Cycle Model Compiler

Version 11.4

User Guide

Cycle Model Compiler

User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
1102-00	15 January 2020	Non-Confidential	Release 11.2
1103-00	13 November 2020	Non-Confidential	Release 11.3
1103-01	20 January 2021	Non-Confidential	Documentation update
1104-00	12 February 2021	Non-Confidential	Release 11.4

Non-Confidential Proprietary Notice

Contents

Cycle Model Compiler User Guide

Preface

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

It contains the following:

- *About this book* on page 7.

About this book

This guide describes how to use the Cycle Model API to generate a Cycle Model from RTL.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction
This chapter provides an overview of the Cycle Model Compiler and how it fits into the Arm Cycle Models system validation workflow.

Chapter 2 Getting started with the Cycle Model Compiler
The Cycle Model Compiler package includes example design files in the `examples` directory. This chapter describes running the `TwoCounter` example, which is a simple design with two counters driven by two clocks.

Chapter 3 Cycle Model Compiler command line options
This chapter describes the command-line options available to control Cycle Model compilation.

Chapter 4 Using Cycle Model Compiler directives
Directives are compiler commands that can be contained in a directives file. Directives help control how the Cycle Model Compiler interprets your design. This chapter describes the Cycle Model Compiler directives.

Chapter 5 Net Control directives
This chapter describes the Cycle Model Compiler directives that control nets.

Chapter 6 Module Control directives
This chapter describes the Cycle Model Compiler directives that control modules.

Chapter 7 Output Control directives
This chapter describes directives that control Cycle Model Compiler output messages.

Appendix A Dumping waveforms in different environments
Dumping waveforms from a simulation is important to the debug process. This appendix describes waveform dumping procedures for various environments.

Appendix B Using DesignWare replacement modules
This appendix describes replacement modules that may help make your design run faster.

Appendix C Using Profiling to find performance problems
This appendix describes the profiling capabilities of the Cycle Model Compiler.

Appendix D Using Design Reducer to troubleshoot design errors
This appendix describes how to use the Design Reducer utility. Use this tool when working with Arm support to troubleshoot your design.

Glossary

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

See the *Arm® Glossary* for more information.

Typographic conventions

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

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

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

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

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

`monospace bold`
Denotes language keywords when used outside example code.

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

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

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

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:
- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

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

- The title *Cycle Model Compiler User Guide*.
- The number 101050_1104_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

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

Other information

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

Chapter 1
Introduction

This chapter provides an overview of the Cycle Model Compiler and how it fits into the Arm Cycle Models system validation workflow.

It contains the following sections:

1.1 Validation methodology

Arm Cycle Models tools provide an integrated environment that places system validation in parallel with the hardware development flow.

The Cycle Model Compiler takes an RTL hardware model and creates a high-performance linkable object, called the Cycle Model, that is cycle- and register- accurate. The Cycle Model Compiler provides an API for interfacing with your validation environment:

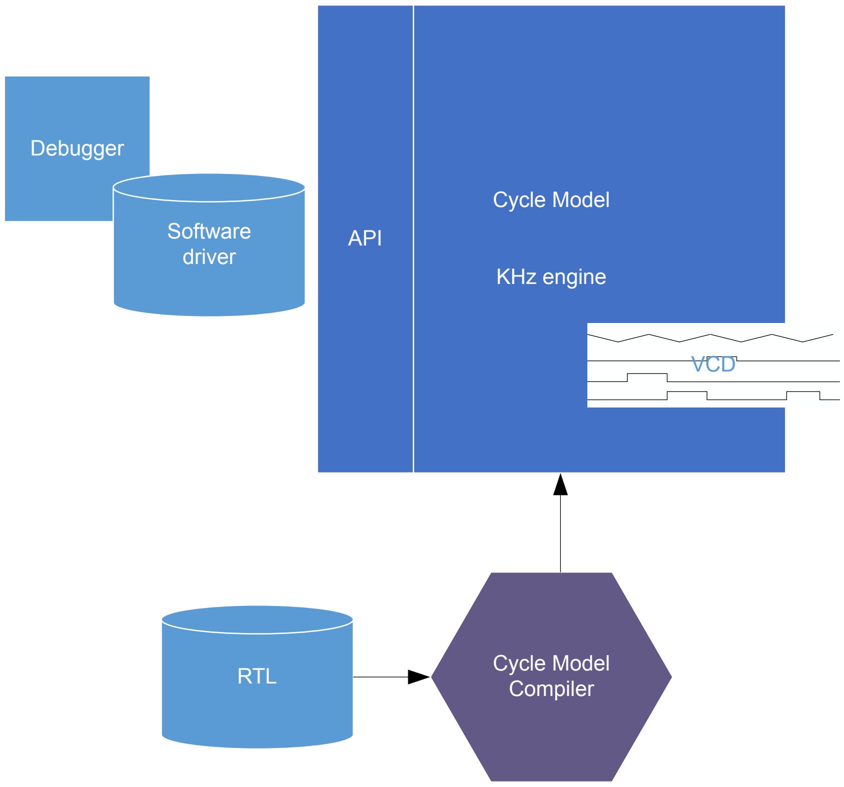

Figure 1-1 Validation environment

1.2 Compiler inputs

To generate a Cycle Model, the Cycle Model Compiler reads certain files and generates a Cycle Model for the design.

The files, and the order in which they are read, are:

1. Options files – Contain command options that provide control and guidance to the Cycle Model Compiler.
2. Directives files – Contain directives that control how the Cycle Model Compiler interprets and builds a Cycle Model.
3. Verilog® design and library files – Golden RTL of the hardware design.

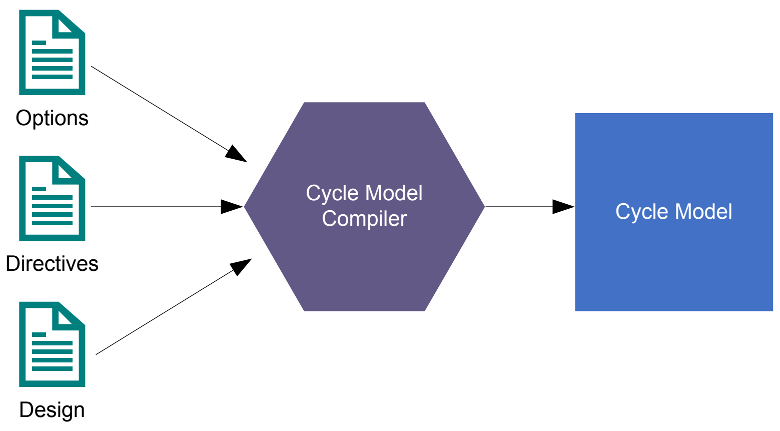

Figure 1-2 Input files for Cycle Model generation

For more information about available options and directives, see *Chapter 3 Cycle Model Compiler command line options* on page 3-26 and *Chapter 4 Using Cycle Model Compiler directives* on page 4-49.

1.3 What is a Cycle Model?

A Cycle Model is a high-performance linkable software object that is generated by the Cycle Model Compiler directly from RTL design files.

The Cycle Model contains a cycle and register-accurate model of the hardware design in the form of a software object file, header file, and supporting binary database. By default, the Cycle Model Compiler generates these files in the current working directory (`./`) as listed below:

- `libdesign.a` – Cycle Model object library
- `libdesign.h` – Cycle Model header file
- `libdesign.symtab.db` – database with information about all internal signals
- `libdesign.io.db` – a subset of `libdesign.symtab.db` that includes top-level inputs, outputs, inouts, and those signals marked as observable or depositable (to the external environment)

In general, when integrating the Cycle Model into a simulation environment, use the `symtab.db`. The `io.db` file is provided for use if you are passing your Cycle Model on to a third-party customer and you want to restrict the visibility of your design.

For information about the Cycle Model Compiler output files, see *2.3 Compilation output files* on page 2-24

1.4 Simulation dependencies and requirements

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

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

1.5 Creating the Cycle Model executable

Cycle Model files must be linked with a standard software compiler, such as GCC, before they can be run in the software test environment.

The following files are required to create a proper executable:
- Cycle Model (`libdesign.a`)
- Cycle Model header file (`libdesign.h`)
- Cycle Model shell (`libcarbon5.so`)
- `carbon_capi.h` – API header file (included in the installation package)

In addition, the `.db` file must be accessible to the Cycle Model at runtime (it should be located in the same directory as the validation executable).

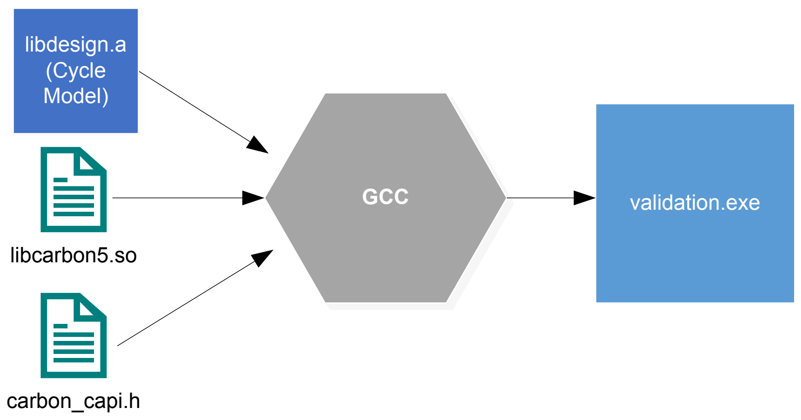

Figure 1-3 Generating an executable model

The Cycle Model is controlled by your software test environment. A software program, such as a driver, can communicate with the hardware model directly through sockets, through the Cycle Model API, or through an Instruction Set (ISS). For embedded software—where the software is loaded into a hardware memory model—a software debugger may be linked to the embedded software through the Cycle Model API.

1.6 Using the Cycle Model API

The Cycle Model API provides the C functions necessary to link a Cycle Model into a software test environment.

You can access the value of any net in the design (including memories), deposit values on nets, dump signal waveforms, and supply time and timescale information to the design.

The Cycle Model API is handle based, meaning that in order to query and manipulate a specific design structure in a Cycle Model, you must use its handle or ID (rather than the full HDL path name). Following are the primary reference structures that provide access into a Cycle Model:

- `CarbonObjectID` – Provides the context for a design, and is used to run the design's core functions.
- `CarbonWaveID` – Provides signal waveform dump control. Standard Verilog VCD and Verdi FSDB formats are supported.
- `CarbonNetID` – Used to access nets in the design. API functions allow you to examine signal values, deposit values on signals, and force signals to specific values.
- `CarbonMemoryID` – Used to access memories in the design. API functions allow you to examine memory values and deposit values into memories.

See the *Cycle Model API Reference Manual* (101067) for detailed information about all API files and functions.

1.7 Remodeling unsupported RTL constructs

There are certain hardware constructs that the Cycle Model Compiler does not currently support; they must be remodeled using supported constructs.

Remodeling is required for proper Cycle Model Compiler function. Tips for additional remodeling that can improve performance can be found in the *Cycle Model Studio RTL Style Guide* (101769).

For a report of supported and unsupported SystemVerilog language constructs used in your design, use the Cycle Model Compiler `-SVInspector` option (see *3.2 General compile control options* on page 3-28.

For detailed information about Cycle Model Compiler support for Verilog and SystemVerilog language constructs, see the *Cycle Model Compiler Verilog and SystemVerilog Language Support Guide* (100972).

Phase-locked loops

PLLs implement behavior that occurs without cycle dependencies, and therefore are not supported. Designs that use PLLs must be remodeled to bypass the PLL and drive the generated clocks from the external environment via the API, or to provide pass-through logic. The Cycle Model API is able to drive PLL-generated clocks without needing to bring the clock to a primary input.

Memories

Vendor-provided memory libraries often use behavioral constructs that the Cycle Model Compiler does not support. These memories need to be remodeled using constructs supported by the Cycle Model Compiler.

Low-level constructs and gate-level modeling

Though the Cycle Model Compiler supports gate-level constructs, the use of high-level Verilog constructs generally yields higher performance Cycle Models and is highly recommended. A common example of a gate-level construct that can be improved with high-level modeling is a pad cell.

1.8 Specifying the Verilog language variant to use when compiling

This section provides information about Cycle Model Compiler compilation modes.

Default setting

By default, the Cycle Model Compiler processes design files using the Verilog 95 language definition (IEEE 1634-1995).

Supported compilation modes

The Cycle Model Compiler includes the following options for selecting the Verilog language variant:

- -95 (alias is -v95). This is the default setting.
- -2001 (aliases are -v2k and -2000). This option includes partial support for Verilog-2005 (IEEE Std 1364-2005) language features. All files encountered during the compilation are treated as Verilog 2001.
- -sverilog. All Verilog files encountered during compilation are treated as SystemVerilog source files.

Full and partial compilation

The Cycle Model Compiler does not support partial compilation using compilation units as described in Section 3.12.1 of the Language Standard; full compilation is supported. This means that you must include a specification of all Verilog files when you issue the `cbuild` command.

Compiler behavior when multiple modes are specified

If more than one Verilog switch is specified, the Cycle Model Compiler issues the following alert, which identifies the variant that the compiler would use, if you demote the alert to a warning:

```
Alert 183: Multiple Verilog compilation mode switches are selected on command line or with
-f file. If the severity of this message is Note or Warning, language variant switch is
used.
```

Arm recommends correcting the command line so that a single variant is specified. However, you also have the option of demoting the alert to a warning. In this case, the message text identifies the variant that the compiler will use.

1.9 Getting information about error codes

Compilation errors are displayed to standard output.

Error notifications consist of an error number and a brief error description. For additional information about errors, see the help file `Arm_CM.help` in the directory `$CARBON_HOME/userdoc/help`.

1.10 Data collection in the Cycle Model Compiler

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

Host information includes:

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

——— **Note** ———

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

Chapter 2
Getting started with the Cycle Model Compiler

The Cycle Model Compiler package includes example design files in the `examples` directory. This chapter describes running the `TwoCounter` example, which is a simple design with two counters driven by two clocks.

It contains the following sections:

2.1 Setting up the example environment

Compile the TwoCounter design from the appropriate input files, then link the resulting software executable to a testbench.

To set up your environment for running the TwoCounter example:

1. Using syntax appropriate to your shell, set the `CARBON_HOME`environment variable:

   ```
   CARBON_HOME =installation directoryPATH =$CARBON_HOME/bin:$PATH
   ```
2. Create a working directory for your experiments. For example:

   ```
   mkdir ~/cycle_model_experiment
   cd ~/cycle_model_experiment
   ```
3. Copy the example files into your local work directory:

   ```
   cp -r $CARBON_HOME/examples/twocounter ./twocounter
   ```
4. Change to the twocounter directory:

   ```
   cd twocounter
   ```

The files in this directory are:

- `Makefile`
- `Makefile.shared`
- `Makefile.notes`
- `twocounter.v` – HDL code for the design
- `twocounter.c` – C code for test harness
- `twocounter.gold` – expected output for test harness

2.2 Running the example

Create the Cycle Model from the TwoCounter example design, then create and test a validation executable using your Cycle Model.

See the `twocounter.v` file to view the Verilog design.

Before you begin

Ensure that your environment meets the requirements in *1.4 Simulation dependencies and requirements* on page 1-13.

It is important to understand how the `Makefile` works before running an example.

During the validation process, the `Makefile` uses variables to invoke the Cycle Model Compiler, to invoke GNU compilers, and to access a list of link libraries. Using these variables in your project's `Makefile` helps ensure smooth operation, and facilitates future product upgrades.

The `Makefile`:
1. Compiles the design file into a Cycle Model using the Cycle Model Compiler.
2. Compiles the software harness into a software harness object using `gcc`.
3. Links the Cycle Model with the software harness object using g++ to produce a software validation executable.
4. Invokes the software validation executable to generate run-time output.
5. Compares the run-time output with the expected values (`twocounter.gold` file).

The `CARBON_LIB_LIST make` variable links the program so that `LD_LIBRARY_PATH` overrides `-rpath`. To avoid library conflicts, ensure you are using a single GCC version within your environment, as described in *1.4 Simulation dependencies and requirements* on page 1-13.

Step 1: Compile the Cycle Model library object

The TwoCounter example is a Verilog design file. To compile a Cycle Model for this design, enter the following command:

```
$ make libtwocounter.a
```

Alternatively, compile the Cycle Model using the `cbuild` command:

```
$ cbuild twocounter.v -o libtwocounter.a
```

The `-o` option specifies a name for the Cycle Model Compiler output files.

The `.a` extension generates a traditional object archive.

Result

The Cycle Model Compiler compiles a Cycle Model object called `libtwocounter.a` for the specified design. The Cycle Model Compiler also creates `libtwocounter.h`, which will be used to link the object into the test environment.

The following message is generated upon successful completion of the compilation:

```
Note 111: Successfully created libtwocounter.
```

If a compilation is unsuccessful, the Cycle Model Compiler generates a message that indicates which phase the error occurred in, as well as the number of warning, error, and alert messages:

```
Note 110: There were errors during the phase name phase, cannot continue. 0 warnings, 1
errors, and 1 alerts detected.
```

The help file `Arm_CM.help`, in the directory `$CARBON_HOME/userdoc/help`, lists all the Cycle Model Compiler error and warning messages. .

Step 2:

Examine the `twocounter.c` file. This file contains Cycle Model API code that directs the executable in test. Notice that the header file `libtwocounter.h` is explicitly included. This header file is part of the generated Cycle Model and is required for linking the object into a test environment.

The Cycle Model is explicitly instantiated with the `carbon_twocounter_create` command. This provides context for the design and is used to run the design's core functions. This is followed by a series of functions that exercise the nets—values are deposited on nets and then examined on a schedule.

Step 3: Compile the software harness

To compile the software harness using GCC:

```
$ make twocounter.o
```

The file `twocounter.c` is compiled into a software harness object.

Step 4: Link the Cycle Model to the software harness

To link the Cycle Model to the software harness object and create a software validation executable using g++:

```
$ make twocounter.exe
```

The Cycle Model Compiler creates many output files. For information about these files, see *2.3 Compilation output files* on page 2-24.

Step 5: Run the executable

After generating the software validation executable, run it:

```
$ make twocounter.out
./twocounter.exe   >   twocounter.out
```

The results of the tests, which are directed by the software harness, are output to the `twocounter.out` file:

```
0: clk1=1 reset1=1 clk2=1 reset2=1 out1=0 out2=0
100: clk1=1 reset1=1 clk2=1 reset2=1 out1=0 out2=0
200: clk1=1 reset1=1 clk2=1 reset2=1 out1=0 out2=0
300: clk1=1 reset1=1 clk2=1 reset2=1 out1=0 out2=0
400: clk1=1 reset1=1 clk2=1 reset2=1 out1=0 out2=0
500: clk1=1 reset1=1 clk2=1 reset2=1 out1=0 out2=0
600: clk1=1 reset1=1 clk2=1 reset2=1 out1=0 out2=0
700: clk1=0 reset1=1 clk2=1 reset2=1 out1=0 out2=0
800: clk1=0 reset1=1 clk2=0 reset2=1 out1=0 out2=0
. . .
```

Step 6: Compare the output with the gold file

To compare this output with the contents of `twocounter.gold`:

```
$ make twocounter
```

If there are any differences between your output and the output that was expected, the differences are output.

Rerunning the example

Before rerunning the example, clean up the `twocounter` directory:

```
$ make clean
```

2.3 Compilation output files

The Cycle Model Compiler writes many files to the working directory. You may find these files useful for interpreting the Cycle Model.

./libtwocounter.a
: The Cycle Model file. The type of file created depends on the extension specified with the `-o` option. The default is `.a`.

./libtwocounter.h
: Cycle Model header file.

./libtwocounter.symtab.db
: Database containing information about all internal signals.

./libtwocounter.io.db
: Database containing information about IOs marked as observable (to the external environment). Used instead of the `*.symtab.db` file if you are passing your Cycle Model to a third-party customer, and you want to restrict the visibility of your design.

./libtwocounter.cmd
: All commands passed to Cycle Model Compiler, including those passed on the command line and those passed in a command file.

./libtwocounter.dir
: All directives that were parsed during the compile, including those in a directives file and any embedded (inline) directives.

./libtwocounter.hierarchy
: Table of every module instance in the design and its corresponding RTL file.

./libtwocounter.designHierarchy
: The instance hierarchy, with the architecture name, for all the design units (instances) used in the design. It also includes the library for each design instance, the location of the design instance, and the ports and parameters used.

./libtwocounter.warnings
: Warnings encountered during the compile.

./libtwocounter.errors
: Errors encountered during the compile. If the compilation succeeds, this file is created anyway, but is empty.

./libtwocounter.suppress
: Messages suppressed during the compile.

The following additional files are used exclusively by the Cycle Model Compiler, and may be ignored:

- `lib`*`design`*`.clocks`
- `lib`*`design`*`.cmodel.dir`
- `lib`*`design`*`.congruence`
- `lib`*`design`*`.costs`
- `lib`*`design`*`.cycles`
- `lib`*`design`*`.dclcycles`
- `lib`*`design`*`.drivers`
- `lib`*`design`*`.flattening`
- `lib`*`design`*`.init.v`
- `lib`*`design`*`.latches`
- `lib`*`design`*`.netvec`
- `lib`*`design`*`.parameters`
- `lib`*`design`*`.prof`

- `lib`*`design`*`.scheduleStatistics`
- `lib`*`design`*`.vfiles`

The directory `.carbon.lib`*`design`* contains files that are used internally. You can delete this directory when cleaning up your compiled files.

Chapter 3
Cycle Model Compiler command line options

This chapter describes the command-line options available to control Cycle Model compilation.

It contains the following sections:

3.1 Command syntax and guidelines

Use the `cbuild` command-line options, along with directives, to provide control and guidance to the Cycle Model Compiler.

Syntax

```
cbuild [options] [package file list] design file list
```

`options`

The command line options you specify. Optional.
Options are processed in the order that they are specified on the command line, whether explicitly or within an options file (see the `-f | -F` option in *3.3 Input file control options* on page 3-32). If an option is defined multiple times, the Cycle Model Compiler generates a warning at each successive encounter, and uses the value of the last option specified.
Options must include any special characters (for example, `-`, `_`, `+`) that appear in the option name. Spaces are not allowed within an option name.
Options are case sensitive.

`package file list`

List of files that contain SystemVerilog package definitions. Optional.
If you are specifying package files, they must precede the design file list on the command line.

`design file list`

List of design files you want to include in the compile. Required.
A Cycle Model may be compiled from several design and library files. When you invoke the Cycle Model Compiler, list all necessary design files and referenced library files.

File processing order

The Cycle Model Compiler processes files in the following order:

1. Command option files. These are specified with the `-f` option. Command option files are resolved by replacing `-f filename` with the contents of the `filename` file. This includes any nested `-f` specifications.
2. Directives files (specified with the `-directive` option).
3. Package, design, and library files are processed in the order that they appear on the command line.

3.2 General compile control options

This section describes the Cycle Model Compiler command line options that control aspects of compilation.

-annotateCode

Annotate the generated C++ code for C++ fault diagnosis. By default, both HDL and implementation annotations are disabled.

- HDL annotation associates a location in the generated C++ model with a file name and line number in the HDL design.
- Implementation annotation associates a location in the generated C++ model with a location in the Cycle Model Compiler implementation.

Consult with your Arm Cycle Models Applications Engineer before using this option.

-clockGlitchDetect

-noClockGlitchDetect

Creates a model that uses glitch detection when the model is run. Glitch detection enables the names of internal clocks to be reported from the model. The default behavior is to create a model that supports glitch detection (-ClockGlitchDetect).
When glitch detection is disabled at compile time using `-noClockGlitchDetect`, it is not possible to enable glitch detection at runtime in the resulting model.

-compileLibOnly

Specifies the library compile-only mode. The source design files are compiled into libraries (with no analysis done except for basic syntax checks), but no Cycle Model is generated.

——— **Note** ———

If any of your Verilog source files contains a `` `protected `` region and you attempt to compile with `-compileLibOnly`, the compilation fails and gives an error message. This limitation preserves the integrity of the `` `protected `` information.

———————

-h | -help

Prints the documentation for compiler options. The compiler help information displays, and then the compiler exits.

-j ***`integer`***

Limits the number of parallel make sub-jobs—the number of Cycle Model compilations running in parallel. The default value is 4. Set this value to 1 for serial runs.

——— **Note** ———

You may set this option to any positive integer value you want, however the Cycle Model Compiler performance may not be optimal if it is set too high. Also note that you may need to use a smaller number if the compile process produces errors that point to a possible memory issue.

———————

-libMap *string*

Builds a library mapping file that contains the mapping between the logical library names used in a design and their physical locations. Typically this is a directory which contains the precompiled versions of the design instances (entities, architectures, package declarations, package bodies and configurations) and modules that make up a library. `-libMap` allows the compiler to find the physical location of any logical library that has been precompiled.

Note

Do not confuse the Cycle Model Compiler `-libMap` option (note upper-case `M`) with `-libmap` switches, such as those provided by VCS and MTI. The Cycle Model Compiler `-libMap` switch contains logical-to-physical library mapping information, and it is different in functionality from the `-libmap` switch in simulators.

The library mapping file generated by `-libMap` contains entries in this form:

```
logical library name => physical library location
```

Usage notes:

- Each time the Cycle Model Compiler is run with this option, a single library is added to the library mapping file. The Cycle Model Compiler assumes that the last library mapping is the `work` library, and it expects the top-level design unit (instance) to be in the work library.
- The same logical library can not have more than one entry. For example, suppose a design has three libraries: A, B, and C. Library B depends on library A, and library C depends on library B. There are two ways to compile this design:
  — Without using the `-libMap` option, the commands are:

  ```
  cbuild -compileLibOnly -vlogLib A files belonging to A cbuild -
  compileLibOnly -vlogLib A -vlogLib B files belonging to B cbuild -vlogLib A
  -vlogLib B -vlogLib C files belonging to C -vlogTop top level design unit
  ```

  — Using the `-libMap` option, the above steps are rewritten as follows:

  ```
  cbuild -compileLibOnly -libMap design.libMap -vlogLib A files belonging to
  A
  cbuild -compileLibOnly -libMap design.libMap -vlogLib B files belonging to
  B
  cbuild -compileLibOnly -libMap design.libMap -vlogLib C files belonging to
  C
  cbuild -libMap design.libMap -vlogTop top level design
  unit
  ```
- The `design.libmap` file maps the logical library name to its physical library location, which allows `cbuild` to load the design instances it needs automatically. The libraries must be compiled in order.

You can edit the entries in the library mapping file directly, or you can add or modify entries using `-vlogLib` on the `cbuild` command line. The command line always has precedence over any entry in the library mapping file. For exam0ple, if a logical library that exists in the library mapping file is specified again on the command line with `-vlogLib`, the entry in the library mapping file is edited to match the command line.

-licq

Enables license queuing for the Cycle Model Compiler compilation process. When specified, the Cycle Model Compiler waits for a license to become available rather than exiting immediately if all licenses are in use.
You can also enable license queuing for the Cycle Model Compiler using the `CM_ENABLE_LICENSE_Q` environment variable. See the licensing appendix of the *Cycle Model Studio Installation Guide* (101106).

Note

License queuing is also available for runtime models. See the licensing appendix of the *Cycle Model Studio Installation Guide* (101106).

-loopUnrollLimit *integer*

Sets the maximum number of loop iterations allowed for static elaborated loops (`FOR`, `GENERATE FOR`, `WHILE`, `FOREVER`, `DO`). The default value is 5000.
Use this option in the event the compiler emits errors related to the loop count limit; for example:

```
foo.v:22: Error 51066: loop count limit of 5000 exceeded; condition is never false
```

-O *string*

Controls the design optimization level. Higher optimizations may result in a faster Cycle Model. The optimization levels are defined in the following table. Note that the `s` value is case sensitive, meaning that `S` is not equivalent to `s`.

Level	Passes to g++	Optimizations
0	-O0	None
1	-O1	Basic
2	-O2	Advanced (default setting)
3	-O3	Advanced, aggressive inlining
s	-Os	Generates a smaller code size for the Cycle Model, which can improve performance for certain designs.

-phaseStats

Prints the time and memory statistics for each compile phase.

-profileGenerate

Use with the `-profileUse` option to generate more efficient (faster) Cycle Models as follows:

1. Run the Cycle Model Compiler using `-profileGenerate`.
2. Link the simulation using GCC `-fprofile-generate` option.
3. Run the Cycle Model in your simulation environment.
4. Rerun the Cycle Model Compiler using the `-profileUse` option.
5. Link the simulation using the GCC `-profileUse` option.

Usage notes

- If you are not using the Cycle Model Compiler version of `g++` to link, then a link error that mentions `__gcov_*` may occur. If this link error occurs, add the following string to the end of the link command (instead of `-fprofile-generate`):

```
$CARBON_HOME/Linux64/gcc73/lib/gcc/x86_64-pc-linux-gnu/7.3.0/libgcov.a
```

- `-profileGenerate` and `-profileUse` are not related to the `-profile` option. The `-profileGenerate` and `-profileUse` options result in a more efficient Cycle Model. The `-profile` option generates a report that shows where time is spent in the runtime model.

-profileUse

Re-executes the last few phases of the Cycle Model Compilation process using feedback from profile-directed optimization. As shown in the `-profileGenerate` flow above, use this option after compiling with `-profileGenerate`, linking with the GCC `-fprofile-generate` option, and simulating the Cycle Model in your validation environment.
`-profileUse` reuses much of the previous compilation, so HDL changes and many of the Cycle Model Compiler options are ignored.

-SVInspector *number_of_lines*

Parses your design and and outputs a report of all supported and unsupported SystemVerilog language constructs. See *3.8 Options for troubleshooting design errors* on page 3-48 for information.

-topLevelParam ***`parameter/generic=value`***

Specifies new values for Verilog parameters of top-level modules. By default, the Cycle Model Compiler compiles parameters using their default values.

You can specify this option multiple times to account for all the parameters in the top-level design instance. Any parameters not specified using this option retain their default values. The case of the parameter name must match.

———— **Note** ————

If special characters, such as single or double quotes, are to be used as part of the value for the parameter, you must use the Escape sequence (backslash) before the special character; for example, `-topLevelParam ABC=20\'h1234`. This allows the compiler to properly convert and store the value. This method works both on the command line and within a `-f` file.

————————————

Errors are generated under the following conditions:

- A parameter specified using this option does not exist in the top-level module.
- The top-level module does not have any parameters.

3.3 Input file control options

-f | -F *string*

Specifies a command file name from which the Cycle Model Compiler reads command-line options. The compiler treats these options as if they have been entered on the command line.
Usage notes:

- Options specified in files are cumulative.
- `string` may be the full path or relative file name after the `-f`. There is no restriction on file naming.
- The syntax of the file is a white-space separated list of options and arguments. It is not necessary to specify each option on a new line. The file may contain comments that use any of the following delimiters: `#` (shell), `//`, or `/* ... */` (C++ style).

The file may also include entries that include environment variable expansion, according to the following rules:

Environment variable	Supported/Unsupported
`$varname`	Supported. The variable name must be followed by a space or a recognized delimiter.
`${varname}`	Supported. Curly braces can be used in cases where there is no recognized delimiter.
`$(varname)`	Not supported. Parenthesis are not recognized as valid delimiters for environment variables. This follows the same rules as the tcsh, csh, bash, and sh shells, as well as the NC and MTI simulators.

The following examples show valid uses of variables in a command file:

```
$TEST_DIR/test1.v     // valid because / is a recognized delimiter
${TEST_DIR}test1.v    // curly braces are required because no / is used
${TEST_DIR}/test1.v   // curly braces are not required in this case, but they are
ignored when not needed
```

——— Note ———

In these examples, the environment variable `TEST_DIR` must be defined before the Cycle Model Compiler uses the environment variable, or an error occurs.

-directive *string*

Includes a directives file on the command line. Multiple instances of the `-directive` option are allowed. Directives specified in files are cumulative.
The syntax of a directives file is line oriented — each directive and its values must be specified on its own line. Enter the full path or relative file name after the `-directive`. There is no restriction on file naming. However, it is standard to use a `.dir` or `.dct` suffix for directives files.
See *Chapter 4 Using Cycle Model Compiler directives* on page 4-49 for more information about directives.

-showParseMessages

Directs the Cycle Model Compiler to write messages to `stdout` as it analyzes HDL source files. The Cycle Model Compiler prints the name of the source file being analyzed, and the modules found within each source file. This can be useful during initial model compilation when you want to ensure that the Cycle Model Compiler uses the correct source files.
Example output:

```
Note 20001: Analyzing source file "test.v" ...
test.v:1: Note 20004: Analyzing module (top).
test.v:5: Note 20004: Analyzing module (child).
Note 20025: 0 error(s)   0 warning(s).
```

3.4 Module control options

This section describes Cycle Model Compiler options related to modules.

-checkpoint

-noCheckpoint

Causes the Cycle Model Compiler to generate checkpoint save and restore support in the compiled Cycle Model. By default, `-checkpoint` is enabled. Save and restore support means that API functions can be used to save and restore a given state of the Cycle Model during validation runtime.

-noFlatten

Disables flattening. By default, the Cycle Model Compiler flattens design modules into their instantiating parent module based on the value of `-flattenThreshold`. Disabling flattening can improve design visibility, but may produce a slower Cycle Model.

Note

See *6.2 Flattening-related module control directives* on page 6-64 for directives that conditionally or unconditionally flatten modules. These directives are effective only if the -`noFlatten` option has not been specified.

-flattenThreshold *`integer`*

Specifiies the largest child module that is considered for flattening into its parent module. The size of a module is computed based on the number of assignment statements (blocking and non-blocking) contained within that module. The default value is `25`.
In general, threshold values between `10` and `50` yield the best results. However, the best threshold is design dependent. Compile, then measure performance with different values and select the optimal value.

-flattenParentThreshold *`integer`*

Specifies the maximum parent module size during flattening.The default value is 10000.
Modules that are too large can decrease the performance of your model. After a module reaches the size specified in this option, no more children will be flattened into the module unless the child modules are tiny (see `-flattenTinyThreshold`).

-flattenTinyThreshold *`integer`*

Flattens modules of this size or smaller, even if their parent modules have reached the limit defined in`-flattenParentThreshold`. The default value is 10.
Ensure this value is less than the value specified in `-flattenThreshold`.

-inlineTasks

Replaces all task and function calls with the contents of the called task or function. This option may enable further optimization, resulting in a faster Cycle Model. However, be aware that inlining large tasks and functions that have multiple calls may increase the size of the Cycle Model and therefore produce a slower model.
To inline selected tasks or functions, see the`-inlineSingleTaskCalls` option (below) and the `inline` directive (*6.1 General module control directives* on page 6-61).
Tasks with hierarchical referrers are not considered for inlining; for example, if a module a calls task `x` through the hierarchical path `a.b.x`, then no call of `x` is inlined.

Note

Using the `-inlineTasks` option on designs that require a great deal of memory to compile may cause a memory allocation error. For these cases, Arm recommends using the `inline` directive.

-inlineSingleTaskCalls

Inlines tasks and functions that have only a single call. In other words, if a task or function is only called once, that call is replaced with the contents of the task or function. This option may enable further optimization, resulting in a faster Cycle Model.
Tasks and functions with hierarchical referrers are not considered for inlining; for example, if a module a calls task `x` through the hierarchical path `a.b.x`, then no call of `x` is inlined.

-tristate {1 | 0 | x | z}

Sets how tristate data appears in waveforms. By default, the Cycle Model Compliler is set to `x`, which provides the best performance. Setting to `z` provides maximum visibility into the Cycle Model.
Values are case-insensitive.
The following table defines the tristate propagation and waveform display for each tristate mode. Note that `x` and `z` propagation always results in `don’t care`.

Setting	**Tristate Propagation / Waveform Display**
-tristate 0	don’t care / 0
-tristate 1	don’t care / 1
-tristate z	don’t care / Z
-tristate x	don’t care / don’t care (default)

3.5 Net control options

-bufferedMemoryThreshold ***integer***

Sets the threshold size to allow larger memories to be buffered. The default is 16,384 bits.
The following code example causes the Cycle Model Compiler to encounter a scheduling conflict, because it wants to schedule the flop that writes to data before scheduling the flop that writes to `out1`. In this case, data could race through:

```
module top (out1, out2, clk, rst, en, in);
  output [3:0] out1, out2;
  input clk, rst;
  input [3:0] en, in;
  reg [3:0] data [1:0];

// Put all the inputs into a memory
  always @ (posedge clk or posedge rst)
    begin
      if (rst)
        begin
          data[0] <= 4'b0;
          data[1] <= 4'b0;
        end
      else
        begin
          data[0] <= en;
          data[1] <= in;
        end
    end

// Create a clock from the enable
  wire [3:0] ren = data[0];
  wire dclk = clk & ren[0];
  reg [3:0] out2;
  always @ (posedge dclk)
    out2 <= data[1];
// Use those clocks and the data
  reg [3:0] out1;
  always @ (posedge clk)
    begin
      out1 <= data[1];
    end
endmodule
```

The Cycle Model Compiler resolves this scheduling conflict by introducing a delay in the data before it gets to `out1`. However, for performance reasons it only does so if the memory is 16,384 bits or less. If the memory is larger than 16,384 bits, then the Cycle Model Compiler issues an alert. For example:

```
bufmem.v:57 top.out1: Alert 1054: A memory `top.data` that is written as part of
clock logic is read in a flop; the conflict could not be resolved. See the
documentation for -bufferedMemoryThreshold for information on this problem.
```

There are three options for dealing with this alert:

- Demote the alert to a warning if the race condition does not matter.
- Increase the buffer memory threshold using the `-bufferedMemoryThreshold` option.
- Remodel the code by writing to the data memory in different `always` blocks for the clock enable and data portions.

-g

Increases visibility of design nets for debugging purposes.
Not all nets in the design are preserved. Dead nets (nets that do not reach primary outputs) are not affected by this option.
See the `observeSignal` directive (*5.2 Signal-related net control directives* on page 5-56) for more information about making signals observable.

-memoryCapacity *`integer`*

Specifies the total amount of runtime memory (in bytes) allocated to memories. The default value is `4,194,304` bytes (4 MB).

Memories that cause the model to require more than this amount of space are coded as sparse memories. By default, memories use a fast array-based representation. Sparse memories use a memory-efficient, hash table-based implementation. A value of:

- `0` - Generates only sparse memories
- `-1` - Does not generate any sparse memories

-checkOOB

-no-OOB

`checkOOB` generates warning messages during validation runtime if any out-of-bounds bit or memory references are made. Use `checkOOB` to ensure that your design has no out-of-bounds references, and then recompile using `-no-OOB` to generate a Cycle Model with a faster execution time.

If `checkOOB` detects any out-of-bounds references, fix the out-of-bounds references in your design as necessary and then recompile with `-no-OOB`. The warning message generated by `checkOOB` does not pinpoint the name of the vector or memory containing the out-of-bounds reference, but gives the following information to help your search:

- Whether the out-of-bounds access occurs on a read or write.
- Whether the accessed object is a memory, register, wire, or net.
- The declared range of the object.
- The invalid index value.

`-no-OOB` disables checking for out-of-bounds bit references. This means the Cycle Model Compiler does not check for such references. As a result, the Cycle Model runtime is faster.

Note

If you specify this option, the Cycle Model exhibits unpredictable behavior during runtime if there are out-of-bounds bit references in your design.

Consider the following:

```
sider the following:
wire [7:0] value;
wire [7:0] index;
. . .
. . .
result = value [index];
```

If index can take on a value larger than 7, this may produce incorrect answers when `-no-OOB` is specified. The index values greater than 7 are out of bounds.

-noCoercePorts

Disables port analysis. Disabling the port analysis functionality disables the Cycle Model Compiler's ability to process complex bidirectional ports.

- Port analysis - By default, the Cycle Model Compiler performs a design-wide port analysis and may alter port directions based on the characteristics of the design. In particular, these modifications may occur when the declared port directions do not model the flow of data within the design. For example:

```
...
wire data;
sub s0(en1,data,out1);
sub s1(en2,data,out2);
...
module sub(en,data,ndata);
    input en;
    output data;
    output ndata;
    assign data = en ? 1'b1 : 1'bz;
    assign ndata = ~data;
endmodule
```

  In this design, the `sub.data` wire has multiple drivers. Converting `sub.data` from `output` to `inout` properly models the fact that a value written in the `s0` instance can be read in the `s1` instance, and vice-versa.

- Primary ports - A unidirectional primary port is considered a stronger statement than a bidirectional primary port. This means that a user-declared primary input must behave as an input. A user-declared primary output must behave as an output. Therefore, inputs/outputs may be coerced to inouts, but not to output/input.

  Primary inouts are handled differently — their bidirectional nature is considered a weaker statement, therefore coercion from inout to either directional port type is allowed.

- Per-bit Port Behavior - Port analysis may determine that all bits in a primary port do not behave in a uniform fashion. If this occurs, different bits may be identified as having different port direction. For example:

```
module top(a,out1,out2);
    input [1:0] a;
    output out1,out2;
    pullsub p0(a[0],out1);
    assign out2 = a[1];
endmodule
module pullsub(in,out);
    input in;
    output out;
    pullup(in);
    assign out = in;
endmodule
```

  In this design, the `top.a` net will be split into two components. The `a[0]` bit will be converted to a bidirectional port to reflect the fact that it is driven by the model. The `a[1]` component of `top.a` remains an input.

-sanitizeCheck

Use this option to check for dirty writes. After every write to a net, it checks for non-zero bits outside the declared size of the net. This helps to detect out-of-bound reads and writes permitted by the `-no-OOB` flag in order to help diagnose problems.

-waveformDumpSizeLimit *integer*

Specifies the maximum size (in bits) of design elements to dump to waveform files. The default value is 1024 bits.
Set this value to 0 in order to dump all waveforms; regardless of size. This option is an alternative to the `carbonDumpSizeLimit()` API function, although the API function has precedence if both are specified.
Multiple instances of this option are allowed.

This section contains the following subsection:

- *3.5.1 Port vectorization options* on page 3-39.

3.5.1 Port vectorization options

To improve runtime performance, the Cycle Model Compiler replaces selected scalar ports and scalar local variables with a vector port or local nets.

-doNetVec
Enable port vectorization. Port vectorization is enabled by default.

-noNetVec
Disable port vectorization. Cases in which you might disable port vectorization include:
- A modeling error has resulted from port vectorization
- Port vectorization has broken visibility for some signals

In either of these cases, initiate a bug report with Arm.

-netVecMinCluster ***`integer`***
Specifies the minimum size of a vector created during net vectorization.

-verboseNetVec
Output information on the vectorized nets.

-reportNetVec ***`filename`***
Write a report to the specified *`filename`*, enumerating the vectorizations discovered by port vectorization.

Options for Vectorizing Primary Ports

The three options in this section are mutually exclusive. The default is `-netVecThroughPrimary`.

Note

`-netVec` line options have been replaced by `-netVec`. If used, a warning message appears, notifying you to switch to the `-netVec` options.

-netVecPrimary
Vectorizes the primary ports of the design.

Note

This option breaks visibility of any primary ports that are vectorized. When scalar primary ports are combined into vectorized ports, the original port names are replaced with a new vectorized port name. You can not make Cycle Model API calls (such as `observeSignal` or `depositSignal`) using the old or new port names.

-netVecThroughPrimary
Does not vectorize the primary ports. However,`-netVecThroughPrimary` allows vectorization opportunities inferred between the primary ports to propagate down the module hierarchy. This is the default setting.

-nonetVecThroughPrimary
Does not infer any vectorization opportunities from the primary ports, and does not allow propagation of opportunities through the primary ports.

3.6 Verilog- and SystemVerilog-specific control options

The options in this section are for use only with Verilog and SystemVerilog design files.

Options that specify compilation mode

Note

To view a report of the supported and unsupported SystemVerilog language constructs used in your design, use the Cycle Model Compiler `-SVInspector` option.

-95

Enables Verilog-95 compilation mode. See *1.8 Specifying the Verilog language variant to use when compiling* on page 1-17 for more information.

-2001

Enables Verilog-2001 compilation mode. This includes partial support for Verilog-2005 (IEEE Std 1364-2005) language features. All files encountered during the compilation are treated as Verilog 2001. Note that you may also use `-2000` or `-v2k` to enable this compilation mode. These three options are equivalent.
See *1.8 Specifying the Verilog language variant to use when compiling* on page 1-17 for more information.

-sverilog

Enables SystemVerilog compilation mode. All Verilog files encountered during compilation are treated as SystemVerilog source files. All Verilog command line options work with SystemVerilog, provided `-sverilog` is specified. See *1.8 Specifying the Verilog language variant to use when compiling* on page 1-17 for more information.

General Verilog- and SystemVerilog- control options

-enableOutputSysTasks

By default, the Cycle Model Compiler issues a warning and ignores system tasks such as:

- `$display`
- `$fclose`
- `$fdisplay`
- `$fflush`
- `$fopen`
- `$fwrite`
- `$sformat`
- `$write`

The `-enableOutputSysTasks` option enables support for these system tasks throughout your design.

Note

This option may impact the performance of the resulting Cycle Model.

For information about supported, unsupported, and partially-supported system tasks, see the *Cycle Model Compiler Verilog and SystemVerilog Language Support Guide* (100972).

You can also enable or disable output system tasks at the module level. See the directives described in *Chapter 6 Module Control directives* on page 6-60.

A system task that appears within an edge-sensitive `always` block is scheduled with that clock. For example, the system task in the following example is scheduled with `clk`:

```
always @(posedge clk)
    $display (a1, a2);
```

The values displayed from calls to `$time` functions may not match the times generated by the design. This can result in the following example output. In the Cycle Model Compiler output, the time does not appear to change, but it displays identical values to the Verilog output for the `out` variable:

```
source verilog
...
always @(posedge clk) begin
begin
    a = 1;
    b = 0;
    out = a
    $display("a time: %t a=%b b=%b out=%b", $time, a, b, out);
    a = 0;
    out = #10 b;
    $display("b time: %t a=%b b=%b out=%b", $time, a, b, out);
    #10 $display("c time: %t a=%b b=%b out=%b", $time, a, b, out);
...
verilog
a time:    0      a=1 b=0 out=1
b time:    10     a=0 b=0 out=0
c time:    20     a=0 b=0 out=0
cycle model compiler output
a time:    0      a=1 b=0 out=1
b time:    0      a=0 b=0 out=0
c time:    0      a=0 b=0 out=0
```

-pragma_prefix *`string`*

When embedding directives in comments, use this option to specify the prefix used for:

- Synthesis-specific compiler directives, such as `translate_off`, `translate_on`, `full_case`, and `parallel_case`.
- Cyle Model-specific compiler directives, such as `observeSignal` and `depositSignal`. For a complete list, see *4.2 Embedding directives in comments* on page 4-52.

`-pragma_prefix` takes a single string argument. To specify multiple prefixes, specify `-pragma_prefix` multiple times. Subsequent prefix specifications do not cancel earlier ones.

For example, to make the following signal observable:

```
reg a; // myPrefix observeSignal
```

you must include the following on the command line:

```
-pragma_prefix myPrefix
```

If you do not use this option to specify a prefix, the synthesis and directives are ignored for all prefixes except `ARM_CM`. The Cycle Model Compiler automatically recognizes the prefix `ARM_CM`.

-synth_prefix *`string`*

Same as the `-pragma_prefix` option.

-u

Converts all identifiers in all referenced Verilog files to upper case. This option makes the design insensitive to identifiers' case.

Note

All references to Verilog identifiers in options and identifiers within directives must be changed to upper case. For example, when specifying `-u`, change `-vlogTop top` to `-vlogTop TOP`.

-topModuleListDumpFile *`string`*

Specifies the name of the file into which the Cycle Model Compiler will place the names of the top-level modules of the Verilog design.

-vlogTop *`string`*

Specifies the top-most module in the Verilog design hierarchy. The Cycle Model Compiler parses only the specified module and its descendents. If you do not specify this option, all modules from the top down in the given design are compiled.

Note

Currently, only one top-level module is supported.

-v *`string`*

Specifies a Verilog source library file. The Cycle Model Compiler scans the file for module definitions that have not been resolved in the specified design files. Enter the full path or relative file name after the `-v`. For example:

```
cbuild -v ../library/vendor.lib 2clock.v
```

Without this option, the Cycle Model Compiler processes only those library modules that are explicitly referenced by the Verilog source files.

-y *string*

Specifies a library directory. The Cycle Model Compiler scans the directory for module definitions that have not been resolved in the specified design and library files. Enter the full or relative directory path after the option. For example:

```
cbuild -y library/cells 2clock.v
```

This command references the library directory `/library/cells` for input design files.
Usage notes:

- The file names within the specified library directory must match the module names that are being searched for.
- If two subdirectories contain a file with the same name, and the top level design file in the current directory uses a single instance of that file, the Cycle Model Compiler uses the definition of the file from the directory that appears first on the command line. For example, in the following case, the definition of the file from the `foo` directory is used:

  ```
  cbuild -q -y foo -y bar +libext+.v test.v
  ```

+define+*string*

Use the `+define+` option to specify Verilog macros to be used during compilation. Enter the variables with values, linked with plus signs (+). Syntax:

```
+define+var1+var2+ ... +varN=value
```

Usage notes:

- The equals sign (=) terminates the string. That is, anything after the equals sign is treated as part of the value of the variable with which it is associated. In the following example, whenever `` `WORD_LENGTH`` appears in the Verilog text, it is replaced with `8`:

  ```
  cbuild 2clock.v +define+WORD_LENGTH=8
  ```
- The `value` parameter is optional. The following example makes `` `ifdef MYDEF`` statements true in the code:

  ```
  cbuild 2clock.v +define+MYDEF
  ```
- You can specify multiple `+define+` options on the command line. If you define the same variable more than once, an Alert is issued. If you demote the Alert, then later `+define+` options take precedence over earlier ones.

+incdir+*path1*+...

Specifies the directories in which the Cycle Model Compiler should search for include files. Enter the list of relative or absolute paths, linked with plus signs (+). The paths are searched in the order specified.
`-incdir` can be used by NC-Verilog simulation users to specify a single directory.
You can enter multiple `+incdir+` options on the command line. If there is a conflict between values in include files, the last value is used.

+libext+*ext1*+...

Specifies extensions on the files you want to reference in a library directory. The default extension is `.v`. Specifying this option on the command line replaces the default.
Usage notes:

- To specify multiple extensions, enter the list of extensions after the option linked with plus signs (+). The following example references the library directory `library/cells`, but uses only files in that directory with the extension `.v`:

  ```
  cbuild -y library/cells +libext+.v 2clock.v
  ```
- A single `+libext+` option can appear on the command line. However, the single option can specify multiple extensions. For example, `+libext++.v+.vlog+` indicates that there are three possible extensions: a null string, `.v`, and `.vlog`.

+mindelay

+typdelay

+maxdelay

Specify the use of the minimum, typical, or maximum value respectively for all expressions.

——— **Note** ———

These options have no effect. They are provided only for compatibility.

———————

3.7 Output control options

Do not edit any files that are generated by the Cycle Model Compiler. Doing so results in unexpected behavior or failure of subsequent processes.

-dumpTokenFile

Creates a single file in the libdesign.tokens directory that contains the full design. This switch is not functional for any part of the design that contains a Verilog-style protected region.

-o ***`string`***

Specifies the name of the compiled Cycle Model. The default is `./lib`*`design`*`.a`.
The Cycle Model Compiler creates additional files whose base names match those used for the Cycle Model. For a list of output files created by Cycle Model Compiler, see *2.3 Compilation output files* on page 2-24.
Usage notes:

- Enter the full path or relative file name after `-o`. You may use alphanumeric characters, period (`.`), hyphen (`-`), underscore (_), and plus sign (`+`) characters in your specification, either on the command line or within a Makefile.
- You must use the `lib` prefix for the file name.
- Do not use white spaces in the string, and do not use existing system library names (for example, `libc` or `libm`).
- Use `-o` to direct model output for multiple platforms from the same directory substructure. For example, if you specify `-o Linux/obj/libtest.a` on the command line, the Cycle Model Compiler writes the output files to the `Linux/obj` directory.
- Copy the following files any new working directories you create:

  1) The Cycle Model (`*.a`) file

  2) The header (`*.h`) file

-noFullDB | -nodb

Use with `-embedIODB` to disable generation of the `.symtab.db`. Only the `.io.db` is created and embedded into the library.

-embedIODB

By default, the Cycle Model Compiler creates and embeds the `.symtab.db` file into the Cycle Model, eliminating the need to search for it during runtime. This option creates and embeds the `.io.db` file into the generated `libdesign.*library` file.
This option is enabled by default.

Note

The component for SystemC and Cycle Model Validation currently requires the full database.

-profile

Enables block profiling of the Cycle Model. This is sample-based profiling that records information about the HDL block or Cycle Model API function that is currently executing. Sampling occurs approximately 100 times per second. The data is an estimate, but provides a general idea of which blocks take the most time during your validation run.

Note

Profiling of Cycle Model API functions requires a profile-enabled version of the library. Select this library by using `$(CARBON_PROFILE_LIB_LIST)` instead of `$(CARBON_LIB_LIST)` in the `Makefile` used to link the executable.

To use profiling:

1. Add the `-profile` option to the command line during compilation.

   The Cycle Model Compiler creates a file named `lib`*`design`*`.prof`, which contains compile-time information needed for profiling. During simulation, the number of samples that occur in each block is counted, then written to `carbon_profile.dat` when the simulation ends.
2. Following simulation, run `carbon profile` at the command prompt.

   The `carbon profile` command reads `carbon_profile.dat`, files matching `*.prof`, and files matching `*.hierarchy` in the current directory. To override which files are read, list specific files on the command line when you issue the `carbon profile` command.

   The profiling results are sent to standard output.

   Subsequent compilation runs overwrite the `lib`*`design`*`.prof` file and `lib`*`design`*`.hierarchy` file, and subsequent simulations overwrite the `carbon_profile.dat` file.

See *Appendix C Using Profiling to find performance problems* on page Appx-C-76 for information about profiling data.

+protect[.ext]

Generates protected source versions of all given Verilog input files. Modified Verilog output files are generated for each input; the output files contain all source between `` `protect`` and `` `endprotect`` compiler directives in encrypted form. Unless a different extension is specified on the command line, the output file is named with a `.vp` extension. (Verilog only.)

Usage notes:

- Only Verilog 2001-style `` `protect`` and `` `endprotect`` are supported. The use of the `-v2001` switch does not cause this support to advance to Verilog 2005 style.
- Conditional code blocks are bounded by `` `ifdef`` (or `` `ifndef``) and `` `endif``. All conditional code blocks must be closed or completed before a `` `protect`` region starts or ends. This means that you can place conditional code blocks before or after a protected block, or even within a `` `protected block``. However, no conditional code block may be open at the point that the `` `protect`` or `` `endprotect`` line is encountered if that file is to be processed by the `+protect` command line option.
- The `+protect` command line option only protects code in the files that are listed on the command line. Files that are included (with the `` `include`` *`filename`* line) in the source Verilog files remain unprotected.

-verboseFlattening *integer*

Outputs a trace of flattening operations to `stdout`. The default value is `0` (no output).

Example 1: Output when flattening operation is successful and `-verboseFlattening` is set to a non-zero value:

```
Flatten: {case.v:7} instance CASE2 of module mycase into module case_top (SUCCESS)
--> Successfully flattened. [parent=2 child=2 comb=4]
```

Example 2: Output when flattening operation fails and `-verboseFlattening` is set to a non-zero value:

```
Flatten: {big.v:5} instance big of module big into module big_top (FAILURE)
--> Both the parent and sub modules were too large. [parent=7 child=7 comb=14]
```

`child=`*`num`* is the Cycle Model Compiler's perceived size for that instantiated submodule. To allow flattending for the submodule, increase the `-flattenThreshold` option to a value larger than `num` (see `-flattenThreshold` in *3.4 Module control options* on page 3-34).

Example 3: Summary of final flattening operation when `-verboseFlattening` is set to a non-zero value:

```
flattening Summary: 2 instances flattened into 1 parent modules.
```

-verboseLatches

Outputs a list of latches found in the design to `stdout`. For example:

```
foo.v: 9 Net is a latch.
```

-q

Enables quiet mode and suppresses all Cycle Model Compiler banner output.

By default, the Cycle Model Compiler outputs the progress of the build including elapsed time and estimated percent-done. The Compiler also prints out a phase number, which is useful to your Arm Applications Engineer if you encounter any problems.

This option has no effect on other information, such as warnings and errors, that are written to `stdout`.

-w

Suppresses all Warnings generated by the Cycle Model Compiler. See *Chapter 7 Output Control directives* on page 7-66 for additional information about message severity levels.

-stats

Prints time and memory statistics for the compilation to `stdout`.

-version

Displays the product version of the Cycle Model Compiler. The version information displays, and then the Cycle Model Compiler exits.

-verboseVersion

Prints various internal version strings. The version information displays, and then the Cycle Model Compiler exits. Your Arm Applications Engineer may ask you to run this command to confirm product version information.

3.8 Options for troubleshooting design errors

The Cycle Model Compiler provides options and resources that help you troubleshoot errors in your design.

-designReducer

Enables a tool that is built in to the Cycle Model Compiler. This tool, called the Design Reducer, helps create a small test case that exhibits a problem that you want to tell Arm support about. The Design Reducer automatically removes parts of an HDL design, while ensuring that the simplified design continues to cause the compiler error or warning that the original design caused. The reduced design is easier to troubleshoot because the non-essential portions have been removed.

For example, the following command runs the Design Reducer on a SystemVerilog design called `myDesign.sv` with a check script called `checkDesign.sh`:

```
build -sverilog -q -designReducer checkDesign.sh -vlogTop top myDesign.sv
```

When you specify this option, no Cycle Model is created.

See *Appendix D Using Design Reducer to troubleshoot design errors* on page Appx-D-83 for more information about using the Design Reducer.

-SVInspector *`number_of_lines`*

Parses your design and outputs a report of all supported and unsupported SystemVerilog language constructs. Specify an integer argument that defines the maximum number of HDL source reference lines for each language construct reported. No Cycle Model is created when you specify this option.

For example:

```
cbuild -SVInspector 100 test/sv/Interfaces/virtual_interface.sv
```

Here, the maximum number of lines is specified as 100. If the integer argument is negative, then the number of lines printed is unlimited. If the argument is 0 or positive then only that number of lines is printed for each language unit.

In the generated report, `LU` stands for Language Unit; unsupported constructs are noted as `-unsupported`. The following sample output includes both supported and unsupported language constructs:

```
---  SystemVerilog Language Unit Report For: test  ---
found 2 LU for VeriAnsiPortDecl:noDimensions:tfParam:inout-
unsupported:inTaskOrFunction:static:isVirtualInterface
   test/sv/Interfaces/virtual_interface.sv:20 ..
found 2 LU for VeriBinaryOperator==
   test/sv/Interfaces/virtual_interface.sv:21 ..
found 6 LU for VeriDataDecl:fromInterface
   test/sv/Interfaces/virtual_interface.sv:5 .
   test/sv/Interfaces/virtual_interface.sv:6 .
   test/sv/Interfaces/virtual_interface.sv:7 .
   test/sv/Interfaces/virtual_interface.sv:5 .
   test/sv/Interfaces/virtual_interface.sv:6 .
   test/sv/Interfaces/virtual_interface.sv:7 .
found 1 LU for VeriSystemFunctionCall:urandom_range-unsupported:withConstArg
   test/sv/Interfaces/virtual_interface.sv:31 .
found 1 LU for VeriVariable:has1unpackedDimensions:isVirtualInterface
   test/sv/Interfaces/virtual_interface.sv:18 .

---  SystemVerilog Language Unit Report End For: test  ---
#  4  Elapsed: 0:00:01 Complete:  5%
#  4  Elapsed: 0:00:01 Complete:  5%
0 warnings, 0 errors, and 0 alerts detected.
---
```

Error message help file

Your Cycle Model Studio installation includes a text-based file that includes additional information about the cbuild diagnostic messages emitted when build errors occur.

The help file is `Arm_CM.help`. It is located in the directory `$CARBON_HOME/userdoc/help`.

Chapter 4
Using Cycle Model Compiler directives

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

It contains the following sections:

4.1 Passing directives to the Cycle Model Compiler

You can provide directives to the Cycle Model Compiler by passing in a directives file with the `-directive` command-line option, or by embedding a set of directives in the Verilog or SystemVerilog source as comments.

Limitations

The following limitation applies when using directives. Additional limitations may be noted in the description of specific directives:

- Directives on packed unions, or on members of packed unions, are not supported.

Directives file syntax and usage

Pass a directives file to the Cycle Model Compiler using the `-directive` command-line option:

- The syntax of a directives file is line oriented. Each directive and its values must be specified on a single line separated only by spaces.
- You must use the back slash (`\`) to indicate line continuation when necessary.
- Any extra white space between the directive name and values is ignored.
- Multiple instances of the `-directive` option are allowed.
- Directives specified in a file are cumulative, meaning there is no precedence. All directives are used.
- Use the pound sign (`#`) to specify comments in your directives file.

Using wildcards and single character match

You may use wildcards (`*`) and single character match (`?`) within a directives file. However, they apply only to one level of hierarchy in the design. To traverse multiple levels, use the following format:

```
observeSignal top.* top.*.* top.*.*.*
```

This example applies the `observeSignal` directive to three levels of the design hierarchy.

In the following example, the `observeSignal` directive applies to all nets in the design:

```
observeSignal *.*
```

Specifying single or multiple signals

For directives that take a list of signals, you may specify a single signal by hierarchy. Or, you can specify signals by module, so that every instance of the module is affected. The following example applies the `observeSignal` directive to all instances of `sig3` under module `sub2`:

```
observeSignal sub2.sig3
```

Note

You can specify only one level of hierarchy when using this syntax. You cannot enter `observeSignal u1.sub2.sig4`.

Using Directives on Signals Inside Generate Blocks

Adding directives on signals inside `generate` blocks, and blocks under `generate` blocks, is done in a slightly different fashion. The following example shows the hierarchical names of the two registers declared inside a `generate for` loop:

```
module top(in, out, clk);

    parameter genblk2 = 0;
    input     in;
    output    out, clk;

    genvar i;

    generate
    for (i=0; i<1; i = i+1)
        begin: named_block
            if (1)
                begin
                    reg c; // top.named_block[0].genblk1.c
                end
            if (1)
                begin
                    reg d; // top.named_block[0].genblk02.d
                end
        end

    endgenerate

endmodule
```

The second unnamed `generate` block is called `genblk02` is because the parameter `genblk2` is already using that name in the module scope. This follows the IEEE Verilog standard for uniquely naming `generate` blocks (IEEE Standard 1364-2001 - Verilog Hardware Description Language).

To make these signals observable, add the following information in a directives file:

```
observeSignal top.named_block[0].genblk1.c
observeSignal top.named_block[0].genblk02.d
```

4.2 Embedding directives in comments

As an alternative to a directives file, you can embed a limited set of directives in the Verilog or SystemVerilog source as comments, with a syntax similar to synthesis pragmas.

An embedded directive is associated with a net or a module.

The prefix `ARM_CM`, shown in the examples below, is automatically recognized by the Cycle Model Compiler. To use a prefix other than `ARM_CM` to identify embedded directives, specify the `-synth_prefix` or `-pragma_prefix` option on the command line.

Following is a sample section of a Verilog file, which uses comments to embed directives:

```
...
module top(in1, in2, clk1, clk2, out, ena);
    input in1, in2;
    input ena;   // ARM_CM tieNet 1'b1
    output out;
    input clk1;
    input clk2;   // ARM_CM collapseClock top.clk1
    reg a;   // ARM_CM observeSignal
            // ARM_CM depositSignal
    always @(posedge clk1)
        if (ena)
            a <= ~in1;
    wire out;   // ARM_CM depositSignal
    flop u2(out, in2, clk2, ena);
endmodule
...
```

Supported embedded net directives

The net directives supported in this fashion are listed below. Net directives apply to the specified net in every instance of the module:

- `asyncReset`
- `collapseClock`
- `depositSignal`
- `exposeSignal`
- `fastReset`
- `forceSignal`
- `ignoreSynthCheck`
- `observeSignal`
- `slowClock`
- `scObserveSignal`
- `scDepositSignal`
- `tieNet`

Net directives apply to the last wire or register declared prior to the embedded directive. If multiple wires or registers are declared on the same line, the directive is applied only to the last wire or register declared. For the following, `depositSignal` is applied only to `cff`:

```
reg aff, bff, cff;  // ARM_CM depositSignal
```

For the following, `observeSignal` is applied to `absig`:\

```
wire absig = en ? a_in : b_in;  // ARM_CM observeSignal
```

To place multiple directives on a wire or register, place the directives on separate lines prior to the declaration of other wires or registers. The following shows how to put both `observeSignal` and `depositSignal` directives on `reg a`:

```
reg a;  // ARM_CM observeSignal
        // ARM_CM depositSignal
```

Supported embedded module directives

Supported module directives are listed below. See the section *Chapter 6 Module Control directives* on page 6-60 for complete usage instructions. Module directives apply to every instance of the module in which they are declared.

- `allowFlattening`
- `disableOutputSysTasks`
- `disallowFlattening`
- `enableOutputSysTasks`
- `flattenModule`
- `flattenModuleContents`
- `hideModule`

To apply a module directive as an embedded directive, add the comment to the same line as the module declaration. To apply multiple embedded module directives, specify the directives on separate lines.

In the following example, the `disallowFlattening` directive is applied to the module called `top` and the `allowFlattening` and `enableOutputSysTasks` directives are applied to the module called `bottom`:

```
module top(clock, in1, in2, out1); // ARM_CM disallowFlattening
...
    bottom u1(clock, in1, in2, out1);
endmodule
module bottom(clock, in1, in2, out1); // ARM_CM allowFlattening
                                // ARM_CM enableOutputSysTasks
...
endmodule
```

Chapter 5
Net Control directives

This chapter describes the Cycle Model Compiler directives that control nets.

It contains the following sections:

5.1 General net control directives

This section describes Cycle Model Compiler directives related to nets.

tieNet

This directive allows you to disable logic within a design by tying it to a constant. This applies to all instances of a module. The syntax for this directive is:

```
tieNet HDL_constant list_of_module_nets
```

where:

- `HDL_constant` is a Verilog constant expression of the form `N'hnumber` or `N'bnumber`. If `X` or `Z` is present in the constant, the Cycle Model Compiler issues an error and the compile fails. If the net is not large enough to hold the specified tie value, then the value is truncated to fit into the net.
- `list_of_module_nets` is a white-space separated list of `module-name.net-name` strings. For vectors, the specified constant applies to the entire net (part or bit selects are not supported). If an instance-based name is used, the Cycle Model Compiler issues an error and the compile fails.

`tieNet` constants are treated as unsigned constants. If the net is signed, its value is equivalent to a simple cast. If the net is smaller than the constant, it is truncated. If the net is larger than the constant, it is zero-extended.
Limitations:

- You can apply the `tieNet` directive only to simple types, like wires in Verilog. More complicated types (for example, arrays of arrays, structs, or enumerations) are not supported.
- You can not apply the `tieNet` directive on a net that has either two or more dimensions, or at least one unpacked dimension.
- You can not apply the `tieNet` directive on a net within a task.

Processing:
All module nets specified with the `tieNet` directive are processed as follows:

1. All assignments to the net are removed.
2. A new `continuous assign` is added to the module with the constant on the right-hand side.

This process is performed before optimizations, and occurs even if optimizations are disabled (see the `-O string` option in *3.2 General compile control options* on page 3-28 for more information). This process could lead to further optimizations, which may improve performance of the resulting Cycle Model.

ignoreSynthCheck ***`list of signals`***

Suppresses sensitivity list checking for the specified nets in Verilog designs. By default, the Cycle Model Compiler performs synthesis checking in the sensitivity list of `always` blocks during the build. You may receive errors and warnings about unsynthesizable constructs in your design. If there are sensitivity list constructs that cannot be fixed, or that you know do not cause issues in your design, use this directive to selectively suppress sensitivity list checking for those nets.

5.2 Signal-related net control directives

The `observeSignal`, `depositSignal`, and `forceSignal` directives provide access to specified signals in the Cycle Model from the external environment. The Cycle Model API functions can be used to observe, deposit, and force signal values.

observeSignal ***`list of signals`***

Identifies the nets in the design that must be observable during runtime. The current values can be retrieved. For a signal to be observable during validation runtime, it must be marked before compilation. Marking a signal as observable does not imply that it will accept deposited values.

——— **Note** ———

This directive can not be applied on a net within a task.

———————

Nets marked as observable are never optimized away by the Cycle Model Compiler, and their values are guaranteed to be correct at all times (values match what the hardware is intended to do). In addition, any dead signal that is marked observable, for example, a signal that does not reach an output of the design, is reanimated.

Nets not marked as observable may be visible when the full symbol table is loaded via the Cycle Model API. However, any unmarked net may be optimized away by the Cycle Model Compiler making the validation runtime faster.

When you plan to use Cycle Model Studio to create a Cycle Model component, you must mark any internal signals with `observeSignal` to make them available in Cycle Model Studio as debug registers or for profiling.

depositSignal ***list of signals***

depositSignalFrequent ***list of signals***

depositSignalInfrequent ***list of signals***

These directives identify the nets in the design that accept deposited values. The nets may be inputs, registers, or un-driven logic. To allow values to be deposited on a signal during validation runtime, the signal must be marked before compilation. Note that marking a signal as depositable does not imply observability. The Cycle Model Compiler does not process deposits as outputs; the driving logic of a depositable net may be eliminated due to liveness checks.

Note

These directives can not be applied on a net within a task.

Deposits on nets can occur frequently, as with a clock, or infrequently, as with a control pin. In some situations, your Cycle Model will run faster if you specify depositable nets as frequent or infrequent during compilation. Follow these guidelines to decide which variation of this directive to use:

- `depositSignal`: The Cycle Model Compiler automatically categorizes a net as frequent or infrequent. In general, all deposits are marked as infrequent unless part of a clock tree, in which case they are marked as frequent. If you are using very few `depositSignal` and `forceSignal` directives, then the `depositSignal` directive is sufficient and does not noticeably impact Cycle Model performance.
- `depositSignalInfrequent`: If your design has many nets that you intend to mark with `depositSignal` or `forceSignal` directives, use the `depositSignalInfrequent` directive to improve performance. To identify infrequently accessed nets, analyze the nets noted in `message 1057` in the compile output and re-run the design, using the `depositSignalInfrequent` directive to mark all infrequently accessed nets (such as control pins and enables).
- `depositSignalFrequent`: To improve performance, specify the `depositSignalFrequent` directive for data pins that change every four schedule calls or more. You do not have to specify the `depositSignalFrequent` directive for clocks and clock tree inputs as the Cycle Model Compiler automatically marks them as frequent.

If both `depositSignal` and one of the other two variations are specified, the Cycle Model Compiler assigns the net according to the more detailed directive. If both of the other two variations are specified simultaneously, the Cycle Model Compiler uses the first one given and prints a warning.

forceSignal ***list of signals***

Identifies the nets in the design that are forcible; the listed signals can be forced to specified values. In order for a signal to be forcible during validation runtime, it must be marked before compilation.

This directive can not be applied on a net within a task. Additionally, it can not be applied on a net that has either:

- two or more dimensions, or
- one or more unpacked dimension

scDepositSignal ***`list of signals`***

scObserveSignal ***`list of signals`***

`scDepositSignal` and `scObserveSignal` provide the same functionality as the `depositSignal` and `observeSignal` command-line options when creating a SystemC component. These directives make the nets available from the Cycle Model API, display the nets in waveforms, etc. They also may be used in HDL comments; for example:

```
reg foo;  // ARM_CM scObserveSignal
```

`scDepositSignal` and `scObserveSignal` must be set prior to compilation of the Cycle Model.

Note

The `scDepositSignal` and `scObserveSignal` directives can not be applied on a net within a task.

Whether using the command line `carbon systemCWrapper` tool or Cycle Model Studio, use these directives to make internal nets available as members of the SystemC component. The member signals can be read and written from SystemC to examine and deposit the values of the `scObserveSignal`/`scDepositSignal` nets.

5.3 Clock- and reset-related net control directives

collapseClock

Indicates that the specified clocks are equivalent, meaning that the values are the same (but they may have completely different logical paths). The syntax for this directive is:

```
collapseClock main_clock list_of_subordinate_clocks
```

`main_clock` is a single, specific hierarchical signal in the design. The Cycle Model Compiler substitutes *`main_clock`* for each subordinate clock it finds in the design. This may improve the performance of the generated object.

A clock equivalency table is generated automatically and written to the file `./`*`design name`*`.clocks` (`./libdesign.clocks` by default). The clock equivalency table shows all clocks that will be treated as equivalent because they were proven equivalent, or because they were specified to be equivalent with the `collapseClock` directive.

Caution

Equivalent does not mean aliased. The use of this directive is an assertion that the subordinate clocks have the same value as the main clock. The Cycle Model Compiler may or may not alias these clocks together (they may or may not share storage), and does not guarantee changes to data propagation. Setting this option incorrectly can also introduce errors which are difficult to debug if non-equivalent clocks are collapsed.

slowClock ***`clock names`***

Identifies clocks that are routed slowly in the chip, so that they are slower than async resets. If a slow clock rises at the same time a reset is deasserted, then the code with the clock block runs. By default, clocks are assumed to be faster than resets, so resets are assumed to be deasserted in a race. A `slowClock` is slower than any reset, but those resets are not necessarily faster than other clocks.

fastReset ***`reset names`***

Identifies resets that are routed fast in the chip, so that they are faster than clocks. If a clock rises at the same time a reset is deasserted, then the code with the clock block runs. By default, clocks are assumed to be faster than resets, so resets are assumed to be deasserted in a race. A `fastReset` is faster than any clock, but those clocks are not necessarily slower than other resets.

asyncReset ***`reset names`***

Identifies synchronous resets that the Cycle Model Compiler treats as asynchronous resets, if possible. Combined with `fastReset` or `slowClock`, this directive allows reset data to be scheduled faster than clocks.

Chapter 6
Module Control directives

This chapter describes the Cycle Model Compiler directives that control modules.

It contains the following sections:

6.1 General module control directives

This section describes Cycle Model Compiler directives related to module control.

hideModule ***list of module names***

Identifies a module in the design hierarchy that you intend to be hidden in the Cycle Model. Note that this directive applies to all instances of the specified module(s). All waveforms for the specified module and its descendents will be suppressed.
The Cycle Model Compiler issues a warning if the specified module names are not found.

enableOutputSysTasks ***list of module names***

disableOutputSysTasks ***list of module names***

Enables or disables output system tasks by module.

Note

The `-enableOutputSysTasks` command-line option (see *3.6 Verilog- and SystemVerilog-specific control options* on page 3-40) enables support for `$display-type` system tasks throughout a design.

Wild cards are not supported in the module specification. You can specify these directives in the source code if you use the HDL comment form:

```
// ARM_CM enableOutputSysTasks
```

substituteModule

Replaces the body of one Verilog module with the body of another Verilog module.

Note

The term 'body' refers to the functional implementation of the module.

There are two forms of the directive:

- Form 1:

```
substituteModule old_module_name new_module_name
```

- Form 2:

```
substituteModule old_module_name \
portsBegin old_port1_name old_port2_name old_port3_name portsEnd
new_module_name \
portsBegin new_port1_name new_port2_name new_port3_name portsEnd
```

The body of all instances of *old_module* is replaced with the body of *new_module*.
Use Form 1 if:

- The module instantiation uses positional port connections for the port list, and the order of the ports in the *old_module* and the *new_module* positionally match up one-for-one.
- The module instantiation uses named port connections, and the formal port names in the *old_module* and the *new_module* match one-for-one.

Use Form 2 if:

- The module instantiation uses a named port connection for the port list, and there is a need to rename one or more of the formal port names. In this case the named port names are paired up:

```
old_port1_name:new_port1_name,
old_port2_name:new_port2_name,
old_port3_name:new_port3_name ...
```

The old formal port names in the instantiation line are replaced with the `new_port` names. When using this form, the order of the ports in the `old_module` and `new_module` definitions do not need to be identical.
Examples:

```
module1 u1(a,b,c);  // positional port
module2 u2(.fA(a), .fB(b), .fC(c)); // named port
```

Usage notes:

- An equal number of named ports is required in the two `portsBegin .. portsEnd` regions.
- The definition for `new_module` must be included and compiled without errors. The definition for `old_module` is not needed and is ignored.
- If the module instantiation specifies parameter values, and lists only values in the `parameter_port_list`, then the parameter values are mapped by position from `old_module` to `new_module`.
- If the module instantiation uses parameter assignments in the `parameter_port_list`, then the values are mapped by the indicated name. For example, `param1` and `param2` in the following instantiation:

```
module3 #(parm1=5,param2=2) u3(a,b,c)
```

inline *module.task*

inline *module.function*

Forces the inlining of the specified *module.task* or *module.function* into all calling scopes.
The Cycle Model Compiler replaces all specified task and function calls with the contents of the called task or function. Wildcards are allowed for both the module and task/function names.
The inline directive for functions in a package is not supported.
If the specified modules or tasks/functions do not exist in the RTL code, the following warnings are produced:

```
Warning 57: Module x does not exist; 'inline' directive ignored.
Warning 106: Task/Function x does not exist in module top; 'inline' directive
ignored within this module.
```

If the specified task or function is called via hierarchical references, the following warning is generated:

```
Warning 107: Task/Function t in module declarations has hierarchical referrers which
may prevent inlining.
```

6.2 Flattening-related module control directives

Flattening directive usage notes

Note the following interactions when using flattening directives:

- If `disallowFlattening` and `allowFlattening` reference the same module, an error occurs.
- The `flattenModule` and `flattenModuleContents` directives may be applied to a descendant module of a module marked `allowFlattening` or `disallowFlattening`.
- The `flattenModule` and `flattenModuleContents` directives mark areas for unconditional flattening. If `allowFlattening` references a module marked by, or contained within, a `flattenModule` or `flattenModuleContents` module, a warning is generated, and the `allowFlattening` has no affect.
- If `disallowFlattening` references a module marked by or contained within a `flattenModule` or `flattenModuleContents` module, an error occurs.
- If `flattenModule` and `flattenModuleContents` reference the same module, a warning is generated and `flattenModule` prevails.
- If a `flattenModuleContents` sub-module occurs within a `flattenModule` module, an error occurs.
- If a `flattenModule` sub-module occurs within a `flattenModuleContents` module, a warning is generated, and the `flattenModule` directive has no effect.
- A warning is generated if any of these directives are present and flattening has been disabled with the `-noFlatten` command-line option.
- An error is generated if the specified module does not exist.

Flattening directives

flattenModule ***`module name`***

Identifies a module to be fully flattened, including the module itself, disregarding any thresholds (specified with command options). This operation flattens all instances of the named module into the instantiating parent module. For example:

```
flattenModule flop
```

All submodules of `flop` are flattened into `flop`. All instances of `flop` are flattened into the instantiating parent module.

flattenModuleContents ***`module name`***

Identifies a module whose contained hierarchy is to be fully flattened, disregarding any thresholds (specified with command options). The specified module becomes a container for all of the flattened contents. This operation occurs for all instances of the named module. For example:

```
flattenModuleContents pad_block
```

All submodules of `pad_block` are flattened into `pad_block`.

allowFlattening ***`module name`***

Allows flattening to occur under a specified module, respecting thresholds defined by command options. This directive can be used in conjunction with `disallowFlattening` (see next).

disallowFlattening ***`module name`***

Disallows flattening under a specified module, respecting thresholds defined by command options. This directive can be used in conjunction with `allowFlattening` (see previous).
Examples:
The following example assumes `top.machine`:

```
disallowFlattening top
allowFlattening machine
```

In this example, flattening is disabled except for submodules under instantiations of the `machine` module. The contents of the `top.machine` hierarchy are able to be flattened.
The `disallowFlattening` directive can be applied to a descendant module under a module marked with the `allowFlattening` directive. The following example assumes `top.machine.crc32`:

```
disallowFlattening top
allowFlattening machine
disallowFlattening crc32
```

Chapter 7
Output Control directives

This chapter describes directives that control Cycle Model Compiler output messages.

It contains the following sections:

7.1 Compiler message severity levels

By default, the Cycle Model Compiler outputs the following message severity levels:

- Note – Informational only.
- Warning – Indicates a condition that will not cause the operation to fail. However, if not addressed, the issue may resurface at runtime.
- Alert – Indicates a demotable error (may be demoted to a Warning or Note, or may be suppressed).
- Error – Indicates a condition that will cause the generation of the Cycle Model to fail. In this case, the Cycle Model Compiler continues to run and report additional errors, alerts, and warnings before it exits.

  ——— **Note** ———

  The compiler may exit before all possible messages are reported.

  ———————————

- Fatal – Indicates a condition that will cause the generation of the Cycle Model to fail. The Cycle Model Compiler exits immediately.

7.2 Output control directives

You can change the severity of a message.

You can change the severity of `Alerts`, `Warnings`, and `Notes`, such as causing a `Note` to be output as an `Error`, or promoting a `Warning` to an `Error`. You cannot demote `Fatal` or `Error` messages.

errorMsg ***`message IDs`***
Identifies the Cycle Model Compiler message numbers that should be treated as `Errors`.

warningMsg ***`message IDs`***
Identifies the Cycle Model Compiler message numbers that should be treated as `Warnings`.

infoMsg ***`message IDs`***
Identifies the Cycle Model Compiler message numbers that should be treated as informational (as a `Note`).

silentMsg ***`message IDs`***
Identifies the Cycle Model Compiler message numbers to be suppressed. Suppressed messages do not appear in the standard output; instead, they are output to the `./libdesign.suppress` file by default.

Note

You can suppress only `Notes`, `Warnings`, and `Alerts`. `Errors` and `Fatals` can not be suppressed.

Appendix A
Dumping waveforms in different environments

Dumping waveforms from a simulation is important to the debug process. This appendix describes waveform dumping procedures for various environments.

It contains the following sections:

A.1 Waveform dumping implementation notes

This section applies to hierarchy paths to nets and instances that are declared below a Verilog `generate` block. These paths are used in FSDB and VCD files when the Cycle Model is dumping waveforms.

Dumping values into a VCD waveform file from arrays that have two or more dimensions is not supported. When dumping values from arrays with two or more dimensions, select the FSDB format. For proper display of the array, ensure that an `observeSignal` has been applied to that array.

See the *Cycle Model Studio Installation Guide* (101106) for required versions of the FSDB writer library.

Note

The Cycle Model Compiler follows standard Verilog naming conventions for hierarchy paths to nets and instances that are declared below a Verilog `generate` block.

If you are using the SpringSoft Novas VCD-to-FSDB converter (VFast) be aware that the switches `-orig_scopename` and `-orig_varname` force VFast to convert the name according to standard Verilog naming conventions.

A.2 Using the Cycle Model API to dump waveforms

This section describes using the Cycle Model API to dump waveforms using a simple C or C++ testbench.

Create wave files after the Cycle Model is created, but before any calls to `carbonSchedule()`:

1. Create the wave file and save its handle for future API calls. Call `carbonWaveInitVCD` for VCD format, or `carbonWaveInitFSDB` for FSDB format. Following are examples of these functions:

   ```
   CarbonWaveID *wave = carbonWaveInitVCD(obj, "design.vcd", e1ns);
   CarbonWaveID *wave = carbonWaveInitFSDB(obj, "design.fsdb", e1ns);
   ```

   where:

   - `obj` is the `CarbonObjectID` handle acquired when the Cycle Model was created.
   - `design.vcd` is the output file name.
   - `e1ns` is the timescale for the waveform. See the *Cycle Model API Reference* (101067) for a full list of the enumerated types for timescales.

2. After the waveform file is created, add functions to control the waveform dump. For example, to dump a portion of a design's hierarchy:

   ```
   carbonDumpVars(wave, 0, "top");
   ```

   where:

   - `wave` is the `CarbonWaveID` for the file that was created.
   - `0` is the number of levels of hierarchy to dump. The value of zero means all levels.
   - `top` indicates the hierarchy to dump. This can be a module instance or signal instance in the design.

   The level and hierarchy arguments operate in the same manner as in the `$dumpvars` Verilog system task. As with `$dumpvars`, you can make multiple calls to `carbonDumpVars()` to dump multiple parts of a design's hierarchy. For example:

   ```
   carbonDumpVars(wave, 1, "top");
   carbonDumpVars(wave, 0, "top.core.fifo0");
   carbonDumpVars(wave, 0, "top.core.fifo1");
   ```

3. Call the `carbonDestroy()` function at the end of the simulation. This function properly flushes and closes its waveform file. It also frees up memory used by structures, and invalidates the Cycle Model.

For more examples of API functions, see the twocounter example included in your installation (`$CARBON_HOME/examples/twocounter/twocounter.c`).

Suspending waveform dumping during simulation

When a waveform file is created, signals are automatically dumped starting with the first `carbonSchedule()` call. During the course of the simulation, you can suspend and re-enable dumping using the following functions:

```
carbonDumpOff(wave);
carbonDumpOn(wave);
```

A.3 Enabling waveform dumping in SystemC environments

The auto-generated SystemC module (the Arm Cycle Models component for SystemC), has built-in support for waveform dumping. You can enable this on the command line, or by calling Cycle Model Compiler functions.

Waveform file naming

By default, waveform files are named `arm_cm_design.{vcd|fsdb}`. If you use the `-o` command line option (see *3.7 Output control options* on page 3-45, the design name you specify is used.

If you are using the Cycle Model Compiler functions to specify waveform generation, see the *SystemC Wrapper Methods Reference Manual* (`arm_cycle_model_systemc_wrapper_methods_reference_manual.pdf`) in the `userdoc/pdf` directory of your installation.

Specifying waveform updates on all clock edges

Because the Cycle Model component for SystemC is optimized for speed, the scheduler only runs when necessary. To specify waveform updates on every clock edge, using the `-DCARBON_SCHED_ALL_CLKEDGES` option on the command line as shown in the following example:

```
-DCARBON_SCHED_ALL_CLKEDGES=1
```

Enabling waveform updates on all clock edges causes all SystemC modules to schedule on every clock edge. This functionality impacts performance.

Enabling waveform dumping on the command line

To unconditionally enable waveform dumping, add a define (depending on the desired file type) to the `g++` command line when compiling `libdesign.systemc.cpp`. For example:

```
g++ -DCARBON_DUMP_VCD ...
g++ -DCARBON_DUMP_FSDB ...
```

These commands activate `carbonSchedule` on all clock edges and dump all variables.

Calling Cycle Model Compiler functions to enable waveforms

You can dump VCD or FSDB waveforms by calling the appropriate functions of the generated `SC_MODULE` in your SystemC testbench. In the following example, the instance of the `SC_MODULE` is called `dut`:

```
dut.carbonSCWaveInitVCD("arm_cm_design.vcd", SC_NS);
dut.carbonSCWaveInitFSDB("arm_cm_design.fsdb", SC_NS);
```

The timescale in this case is the SystemC type.

Specify the level and hierarchy to dump as follows:

```
dut.carbonSCDumpVars(0, "top");
```

To create a VCD file with a root file name that matches the module name, with timescale `SC_PS`, and to dump the entire design hierarchy, enter:

```
dut.carbonSCWaveInitVCD();
dut.carbonSCDumpVars();
```

For more information about dumping waveforms using the `SC_MODULE` functions, see the *SystemC Wrapper Methods Reference Manual* (`arm_cycle_model_systemc_wrapper_methods_reference_manual.pdf`) in the `userdoc/pdf` directory of your installation.

Appendix B
Using DesignWare replacement modules

This appendix describes replacement modules that may help make your design run faster.

It contains the following sections:

B.1 Improving speed by replacing DesignWare modules

You can use Arm accelerated DesignWare replacement modules to replace existing Synopsys DesignWare® components in your design. These replacement modules are optimized for the Cycle Model Compiler and generally yield faster validation runtimes.

The replacement modules can not be used independently of DesignWare modules; your design must include DesignWare libraries for the replacements to be used.

For a complete list of available replacement modules, see the directive file `DW.dir`.

Syntax

```
cbuild -y Synopsys_DW_path +libext+.v+ -vlogTop myTop myTop.v -f $CARBON_HOME/lib/dw/DW.f
```

where:

- *`Synopsys_DW_path`* is the directory containing your DesignWare files.
- *`myTop`* is the name of the top module in your design.
- *`myTop`*`.v` is the top module.
- `$CARBON_HOME/lib/dw/DW.f` is the command file included in your Cycle Model Compiler installation. This file includes:
  — The `-2001` option to enable Verilog-2001 compilation mode.
  — The full path to the DesignWare replacement library: `$CARBON_HOME/lib/dw/DW_all.vp`.
  — The directive file `$CARBON_HOME/lib/dw/DW.dir`. This file contains the full list of `substituteModule` directives.

Optionally, you can use the following syntax:

```
cbuild –v Synopsys_DW_path/dwlib.v -vlogTop myTop myTop.v -f $CARBON_HOME/lib/dw/DW.f
```

Note

If your design includes SystemVerilog RTL files, then you must direct the Cycle Model Compiler to process your full design as SystemVerilog source. This is because the `DW.f` file internally specifies `-2001` as the compilation mode. To specify SystemVerilog compilation, add the `-sverilog` option to the command line after the command file name. The `-sverilog` command line option overrides the internally-specified `-2001` specification.

Verifying the Replacements

The DesignWare replacement modules append the string `_carbon` to the original DesignWare name. After the compilation is complete, you can see a list of the replacements that were made by the Cycle Model Compiler. Enter the Linux `grep` command::

```
grep DW lib*.hierarchy
```

B.2 Troubleshooting DesignWare replacement issues

Warning 3096

Failure to specify a replacement library path is indicated by warnings similar to the following examples:

```
Warning 3096: substituteModule directive could not find module: DW01_add_carbon
Warning 3096: substituteModule directive could not find module: DW01_add
```

If the module that cannot be found ends in the string `_carbon`, as in the first example, then add `$CARBON_HOME/lib/dw/DW_all.vp` to the `cbuild` command.

If the unfound module does not end in the string `_carbon`, as in the second example, then you can ignore the warning. In this case, the warning indicates that the Synopsys DesignWare module is not being used in the original design. A substitution exists for the unused module.

Error 3030

Failure to identify the top-level module is indicated by an error similar to the following:

```
Error 3030: Multiple top-level modules found:  myTop, DW01_absval_carbon, DW01_sub_carbon …
```

The solution is to add the option `-vlogTop` *`myTop`* to identify the top-level module in your design.

Alert 43003

Failure to specify the Synopsys DesignWare libraries is indicated by an alert similar to the following:

```
Alert 43003: Design unit DW01_add is not found.
```

The Synopsys DesignWare libraries are not being referenced correctly. Verify that the DesignWare libraries are referenced correctly in your `cbuild` command.

Appendix C
Using Profiling to find performance problems

This appendix describes the profiling capabilities of the Cycle Model Compiler.

It contains the following sections:

C.1 Profiling and profiling hotspots

Profiling allows you to learn where a program spends time and which functions call which other functions during execution. This information can help pinpoint performance issues and bugs.

For general information about compiling, running and interpreting the output of the Arm Cycle Model profiling tools, see the `-profile` option in *3.7 Output control options* on page 3-45.

After running profiling to identify the functions that take the most time, examine the items that appear at the top of the profile list, called profiling hotspots, to determine actions that you can take to speed up the Cycle Model simulation time. The information in this section can help you locate and fix performance problems in your code.

Types of performance problems

Performance problems can have many causes, including:

- Time-consuming functions can appear as profiling hotspots. This may be reasonable; however, you can examine these functions to determine if it is possible to re-write them to run faster. See the *Cycle Model Studio RTL Style Guide* (101769) for information about remodeling code to improve performance.
- Even a fast function, if executed often, takes a significant amount of runtime. Therefore, profiling may point to a small function as being a profiling hotspot. In this case, verify that the number of calls to the function are justified.
- If there are multiple calls to a function, some of the function calls may be slow and some are fast.
- The profiler looks at the lowest levels of the design, and therefore may identify a simple library cell as a profiling hotspot. While this may seem counterintuitive, often the use of the cell is important (see Example 1 in *C.3 Re-writing RTL to Improve Performance* on page Appx-C-79).

C.2 Locating the RTL source of a profiling hotspot

When a section of RTL code is identified as a profiling hotspot, it may not be immediately apparent why the RTL code is time-consuming. The next step is to determine how the hotspot's RTL code is used in the design, to determine why it is taking so much time.

Use two techniques to determine how the hotspots are used in the RTL design:
- Use the hierarchy file
- Comment out the hotspots

Use the hierarchy file

This is the simplest approach, but it only works for architectures/modules, not for functions:
1. Check the hierarchy file, `lib`*`design`*`.hierarchy`, to find the parent instances for any module.
2. The hierarchy file lists the module name, with child architectures/modules indented under their parent, the instance name in the second column, and the source file name and line number in the last column. For example:

```
Module    Instance  Location
--------------------------------
bug5391_1           clock1.v:8
m0        u0        clock1.v:27
m1        a1        clock1.v:40
   m1_1   u1        clock1.v:52
m2        u2        clock1.v:66
```
3. To find the name of the hotspot's RTL code, match the location (source file and line number) from the profiling output to the location in the hierarchy file. The calling code for the hotspot is the parent architecture/module.
4. Examine the calling code to identify the reason for the performance problem (see examples in *C.3 Re-writing RTL to Improve Performance* on page Appx-C-79).

Comment out the problem function

Identify the calling code by commenting out the hotspot function:
1. In the RTL source, comment out the hotspot's code block.
2. Recompile the design with the Cycle Model Compiler.
3. The Cycle Model Compiler prints error messages when a part of the code attempts to call the commented-out function. Use the error messages to identify the RTL code (the "calling code") that calls the hotspot.
4. Examine the calling code to identify the reason for the performance problem (see examples in *C.3 Re-writing RTL to Improve Performance* on page Appx-C-79).

   This process identifies one calling source at a time. To locate other sections of code that call the identified hotspot, comment out the identified hotspot plus any previously identified calling code. Then repeat these steps.

Confirm that the identified calling code leads to the profiling hotspot

After identifying the RTL calling code for a profiling hotspot, confirm that these code sections are responsible for the profiling hotspot as follows:
1. Cut and paste the hotspot's RTL code, and create an exact duplicate with a different name. For example: `DUPLICATE_HOTSPOT`.
2. Alter the identified calling code to make it call the newly-created duplicate function.
3. Rerun the profiling process. If the duplicate replaces its original as a profiling hotspot, you have identified the correct calling code for that hotspot.
4. Consider the interaction between the calling code and the hotspot and, if possible, re-write one or the other to speed up the Cycle Model (see *C.3 Re-writing RTL to Improve Performance* on page Appx-C-79) for examples of remodeling slow code).

C.3 Re-writing RTL to Improve Performance

The examples in this section demonstrate how to improve performance in the RTL source.

Example 1: A simple library cell as a profiling hotspot

After running the profiler, the following AND function appears as a profiling hotspot:

```
TEMP := (others => R);
return (TEMP and L);
```

The RTL calling code is:

```
out1 = (mem[i] && in1) ? val1 :
       (mem[j] && in2) ? val2 :
       (mem[k] && in3) ? val3 ...
```

You can conclude that the L leg of the AND function (the profiling hotspot) is an expensive unconditional memory read, and the R leg is inexpensive.

Example 1 Solution

Remodel the condition calculation so that in* is checked first, and then the mem[] extraction is only done once. this assumess that in* is one-hot.

```
for ( ii = 0 ; ii < 1000; ii++)
 for (jj = 0; jj < 1000; jj++)
   begin
      out[(ii*1000)+jj] = mem[ii] & mem2[jj];
   end
```

This moves the mem[ii] out to the ii loop, so it is only performed once.

Example 2: Infrequently occurring modules as profiling hotspots

Sometimes you can gain performance by improving the modeling of infrequently-occurring modules.

Consider an example in which a reg function, which is essentially a scalar flop, is identified as a profiling hotspot. The lib*design*.hierarchy file shows that there are not many instances of this reg function. However, the hierarchy files indicate a few places where the reg function is used in generates.

Example 2 Solution

Convert the reg instances to reg_vec instances, which perform vectored operations. This results in better performance.

Example 3: Iterative profiling

Profiling is an iterative process. After fixing the most expensive performance problems, rerun profiling to identify the logic that is now at the top of the profile.

For example, consider a case where you rerun profiling after converting the reg instances to reg_vec instances. The results show that the reg_vec function is now a profiling hotspot. The lib*design*.hierarchy file shows a number of FIFOs that are built with registers.

In this case, the performance issue is the decoder logic used to access the entries of the FIFO. Each register's write enable is computed with logic in a generate statement similar to the following:

```
(WADDR == i)
```

If instead the memory is accessed directly with WADDR, none of this logic would be necessary.

Example 3 solution

Remodel the `write` logic as a memory `write` process as follows:

```
if (WCLK == '1')
    DATA_REG[WADDR] <= DIN;
```

This remodeling improves the performance slightly. The greatest performance gain is from remodeling the most used architectures; remodeling the smaller and less-used FIFOs may not yield a big gain.

C.4 Flat and hierarchical profiles

Use the `carbon profile` command in combination with the `-profile` option to enable block profiling of the Cycle Model. Profiling data is sent to standard output.

The following is a sample of `carbon profile` output:

```
Profiling data collected May 21, 2008 08:03
Model: design
  %  Cum %     Time Type                   Parent               Location
35.7  35.7     6.63 carbonSchedule
30.1  65.8     5.58 carbonDeposit
17.5  83.2     3.24 <outside of Arm CycleModels>
 2.4  85.6     0.44 carbonReadMemFile
 1.8  87.4     0.33 AlwaysBlock            sub                  profile1.v:26
 1.2  88.6     0.23 AlwaysBlock            flop                 profile1.v:53
 1.0  89.7     0.19 AlwaysBlock            flop                 profile1.v:97
 0.9  90.6     0.17 (none)                 (none)               (none)
 0.9  91.4     0.16 AlwaysBlock            flop                 profile1.v:77
 0.9  92.3     0.16 AlwaysBlock            flop                 profile1.v:85
 0.9  93.2     0.16 AlwaysBlock            flop                 profile1.v:89
 0.8  94.0     0.15 AlwaysBlock            flop                 profile1.v:73
 0.8  94.7     0.14 AlwaysBlock            flop                 profile1.v:69
 0.6  95.4     0.12 AlwaysBlock            flop                 profile1.v:45
 0.6  96.0     0.11 AlwaysBlock            flop                 profile1.v:93
 0.5  96.5     0.10 AlwaysBlock            flop                 profile1.v:61
 0.5  97.0     0.10 AlwaysBlock            sub                  profile1.v:32
 0.5  97.6     0.10 AlwaysBlock            flop                 profile1.v:81
 0.5  98.1     0.10 AlwaysBlock            flop                 profile1.v:105
 0.5  98.7     0.10 AlwaysBlock            flop                 profile1.v:57
 0.5  99.1     0.09 AlwaysBlock            flop                 profile1.v:65
 0.5  99.6     0.09 AlwaysBlock            flop                 profile1.v:49
 0.4 100.0     0.07 AlwaysBlock            flop                 profile1.v:101

Top-Down Hierarchy and Component View
Order Level  Self%  SelfTime  Inst%  InstTime  Comp% CompTime  Instance (Component) Location
----- -----  -----  --------  -----  --------  ----- --------  ------------------------------
    0     1   0.00      0.00  13.47      2.50   0.00     0.00    top (top)  profile1.v:1
    1     2   1.16      0.21   6.73      1.25   2.32     0.43    S1 (sub)   profile1.v:12
    2     3   1.39      0.26   1.39      0.26  11.15     2.07    F0 (flop)  profile1.v:37
    3     3   1.39      0.26   1.39      0.26  11.15     2.07    F1 (flop)  profile1.v:37
    4     3   1.39      0.26   1.39      0.26  11.15     2.07    R0 (flop)  profile1.v:37
    5     3   1.39      0.26   1.39      0.26  11.15     2.07    R1 (flop)  profile1.v:37
    6     2   1.16      0.21   6.73      1.25   2.32     0.43    S0 (sub)   profile1.v:12
    7     3   1.39      0.26   1.39      0.26  11.15     2.07    F0 (flop)  profile1.v:37
    8     3   1.39      0.26   1.39      0.26  11.15     2.07    F1 (flop)  profile1.v:37
    9     3   1.39      0.26   1.39      0.26  11.15     2.07    R0 (flop)  profile1.v:37
   10     3   1.39      0.26   1.39      0.26  11.15     2.07    R1 (flop)  profile1.v:37

Key: Order - Use to revert to original order if the lines are sorted
     Self  - The time in this instance of the component not including sub-instances
     Inst  - The time in this instance including sub-instances
     Comp  - The time for all instances of this component not including sub-components
```

The profiling output is broken out into two sections. The first section is a flat profile and the second is a hierarchical profile.

Flat profile

The flat profile lists sampled blocks in decreasing order of runtime usage:

- Column 1 - Percent of time spent in the block.
- Column 2 - Running total (cumulative) percentage.
- Column 3 - Number of seconds spent executing the block.

The `<outside of Arm Cycle Models>` category contains:

- Time spent executing non-Cycle Model code; that is, user code or third-party code.
- Time spent by the testbench waiting for user interaction.

The sample output shows that the model design spends approximately 17% of its time outside of Cycle Model code, and almost two-thirds of its time running `carbonDeposit` and `carbonSchedule` API calls. Approximately 13% of the time is spent in the listed `always` and `continuous assignment` blocks.

- Parent column - Verilog module for that block of code.
- Location column - Source locator for blocks that correspond to RTL. The `carbon profile` command does not specify locations for testbench code or API functions (for example, `carbonSchedule`) that are found in the `<outside of Arm Cycle Models>` category. This is because these items are not found in the original RTL.

———— **Note** ————

The time displayed for `carbonSchedule` does not include time spent in blocks.

————————————

Hierarchical profile

The hierarchical profile uses the same profile data, but reorganizes the data into a hierarchical view.

The logic buckets are grouped into modules and instances. There are two sub-views of the data: elaborated and unelaborated:

- `Order` column - Indicates the original order of the profiling lines. This is useful if the data is loaded into a spreadsheet and sorted by different data columns. It allows the spreadsheet to return to the original order.
- `Level` column - Elaborated depth for the next four columns from the top of the design, where `1` indicates the top-level component. The elaborated view is an instance-based view of the data. This means that if there are multiple instances of a component, the time in the instance is a fraction of the time for the component as a whole.
- `Self%` and `SelfTime` columns - Percentage and time for a component instance. These values do not include the time for any sub-instances. If multiple instances of a component exist, then the value is the total time for the component divided by the total number of instances. For example, in the sample, `sub` has two instances: `S0` and `S1`. Each instance takes 1.16% of the time. This means the `sub` component takes 2.32% of the time.
- `Inst%` and `InstTime` columns - Percentage and time for an instance and its child instances. For example, in the sample profile, the `Instance` time for `S1 (sub)` is the `Self` time for `S1` (`.21s`) plus the `Self` time for each of its four children:
  - `F0` (`.26s`)
  - `F1` (`.26s`)
  - `R0` (`.26s`)
  - `R1` (`.26s`)

  This adds up to 1.25 seconds, or 6.73% of the time.
- `Comp%` and `CompTime` columns - Unelaborated view of the design. Total time for all instances of that component, excluding any sub-components. For example, there are eight instances of the `flop` component. Each instance takes 1.39% of the time and the component as a whole takes 11.15% of the time.
- `Instance (Component) Location` column - Shows the design hierarchy, including the instance name, the component name in parentheses, and the file location for the component.

Appendix D
Using Design Reducer to troubleshoot design errors

This appendix describes how to use the Design Reducer utility. Use this tool when working with Arm support to troubleshoot your design.

It contains the following sections:

D.1 Introduction to the Design Reducer

In cases where the Cycle Model Compiler fails to properly compile a Cycle Model, or the generated Cycle Model emits an error when used, the Cycle Model Compiler provides a descriptive message of the error. However, determining how to fix the problem can still be challenging, especially in large designs.

A common approach to troubleshooting the error is to manually modify the design file by removing large portions of the design that seem unrelated to the error. The new design is then manually checked to determine if the error is still present. The Design Reducer automates this approach.

Design Reducer functionality

When you invoke the Design Reducer, it:

1. Parses and elaborates the design, writing the entire design to the file `libdesign_designInitial.sv` in the directory from which the Cycle Model Compiler was called. If a file with the same name exists, it is overwritten.
2. Re-parses this file and removes comments to produce the first candidate design, which is passed to the Check Design script. If this initial, un-reduced (but comment-clean) design reproduces the error as defined by the check script, the Design Reducer begins the reduction process.
3. Design Reducer performs reductions in one or more runs. Each run performs some reduction before calling the Check Design script to verify that the reduced design continues to produce the error.

Each run begins by printing the run number, in addition to the current strategy. It then performs reduction actions on the file and outputs the name of the candidate design file, which is in the form `libdesign_reduction_`*`id`*`.sv`. Additionally, it reports the line-count of the reduced file.

All design files that reproduce the error are kept over the course of Design Reducer operation. The Design Reducer requires the two most recent files to exist, so do not delete them. One is the candidate design being tested, and the other is the backup of the last design that reproduced the error.

This section contains the following subsections:

- *D.1.1 Use cases and limitations* on page Appx-D-84.
- *D.1.2 Design Reducer strategies* on page Appx-D-84.
- *D.1.3 Design Reducer context restoration* on page Appx-D-85.
- *D.1.4 Completing the reduction* on page Appx-D-85.

D.1.1 Use cases and limitations

The Design Reducer is most useful in the following cases.

- Cycle Model internal errors while running the Cycle Model Compiler.
- Compilation errors that occur in GCC during the backend compilation phase.
- Duplicating a specific error or warning message using a smaller design.

The Design Reducer may not be the best tool in the following cases:

- Simulation errors, such as mismatches. Design Reducer reductions change the behavior of the design, so it's unlikely that the same simulation issue can be reproduced.
- Verilog parse errors. The Design Reducer requires the design RTL to be synthesizable. A design that can not be parsed can not be reduced.

D.1.2 Design Reducer strategies

The Design Reducer performs its reductions as a group of strategies, run in sequence. This allows the Design Reducer to make more valuable reductions before performing smaller ones. This approach reduces the amount of time the Design Reducer takes to run.

Following are some of the strategies that may be applied to reduce the size of the design file:

- Change the top-level module
- Remove instances
- Remove statements
- Remove unused module or package items

The switch `-reducerStrategies` allows you to specify a subset of the strategies to apply to the design. Contact Arm Support for more information.

D.1.3 Design Reducer context restoration

All design files produced by the Design Reducer include a header comment at the top. This header comment contains information about the progress of the Design Reducer, such as the current strategy being run and the command line arguments that are supplied to the Check Design script.

The header also includes a record of work that the Design Reducer has already performed. This means that you can manually stop and restart the Design Reducer with little loss of progress. Be sure to supply the most recent file as the new starting file.

You can manually modify this header file to change the progress of the Design Reducer, as the Design Reducer never leaves the design in a state that does not reproduce the error. When modifying the header file, however, be aware that your modifications may cause the Design Reducer to redo previous failed reduction attempts, or to skip over sections that have not yet been reduced.

A useful modification is resetting the current strategy. To do so, delete the following:
- Line and column number, known as "CurrentCursor" (set to 0:0)
- Current module (set to blank)
- Args (set to blank)

By making these changes and restarting the Design Reducer with the modified file, the reduction process starts the identified strategy over again. This means, however, that it may re-attempt reductions that are already known to not reproduce the design.

D.1.4 Completing the reduction

After all strategies have been performed, the Design Reducer prints the final design to `libdesign_designReduced.sv`, and displays the original and final sizes of the files, along with the number of runs that reduced the file.

As a next step, you can copy the `libdesign_designReduced.sv` file into another file name and manually modify it yourself to try to reduce the design size further. You can then call the Design Reducer again on the manually modified file; it will attempt to make further reductions.

D.2 Invoking the Design Reducer

You can invoke the Design Reducer on the command line, or using a command options file.

As an argument to the `-designReducer` option, provide a Check Design script that defines the errors, warnings, or other output for the Design Reducer to look for in your design.

You must set the executable flag on the Check Design script.

This section contains the following subsections:
- *D.2.1 Invoking the Design Reducer using the command line* on page Appx-D-86.
- *D.2.2 Invoking the Design Reducer using a command options file* on page Appx-D-86.

D.2.1 Invoking the Design Reducer using the command line

The command line method of invoking Design Reducer is described in this section.

Syntax

```
cbuild verilog_language_variant -q -designReducer [-reducerStrategies N]
check_design_script -vlogTop top_level_module source_file
```

Options

verilog_language_variant
Required. Verilog language variant. See *1.8 Specifying the Verilog language variant to use when compiling* on page 1-17.

`-reducerStrategies` ***N***
Optional. By default, the Design Reducer runs all strategies. This option allows you to specify a subset of the strategies to apply to the design. Contact Arm Support for details.

check_design_script
Required. Script that you design. See *D.3 Defining errors to identify with the Check Design script* on page Appx-D-88.

top_level_module

Required. See *3.6 Verilog- and SystemVerilog-specific control options* on page 3-40.

source_file

Required. The design you want to reduce.

Example

The following command-line example runs the Design Reducer on a SystemVerilog design source file called `myDesign.sv` with a Check Design script called `checkDesign.sh`:

```
cbuild -sverilog -q -designReducer checkDesign.sh -vlogTop top myDesign.sv
```

D.2.2 Invoking the Design Reducer using a command options file

You can insert commands into a command options file and call the file on the `cbuild` command line.

Syntax

```
cbuild verilog_language_variant -q -f myCommands.cmdOpts myDesign.sv
```

Options

verilog_language_variant
Required. Verilog language variant. See *1.8 Specifying the Verilog language variant to use when compiling* on page 1-17.

***myCommands*.cmdOpts**

Command options file that contains the Design Reducer invocation, the name of the Check Design script, and `-vlogTop`, which specifies the top-most module in the design hierarchy. See the `-vlogTop` option in *3.6 Verilog- and SystemVerilog-specific control options* on page 3-40). The contents of *`myCommands`*`.cmdOpts` are:

```
-designReducer checkDesign.sh
-vlogTop top
```

D.3 Defining errors to identify with the Check Design script

The Cycle Model Compiler installation includes sample Check Design scripts, located in the directory `$CARBON_HOME/util/designreducer/`.

The scripts are named:

- `checkDesign.sh` (Bourne shell)
- `checkDesign.csh` (C shell)

Using one of these scripts as a template, modify it to define errors you want to identify in your design.

The Check Design script included with Cycle Model Compiler runs an additional instance of the Cycle Model Compiler on the design, and sends its output to a `grep` command (which `grep`s for the error message from the original design).

Modifying the Check Design script

When modifying the Check Design script (or creating your own), refer to the following guidelines:

- You must set the executable flag on the Check Design script.
- Error messages should be general enough to allow for reduction, but not too general. For instance, do not include the exact module hierarchy of the error as part of the grep command, because this prevents the Design Reducer from changing the top-level module of the design. The error message of the reduced design may not reproduce the entire hierarchy.
- The script must have a return value of `0`, `1` or `2`. These return values mean:
  — `0` — The error was reproduced successfully; the reductions made are valid.
  — `1` — No error (as defined by the check design file) was produced. The reductions made are not valid and should be reverted. The newly-created design file is deleted.
  — `2` — An error was produced, but it was not the original error. While the reductions must be reverted, the design file is kept with the name extension `.newerror`.
  — Any other value — The script did not function as intended. If this is returned the Design Reducer stops. You can use this value to direct the Design Reducer to terminate.
- When the Design Reducer calls the Check Design script, it passes the Check Design script the following command line arguments. These may be used in other Cycle Model Compiler invocations, logging, or any other calls the script may make:

```
verilog_language_variant -q reduction_candidate_file -vlogTop top_level_module
```

  — For a C-shell script, you can access these arguments with `$argv` in the script
  — For a bash shell, you can access these arguments with `$@` in the script