Cycle Model Studio

Version 9.2

SystemC User Manual

Non-Confidential

ARM®

Cycle Model Studio
SystemC User Manual

Release Information

The following changes have been made to this document.

Change History

Date	Issue	Confidentiality	Change
November 2016	A	Non-Confidential	Restamp release with 9.0.0
May 2017	B	Non-Confidential	Updated for v9.2

Non-Confidential Proprietary Notice

Confidentiality Status

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

Product Status

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

Web Address

http://www.arm.com

Contents

Chapter 1

Overview

This guide discusses how to integrate a Cycle Model into a SystemC™ development environment. Specifically, we examine the case of replacing an existing module in a design with a Cycle Model. This process includes building the Cycle Model with the Cycle Model Compiler, automatically generating a system module to include this Cycle Model into a SystemC environment, and compiling the design into an executable.

1.1 Integration Benefits

SystemC is a design environment that provides the user with an effective method for refining a design from a high level of abstraction to an implementation level model. A SystemC model can be used to verify the design and integrate system software. However, when a SystemC model is refined down to an implementation level it begins to suffer from performance bottlenecks and much of its value is diminished.

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

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

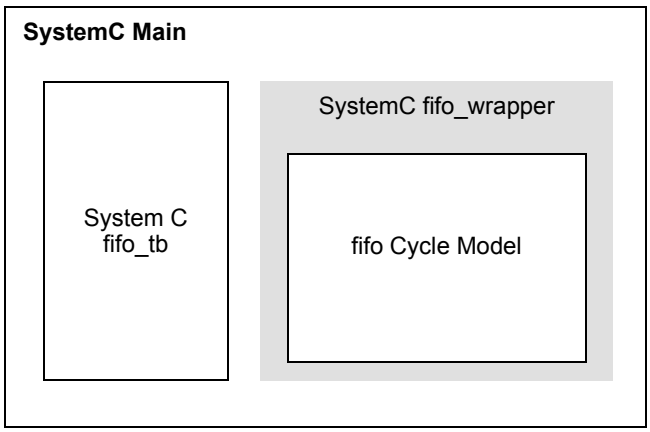
1.2 Accellera SystemC Installation

Cycle Model Compiler code works with SystemC Version 2.3.1. To dowload it, visit http://accellera.org/downloads/standards/systemc.

1.3 Example Design

The 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 and makes sure that data is correctly transferred in and out of the Cycle Model. The following figure illustrates the design referenced in this guide.



Chapter 2

Integration Example

The basic integration procedure includes the following steps (details are included in the subsections that follow):

1. Identify the Verilog 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 (see "Creating the Component for SystemC" on page 2-4 for details):
   - If you are not using Cycle Model Studio:

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

   - If you are using Cycle Model Studio:

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

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

2.1 Setup

The general integration concepts discussed in the following steps assumes a fully compiled SystemC build area, as well as a working knowledge of the following:

- SystemC
- The Cycle Model Compiler command line and options

All of the following steps are executed automatically when you run `make` in the `$CARBON_HOME/app_notes/systemC` directory. In order to compile the design correctly, the Makefile for the example needs to be modified.

- `SYSTEMC_ARCH` should be set based on the architecture of the computer where the compile is taking place. It should be set to `linux` for Linux machines.
- `SYSTEMC_HOME` should be set 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.
- `CARBON_HOME` must be set to the ARM installation directory.
- When running on a 64-bit Linux machine, `CARBON_HOST_ARCH` and `CARBON_TARGET_ARCH` must be set correctly depending on whether you want to create a 32- or 64-bit Cycle Model. See “Setting System Architecture Variables” in the *Cycle Model Studio Installation Guide* for more information.
- Optionally, 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
```

2.2 Creating the Cycle Model

Create the Cycle Model for the FIFO. This is done by invoking the Cycle Model Compiler and referencing the RTL for the FIFO module. The Verilog source and Makefile for this example are located in the `app_notes/systemC/` subdirectory.

```
> make libfifo.a
```

2.3 Creating the Component for SystemC

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 “Command Line Reference” on page 3-9 for more command line options.

2.3.1 Customizing the Component for SystemC

At this point, if you want to customize the component for SystemC while ensuring that the edits are reused if you rerun the SystemC wrapper tool, you can edit the "Carbon User Code" sections of the `libfifo.systemc.cpp` and `libfifo.systemc.h` files, as shown in 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 will be ignored.

2.4 Compiling the Component for SystemC

The SystemC module that encapsulates the Cycle Model needs to be compiled into an object so that it can be linked into an executable. The testbench and top-level files need to 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 "Enabling Waveform Dumping" on page 5-15.

Note: The CARBON_LIB_LIST make variable links the program so that LD_LIBRARY_PATH overrides -rpath, therefore, a single GCC version should be used within your environment to avoid library conflicts. While a Cycle Model itself has no dependencies on compiler libraries,

custom code compiled with the ARM-provided GCC may. 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, it is recommended that the GCC provided by the third-party tool be used to compile the custom code.

2.5 Linking the Executable

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
```

To compile a component for SystemC, the following are the required g++ options (using Make-file notation):

Non-SystemC includes, options, and sources required by the Cycle Model:

```
-I$(CARBON_HOME)/include -I$(CARBON_MODEL_DIR)
-c $(CARBON_MODEL_DIR)/lib<design>.systemc.cpp
-m32              # or -m64 for Linux64
```

SystemC-related includes, libraries, and options that are required:

```
-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 required by the Cycle Model:

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

SystemC-related includes, libraries, and options that are required:

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

where user-defined environment variables are as follows:

`CARBON_HOME` — Cycle Models installation directory

`SYSTEMC_HOME` — SystemC installation directory

`CARBON_MODEL_DIR` — Location of Cycle Model and component for SystemC

To run the SystemC simulation with a Cycle Model, the following shared library ***must*** be accessible at link time location or LD_LIBRARY_PATH, etc.

libcarbon5.so

This file can be found in the following locations:

Linux: $CARBON_HOME/Linux/lib/gcc/shared/libcarbon5.so

Linux64: $CARBON_HOME/Linux64/lib/gcc/shared/libcarbon5.so

2.6 Running the Testbench

Once the example is built using the Makefile, type:

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

to execute the design with the embedded Cycle Model. The SystemC copyright should be displayed, followed a few seconds later by the following:

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

If any 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

Command Line Reference

The `carbon systemCWrapper` tool has the following command line options:

-ccfg *design*.ccfg

If you are using Cycle Model Studio, you should designate the `.ccfg` file on the command line.

-portType <portName>=<systemCType>

You can use the following `<systemCType>` declarations:

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

The `uint`, `unsigned`, `long`, `ulong`, and `uchar` aliases remove the need for quotes in the `portType` declaration, as shown in the following equivalent declarations:

```
-portType "a=unsigned long"
-portType a=ulong
```

The `-portType` option is not necessary if you are using Cycle Model Studio, as the declarations can be made in the `.ccfg` file.

-moduleName <name>

Allows you to specify the name of the generated `SC_MODULE` on the command line.

The `-moduleName` option is not necessary if you are using Cycle Model Studio, as the 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

Accessing Internal Signals

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

4.1 Setting Directives During the Cycle Model Compiler Run

There are two types of directives that can be used to access internal signals:

- The `sc` directives: `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 via 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.

  Because the value of `scObserveSignal` and `scDepositSignal` nets are updated every time the Cycle Model is executed, you should use them only when necessary for optimal performance.

- The standard directives: `observeSignal`, `depositSignal`, and `forceSignal`.

  Memories must be accessed with these directives. In addition, all signals that only need to be observed or deposited a few times should be accessed with these directives for best performance. These directives require that you access the signals from the testbench SC_MODULE:

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

Nets declared with `observeSignal`, `depositSignal`, and `forceSignal` are not automatically shadowed using SystemC member variables, however, the Cycle Model API can be used to access them via net handles in the testbench as shown in the next section. The API uses the underlying `CarbonObjectID*`, which is available in the generated `SC_MODULE`.

4.2 Accessing Signals During Simulation

After setting the directives during the Cycle Model Compiler run as outlined above, how you access those signals during simulation depends on whether you access them through the testbench or through the component for SystemC.

1. Accessing signals via the component for SystemC.

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

   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;
   ```

2. Accessing signals via the testbench member variables.

   You can use the testbench to access member variables created with either type of directive; e.g. `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/app_notes/systemC/twocounter`. The Makefile contains the build rules for the examples. To build the twocounter example, type:
   `make twocounter`

   The `twocounter.cpp` file contains the SystemC 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. Note that 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 C API 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* for more information.

Chapter 5

Waveforms

There are two ways to affect waveforms for simulations:

- Waveform dumping
- Waveform, profile data flushing, and exit() calls

5.1 Enabling Waveform Dumping

The generated SystemC module exposes the following Cycle Model API functions as methods. Please see the *SystemC Wrapper Methods Reference Manual* for details about these functions.

- `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()`

There are two ways to turn on waveform dumping during the simulation run; you can dump all variables or you can control what is dumped.

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 control the variables that are dumped, the timescale, and the 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 // initializes wavefile
   carbonSCDumpVars | carbonSCDumpVar | carbonSCDumpStateIO // adds nets
                                                            to wavefile
   ```

   For example,

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

To simplify usage even further, the following additional wave dumping logic is included with the component for SystemC:

If `carbonSCDumpVar`, `carbonSCDumpVars`, or `carbonSCDumpStateIO` is called before a call to `carbonWaveInit{VCD|FSDB}`, then a VCD wave file named `carbon_<module_instance_name>.vcd` 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();
```

5.2 Waveform, Profile Data Flushing, and exit() Calls

To improve performance, waveforms and profile data are cached, and written to their respective files 1) when the buffer is full, and 2) when the object is destroyed. However, if you call `exit()` prior to the object being destroyed, the program is killed before the cached data is flushed, and the data is lost.

To ensure waveform dumping, use one of the following methods:

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

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

- Explicitly flush the waveforms immediately before the `exit()` call. Note that 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 carbonSCDump-Flush approach only writes out waveforms.

Chapter 6

Testbench

To simulate a SystemC Cycle model with data in a vector file, you can create a SystemC testbench around an SC_MODULE by executing the following command:

```
carbon systemCTestbench lib<design>.io.db <vectorFile>
```

The testbench file, called `lib<design>.sctestbench.cpp`, is placed in your run directory. Clocks and resets are included in the vectors, so they are treated as ordinary inputs.