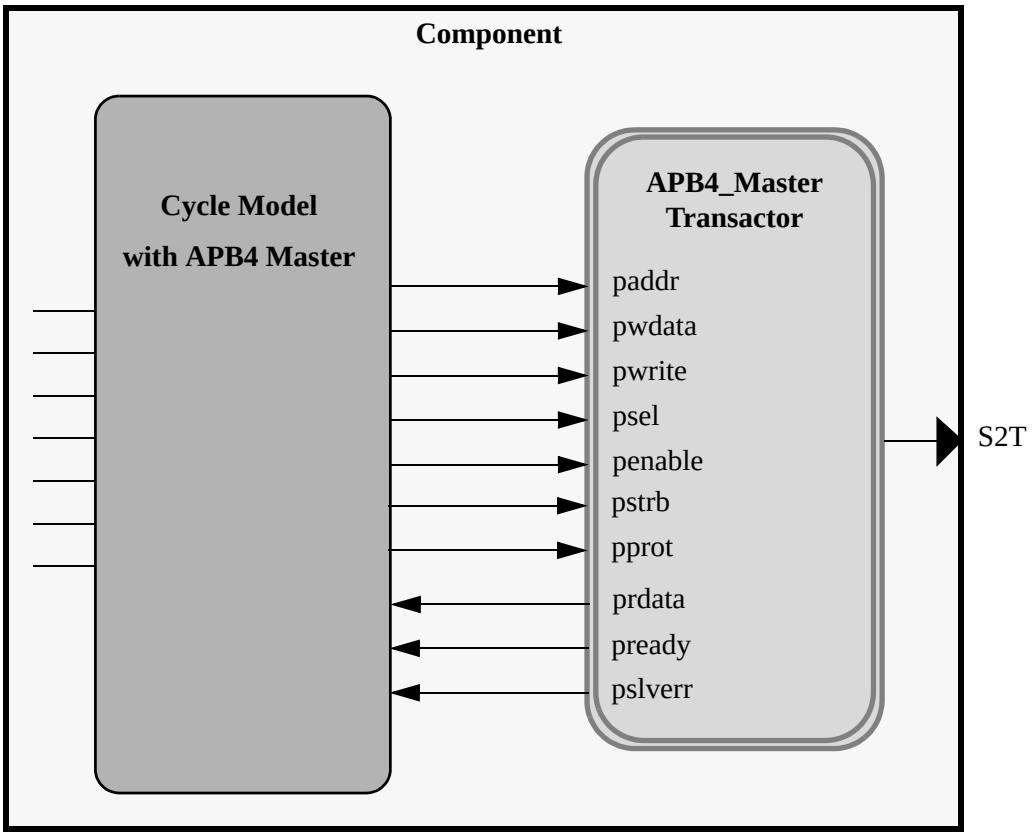


Figure A-24 APB4_Master Transactor

APB4_Master parameters

Parameter: `Enable Debug Messages`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: When set to *true*, the adaptor outputs debug messages while running.

Parameter: `Base Address`

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = 0.

Scope: Init-time

Description: Base address of the APB memory region.

Parameter: `Size`

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = null.

Scope: Init-time

Description: Size of memory region.

Parameter: `PReady Default High`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: This parameter affects the value of the PReady signal while not in a transaction. When *true*, PReady is High between transactions. When *false*, PReady is Low.

Parameter: `Protocol Variant`

Legal value: APB4

Scope: Fixed (not changeable)

Description: Specifies APB4 protocol

A.2.4.2 APB4_Slave

A transactor that converts APB4 transactions from a transaction slave port to APB4 Slave interface signals in the Cycle Model. These transactors can communicate with SoC Designer Plus components that implement the CASI APB4 transaction interface.

APB4 adds two signals: PSTRB and PPROT. PSTRB, a byte strobe for write commands, and PPROT, which selects the protection type.

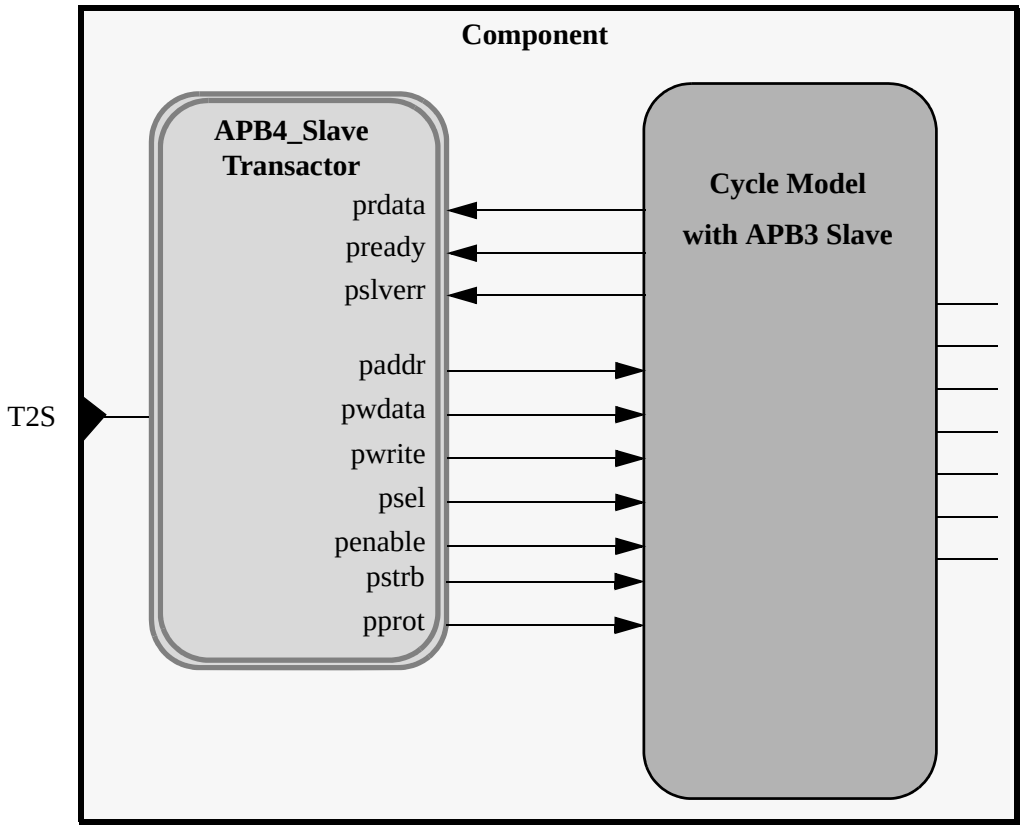


Figure A-25 APB4_Slave Transactor

APB4_Slave parameters

Parameter: `Enable Debug Messages`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: When set to *true*, the adaptor outputs debug messages while running.

Parameter: `Base Address`

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = 0.

Scope: Init-time

Description: Base address of the APB Slave memory region.

Parameter: `Size`

Parameter Type: value

Legal Values: 0 - Max Address Size. Default = null.

Scope: Init-time

Description: Size of memory region.

Parameter: `Protocol Variant`

Legal value: APB4

Scope: Fixed (not changeable)

Description: Specifies APB4 protocol

A.2.5 AXI Transactors

There are four variants of the AXI transactors:

- *AXI_Master* and *AXI_Slave*: These are the original AXI v1 transactors that have been available through Carbon Model Studio for many releases.
- *AXI_Flowthru_Master* and *AXI_Flowthru_Slave*: These flow-through versions of the AXI transactors (v2) should be used if the Cycle Model has flow through (asynchronous) paths in the design. These transactors are compatible with the ARM AMBA3/AXI protocol.
- *AXI4_Master* and *AXI4_Slave*: These transactors support the ARM AMBA4/AXI protocol.

Note: A component built with the original AXI transactors cannot be directly connected to a component built with flow-through AXI transactors. The SoC Designer Plus release provides adaptors to connect two such components.

Note: The AXI_Master and AXI_Slave transactors are deprecated and will be removed in 2012 Q2.

A.2.5.1 AXI_Master and AXI_Flowthru_Master

A transactor that converts AXI Master signals from the Cycle Model into transactions that are communicated through the transaction master port. These transactors can communicate with SoC Designer Plus components that implement the CASI AXI transaction interface.

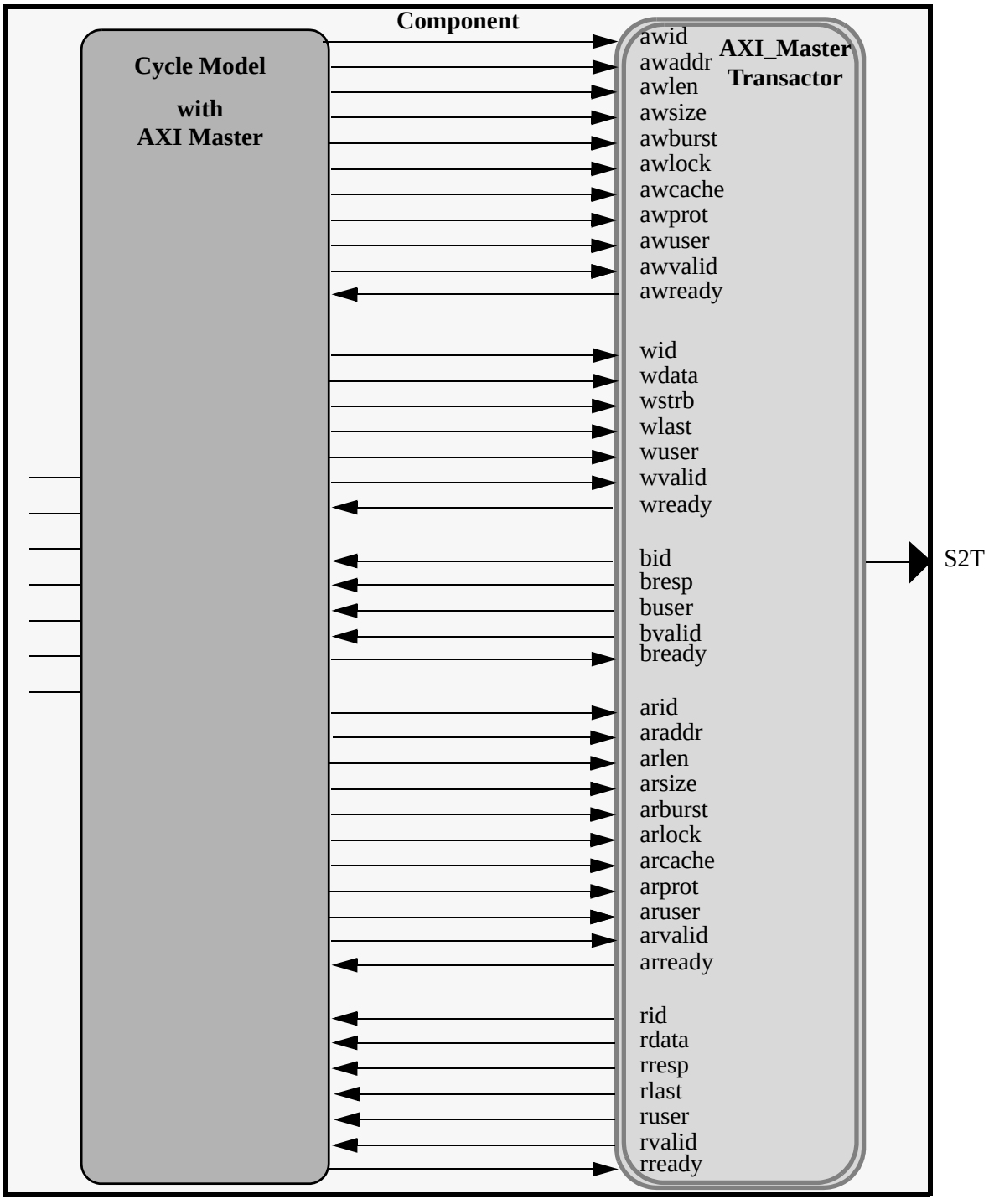


Figure A-26 AXI_Master Transactor

AXI_Master and AXI_Flowthru_Master parameters

Parameter: `Enable Debug Messages`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: When set to *true*, the adaptor outputs debug messages while running.

A.2.5.2 AXI_Slave and AXI_Flowthru_Slave

A transactor that converts AXI transactions from a transaction slave port to AXI Slave interface signals in the Cycle Model. These transactors can communicate with SoC Designer Plus components that implement the CASI AXI transaction interface.

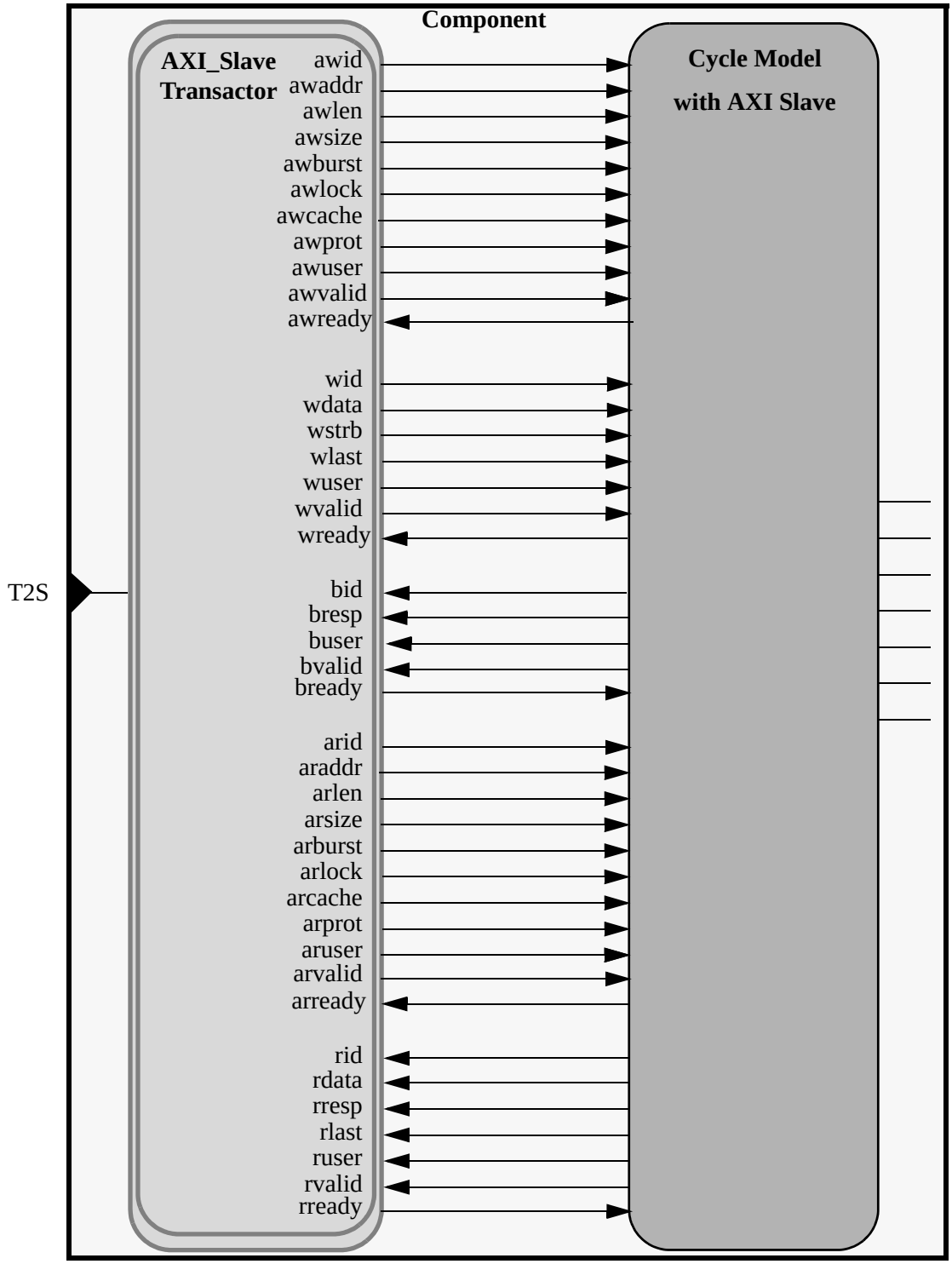


Figure A-27 AXI_Slave Transactor

AXI_Slave and AXI_Flowthru_Slave parameters

Parameter: `Enable Debug Messages`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: When set to *true*, the adaptor outputs debug messages while running.

Parameter: `axi_start`*`[0-5]`*

Parameter type: value

Legal Values: 0 - Max Address Size. Default = 0.

Scope: Init-time

Description: Start Address of AXI memory regions 0 through 5.

Parameter: `axi_size`*`[0-5]`*

Parameter type: value

Legal Values: 0 - Max Address Size. Default = 0x10000000
0 for regions 1-5

Scope: Init-time

Description: Size of AXI memory region 0.

A.2.5.3 AXI4_Master and AXI4_Slave

An AXI4_Master transactor converts AXI4 Master signals from the Cycle Model into transactions that are communicated through the transaction master port.

An AXI4_Slave transactor converts AXI4 transactions from a transaction slave port to AXI Slave interface signals in the Cycle Model.

The AXI4 Master and Slave transactors implement a number of AXI4 protocol variants including:

- AXI4-Lite
- AXI4
- ACE-Lite
- ACE-Lite+DVM
- ACE

This is implemented as a transactor parameter called Protocol Variant. You can set this value in either of the following ways:

- In CMS, by accessing the Parameter pulldown menu on the Ports tab.
- In SoC Designer Canvas, by accessing the parameter on the component that contains the transaction interface. There is one parameter per AXI4 interface in the SoC Designer Plus component. The name is prefixed by the transaction interface name to differentiate each interface's parameter.

These transactors can communicate with SoC Designer Plus components that implement the CASI AXI transaction interface. The following table describes the ports for the AXI4_Master and AXI4_Slave transactors.

Table A.3: AXI4_Master and AXI4_Slave Transactor Ports

Channel	Port	Master Direction	Slave Direction
Write Address	awid	Model to transactor	Transactor to model
	awaddr	Model to transactor	Transactor to model
	awlen	Model to transactor	Transactor to model
	awsize	Model to transactor	Transactor to model
	awburst	Model to transactor	Transactor to model
	awlock	Model to transactor	Transactor to model
	awcache	Model to transactor	Transactor to model
	awprot	Model to transactor	Transactor to model
	awuser	Model to transactor	Transactor to model
	awvalid	Model to transactor	Transactor to model
	awready	Transactor to model	Model to transactor
	awqos	Model to transactor	Transactor to model
	awregion	Model to transactor	Transactor to model
	awdomain	Model to transactor	Transactor to model
	awsnoop	Model to transactor	Transactor to model
	awbar	Model to transactor	Transactor to model
Write Data	wdata	Model to transactor	Transactor to model
	wstrb	Model to transactor	Transactor to model
	wlast	Model to transactor	Transactor to model
	wuser	Model to transactor	Transactor to model
	wvalid	Model to transactor	Transactor to model
	wready	Transactor to model	Model to transactor
Write Response	bid	Transactor to model	Model to transactor
	bresp	Transactor to model	Model to transactor
	buser	Transactor to model	Model to transactor
	bvalid	Transactor to model	Model to transactor
	bready	Model to transactor	Transactor to model
	wack	Model to transactor	Transactor to model

Table A.3: AXI4_Master and AXI4_Slave Transactor Ports (continued)

Channel	Port	Master Direction	Slave Direction
Read Address	arid	Model to transactor	Transactor to model
	araddr	Model to transactor	Transactor to model
	arlen	Model to transactor	Transactor to model
	arsize	Model to transactor	Transactor to model
	arburst	Model to transactor	Transactor to model
	arlock	Model to transactor	Transactor to model
	arcache	Model to transactor	Transactor to model
	arprot	Model to transactor	Transactor to model
	aruser	Model to transactor	Transactor to model
	arvalid	Model to transactor	Transactor to model
	arready	Transactor to model	Model to transactor
	arqos	Model to transactor	Transactor to model
	arregion	Model to transactor	Transactor to model
	ardomain	Model to transactor	Transactor to model
	arsnoop	Model to transactor	Transactor to model
	arbar	Model to transactor	Transactor to model
Read Data	rack	Transactor to model	Model to transactor
	rid	Transactor to model	Model to transactor
	rdata	Transactor to model	Model to transactor
	rresp	Transactor to model	Model to transactor
	rlast	Transactor to model	Model to transactor
	ruser	Transactor to model	Model to transactor
	rvalid	Transactor to model	Model to transactor
	rready	Model to transactor	Transactor to model
Snoop Address	acvalid	Transactor to model	Model to transactor
	acready	Model to transactor	Transactor to model
	acaddr	Transactor to model	Model to transactor
	acsnoop	Transactor to model	Model to transactor
	acprot	Transactor to model	Model to transactor

Table A.3: AXI4_Master and AXI4_Slave Transactor Ports (continued)

Channel	Port	Master Direction	Slave Direction
Snoop Response	crvalid	Model to transactor	Transactor to model
	crready	Transactor to model	Model to transactor
	crresp	Model to transactor	Transactor to model
Snoop Data	cdvalid	Model to transactor	Transactor to model
	cdready	Transactor to model	Model to transactor
	cddata	Model to transactor	Transactor to model
	cdlast	Model to transactor	Transactor to model

AXI4_Master and AXI4_Slave parameters

Parameter: `Enable Debug Messages`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: When set to *true*, the adaptor outputs debug messages while running.

Parameter: `axi_start`*`[0-5]`*

Parameter type: value

Legal Values: 0 - Max Address Size. Default = 0.

Scope: Init-time

Description: Start Address of AXI memory regions 0 through 5.

Parameter: `axi_size`*`[0-5]`*

Parameter type: value

Legal Values: 0 - Max Address Size. Default = 0x10000000
0 for regions 1-5

Scope: Init-time

Description: Size of AXI memory region 0.

A.2.6 ARM926_DTCM Transactors

The ARM926_DTCM_Slave and ARM926_DTCM_Master transactors are provided for use with the SoC Designer Plus model for the ARM926EJS with external Data TCM.

A.2.6.1 ARM926_DTCM_Slave and ARM926_DTCM_Master Ports

The transactor signal names are similar to the names assigned by ARM, but do not include the “D” prefix (see the *ARM926EJ-S Technical Reference Manual*, rev r0p5). Refer to Figure A-28 and Figure A-29 for port directions..

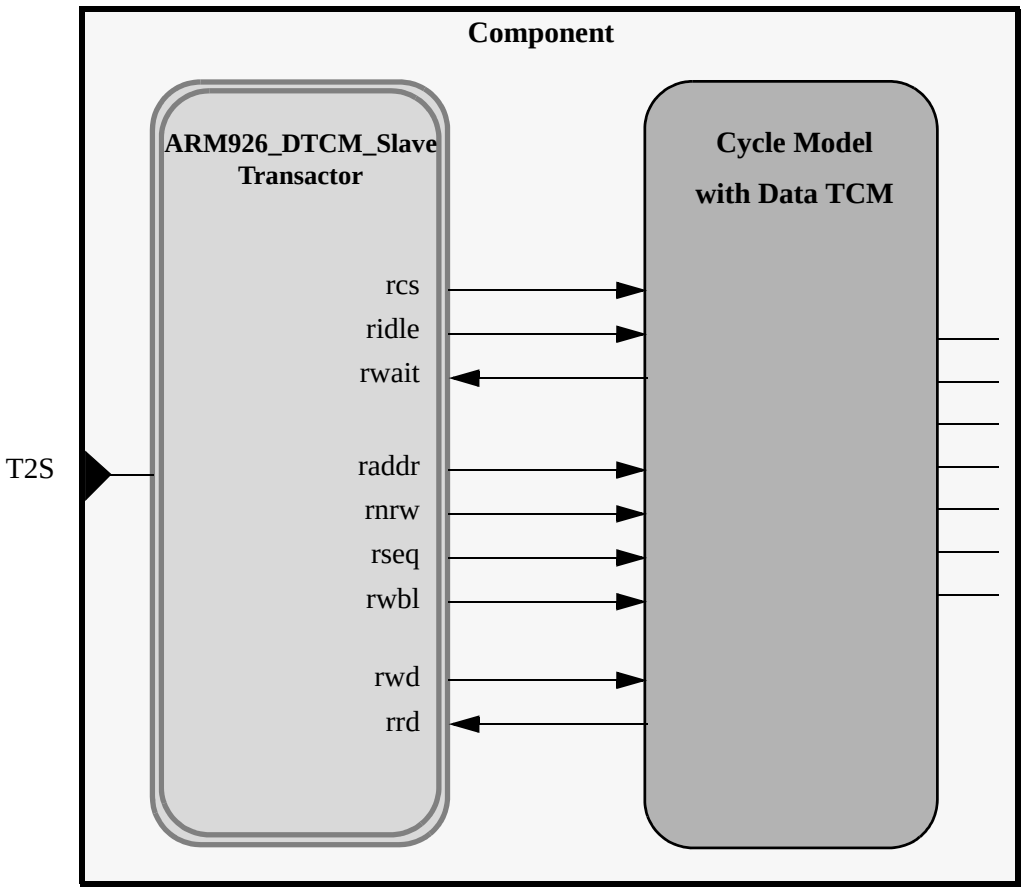


Figure A-28 ARM926_DTCM_Slave Transactor

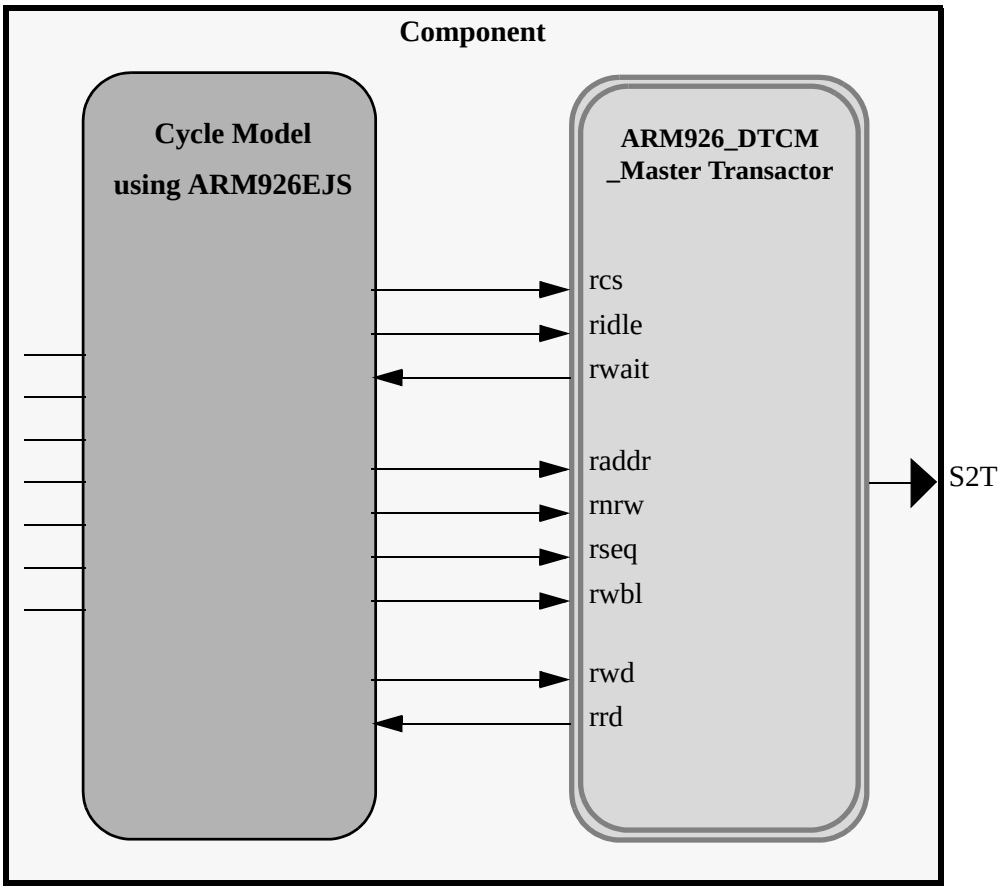


Figure A-29 ARM926_DTCM_Master Transactor

A.2.7 CHI Transactors

The CHI transactors are separated into two groups: CHIInt* and CHINode*. The CHIInt* transactors convert signals from a CHI ICN (Interconnect) port to link layer flits, which are transacted to/from a CHI Node port. The CHINode* transactors convert signals from a CHI Node port to link layer flits, which are transacted to/from a CHI ICN port.

These transactors communicate with SoC Designer Plus components that implement the AMBA CHI Port Interface classes. Refer to the *AMBA CHI Protocol Bundle User Guide* for more information.

A.2.7.1 CHIInt* Transactors

The CHIInt* set of transactors includes:

- CHIIntRNDXtor (CHI Completer)
- CHIIntRNFXtor (CHI Completer)
- CHIIntRNIXtor (CHI Completer)
- CHIIntRequestorXtor (CHI Completer)
- CHIIntSNFXtor (CHI Requestor)
- CHIIntSNIXtor (CHI Requestor)
- CHIIntSlaveXtor (CHI Requestor)

This section describes the available signals and parameters for this set of transactors.

CHIInt* Transactor Signals

Table A.4 describes the available signals for each CHIInt* transactor.

Table A.4: CHIInt* Transactor Signals

Channel	Signal	Direction	Signal Availability Per Transactor						
			CHI Int RND Xtor	CHI Int RNF Xtor	CHI Int RNI Xtor	CHI Int Requestor Xtor	CHI Int SNF Xtor	CHI Int SNI Xtor	CHI Int Slave Xtor
REQ	TXREQFLIT	Model to Transactor				•	•	•	•
	TXREQFLITPEND	Model to Transactor				•	•	•	•
	TXREQFLITV	Model to Transactor				•	•	•	•
	TXREQLCRDV	Transactor to Model				•	•	•	•
	RXREQFLIT	Transactor to Model	•	•	•	•			•
	RXREQFLITPEND	Transactor to Model	•	•	•	•			•
	RXREQFLITV	Transactor to Model	•	•	•	•			•
	RXREQLCRDV	Model to Transactor	•	•	•	•			•

Table A.4: CHIInt* Transactor Signals (continued)

Channel	Signal	Direction	Signal Availability Per Transactor						
			CHI Int RND Xtor	CHI Int RNF Xtor	CHI Int RNI Xtor	CHI Int Requestor Xtor	CHI Int SNF Xtor	CHI Int SNI Xtor	CHI Int Slave Xtor
RSP	TXRSPFLIT	Model to Transactor	•	•	•	•			•
	TXRSPFLITPEND	Model to Transactor	•	•	•	•			•
	TXRSPFLITV	Model to Transactor	•	•	•	•			•
	TXRSPLCRDV	Transactor to Model	•	•	•	•			•
	RXRSPFLIT	Transactor to Model	•	•	•	•	•	•	•
	RXRSPFLITPEND	Transactor to Model	•	•	•	•	•	•	•
	RXRSPFLITV	Transactor to Model	•	•	•	•	•	•	•
	RXRSPLCRDV	Model to Transactor	•	•	•	•	•	•	•
SNP	TXSNPFLIT	Model to Transactor	•	•		•			•
	TXSNPFLITPEND	Model to Transactor	•	•		•			•
	TXSNPFLITV	Model to Transactor	•	•		•			•
	TXSNPLCRDV	Transactor to Model	•	•		•			•
	RXSNPFLIT	Transactor to Model				•			•
	RXSNPFLITPEND	Transactor to Model				•			•
	RXSNPFLITV	Transactor to Model				•			•
	RXSNPLCRDV	Model to Transactor				•			•

Table A.4: CHIInt* Transactor Signals (continued)

Channel	Signal	Direction	Signal Availability Per Transactor						
			CHI Int RND Xtor	CHI Int RNF Xtor	CHI Int RNI Xtor	CHI Int Requestor Xtor	CHI Int SNF Xtor	CHI Int SNI Xtor	CHI Int Slave Xtor
DAT	TXDATFLIT	Model to Transactor	•	•	•	•	•	•	•
	TXDATFLITPEND	Model to Transactor	•	•	•	•	•	•	•
	TXDATFLITV	Model to Transactor	•	•	•	•	•	•	•
	TXDATLCRDV	Transactor to Model	•	•	•	•	•	•	•
	RXDATFLIT	Transactor to Model	•	•	•	•	•	•	•
	RXDATFLITPEND	Transactor to Model	•	•	•	•	•	•	•
	RXDATFLITV	Transactor to Model	•	•	•	•	•	•	•
	RXDATLCRDV	Model to Transactor	•	•	•	•	•	•	•
Control Signals	TXLINKACTIVEACK	Transactor to Model	•	•	•	•	•	•	•
	TXLINKACTIVEREQ	Model to Transactor	•	•	•	•	•	•	•
	TXSACTIVE	Model to Transactor	•	•	•	•	•	•	•
	RXLINKACTIVEACK	Model to Transactor	•	•	•	•	•	•	•
	RXLINKACTIVEREQ	Transactor to Model	•	•	•	•	•	•	•
	RXSACTIVE	Transactor to Model	•	•	•	•	•	•	•

CHIInt* Parameters

Parameter: `Data Bus Width`

Parameter Type: integer value

Legal Values: 128, 256, 512. Default = 128.

Scope: Compile-time

Description: Width of rdata and wdata busses.

Parameter: `Enable Debug Messages`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: When set to *true*, outputs debug messages while running.

Parameter: `Protocol Variant`

Parameter Type: string

Legal Values: Fixed. Default = Protocol for this transactor (i.e., CHI-RND).

Scope: Init-time

Description: Specifies the protocol variant for the transactor.

A.2.7.2 CHINode* Transactors

The CHINode* set of transactors includes:

- CHINodeRNDXtor (CHI Requestor)
- CHINodeRNFXtor (CHI Requestor)
- CHINodeRNIXtor (CHI Requestor)
- CHINodeRequestorXtor (CHI Requestor)
- CHINodeSNFXtor (CHI Completer)
- CHINodeSNIXtor (CHI Completer)
- CHINodeSlaveXtor (CHI Completer)

This section describes the available signals and parameters for this set of transactors.

CHINode* Transactor Signals

Table A.5 describes the available signals for each CHINode* transactor.

Table A.5: CHINode* Transactor Signals

Channel	Signal	Direction	Signal Availability Per Transactor						
			CHI Node RND Xtor	CHI Node RNF Xtor	CHI Node RNI Xtor	CHI Node Requestor Xtor	CHI Node SNF Xtor	CHI Node SNI Xtor	CHI Node Slave Xtor
REQ	TXREQFLIT	Model to Transactor	•	•	•	•			•
	TXREQFLITPEND	Model to Transactor	•	•	•	•			•
	TXREQFLITV	Model to Transactor	•	•	•	•			•
	TXREQLCRDV	Transactor to Model	•	•	•	•			•
	RXREQFLIT	Transactor to Model				•	•	•	•
	RXREQFLITPEND	Transactor to Model				•	•	•	•
	RXREQFLITV	Transactor to Model				•	•	•	•
	RXREQLCRDV	Model to Transactor				•	•	•	•

Table A.5: CHINode* Transactor Signals (continued)

Channel	Signal	Direction	Signal Availability Per Transactor						
			CHI Node RND Xtor	CHI Node RNF Xtor	CHI Node RNI Xtor	CHI Node Requestor Xtor	CHI Node SNF Xtor	CHI Node SNI Xtor	CHI Node Slave Xtor
RSP	TXRSPFLIT	Model to Transactor	•	•	•	•	•	•	•
	TXRSPFLITPEND	Model to Transactor	•	•	•	•	•	•	•
	TXRSPFLITV	Model to Transactor	•	•	•	•	•	•	•
	TXRSPLCRDV	Transactor to Model	•	•	•	•	•	•	•
	RXRSPFLIT	Transactor to Model	•	•	•	•			•
	RXRSPFLITPEND	Transactor to Model	•	•	•	•			•
	RXRSPFLITV	Transactor to Model	•	•	•	•			•
	RXRRSPLCRDV	Model to Transactor	•	•	•	•			•
SNP	TXSNPFLIT	Model to Transactor				•			•
	TXSNPFLITPEND	Model to Transactor				•			•
	TXSNPFLITV	Model to Transactor				•			•
	TXSNPLCRDV	Transactor to Model				•			•
	RXSNPFLIT	Transactor to Model	•	•		•			•
	RXSNPFLITPEND	Transactor to Model	•	•		•			•
	RXSNPFLITV	Transactor to Model	•	•		•			•
	RXSNPLCRDV	Model to Transactor	•	•		•			•

Table A.5: CHINode* Transactor Signals (continued)

Channel	Signal	Direction	Signal Availability Per Transactor						
			CHI Node RND Xtor	CHI Node RNF Xtor	CHI Node RNI Xtor	CHI Node Requestor Xtor	CHI Node SNF Xtor	CHI Node SNI Xtor	CHI Node Slave Xtor
DAT	TXDATFLIT	Model to Transactor	•	•	•	•	•	•	•
	TXDATFLITPEND	Model to Transactor	•	•	•	•	•	•	•
	TXDATFLITV	Model to Transactor	•	•	•	•	•	•	•
	TXDATLCRDV	Transactor to Model	•	•	•	•	•	•	•
	RXDATFLIT	Transactor to Model	•	•	•	•	•	•	•
	RXDATFLITPEND	Transactor to Model	•	•	•	•	•	•	•
	RXDATFLITV	Transactor to Model	•	•	•	•	•	•	•
	RXDATLCRDV	Model to Transactor	•	•	•	•	•	•	•
Control Signals	TXLINKACTIVEACK	Transactor to Model	•	•	•	•	•	•	•
	TXLINKACTIVEREQ	Model to Transactor	•	•	•	•	•	•	•
	TXSACTIVE	Model to Transactor	•	•	•	•	•	•	•
	RXLINKACTIVEACK	Model to Transactor	•	•	•	•	•	•	•
	RXLINKACTIVEREQ	Transactor to Model	•	•	•	•	•	•	•
	RXSACTIVE	Transactor to Model	•	•	•	•	•	•	•

CHINode* Parameters

Parameter: `Data Bus Width`

Parameter Type: integer value

Legal Values: 128, 256, 512. Default = 128.

Scope: Compile-time

Description: Width of rdata and wdata busses.

Parameter: `Enable Debug Messages`

Parameter Type: boolean

Legal Values: true, false. Default = *false*.

Scope: Run-time

Description: When set to *true*, outputs debug messages while running.

Parameter: `Protocol Variant`

Parameter Type: string

Legal Values: Fixed. Default = Protocol for this transactor (i.e., CHI-RND).

Scope: Init-time

Description: Specifies teh protocol variant for the transactor.

A.3 Transactors that are Internal to the Cycle Model

The following transactors are instantiated in the RTL code and therefore exist inside the Cycle Model (see Figure A-30). The Carbon Model Studio is used only to add transaction ports to the generated component, as the connections between the transactors and the Cycle Model have already been made prior to the Carbon compiler run. These transactors can be used with ARM-supplied components or user-written components.

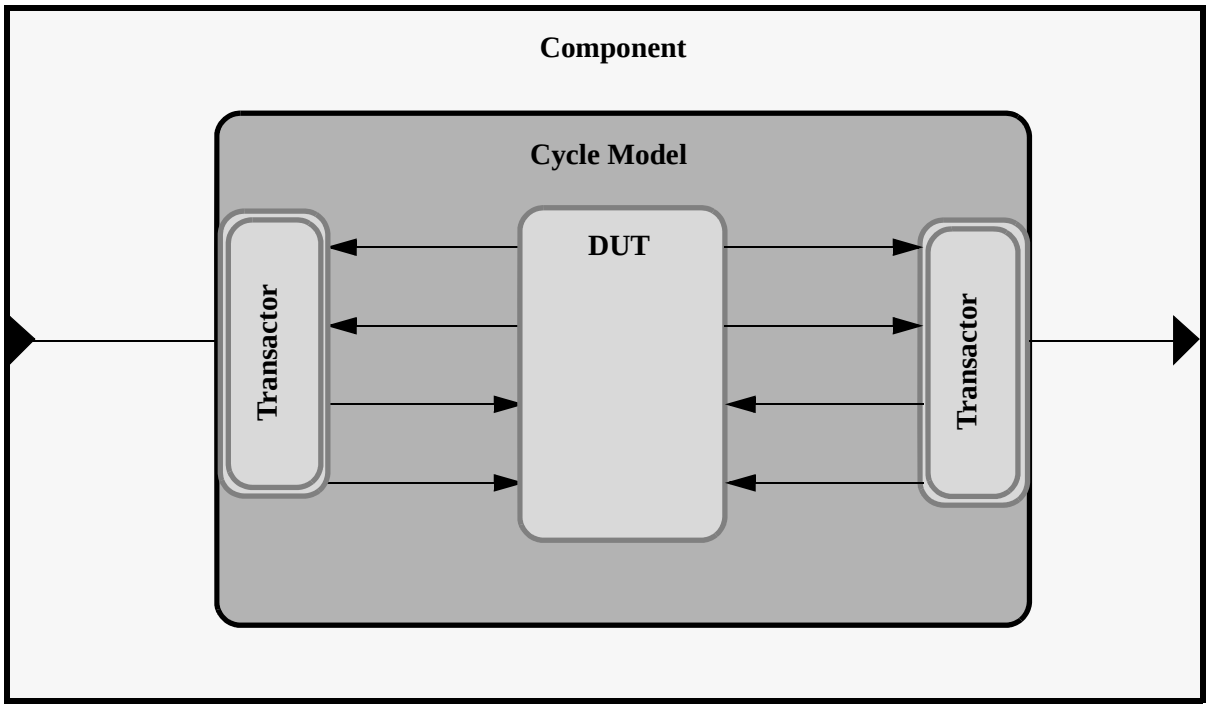


Figure A-30 Generic Transactor Internal to Cycle Model

For more information about each of these transactors, refer to the following User Manuals in the Transactor Guides Library:

- CarbonX_EnetMII_Master and _Slave — *10/100-Megabit Ethernet User Manual*
- CarbonX_EnetGMII_Master and _Slave — *1-Gigabit Ethernet User Manual*
- CarbonX_EnetRGMII_Master and _Slave — *1-Gigabit Ethernet-Reduced Pin Interface User Manual*
- CarbonX_EnetXGMII_Master and _Slave — *10-Gigabit Ethernet User Manual* or *10 Gigabit XGMII/XAUI Ethernet User Manual*
- CarbonX_PCI_Initiator and _Target — *PCI Transactor User Manual*
- CarbonX_Pcie_xLane_10Bit — *PCI Express Transactor User Manual*
- CarbonX_Pcie_xLane_Pipe — *PCI Express Transactor User Manual*

Appendix B

User-Defined Transactors with SoC Designer Plus

In addition to using the ARM-supplied transactors and the SoC Designer Plus transactors, Carbon Model Studio also allows you to define your own transactors. In essence, this allows you to create a named object with named input and output ports that can be connected, via the Carbon Model Studio, to Cycle Model ports.

This appendix describes how to write your own transactor and incorporate it into the component. The first two sections describe the Carbon Model Studio's requirements for the transactor's XML and C++ files. The final section describes how to use the Carbon Model Studio to integrate your transactor into the component.

B.1 Writing the Transactor XML File

Every transactor must be described so that the Carbon Model Studio wizard can provide user interface for port connections and generate the connecting code in the component. Your transactor file must contain the following features:

- The name of your transactor.

  This name should be unique; that is, it should not duplicate the name of another transactor. In addition, avoid using a "CarbonX_" prefix for your transactor.
- The name of the class instantiated by the component.
- The name of the header file that defines the above class.
- A port definition for each of the transactor's signal ports, if any.

You can use the transactor definitions found in the file `$CARBON_HOME/lib/xactors/xactors.xml` as a guide.

Sample user-defined transactor XML file

```
<?xml version="1.0"?>
<transactors version="1.0">

   <!-- Define your transactor with a ‘component’ element. The ‘name’
          attribute is the transactor name and shows up in the Carbon
          Model Studio menu.
   -->
   <component name="MyTransactor">

      <!--
         The name of the class that is instantiated by the Carbon
         component
      -->
      <className>MyXtorClass</className>

      <!--
         The header file that contains the definition of the class
         given by the className element
      -->
      <includeFile>userXtors.h</includeFile>

      <!--
         Port definitions. A transactor may have any number of ports,
         including none. Each port has a name, width, and direction,
         as follows:

         name: The name of the port. This name is used:
             - In the Carbon Model Studio wizard to connect to RTL
               signals.
             - To find runtime port objects via the 'portFactory' and
               'carbonGetPort' interfaces.

         width: The width, in bits, of the port.

         direction: "input" or "output"
      -->
      <port name="input_1" width="1" direction="input">
         <!-- a description of the port is optional -->
         <description>Input One</description>
      </port>

      <port name="output_1" width="1" direction="output">
         <description>Output One</description>
      </port>
   </component>

</transactors>
```

B.2 Writing the Transactor C++ Code

The structure of a user-defined transactor is similar to the structure of an SoC Designer component.

Main transactor class

You must provide a C++ class with these methods:

```
class MyXtorClass
{
    // constructor
    MyXtor(sc_mx_module* carbonComp, const char *xtorName,
        CarbonObjectID **carbonObj, CarbonPortFactory *portFactory);

    // destructor
    ~MyXtor();

    // provide input port handles to the Carbon component
    sc_mx_signal_slave *carbonGetPort(const char *name);

    // similar to sc_mx_module
    void communicate();
    void update();
    void init();
    void terminate();
    void reset(MxResetLevel level, const MxFileMapIF *filelist);
    void setParameter(const string &name, const string &value);
};
```

Transactor output ports

Transactor output ports are connected to Cycle Model input ports by the Carbon Model Studio wizard. The transactor writes signal values to the Cycle Model by calling the `driveSignal` method of output port objects.

Output port handles are obtained by name via the `portFactory` function passed to the constructor. The `portFactory` function returns a pointer of type `CarbonXtorAdaptorToVhmPort`:

```
MyXtor::MyXtor(sc_mx_module* carbonComp,
          const char *xtorName,
          CarbonObjectID **carbonObj,
          CarbonPortFactory *portFactory)
{
    mOutPort = portFactory(carbonComp,
                 xtorName, "output_1", carbonObj);
}
```

Writing to output ports

Ports returned by the `portFactory` have the same '`driveSignal`' interface as `sc_mx_signal_slave`:

```
void driveSignal(MxU32 value, MxU32* extValue)

    value = low 32 bits
    extValue = pointer to the rest of the bits
```

Transactor input ports

Input ports are connected to Cycle Model output signals by the Carbon Model Studio wizard. The Cycle Model sends signal value changes to the transactor by calling the `driveSignal` method of input port objects.

Implement `sc_mx_signal_slave` ports just as you would for an SoC Designer component:

- Subclass `sc_mx_signal_slave.`
- Override `driveSignal()` to do what you need.

Implement carbonGetPort

The component needs to be able to get a handle to your input ports so that it can write values to them.

```
sc_mx_signal_slave *MyXtor::carbonGetPort(const char *name)
{
    sc_mx_signal_slave *port = NULL;

    if (strcmp("input_1", name) == 0) {
        port = mInput1;
    }
    return port;
}
```

init, reset, communicate, update, terminate

The `init`, `reset`, `communicate`, `update`, and `terminate` functions are all called by the component when its own method of the same name is invoked. See the *SoC Designer User Guide* for information on how these methods are used.

For more information about the C++ classes and member functions used to write your transactor, contact ARM Technical Support.

B.3 Adding your Transactor to the Component

To connect your transactor appropriately in the Carbon Model Studio tool:

1. Before creating the SoC Designer Component, specify the name of your user-defined transactor in the *Transactor Definition File* field on the SoC Designer Component Properties page.

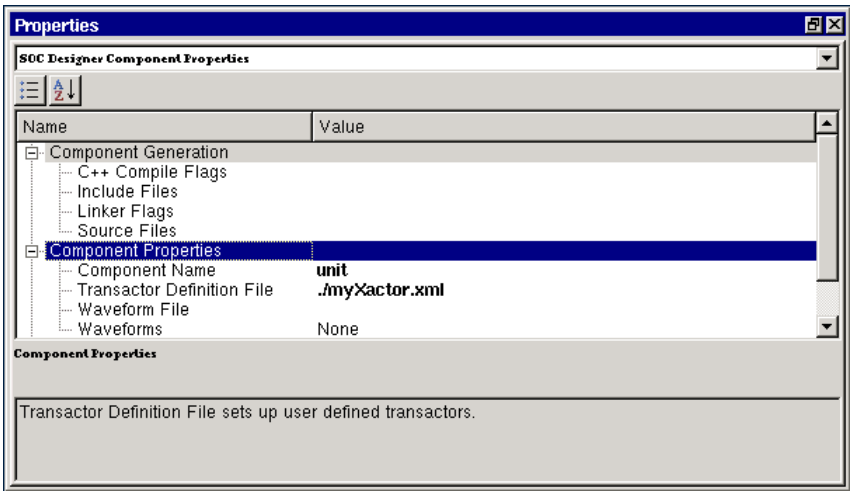


Figure B-1 SoC Designer Component Settings

After you generate the component for SoC Designer, your transactor is available from the list of available transactors in Carbon Model Studio.

2. On the SoC Designer Component Editor *Ports* tab:
   - Click the **Add Xtor** button.
   - In the pulldown menu next to the **Add Xtor** button, select the name of your transactor.
   - Map your transactor ports to RTL ports as outlined in "Adding Transactors and Other Interface Entities" on page 103.
3. On the SoC Designer Component Properties page, add any compiler or linker flags needed by your transactor.

Appendix C

ARM-Supplied RTL Models

ARM provides a family of simple RTL models for use in designs compiled by Carbon Model Studio. Designed for high performance and easy integration, these models are instantiated directly in your RTL design and compiled together to form a single Cycle model. When targeting the SoC Designer Plus platform, Carbon Model Studio automatically detects the presence of one or more of these Cycle models and adds the appropriate parameters, registers, and memory views to the SoC Designer Plus component.

This appendix includes the following sections:

- Overview
- Using ARM-supplied RTL Models
- Specific Requirements for DDRx Memory Models

C.1 Overview

ARM provides a family of simple double data rate (DDR) memory models. In this document, we refer to the family, which includes such models as DDR, DDR2, DDR3, DDR4, and LPDDR2, as *DDRx Memory Models*.

Designed for high performance and easy integration, these Verilog memory models connect directly to RTL-based memory controllers. A wide range of memory configurations are supported through compile time parameters, and each model's interface timing can be adjusted to simplify the connection to the controller's physical layer interface.

The DDRx Memory Models are simple cycle-accurate functional memory models designed to be embedded along with a memory controller and its physical layer interface to form a single memory subsystem component model. The DDRx Memory Model interfaces with the memory controller physical layer interface at the pin level, as defined by the appropriate JEDEC stan-

dard. You must provide the top level wrapper that instantiates the RTL memory controller and the DDRx Memory Model in a single module, and compile this using Carbon Model Studio to create the memory subsystem model. Figure C-1 illustrates this usage.

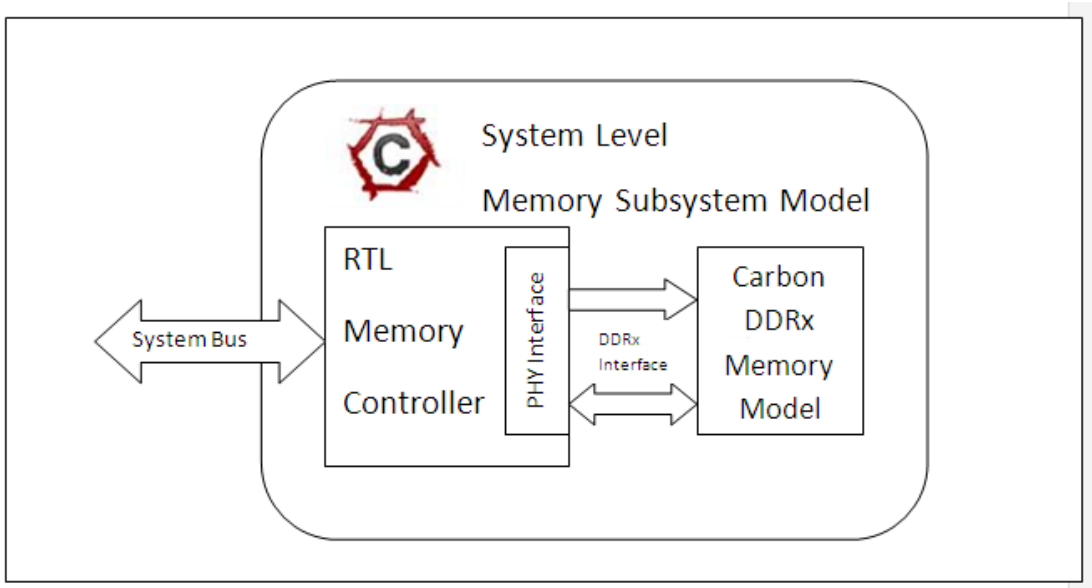


Figure C-1 DDRx Memory Model Usage

Figure C-2 illustrates the internal hierarchy of the DDRx Memory Model shown on the right in Figure C-1.

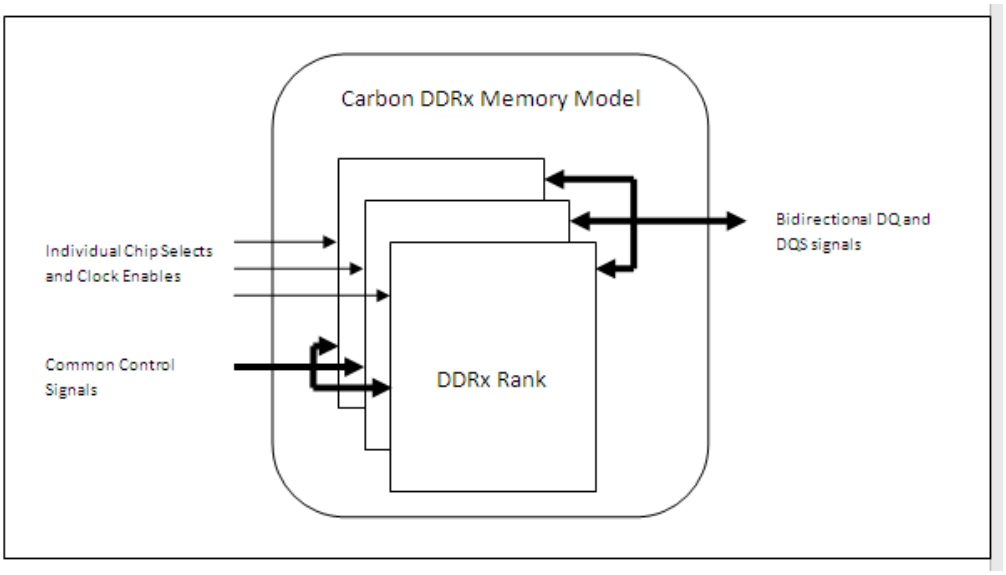


Figure C-2 Internal Hierarchy of DDRx Memory Models

C.1.1 DDRx Memory Model Location

The ARM-supplied RTL models and their command files are distributed as part of the Carbon Model Studio release and are located in the immediate sub-directories of:

```
$CARBON_HOME/lib/carbonRTL
```

For example, the `carbon_memory_ddr` subdirectory contains the following files:

```
carbon_memory_ddr.cmd
carbon_memory_ddr.v
```

The readme file for this release contains a list of models that are currently supported. Specific details about the supplied RTL model interface and parameters can be found in the file that defines the model, located in `$CARBON_HOME/lib/carbonRTL/*`.

C.2 Using ARM-supplied RTL Models

To use an ARM-supplied RTL model, you must perform two steps, which are described in the sections that follow:

1. Instantiate the desired module or modules in your design with the appropriate parameters (described in the section Instantiating the Module).
2. Add an ARM-supplied command file to the project for each model type you use (described in the section Adding the ARM-supplied Command File).

This section also describes:

- Configuring DDRx Size and Signal Timing
- DDRx Mode Registers
- DDRx Profiling Capabilities

C.2.1 Instantiating the Module

The following is a Verilog example of instantiating a module. In this example, an instance of the carbon_memory_ddr2 module is included in the module named **ddr2_memory_block**.

```
module ddr2_memory_block();
…
   carbon_memory_ddr2 #(.CS_BITS(3),
                        .DATA_BITS(64),
                        .MAX_ADDR_BITS(33),
                        .DQS_IN_DELAY(1),
                        .DQS_OUT_DELAY(0)) myDDR2(.ck(ck),
                                                  .ckbar(ckbar),
                                                  .cke(cke),
                                                  .cs_n(cs_n),
                                                  .ras_n(ras_n),
                                                  .cas_n(cas_n),
                                                  .we_n(we_n),
                                                  .dm(dm),
                                                  .ba(ba),
                                                  .addr(addr),
                                                  .dq(dq),
                                                  .dqs(dqs));
```

```
...
endmodule
```

C.2.2 Adding the ARM-supplied Command File

The command file for the module is the module name with a `.cmd` extension. For example, the name of the command file for the carbon_memory_ddr2 module is:

```
$CARBON_HOME/lib/carbonRTL/carbon_memory_ddr2/
carbon_memory_ddr2.cmd
```

This command file for a memory is located in the immediate sub-directory of `$CARBON_HOME/lib/carbonRTL` that corresponds to the memory name.

Add the file to the Carbon Model Studio project by using the **-f** switch on the SoC Designer Component Properties page as follows:

1. Click *RTL Sources* in the *Project Explorer* pane.
2. Choose **Project > Compiler Setting**. The *Compiler Options* screen appears.
3. Under *Input File Control*, enter the full pathname of the `.cmd` file.

   If your design includes multiple instances of a particular model, only one copy of the `.cmd` file needs to be included as an argument to the **-f** switch.

   $CARBON_HOME/lib/carbonRTL/

See the Compiler Settings section "Using the Project Menu" on page 32 for more information on setting compiler flags.

C.2.3 Configuring DDRx Size and Signal Timing

Several compile time Verilog parameters are available to configure the size of the DDRx Memory Model and the timing it uses for the DQS/DQ signaling.

Table C-1 describes the compile time Verilog parameters that are available. "Specific Requirements for DDRx Memory Models" on page 239 and the pages following describe which must be specified for each memory model.

Table C-1 Compile-time Verilog parameters

Parameter	Description
CS_BITS	The number of chip selects and clock enable inputs to be generated. These signals are provided in vector form. **CS_BITS** determines the number of memory ranks in the model. For any given instance of a DDR*x* memory model, all ranks have the same configuration as determined by the other parameters. If you need different rank configurations, then you must have different instantiations of the model. While this use case is not precluded, the integration into a target platform environment is more complicated.
DATA_BITS	The data bus width. While the underlying Verilog code does not place any requirements on the width of the data bus, the integration-related code assumes 8, 16, 32, 64, and 128 are the only valid values. Note that widths that include error checking/correction bits are not supported. Disable error checking/correction bits in the controller logic.
MAX_ADDR_BITS	The maximum number of bits that together make up the bank, row, and column address fields. The specific values for the widths of the bank, row, and column address fields are set using SoCD parameters and their sum cannot exceed the **MAX_ADDR_BITS** value.
DQS_IN_DELAY	This is the number of ½ clock cycle delays that should be applied to the **DQS_IN** signal before being used internally by the model as an internal strobe. If the memory controller is remodeled to generate an early **DQS_IN** signal, set this parameter to 1, otherwise set it to 0. (**DQS_IN_DELAY** = 0 means the signal can already be directly used to strobe in the data.)

Table C-1 Compile-time Verilog parameters (continued)

Parameter	Description
DQS_OUT_DELAY	This is the number of ½ clock cycle delays that should be applied to the **DQS_OUT** signal. If the memory controller expects the nominal early DQS signal and adjusts the DQS timing itself, set **DQS_OUT_DELAY** to 0. However, if it is more convenient to remodel the memory controller data capture circuits to directly use the DQS strobe provided by memory model, then set DQS_OUT_DELAY to 1. For more details concerning DQS timing and modeling, refer to the application note *Cycle-based Modeling of DDR-style Data Transfers*.
WRITE_PREAMBLE (DDR3, DDR4, and LPDDR3 only)	Determines the number of DQS Write preambles the memory should expect. • DDR3 defaults to 1. Allowed values are 0 and 1. • DDR4 defaults to 0. Allowed values are 0, 1, and 2. • LPDDR3 defaults to 0. Allowed values are 0 and 1.
READ_PREAMBLE (DDR4 only)	Determines the number of DQS Read preambles the memory should expect. Defaults to 1. Allowed values are 0, 1, and 2.
MR8_SETTING (LPDDR2 and LPDDR3 only)	Defines the reset value of Mode Register 8 (also known as Basic Configuration 4), which defines the I/O width, Density, and Type of memory. The default is 1.

C.2.4 DDRx Mode Registers

Each embedded DDRx memory has internal mode registers; they are slightly different for each memory model type. These registers are implemented according to the JEDEC specification for each protocol, and configure the operation of the memory — for example, the number of clock cycles it takes the memory to respond to Read and Write commands is set by programming the appropriate values in the mode registers. The Memory Model Mode registers are automatically made visible via CADI registers for each Chip Select.

Figure C-3 shows the Mode Registers tab for the DDR3 memory model (the Commands, Bank Commands, and Events tabs display registers related to profiling, and are discussed in the next section). See the individual JEDEC protocol specification for details about the function of each register.

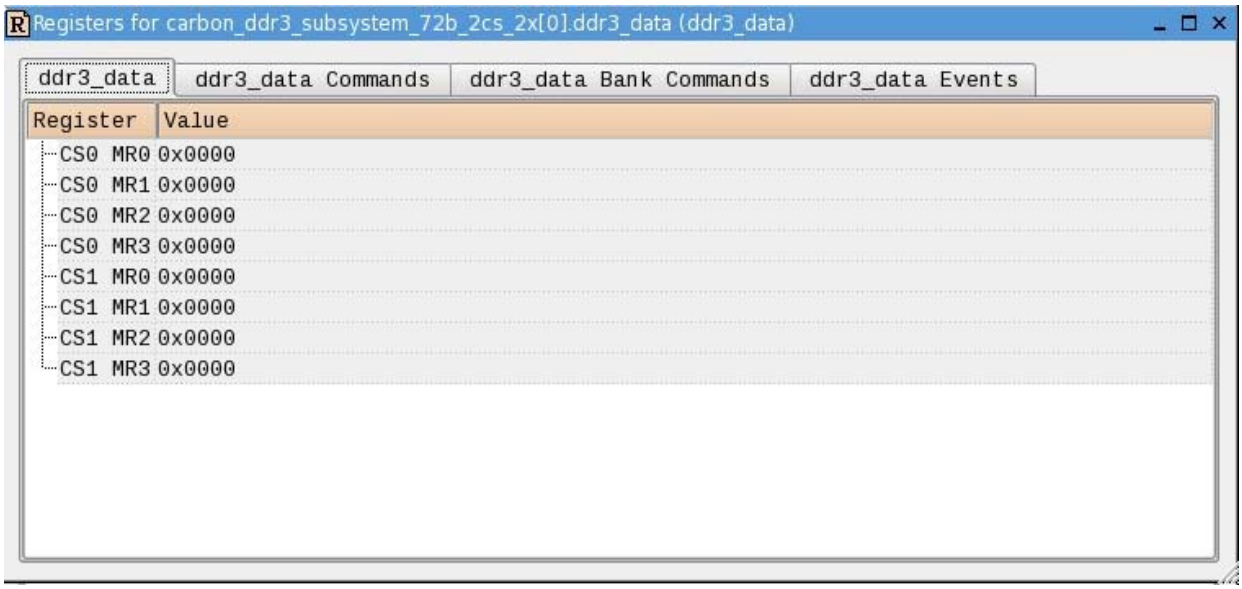


Figure C-3 DDRx Mode Registers

C.2.5 DDRx Profiling Capabilities

The profiling capabilities discussed in this section are supported by all memory models except the GDDR5.

The memory models support profiling through the use of counters in the RTL that track DDR command-related events. These RTL counters and event signals are automatically instrumented for models built using Carbon Model Studio.

Note that only Read/Write CADI register profiling is supported; CAPI profiling streams are not generated.

Profiling data includes the following events and counters:

- **Command Counters** — For each Chip Select:
  - Provides the total count of ACTIVATE, READ, WRITE, PRECHARGE, and REFRESH commands.
  - Counts clock cycles where READ DATA or WRITE DATA is transferred. For example, a single Write Burst 8 DDR3 transfer increments the WRITE count by 1, and the WRITE DATA by 4.
  - Calculates the bus utilization percentages (READ, WRITE, and Total) over a sliding 100 cycle window. In other words, a history of the last 100 cycles is maintained, and the utilization is the count of READ DATA/WRITE DATA events during that time.
- **Bank Command Counters** — Provides additional bank-level detail for the ACTIVATE, READ, and WRITE commands. This allows analysis of how the accesses are distributed across banks as well as the efficiency of ACTIVATE commands at the bank level.
- **Events** — Raw event registers for each Chip Select including ACTIVATE, READ, WRITE, BURST, PRECHARGE, REFRESH, READ DATA, and WRITE DATA. Values are pulsed for one cycle for the event, with the bit position representing the bank where the command is active. For example, an ACTIVATE Bank 3 command pulses the value 0x0008 on the ACTIVATE Event (bit 3 is asserted).

C.3 Specific Requirements for DDRx Memory Models

The following sections provide details of the file locations and compile-time parameter requirements of each DDRx Memory Model that Carbon Model Studio supports:

- DDR Memory
- DDR2 Memory
- DDR3/DDR4 Memory
- LPDDR2/LPDDR3 Memory
- GDDR5 Memory

C.3.1 DDR Memory

Directory

$CARBON_HOME/lib/carbonRTL/carbon_memory_ddr

RTL File

$CARBON_HOME/lib/carbonRTL/carbon_memory_ddr/carbon_memory_ddr.v

Compilation Command File

$CARBON_HOME/lib/carbonRTL/carbon_memory_ddr/carbon_memory_ddr.cmd

Verilog Parameters

See "Configuring DDRx Size and Signal Timing" *on page 235 for details.*

CS_BITS
DATA_BITS
MAX_ADDR_BITS
DQS_IN_DELAY
DQS_OUT_DELAY

SoC Designer Plus Parameters for DDR

The following parameters can be changed in the SoC Designer Canvas only. Init-time parameters can be changed in the SoC Designer Canvas only; run-time parameters can be changed in the SoC Designer Canvas and at runtime during simulation.

Table C-2 DDR Parameters

Parameter	Description
<*component_name*>_Address_Format	Allows you to select between Row_Bank_Column (**RBC**) or Bank_Row_Column (**BRC**) format.
<*component_name*>_Bank_Bits	Sets the number of bits used for the bank portion of the address. The maximum value is 2.
<*component_name*>_Row_Bits	Sets the number of bits used in the row portion of the address. The maximum value is 16.
<*component_name*>_Column Bits	Sets the number of bits used in the column portion of the address. The maximum value is 15.

C.3.2 DDR2 Memory

Directory

$CARBON_HOME/lib/carbonRTL/carbon_memory_ddr2

RTL File

$CARBON_HOME/lib/carbonRTL/carbon_memory_ddr2/carbon_memory_ddr2.v

Compilation Command File

$CARBON_HOME/lib/carbonRTL/carbon_memory_ddr2/carbon_memory_ddr2.cmd

Verilog Parameters

See "Configuring DDRx Size and Signal Timing" *on page 235 for details.*

CS_BITS
DATA_BITS
MAX_ADDR_BITS
DQS_IN_DELAY
DQS_OUT_DELAY

SoC Designer Plus Parameters for DDR2

The following parameters can be changed in the SoC Designer Canvas only. Init-time parameters can be changed in the SoC Designer Canvas only; run-time parameters can be changed in the SoC Designer Canvas and at runtime during simulation.

Table C-3 DDR2 Parameters

Parameter	Description
<*component_name*>_Address_Format	Allows you to select between Row_Bank_Column (**RBC**) or Bank_Row_Column (**BRC**) format.
<*component_name*>_Bank_Bits	Sets the number of bits used for the bank portion of the address. The maximum value is 3.
<*component_name*>_Row_Bits	Sets the number of bits used in the row portion of the address. The maximum value is 16.
<*component_name*>_Column Bits	Sets the number of bits used in the column portion of the address. The maximum value is 15.

C.3.3 DDR3/DDR4 Memory

Directory

- DDR3 — $CARBON_HOME/lib/carbonRTL/carbon_memory_ddr3
- DDR4 — $CARBON_HOME/lib/carbonRTL/carbon_memory_ddr4

RTL File

- DDR3 — $CARBON_HOME/lib/carbonRTL/carbon_memory_ddr4/carbon_memory_ddr3.v
- DDR4 — $CARBON_HOME/lib/carbonRTL/carbon_memory_ddr4/carbon_memory_ddr4.v

Compilation Command File

- DDR3 — $CARBON_HOME/lib/carbonRTL/carbon_memory_ddr3/carbon_memory_ddr3.cmd
- DDR4 — $CARBON_HOME/lib/carbonRTL/carbon_memory_ddr4/carbon_memory_ddr4.cmd

Verilog Parameters

See "Configuring DDRx Size and Signal Timing" *on page 235 for details.*

CS_BITS
DATA_BITS
MAX_ADDR_BITS
DQS_IN_DELAY
DQS_OUT_DELAY
WRITE_PREAMBLE
READ_PREAMBLE (DDR4 only)

SoC Designer Plus Parameters for DDR3/DDR4

The following parameters can be changed in the SoC Designer Canvas only. (Init-time parameters can be changed in the SoC Designer Canvas only; run-time parameters can be changed in the SoC Designer Canvas and at runtime during simulation.)

Table C-4 DDR3/DDR4 Parameters

Parameter	Description
<*component_name*>_Address_Format	Allows you to select between Row_Bank_Column (**RBC**) or Bank_Row_Column (**BRC**) format.
<*component_name*>_Bank_Bits	Sets the number of bits used for the bank portion of the address. The maximum value is 3.
<*component_name*>_Row_Bits	Sets the number of bits used in the row portion of the address. The maximum value is 16.
<*component_name*>_Column Bits	Sets the number of bits used in the column portion of the address. The maximum value is 15.

C.3.4 LPDDR2/LPDDR3 Memory

This section describes both the LPDDR2 and LPDDR3 memory types.

Directory

- LPDDR2 — $CARBON_HOME/lib/carbonRTL/carbon_memory_lpddr2
- LPDDR3 — $CARBON_HOME/lib/carbonRTL/carbon_memory_lpddr3

RTL File

- LPDDR2 — $CARBON_HOME/lib/carbonRTL/carbon_memory_lpddr2/carbon_memory_lpddr2.v
- LPDDR3 — $CARBON_HOME/lib/carbonRTL/carbon_memory_lpddr3/carbon_memory_lpddr3.v

Compilation Command File

- LPDDR2 — $CARBON_HOME/lib/carbonRTL/carbon_memory_lpddr2/carbon_memory_lpddr2.cmd
- LPDDR3 — $CARBON_HOME/lib/carbonRTL/carbon_memory_lpddr3/carbon_memory_lpddr3.cmd

Verilog Parameters

See "Configuring DDRx Size and Signal Timing" *on page 235 for details.*

CS_BITS
DATA_BITS
MAX_ADDR_BITS
DQS_IN_DELAY
DQS_OUT_DELAY
WRITE_PREAMBLE (LPDDR3 only)
MR8_SETTING

SoC Designer Plus Parameters for LPDDR2 and LPDDR3

The following parameters can be changed in the SoC Designer Canvas only. Init-time parameters can be changed in the SoC Designer Canvas only; run-time parameters can be changed in the SoC Designer Canvas and at runtime during simulation.

Table C-5 LPDDR2 AND LPDDR3 Parameters

Parameter	Description
<*component_name*>_Address_Format	Allows you to select between Row_Bank_Column (**RBC**) or Bank_Row_Column (**BRC**) format.
<*component_name*>_Bank_Bits	Sets the number of bits used for the bank portion of the address. The maximum value is 3.
<*component_name*>_Row_Bits	Sets the number of bits used in the row portion of the address. The maximum value is 16.
<*component_name*>_Column Bits	Sets the number of bits used in the column portion of the address. The maximum value is 14.
<*component_name*>_tdqsck	Sets the value used to specify any additional delay from nominal clock cycles. Note you can specify half cycles. Default value is 0.0.

C.3.5 GDDR5 Memory

Due to the unique requirements of the GDDR5 protocol, the GDDR5 Memory Model is slightly different from the other DDRx Memory Models. Rather than modeling the full memory subsystem in an abstract manner with multiple chip selects and common control signals, the GDDR5 Memory Model implements a single logical memory block while exposing the underlying individual device model signals. This allows total flexibility in integrating the model with a memory controller, while still providing a consolidated logical debug view of the memory.

For more details concerning the supported features and limitations of the GDDR5 Model, please examine the RTL source file.

Directory

$CARBON_HOME/lib/carbonRTL/carbon_memory_gddr5

RTL File

$CARBON_HOME/lib/carbonRTL/carbon_memory_gddr5/carbon_memory_gddr5.v

Compilation Command File

$CARBON_HOME/lib/carbonRTL/carbon_memory_gddr5/carbon_memory_gddr5.cmd

Verilog Parameters

DEVICE_BYTE_LANES — Number of bytes each underlying device implements. Allowed values are 2 and 4, representing 16 bit and 32 bit wide devices, respectively.

DEVICES_PER_RANK — Number of devices present in this memory system. The total data width of the memory model is given by 8*DEVICE_BYTE_LANES*DEVICES_PER_RANK.

MAX_ADDR_BITS — As with other DDRx Memory Models, this parameter represents the maximum number of bits that together make up the bank, row, and column address fields. It is recommended that this parameter is left at its default setting of 24, which is the maximum allowed by standard GDDR5 devices.

SoC Designer Plus Parameter Description

The following parameters can be changed in the SoC Designer Canvas only. Init-time parameters can be changed in the SoC Designer Canvas only; run-time parameters can be changed in the SoC Designer Canvas and at runtime during simulation.

Table C-6 GDDR5 Parameters

Parameter	Description
<*component_name*>_Address _Format	Allows you to select between Row_Bank_Column (**RBC**) or Bank_Row_Column (**BRC**) format.
<*component_name*>_Bank_Bits	Sets the number of bits used for the bank portion of the address. Allowed values are 3 and 4.
<*component_name*>_Row_Bits	Sets the number of bits used in the row portion of the address. Allowed values are 12 and 13.
<*component_name*>_Column Bits	Sets the number of bits used in the column portion of the address. Allowed values are 6 and 7.

Index

D

E

F

G

R

S

T

U

V

W

X