Cycle Model Studio SystemC Integration

Version 11.4

Application Note

arm

101198_1104_00_en

Cycle Model Studio SystemC Integration

Application Note

Release Information

Document History

Issue	Date	Confidentiality	Change
0905-00	01 February 2018	Non-Confidential	Release 9.5
1000-00	01 August 2018	Non-Confidential	Release 10.0
1100-00	01 May 2019	Non-Confidential	Release 11.0
1103-01	20 January 2021	Non-Confidential	Documentation update
1104-00	12 February 2021	Non-Confidential	Release 11.4

Non-Confidential Proprietary Notice

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Web Address

developer.arm.com

Progressive terminology commitment

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

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

Contents

Cycle Model Studio SystemC Integration Application Note

Preface

This guide describes how to integrate a Cycle Model into a SystemC development environment.

This preface introduces the *Cycle Model Studio SystemC Integration Application Note*.

It contains the following:

This guide describes how to integrate a Cycle Model into a SystemC development environment.

About this book

This guide describes how to integrate a Cycle Model into a SystemC development environment.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction

This guide describes how to integrate a Cycle Model into a SystemC development environment.

Chapter 2 Integrate a Cycle Model with the SystemC environment

This section describes the process for integrating a Cycle Model with SystemC.

Chapter 3 SystemC Wrapper tool command line reference

This section describes the command line options for the Cycle Model Studio SystemCWrapper tool.

Chapter 4 Access internal signals

This section describes how to access internal signals in an SC_MODULE.

Chapter 5 Enable waveform dumping

This section describes how to configure waveform dumping.

Chapter 6 Simulate a Cycle Model with a vector file

This section describes how to use a vector file during simulation.

Glossary

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

See the *Arm® Glossary* for more information.

Typographic conventions

italic

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

bold

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

`monospace`

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

`monospace`

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

`monospace italic`

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

`monospace bold`

Denotes language keywords when used outside example code.

`<and>`

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

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

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

Feedback

Feedback on this product

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

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

Feedback on content

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

- The title *Cycle Model Studio SystemC Integration Application Note*.
- The number 101198_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 guide describes how to integrate a Cycle Model into a SystemC development environment.

It contains the following sections:

1.1 Benefits of SystemC integration

The SystemC design environment provides you with an effective method for refining a design from a high-level of abstraction to an implementation-level model.

You can use a SystemC model to verify the design and integrate system software. However, when a SystemC model is refined down to an implementation level, it may suffer from performance bottlenecks, diminishing its value.

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

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

1.2 Install Accellera SystemC

Cycle Model Compiler code works with SystemC Version 2.3.1.

To dowload SystemC, visit *http://accellera.org/downloads/standards/systemc*.

1.3 Example design

The directory `$CARBON_HOME/examples/systemC` includes example files for generating a SystemC pin-level model with a wrapper from RTL.

One example design used to demonstrate this application is a simple FIFO with separate read and write clocks. A SystemC test-bench is included that randomly loads and unloads data into the FIFO.

To integrate this design into SystemC, the SystemC wrapper tool generates a component for SystemC (the wrapper) that invokes the Cycle Model at the correct times. The wrapper ensures that data is correctly transferred in and out of the Cycle Model.

The following figure illustrates the design referenced in this guide.

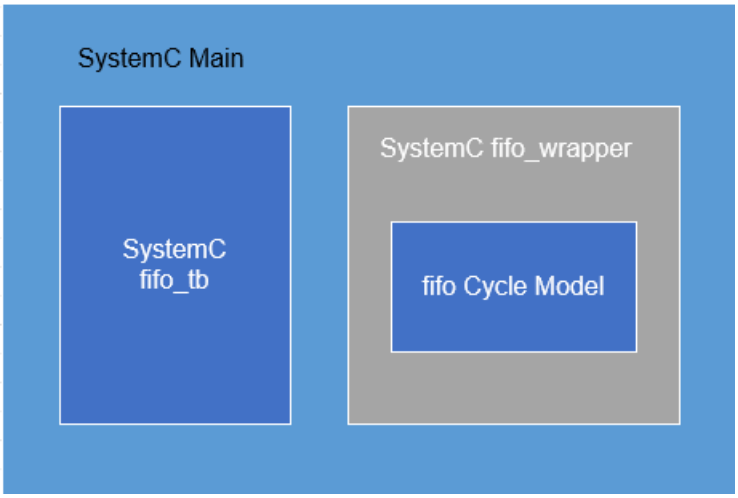

Figure 1-1 Example design

Chapter 2
Integrate a Cycle Model with the SystemC environment

This section describes the process for integrating a Cycle Model with SystemC.

It contains the following sections:

2.1 Overview of the process

This section provides a high-level description of the integration procedure. Details are included in the sections that follow.

1. Identify the Verilog or SystemVerilog module to be used in a SystemC simulation, and compile it with Cycle Model Studio. This generates the Cycle Model.
2. Run the SystemC wrapper tool using the following command:
   - Using Cycle Model Studio:

     ```
     carbon systemCWrapper -ccfg design.ccfg
     ```

   - If you are not using Cycle Model Studio:

     ```
     carbon systemCWrapper libdesign.io.db
        [-portType portName=systemCType]
        [-moduleName name]
     ```

3. Link the Cycle Model and component for SystemC with the other testbench code used in the existing environment.

2.2 Modify the Makefile

When you run `make` in the `$CARBON_HOME` build area, the steps to create the Cycle Model are executed automatically. To compile the design correctly, certain modifcations to the Makefile are required:

- Set `SYSTEMC_ARCH` based on the architecture of the computer running the compilation. Set to `linux` for Linux machines.
- Set `SYSTEMC_HOME` to the top directory in which the SystemC release has been compiled. This example has been validated against the SystemC shipped with the Arm tools.
- Set `CARBON_HOME` to the Arm Cycle Model Studio installation directory.
- If you want to dump waveforms during runtime, uncomment one of the following lines in the Makefile:

```
#WAVE_DUMP = -DCARBON_DUMP_VCD=1
#WAVE_DUMP = -DCARBON_DUMP_FSDB=1
```

2.3 Create the Cycle Model

This section describes the command to create the Cycle Model.

Prerequisites

Make required modifications to the `Makefile`.

Create the Cycle Model

To create the Cycle Model for the FIFO, invoke the Cycle Model Compiler and reference the RTL for the FIFO module:

```
> make libfifo.a
```

2.4 Create the SystemC wrapper

This section describes the commands to create the SystemC wrapper and perform any customizations.

Prerequisites

Create the Cycle Model

Create the SystemC wrapper.

Use the SystemC wrapper tool to create the component for SystemC (a SystemC wrapper around the Cycle Model):

```
> carbon systemCWrapper libfifo.io.db
```

The SystemC wrapper tool generates two SystemC files: `libfifo.systemc.cpp` and `libfifo.systemc.h`.

See *3.1 SystemC Wrapper tool command line options* on page 3-22 for more command line options.

Customize the component for SystemC (optional)

You may want to customize the component for SystemC, while ensuring that the edits are reused if you rerun the SystemC wrapper tool. To do so, edit the Carbon User Code sections of the libfifo.systemc.cpp and libfifo.systemc.h files. See the following example:

```
// from libdesign.systemc.cpp
void component::end_of_elaboration()
{
    // CARBON USER CODE [PRE componentEnd Of Elaboration] BEGIN
    /* code that goes here is preserved when the wrapper is regenerated
    */
    cout
    //"before end of elaboration";

    // CARBON USER CODE END
    ... generated code to initialize model outputs goes here ...
    // CARBON USER CODE [POST componentEnd Of Elaboration] BEGIN
    cout
    //"after end of elaboration";

    // CARBON USER CODE END
}
```

Empty `Carbon User Code` sections or sections containing white space are ignored. Any `Carbon User Code` sections that are unused during component generation are appended to the end of the generated source in an `#if 0` block and a warning is generated.

Any changes made to the `Carbon User Code` section of the source files are retained when you re-run SystemC using the configuration file, assuming that you do not move the source files from the output directory.

You must use the generated `Carbon User Code` sections. If you insert your own `Carbon User Code` sections, they are ignored.

2.5 Compile the component for SystemC

The SystemC module that encapsulates the Cycle Model must be compiled into an object so that it can be linked into an executable.

The testbench and top-level files must be compiled as well. In the commands below, be sure to replace `SYSTEMC_INCLUDE_PATH` with the correct path for your SystemC installation:

```
> g++ -c -I SYSTEMC_INCLUDE_PATH -I$CARBON_HOME/include libfifo.systemc.cpp
> g++ -c -I SYSTEMC_INCLUDE_PATH -I$CARBON_HOME/include fifo_tb.cpp
> g++ -c -I SYSTEMC_INCLUDE_PATH -I$CARBON_HOME/include main.cpp
```

or:

```
> make libfifo.systemc.o fifo_tb.o main.o
```

In this example, the Makefile contains the comments which can be edited to turn on waveform dumping. For details about controlling waveform dumping, see *Chapter 5 Enable waveform dumping* on page 5-27.

Note

The `CARBON_LIB_LIST make` variable links the program so that `LD_LIBRARY_PATH` overrides `-rpath`. This means that you must use a single GCC version within your environment to avoid library conflicts. While a Cycle Model has no dependencies on compiler libraries, custom code compiled with the Arm-provided GCC may have dependencies. If this code is integrated into an environment that uses a different version of GCC (for example, a third-party tool), runtime errors may occur. In environments such as this, we recommend that the GCC provided by the third-party tool be used to compile the custom code.

2.6 Link the executable

This section describes the required options for compilation.

Prerequisites

Compile the component for SystemC.

Required options

All of the files that were generated now need to be linked into a single executable. There are a number of required libraries that need to be linked to form the executable.

```
> make run.x
```

In the following description of required options, the user-defined environment variables are defined as follows:

- `CARBON_HOME` — Cycle Models installation directory
- `SYSTEMC_HOME` — SystemC installation directory
- `CARBON_MODEL_DIR` — Location of Cycle Model and component for SystemC

Required g++ options (using Makefile notation):

- Non-SystemC includes, options, and sources:

```
-I$(CARBON_HOME)/include -I$(CARBON_MODEL_DIR)
-c $(CARBON_MODEL_DIR)/lib<i>design</i>systemc.cpp
```

- SystemC-related includes, libraries, and options:

```
-I$(SYSTEMC_HOME)/include
-fexceptions
```

To link a Cycle Model to SystemC, the following are the required g++ options (using Makefile notation):

- Non-SystemC includes, library, options, and objects

```
-I$(CARBON_HOME)/include -I$(CARBON_MODEL_DIR)
-L$(CARBON_HOME)/Linux/lib/gcc/shared -lcarbon5
$(CARBON_MODEL_DIR)/lib<i>design</i>.a lib<i>design</i>.systemc.o
```

- SystemC-related includes, libraries, and options:

```
-I.  -I$(SYSTEMC_HOME)/include
-L.  -L$(SYSTEMC_HOME)/lib-$(SYSTEMC_ARCH)
-fexceptions
```

2.7 Run the testbench

This section defines prerequisites and describes how to execute the design with the embedded Cycle Model.

Prerequisites

Before running the SystemC simulation with a Cycle Model, define the shared library `libcarbon5.so` in the `LD_LIBRARY_PATH` environment variable. `libcarbon5.so` is found in the following locations:

- Linux: `$CARBON_HOME/Linux/lib/gcc/shared/libcarbon5.so`
- Linux64: `$CARBON_HOME/Linux64/lib/gcc/shared/libcarbon5.so`

Procedure

After the example has been built using the Makefile, enter the following command to execute the design with the embedded Cycle Model:

```
> ./run.x
```

The SystemC copyright is displayed, followed a few seconds later by the following output:

```
Simulation exited at time 50000000 No errors detected.
```

If errors are detected during execution, a message is displayed that provides the timestamp, expected value, and actual value. Errors can be forced by modifying the initial value of `x_data_out` in the `fifo_tb.cpp` file.

Chapter 3
SystemC Wrapper tool command line reference

This section describes the command line options for the Cycle Model Studio SystemCWrapper tool.

It contains the following section:

3.1 SystemC Wrapper tool command line options

The SystemCWrapper tool includes the following command line options:

-ccfg design.ccfg
Configuration file name.

portType *portName=systemCType*
Specifies Type declaration.
Optional. If you are using Cycle Model Studio, declarations can be made in the `.ccfg` file.
Valid types:

- `int`
- `uint` | `unsigned` | `unsigned int`
- `sc_int`
- `sc_bigint`
- `long` | `long int`
- `ulong` | `unsigned long` | `unsigned long int`
- `char`
- `uchar` | `unsigned char`
- `sc_logic`
- `sc_lv`
- `bool`
- `sc_bv`

When using `uint`, `unsigned`, `long`, `ulong`, and `uchar` aliases, you do not need to use quotes in the `portType` declaration. For example, the following are equivalent declarations:

```
-portType “a=unsigned long”
-portType a=ulong
```

-moduleName *name*
Specifies the name of the generated SC_MODULE on the command line.
Optional. If you are using Cycle Model Studio, declarations can be made in the `.ccfg` file.

-forceUpdate
If this option is specified, calls to `sc_prim_channel::request_update` are forced for all input changes. If this is not specified, `request_update` is called only for clock, reset, and feed-through inputs.

Chapter 4
Access internal signals

This section describes how to access internal signals in an SC_MODULE.

It contains the following sections:

4.1 Set directives during the Cycle Model Compiler run

To access internal signals in an SC_MODULE, set directives and then access the correct member variables during simulation.

There are two types of directives that can be used to access internal signals: `sc` directives and standard directives.

SC directives

The `sc` directives are `scObserveSignal` and `scDepositSignal`. These directives are similar to the `observeSignal` and `depositSignal` directives in that they allow external access into the design. These directives also add an `sc_signal` variable to the generated SC_MODULE. This allows the surrounding SystemC environment to directly access the internal signal using SystemC functions and data types. Nets declared with `scObserveSignal` and `scDepositSignal` are shadowed by System C member variables in the SC_MODULE for the DUT.

The values of `scObserveSignal` and `scDepositSignal` nets are updated every time the Cycle Model is executed. For this reason, use them only when necessary to achieve optimal performance.

Standard directives

The standard directives are `observeSignal`, `depositSignal`, and `forceSignal`.

You must use these directives to access memories. In addition, for best performace, use these directives to access signals that need to be observed or deposited only a few times.

These directives require that you access the signals from the testbench SC_MODULE:

```
carbonNetID* popHandle; // works for
    // sc directives (scObserveSignal) and
    // standard directives (observeSignal)
```

Nets declared with `observeSignal`, `depositSignal`, and `forceSignal` are not automatically shadowed using SystemC member variables. However, you can use the Cycle Model API to access them with net handles in the testbench. The API uses the underlying `CarbonObjectID*`, which is available in the generated SC_MODULE.

4.2 Access signals during simulation

You can access signals during simulation through the testbench simulation, or through the component for SystemC.

Prerequisites

Set directives during the Cycle Model Compiler run (see *4.1 Set directives during the Cycle Model Compiler run* on page 4-24).

Access signals using the component for SystemC

To use the SystemC wrapper member variable to access your signals, the signals must have been marked with the `scObserveSignal` directive.

An example of this is shown in the example run earlier. The internal signal `num_pops` is marked with an `scObserveSignal` directive. As a result, its value can be seen directly in `main.cpp`:

```
cout << “Fifo was popped “ << FIFO.num_pops << “ times” << endl;
```

Access signals during simulation

You can use the testbench to access member variables created with either type of directive (`scObserveSignal` or `observeSignal`).

The following SystemC example, `twocounter`, creates an SC_MODULE to encapsulate the testbench, storing internal RTL signals as `CarbonNetIDs`. The DUT is instantiated underneath the testbench. The `twocounter` example, `twocounter.v`, contains the DUT and is located in `$CARBON_HOME/examples/systemc/twocounter`.

The Makefile contains the build rules for the examples. To build the `twocounter` example, enter `make twocounter`.

The `twocounter.cpp` file contains the SystemC testbench, `carbon_testbench`. `carbon_testbench` has declarations for member variables to access internal signals:

```
SC_MODULE( carbon_testbench ) {
    public:
    // Member variable declaration for handles to internal
    // Carbon signals
    CarbonNetID* mInternalSignal1; // marked as forcible in directive
    CarbonNetID* mInternalSignal2; // marked as observable in directive
    ...
```

These signals are initialized in the constructor, right after the ports are hooked up between the testbench and the DUT:

```
// Constructor for the testbench
SC_CTOR( carbon_testbench ) : mTwoCounter("inst") {
....
    // Member variable initialization for handles to internal
    // Carbon signals
    mInternalSignal1 = carbonFindNet(mTwoCounter.carbonModelHandle,
                                     "twocounter.internalSignal1");
    assert(mInternalSignal1);
    mInternalSignal2 = carbonFindNet(mTwoCounter.carbonModelHandle,
                                     "twocounter.internalSignal2");
    assert(mInternalSignal2);
```

Using assertions forces SystemC to abort if the hierarchical name lookup fails. This could happen if the DUT signal names are changed and the DUT is recompiled without updating the testbench code. The Cycle Models C API also prints an error message to `stderr` if the lookup fails. The error message can be redirected with `carbonAddMsgCB`.

From this point forward, you can use the member variables `mInternalSignal1` and `mInternalSignal2` with Cycle Models CAPI functions to manipulate the internal signals:

```
carbonExamine(mTwoCounter.carbonModelHandle, mInternalSignal1, &v1,&drive);
carbonExamine(mTwoCounter.carbonModelHandle, mInternalSignal2, &v2,&drive);
carbonForce(mTwoCounter.carbonModelHandle, mInternalSignal1,&mVectorIndex);
carbonRelease(mTwoCounter.carbonModelHandle, mInternalSignal1);
```

To simplify use of some Cycle Model API functions, helper methods are included in the generated SC_MODULE. See the SystemC Wrapper Methods Reference Manual (101070) for more information.

Chapter 5
Enable waveform dumping

This section describes how to configure waveform dumping.

It contains the following sections:

5.1 API methods for waveform dumping

The generated SystemC module exposes Cycle Model API functions as methods, which you can use to control waveform dumping.

Use the following functions to configure waveform behavior. For details about these functions, see the SystemC Wrapper Methods Reference Manual (101070). This manual is located in the CMS installation directory under `userdoc/pdf`.

- `carbonSCWaveInitVCD(const char* fileName, sc_time_unit timescale)`
- `carbonSCWaveInitFSDB(const char* fileName, sc_time_unit timescale)`
- `carbonSCWaveInitFSDBAutoSwitch (const char* fileNamePrefix=NULL, sc_time_unit timescale=SC_PS, unsigned int limitMegs=1000, unsigned int maxFiles=0)`
- `carbonSCDumpVars(unsigned int depth = 0, const char* listOfScopesNets = NULL)`
- `carbonSCDumpVar(CarbonNetID *handle)`
- `carbonSCDumpStateIO(unsigned int depth = 0, const char* listOfScopesNets = NULL)`
- `carbonSCDumpOn() carbonDumpOff()`

You can enable waveform dumping in two ways during the simulation run:

- Dump all variables
- Specify individual variables, timescale, and filename

These are described in the following sections.

To dump all variables

Add one of the following to the `gcc compile` line (depending on the desired file type) of the C++ module that instantiates the design:

- `-DCARBON_DUMP_VCD=1`
- `-DCARBON_DUMP_FSDB=1`

These commands activate `carbonSchedule` on all clock edges and make all necessary API calls for you.

To specify variables, timescale, and filename

1. Activate complete waveforms by adding the following macros to the `gcc compile` line for lib*design*`.systemc.cpp`:
   - `-DCARBON_SCHED_ALL_CLKEDGES=1`
2. Turn on waveform dumping by calling these methods:
   - `carbonSCWaveInitVCD`, `carbonSCWaveInitFSDB` - Initialize wavefile.
   - `carbonSCDumpVars`, `carbonSCDumpVar`, `carbonSCDumpStateIO` - Adds nets to wavefile. If called before calling `carbonWaveInit()`, then a wave file called `carbon_module_instance_name` is automatically initialized. This makes it possible to enable dumping of all signals in a given model instance with a single line of SystemC code:

     ```
     modelInstance.carbonSCDumpVars();
     ```

   For example:

   ```
   mTwoCounter.carbonSCWaveInitVCD(myFile.vcd, SC_NS);
   mTwoCounter.carbonSCDumpVars(1);
   ```

5.2 Best practices for waveform dumping

To improve performance, waveforms and profile data are cached, and written to their respective files when the buffer is full, and when the object is destroyed.

If you call `exit()` before the object is destroyed, then the program ends before the cached data is flushed, and the data is lost.

To avoid losing data, use one of the following approaches:

- Call `sc_stop()` rather than `exit()`:

```
sc_start(10000);
...
sc_stop();
```

- Explicitly flush the waveforms immediately before the `exit()` call. This requires using a handle to the SystemC module:

```
Foo *foo = new Foo(...);
...
foo->carbonSCDumpFlush(); // explicitly flush waveforms
exit();
...
delete foo;
```

Note

The `sc_stop()` approach writes out both waveforms and profiling data. The `carbonSCDumpFlush` approach only writes out waveforms.

Chapter 6
Simulate a Cycle Model with a vector file

This section describes how to use a vector file during simulation.

It contains the following section:

6.1 Simulate with a vector file

This section describes creating a SystemC testbench around an SC_MODULE.

To simulate a SystemC Cycle Model with data in a vector file, execute the following command:

```
> carbon systemCTestbench libdesign.io.db vectorFile
```

This creates a SystemC testbench file called `libdesign.sctestbench.cpp`. The file is placed in your run directory.

Clocks and resets are included in the vectors, so they are treated as ordinary inputs.