SystemC Cycle Models

Version 11.0

CPAK Getting Started Guide

arm

101497_1100_00_en

SystemC Cycle Models

CPAK Getting Started Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
1100-00	31 May 2019	Non-Confidential	First release

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Web Address

http://www.arm.com

Contents

SystemC Cycle Models CPAK Getting Started Guide

Preface

This preface introduces the *SystemC Cycle Models CPAK Getting Started Guide*.

It contains the following:

About this book

This guide describes downloading, installing, and running SystemC-based Cycle Model Performance Analysis Kits (CPAKs).

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction to CPAKs

This chapter introduces Arm Cycle Model Performance Analysis Kits (CPAKs).

Chapter 2 Building and running the default CPAK

This chapter describes downloading, compiling, and simulating the CPAK default system.

Chapter 3 Introduction to the makefiles

This chapter summarizes the makefiles that are included in CPAKs and describes the available targets.

Chapter 4 Modifying CPAKs

This chapter describes modifications you can make to SystemC CPAKs and how to rebuild the CPAK.

Chapter 5 Troubleshooting

This chapter provides solutions for problems that may occur when working with CPAKs.

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 *SystemC Cycle Models CPAK Getting Started Guide*.
- The number 101497_1100_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® Information Center*.
- *Arm® Technical Support Knowledge Articles*.
- *Technical Support*.
- *Arm® Glossary*.

Chapter 1
Introduction to CPAKs

This chapter introduces Arm Cycle Model Performance Analysis Kits (CPAKs).

It contains the following sections:

1.1 Introduction to CPAKs

CPAKs are prepackaged systems ready to simulate with a default application.

The applications and components that ship with CPAKs may differ, but all CPAKs have the same general directory structure, environment configuration, and processes for creating and executing the test bench.

After the CPAK is simulating with the default application, you can:

- Make changes to it by modifying the CPAK testbench and corresponding build system to include, instantiate, and connect new or updated models.
- Copy and migrate a SystemC Cycle Model that is part of a CPAK, and build it into your own custom system.
- Modify an Arm CPAK system by adding your own SystemC model classes to the CPAK.

For use cases and instructions, see *4.1 CPAK modification use cases* on page 4-24.

Included in the package

CPAKs include:

- Model libraries for the models included in the CPAK.
- SystemC top-level system.
- Sample applications.
- Setup scripts.
- Cycle Model SystemC Runtime, which also includes:
  - The Cycle Model Studio runtime.
  - SystemC Version 2.3.1 (64-bit Linux).
  - Google Protocol Buffer.
  - The Cycle Model Configuration tool, a command-line utility that simplifies integration of Arm SystemC Cycle Models into custom systems. For usage, see the *SystemC Cycle Model User Guide* for the CPU in your CPAK.

Related information

- *SystemC Cycle Model User Guide* for the CPU in your CPAK. This guide is located in `MODELS/CPU/gccversion`.
- *Arm® SystemC Cycle Model Runtime Installation Guide* (101146).

1.2 System requirements and prerequisites

This section describes space, operating system, and software requirements for running Arm SystemC CPAKs.

Disk space

The Cycle Model SystemC Runtime requires 200 MB of disk space.

For CPAKs, additional space is required. The amount of space needed varies depending on the complexity of the CPAK.

Supported operating systems

The supported Linux operating systems are:

- Red Hat Enterprise Linux 6.6 (64-bit)
- CentOS 6.6 (64-bit)

SystemC CPAKs are not supported on Windows.

Supported GCC versions

For rebuilding SystemC CPAKs, GCC version 4.8.3 and GCC version 6.4.0 are supported.

Licensing

You must have a valid, installed license for each runtime and Cycle Model. Visit the Arm licensing portal: *https://developer.arm.com/support/licensing/generate* and use your serial numbers to generate the licenses. Contact Arm Techical Support (support-esl@arm.com) if you need more information.

1.3 CPAK directory structure

This section describes the directory structure that all CPAKs share.

Using a Cortex-R52 CPAK as an example, the following figure describes the general CPAK directory structure and summarizes its contents. Certain CPAKs may have additional directories.

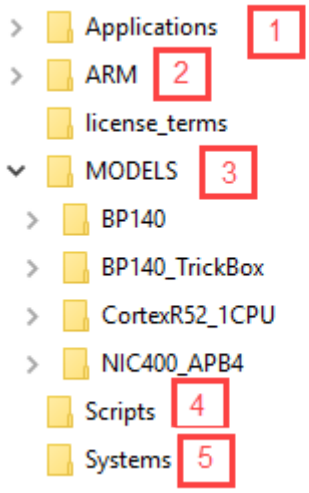

Figure 1-1 CPAK directory structure

1. CPAKs come with sample applications that execute on the processor and application build scripts; the `Applications` directory contains the source code, build files, and executable images for these applications.
2. The `ARM` directory contains the Cycle Models SystemC Runtime, Cycle Model Studio runtime, and third-party files for SystemC and Google Protocol Buffer.
3. The `MODELS` directory contains model libraries and SystemC wrapper source code for all the models used to build the system.
4. The `Scripts` directory contains setup scripts (`setup.sh` and `setup.csh`) used to configure environment variables. This directory may also contain scripts that the CPAK uses for application loading or compilation.
5. The `Systems` directory contains top-level design files used to connect models. It also includes the `system_test.cpp` file (where the `sc_main` function is located), and the `makefile` used to build the CPAK.

Chapter 2
Building and running the default CPAK

This chapter describes downloading, compiling, and simulating the CPAK default system.

It contains the following sections:

2.1 Download a CPAK from IP Exchange

Download a CPAK from IP Exchange.

Before you begin

- You must have a valid account on *http://armipexchange.com/cpaks*. Create one if necessary.

Download

To download a CPAK from Arm IP Exchange:

1. Visit *http://armipexchange.com/cpaks* and download a SystemC CPAK. CPAKs are packaged in a `.tgz` file.

Next steps

Proceed to *2.2 Decompress the CPAK package file* on page 2-15.

2.2 Decompress the CPAK package file

After download, decompress the CPAK package file and review the `README.txt` file.

Before you begin

- Download a SystemC CPAK from Arm IP Exchange (*http://armipexchange.com/cpaks*).
- Review the system and licensing requirements in *1.2 System requirements and prerequisites* on page 1-11.

Decompress the CPAK and review the Readme

1. Untar the CPAK `.tgz` file. For example:

```
$ tar xzvf R52-MP2-MC2-SysC-V10.0.0-CMS10.0.0-MK2018.09.17-SOCD9.6.0.tgz
```

   The CPAK directory structure is created. See *1.3 CPAK directory structure* on page 1-12 for an overview of the structure of the CPAK.
2. Open and review the `README.txt` file located at the top of the CPAK directory structure:

```
$ cd R52-MP2-MC2-SysC-V10.0.0-CMS10.0.0-MK2018.09.17-SOCD9.6.0
$ ls
Applications  ARM  license_terms  Makefile  MODELS  Readme_Debug_Memory_notes.txt
README.txt  Scripts    Systems
$ less README.txt
```

   The `README.txt` file:
   - Lists environment variables that are required to be set for this CPAK.
   - Lists other requirements and dependencies specific to your CPAK.
   - For pin-based CPAKs, states the application that the CPAK runs by default. See the `Applications` directory to determine all of the applications your CPAK includes. See *2.4 Run the CPAK* on page 2-17 for instructions on running applications.
   - Describes scripts included with your CPAK.

Next steps

Proceed to *2.3 Build the CPAK* on page 2-16.

Related information

- *2.1 Download a CPAK from IP Exchange* on page 2-14
- *1.3 CPAK directory structure* on page 1-12

2.3 Build the CPAK

Build the CPAK executable.

Before you begin

- Decompress the CPAK package file and review the `README` as described in *2.2 Decompress the CPAK package file* on page 2-15.

Source the setup script and set required environment variables

Note

Setup scripts for C Shell and Bash are supported. Ensure you are in the correct environment.

1. In the CPAK `Scripts` directory, source the `setup.sh` or `setup.csh` script. You can do this by including the source command in a `makefile`, or from the command line:

```
$ cd Scripts/
$ ls
create_dat_file.sh  setup.csh  setup.sh  vars.csh  vars.sh
$ source setup.sh
Arm Cycle Model SystemC setup completed.
Setup complete.
$
```

2. If you are using a version of the Cycle Model Studio runtime other than the one included in the CPAK, set CARBON_HOME. If you are using the version of the Cycle Model Studio runtime included in the CPAK, skip this step.

Build the CPAK executable

1. `cd` to the CPAK `Systems` directory.
2. Build the CPAK system executable (`system_test`) by entering `make`:

```
$ cd Systems
$ make
-std=c++11 -I/home/CPAKs/v11/R52-MP2-MC2-SysC-Vmainline-CMSmk.snapshot-MKmainline-
SOCDmainline/ARM/CycleModels/Runtime/cm_sysc/mainline -I/home/sandbox/CPAKs/v11/R52-MP2-
MC2-SysC-Vmainline-CMSmk.snapshot-MKmainline-SOCDmainline/ARM/CycleModels/Runtime/cm_sysc/
mainline/include
.
.
.
$
```

Next steps

Proceed to *2.4 Run the CPAK* on page 2-17.

Related information

- *Chapter 5 Troubleshooting* on page 5-33

2.4 Run the CPAK

Simulate with an application included with the CPAK.

Before you begin

- Build the CPAK executable as described in *2.3 Build the CPAK* on page 2-16.
- Ensure that the environment variable `ARMLMD_LICENSE_FILE` is set to your license server; for example, `export ARMLMD_LICENSE_FILE=port@host`.

Set runtime environment variables and run the simulation

1. Source the setup script to set environment variables that are required at runtime:

```
$ source ../Scripts/setup.sh
Arm Cycle Model SystemC setup completed.
Setup complete.
$
```

2. From the `Systems` directory, simulate the CPAK application using the `system_test` test bench:

```
$ ./system_test -a ../Applications/hello_world/armcc/elf/test.elf
```

- TLM-based CPAKs require you to specify the application to run using the `-a` or `--application` argument.
- For pin-based CPAKs, do not use `-a` or `--application`. Run `./system_test` to run the default application. The CPAK `README.txt` file lists the default application for pin-level CPAKs; see the `Applications` directory for additional applications provided with your CPAK.

See *4.2 Application loading* on page 4-28 for more information about application loading.

Result

The test bench starts the simulation, and loads and runs the application specified by `system_test.cpp`. Following is sample output from a TLM-based Cortex-R52 CPAK:

```
$ ./system_test -a ../Applications/hello_world/armcc/elf/test.elf
Starting Simulation
[kite_tarmac] skipping configuration file: kite_tarmac.cfg
[kite_tarmac] (KITE.g_dcls_cpu_container.g_near_cpu_group[0].u_cpu.u_noram.u_follower)
detected capture module (enabled)
[M] Constructing kite follower
[kite_tarmac] (KITE.g_dcls_cpu_container.g_near_cpu_group[0].u_cpu.u_noram.u_follower)
created filter 'DECODE' (kite_tarmac_decode)
[kite_tarmac] (KITE.g_dcls_cpu_container.g_near_cpu_group[0].u_cpu.u_noram.u_follower)
sending captured stream to 'DECODE'
[kite_tarmac] (KITE.g_dcls_cpu_container.g_near_cpu_group[1].u_cpu.u_noram.u_follower)
detected capture module (enabled)
[M] Constructing kite follower
[kite_tarmac] (KITE.g_dcls_cpu_container.g_near_cpu_group[1].u_cpu.u_noram.u_follower)
created filter 'DECODE' (kite_tarmac_decode)
[kite_tarmac] (KITE.g_dcls_cpu_container.g_near_cpu_group[1].u_cpu.u_noram.u_follower)
sending captured stream to 'DECODE'
[kite_tarmac] (KITE.g_dcls_cpu_container.g_near_cpu_group[0].u_cpu.u_noram.u_follower)
detected capture module (enabled)
[M] Constructing kite follower
[kite_tarmac] (KITE.g_dcls_cpu_container.g_near_cpu_group[0].u_cpu.u_noram.u_follower)
created filter 'DECODE' (kite_tarmac_decode)
[kite_tarmac] (KITE.g_dcls_cpu_container.g_near_cpu_group[0].u_cpu.u_noram.u_follower)
sending captured stream to 'DECODE'
[kite_tarmac] (KITE.g_dcls_cpu_container.g_near_cpu_group[1].u_cpu.u_noram.u_follower)
detected capture module (enabled)
[M] Constructing kite follower
[kite_tarmac] (KITE.g_dcls_cpu_container.g_near_cpu_group[1].u_cpu.u_noram.u_follower)
created filter 'DECODE' (kite_tarmac_decode)
[kite_tarmac] (KITE.g_dcls_cpu_container.g_near_cpu_group[1].u_cpu.u_noram.u_follower)
sending captured stream to 'DECODE'
Starting Sim
UART0: Hello World!<0a>
UART1: Hello World!<0a>
UART2: Hello World!<0a>
UART3: Hello World!<0a>
UART0: My name is Kite<0a>
UART1: My name is Kite<0a>
UART2: My name is Kite<0a>
UART3: My name is Kite<0a>
```

```
UART0: I wish you a great day<0a>
UART1: I wish you a great day<0a>
UART2: I wish you a great day<0a>
UART3: I wish you a great day<0a>
UART0: ** TEST PASSED OK **<0a>
UART1: ** TEST PASSED OK **<0a>
UART2: ** TEST PASSED OK **<0a>
UART3: ** TEST PASSED OK **<0a>
UART0: <04>
UART1: <04>
.                                                                                .
.
.
$
```

The simulation results show that the Hello World application ran successfully the four Cortex-R52 cores (output is sent through the UART model attached to each CPU). Simulation results also include performance monitoring data for each CPU (not shown in the example).

Next steps

When your CPAK simulates properly, see *2.5 Next steps* on page 2-19.

Related information

- *Chapter 5 Troubleshooting* on page 5-33
- *4.2 Application loading* on page 4-28

2.5 Next steps

When your default CPAK is simulating properly, learn about and change its operation.

Review the processor model guide

The *SystemC Cycle Model User Guide* for the processor model in your CPAK describes the functionality of the model compared to the hardware.

The model guide includes instructions for viewing model parameters, enabling waveform dumping, resetting the model, and connecting to a debugger.

View the list of processor model parameters

The set of available parameters varies, depending on the type and functionality of the processor model. To list model parameters:

- Enter `./system_test --list-params`.
- View the *`model`*`_params.cfg` file located in the directory *`CPAK`*`/MODELS/`*`CPU`*`/gcc`*`version`*`/SystemC`.

Modify the CPAK

You can change the CPAK to include different or additional models, make changes to the default application, or run the CPAK with a different application. See *Chapter 4 Modifying CPAKs* on page 4-23.

Related information

- *SystemC Cycle Model User Guide* for the CPU in your CPAK. This guide is located in `MODELS/`*`CPU`*`/gcc`*`version`*.
- *Arm® Development Studio Getting Started Guide* (101469).
- *Arm® Development Studio User Guide* (101470).

Chapter 3
Introduction to the makefiles

This chapter summarizes the makefiles that are included in CPAKs and describes the available targets.

It contains the following section:

3.1 Makefiles included in the CPAK

This section explains what you need to know about makefiles included in the CPAK.

The CPAK directory includes the following `makefiles`:

/Systems makefile

The `Makefile` you will use most frequently when working with CPAKs is the `Systems/Makefile`. This is the command script for your build infrastructure. When you make a change to the CPAK, such as adding a model to your design or changing one model for another, rebuild the CPAK using this `Makefile`.

See *3.1.1 Make options for the CPAK makefile and Systems/makefile* on page 3-21 for CPAK build, run, and simulation options.

CPAK makefile

You can use the `Makefile` at the top level of the CPAK to execute the `Systems/Makefile`. This top-level `Makefile` sets certain environment variables.

/MODELS makefiles

Each model in a CPAK system includes its own makefiles, located in `MODELS/`*`CPU_name/gcc_version/`*`SystemC/makefile`.

Note

Do not modify the makefiles in the `MODELS` directories. If you are adding or replacing a model in the CPAK, see *Chapter 4 Modifying CPAKs* on page 4-23.

/Applications makefile

The `Applications/Makefile` defines the application that is launched by default when you run the CPAK simulation. Modify this `Makefile` if you want to simulate using a different application. Do not modify the initialization code contained in the `Applications/Makefile`.

This section contains the following subsection:

3.1.1 Make options for the CPAK makefile and Systems/makefile

This section describes the available build, run, and simulation targets; targets that are available only in the top-level CPAK `Makefile` are noted.

Use the `Systems/Makefile` to rebuild the reference system after making modifications. This `makefile` includes the following available targets:

`make all`
Builds the system.

`make app-setup [ APP=path ]`
Sets up the application that the system uses during simulation. `path` is the path to the compiled application (`.elf`) file. This option is available only in the top-level CPAK `makefile`, not the `Systems/makefile`. See *4.2 Application loading* on page 4-28 for information about application loading.

`make clean`
Removes all the binaries and object files.

`make help`
Prints all available targets. This option is available only in the top-level CPAK `makefile`, not the `Systems/makefile`.

`make req`

Prints out all the components and dependencies for the CPAK.

`make run [ APP=path ] [ RUNLOG=file_name ]`

Runs the simulation. `path` is the path to the compiled application (`.elf`) file and `file_name` is the name of the file for log output. If `APP` is not specified, it runs the default application or the one that was last set up using `app-setup`.

Note

Run this command only with TLM-based CPAKs, not with pin-based CPAKs. This command uses the `-a` flag as a means of passing in the application, and pin-based CPAKs do not accept the`-a` flag. See *4.2 Application loading* on page 4-28 for more information.

`make system`

Uses the CPAK test bench (`system_test.cpp`) and the library files to generate the `system_test` binary.

Chapter 4
Modifying CPAKs

This chapter describes modifications you can make to SystemC CPAKs and how to rebuild the CPAK.

It contains the following sections:

4.1 CPAK modification use cases

This section provides use cases and instructions for CPAK modifications.

Prerequisites

- Before making changes to a CPAK, validate the operation of its default configuration as described in *Chapter 2 Building and running the default CPAK* on page 2-13.
- Review the *Cycle Model User Guide* for your IP for information about using the Cycle Models Configuration Tool. The Cycle Models Configuration Tool is a command-line utility included with the SystemC Cycle Model Runtime. This tool simplifies CPAK modifications by extracting required build and link options for models in a system, and flagging incompatibilities that may be present.

This section contains the following subsections:

4.1.1 Internal changes to a model in the default CPAK

This use case describes making a change to a model that exists in the CPAK.

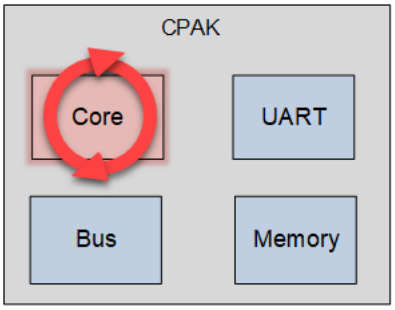

Figure 4-1 Change to existing model

You can make changes in the source files for models included in the default CPAK. For example, you might drive an additional input by commenting out a port binding in the CPAK `MODELS/model/gcc_version/SystemC/modelResetImp.h` file. In this case, ensure that you also drive the value of the port.

1. Make your modifications in `MODELS/model/gcc_version/SystemC/modelResetImp.h`.
2. Rebuild the CPAK using the `Systems/makefile`.

   See the *SystemC Cycle Model User Guide* for your IP for information about binding or tying ports.

Related information

- *SystemC Cycle Model User Guide* for the CPU in your CPAK. This guide is located in `MODELS/CPU/gccversion`

4.1.2 Changing the CPU used in the CPAK

This use case describes replacing the CPU in a CPAK.

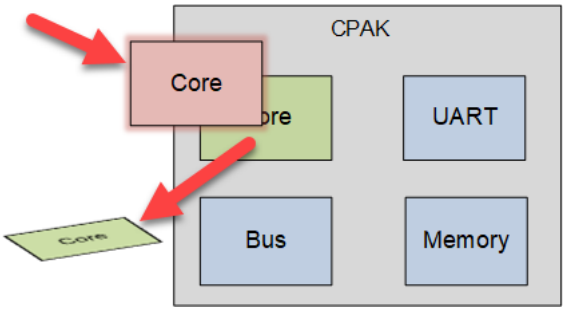

Figure 4-2 Changing the CPU

You can replace the CPU included in the CPAK by default with:

- a CPU of the same IP type, but with a different build configuration.
- a CPU of a different IP type.

To do so:

1. Build the new or revised CPU on IP Exchange.
2. Replace the existing CPU in the CPAK `MODELS` directory.
3. If the new CPU is of a different IP type than the original CPU, update the `system_test.cpp` test bench to reference the header files, ports, and bindings of the new CPU.
4. In the `Systems/makefile`, update the `COMP_NAMES` variable and the model directory to reflect the new model. To get the exact component name, run the Cycle Models Configuration tool with the `--list` argument and look at the `Component Type: model:` section. For example, in the following output, `CortexR52` is the string to use in the `COMP_NAMES` variable:

```
$ cm_config --list
Component Type: model:
CortexR52                mainline        /home/CPAKs/R52-MP2-MC2-SysC/MODELS/
CortexR52_2CPU/gcc640/SystemC/.data/CortexR52.xml
CortexR52                mainline        /home/CPAKs/R52-MP2-MC2-SysC/MODELS/
CortexR52_2CPU/gcc483/SystemC/.data/CortexR52.xml
cm_sysc_models           mainline        /home/CPAKs/R52-MP2-MC2-SysC/ARM/CycleModels/
Runtime/cm_sysc/mainline/etc/.data/cm_sysc_models.xml
Component Type: runtime:
.
.
.
```

   For more information about the Cycle Models Configuration Tool, see the *SystemC Cycle Model User Guide* for the CPU model included in your CPAK.
5. Rebuild the CPAK using the `Systems/makefile`.

Related information

- *SystemC Cycle Model User Guide* for the CPU in your CPAK. This guide is located in `MODELS/CPU/gccversion`

4.1.3 Changes to the composition of the CPAK

This use case describes changes to the models that make up a CPAK.

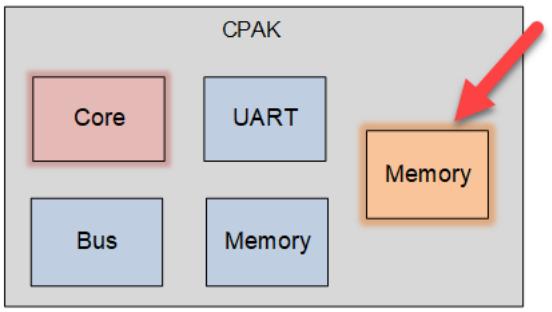

Figure 4-3 Adding a flash memory model to an existing CPAK

This section describes adding models to, or removing models from the CPAK platform; for example, adding a flash memory to the existing system.

1. If you are adding a model to the CPAK, add its directory to the `CPAK/MODELS` directory.
2. If you have removed a model from the CPAK, remove its corresponding directory from `CPAK/MODELS`.
3. Update `Systems/system_test.cpp`. This file defines the models included in the CPAK and their port bindings.
4. If the name of the model directory has changed, further modifications to the `Systems/Makefile` are required. Update the `COMP_NAMES` variable to include models and directories you are adding, and exclude models and directories you are replacing. To get the exact component name, run the Cycle Models Configuration tool with the `--list` argument and look at the `Component Type: model:` section. For example, in the following output, `CortexR52` is the string to use in the `COMP_NAMES` variable:

```
$ cm_config --list
Component Type: model:
CortexR52                mainline        /home/CPAKs/R52-MP2-MC2-SysC/MODELS/
CortexR52_2CPU/gcc640/SystemC/.data/CortexR52.xml
CortexR52                mainline        /home/CPAKs/R52-MP2-MC2-SysC/MODELS/
CortexR52_2CPU/gcc483/SystemC/.data/CortexR52.xml
cm_sysc_models           mainline        /home/CPAKs/R52-MP2-MC2-SysC/ARM/CycleModels/
Runtime/cm_sysc/mainline/etc/.data/cm_sysc_models.xml
.
.
.
```

5. Rebuild the CPAK using the `Systems/makefile`.

Related information

- *SystemC Cycle Model User Guide* for the CPU in your CPAK. This guide is located in `MODELS/CPU/gccversion`

4.1.4 Creating custom systems that include Arm® models

This use case describes adding an Arm model to a custom system.

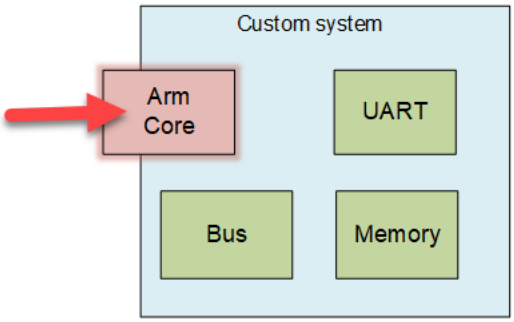

Figure 4-4 Adding an Arm CPU to a custom system

You can build your own, custom SystemC platform that includes a Cycle Model downloaded from Arm IP Exchange. Use the Cycle Models Configuration Tool, included in the SystemC Cycle Models Runtime, to extract its build options. See the *SystemC Cycle Model User Guide* for your IP for more information about the Cycle Models Configuration tool.

Contact Arm Support for more information.

Related information

- *SystemC Cycle Model User Guide* for the CPU in your CPAK. This guide is located in `MODELS/CPU/gccversion`

4.2 Application loading

This section describes what you need to know about loading applications for CPAK simulations.

Applications included with your CPAK are found in the CPAK `Applications` directory. Pin-level CPAKs and TLM-based CPAKs handle application loading differently.

Pin-level CPAK application loading

CPAKs implemented through pin-level connections use the memory-loading capabilities of the memory component included in the CPAK. The application to be loaded is determined by the contents of the `.hex` files in the CPAK `Systems` directory. These hex files are created using the `create_dat_file.sh` script located in the `Scripts` directory. The application is loaded into the CPAK upon initialization of the memory.

Alternate applications must be in `.elf` file format. To change the application used by the CPAK:

1. Run the `create_dat_file.sh` script on the new `.elf` file to create the required hex files.
2. Run `system_test`.

See the `README.txt` file for more information about the application run by default.

Note

Do not use the `-a` or `--application` command line arguments to specify applications for CPAKs with pin-level models. These arguments have no effect on pin-level CPAKs, and result in multiple warnings.

TLM-based CPAK application loading

The information in this section applies to TLM-based CPAKs that support debugging. These CPAKs use the standard SystemC framework to determine which application to load. This means that you must specify which application to load.

Specify the application on the command line using the `-a` or `--application` argument. For example:

```
$ ./system_test -a test.elf -S
```

4.3 CPAK test bench modifications

Each CPAK has its own test bench called `system_test.cpp`, which is located in the `Systems` directory.

`system_test.cpp`:

- Instantiates the models
- Defines the connections between models
- Initializes model parameters
- Provides simulation controls (start, stop, run, specific number of cycles, etc.)

To make changes to the CPAK system, alter the CPAK test bench. After altering the test bench, recompile the system. Models are recompiled automatically as part of the system-level recompile.

This section contains the following subsections:

- *4.3.1 Modifying the test bench for pin-level models* on page 4-29.
- *4.3.2 Modifying the test bench for TLM models* on page 4-31.

4.3.1 Modifying the test bench for pin-level models

This section describes the areas of the CPAK test bench you might want to change.

Note

See *4.2 Application loading* on page 4-28 for information about how pin-level CPAKs handle application loading.

Required includes

The test bench contains the SystemC wrapper files for any models in the system, the corresponding reset module file for each model (for pin-level models only), and `includes` required by the Fast Models runtime. Ensure you add these files for any new models added to your system. Here is an example of the `includes` section of a CPAK test bench:

```
// Include the systemc wrapper files for the models
#include "libCortexR8.systemc.h"            // CortexR8 CPU
#include "libNIC400.systemc.h"              // NIC400 Interconnect
#include "libBP140.systemc.h"               // BP140 Memory
#include "libBP140_TrickBox.systemc.h"      // BP140_TrickBox UART

// Include the reset modules for the above models (pin-level models only)
#include "CortexR8ResetModule.h"
#include "NIC400ResetModule.h"
#include "BP140ResetModule.h"
#include "BP140_TrickBoxResetModule.h"
#include "../perf_common.h"
#include <iostream>
#include <time.h>

// These includes are need by the SCX FastModel Runtime
#include <scx/scx.h>
#include <scx/scx_signal_sizer.h>
#include <scx_simcontroller.cpp>
#include <scx_scheduler_mapping.cpp>
#include <scx_report_handler.cpp>
```

Initialization of the simulation environment and model instantiation

The `sc_main()` section of the test bench must first initialize the SCX simulation environment. It must state clocking specifications, then instantiate all IP and reset models (for pin-level models only) for any models included in the system. For example:

```
int sc_main(int argc, char *argv[])
{
  // Debug initialization
  scx::scx_initialize("R8-SysC-Debug");

  // If you want to see messages about 'port not bound' change SC_DO_NOTHING to SC_DISPLAY.
```

```
  // If you want it to abort on a 'port not bound' error comment out the line below.
  sc_report_handler::set_actions(SC_ID_COMPLETE_BINDING_,SC_ERROR,SC_DO_NOTHING);

  // Clock Object
  sc_clock clk("clk", 1, SC_NS, 0.5);

// Instantiate IP
  CortexR8 core("cortexR8");
  NIC400 nic400("NIC400");
  BP140 bp140("BP140");
  BP140_TrickBox bp140_trickbox("BP140_TrickBox");
  ARM::Models::Cycle::ModelCortexR8::CortexR8ResetModule core_reset("core_reset");
  ARM::Models::Cycle::ModelNIC400::NIC400ResetModule nic400_reset("nic400_reset");
  ARM::Models::Cycle::ModelBP140::BP140ResetModule bp140_reset("bp140_reset");
  ARM::Models::Cycle::ModelBP140_TrickBox::BP140_TrickBoxResetModule
bp140_trickbox_reset("bp140_trickbox_reset");
```

Signal bindings

The test bench specifies all signal bindings, including those for reset modules.

- Declare bindings using an `sc_signal` call in the test bench file. The signal must be the same type and width as the two ports being connected. If the ports are the same type but different widths, use `scx_signal_sizer` instead of `sc_signal`. For example:

  ```
  scx::scx_signal_sizer<sc_uint<13>, sc_uint<16> >ARIDsignal1;
  core.ARIDM0.bind(ARIDsignal);
  nic400.arid_s_axi_64.bind(ARIDsignal);
  ```

- Signals must be bound to both ports. For example:

  ```
  sc_signal(bool) signal1;
  Inst1.port1.bind(signal1);
  sc_signal(bool) signal2;
  Inst2.port1.bind(signal2);
  ```

Clock bindings

All models must be bound to the system clock; for example:

```
// Bind all the models to the system (cpu) clock
core.CLKIN.bind(cpu_clk);
mem.ACLK.bind(cpu_clk);
bus.mainclk.bind(cpu_clk);
uart.ACLK.bind(cpu_clk);
core_reset.clk.bind(cpu_clk);
bus_reset.clk.bind(cpu_clk);
bp140_reset.clk.bind(cpu_clk);
bp140_trickbox_reset.clk.bind(cpu_clk);
```

Functions to generate performance data

The following example shows the use of SCX API functions to specify PMU and Tarmac output. See the *SystemC Cycle Model User Guide* for the CPU in your CPAK for more information:

```
scx::scx_set_parameter("cortexR8_core.PMU_ENABLED",true);
scx::scx_set_parameter("cortexR8_core.TARMAC_ENABLED",true);
```

Parsing of command line arguments

The test bench calls the SCX `scx_parse_and_configure()` function to parse any command line arguments used by the SCX runtime:

```
scx::scx_parse_and_configure(argc, argv);
```

Simulation call

To simulate the system, the test bench calls `sc_start()`:

```
sc_start();
```

Specify all includes, initializations, bindings, functions, and command line options before `sc_start()`.

Related information

- *SystemC Cycle Model User Guide* for the CPU in your CPAK. This guide is located in `MODELS/CPU/gccversion`.

4.3.2 Modifying the test bench for TLM models

The `sc_main()` function in the test bench has the same basic flow for both pin-level and TLM models. One difference is that the TLM models are connected using TLM sockets rather than pins. This section describes the TLM-specific instructions.

Note

See the file `libcomponent.tlm.h` in your installation directory for socket names.

Note

See *4.2 Application loading* on page 4-28 for information about how TLM-based CPAKs handle application loading.

Required includes

The SystemC pin-level models are wrapped with TLM functionality. The test bench includes the SystemC wrapper files for any models included in the system, and `includes` required by the Fast Models runtime. Ensure you add these files for any new models added to your system. Here is an example of the `includes` section of a CPAK test bench:

```
// Include the systemc wrapper files for the models

#include "models/SimpleMem.h"
#include "models/SimpleFlash.h"
#include "models/SimpleFlashImp.h"
#include "models/RAMBlock.h"
#include "models/SimpleBus.h"
#include "models/BasicUART.h"
#include "models/ElfLoader.h"
#include "libCORTEXR8.tlm.h"                 // CPU
#include <tlm_utils/simple_initiator_socket.h>

#include <iostream>

// These includes are need by the SCX FastModel Runtime
#include <scx/scx.h>
#include <scx_simcontroller.cpp>
#include <scx_scheduler_mapping.cpp>
#include <scx_report_handler.cpp>
```

Initialization of the simulation environment and model instantiation

The `sc_main()` section of the test bench must first initialize the SCX simulation environment. It states clocking specifications, then instantiates any models included in the system. Ensure you instantiate any new models added to your system. For example:

```
int sc_main(int argc, char *argv[])
{
// Debug initialization
  scx::scx_initialize("R8-SysC");

  // If you want to see messages about 'port not bound' change SC_DO_NOTHING to SC_DISPLAY.
  // If you want it to abort on a 'port not bound' error comment out the line below.
  sc_report_handler::set_actions(SC_ID_COMPLETE_BINDING_,SC_ERROR,SC_DO_NOTHING);

  // Clock Object
  sc_clock cpu_clk("clk", 1, SC_NS, 0.5);

  ARM::Models::Cycle::ModelCortexR8::CortexR8Imp core("CortexR8");

  // Main Memory
```

```
ARM::Models::RAMBlock ram_block;
ARM::Models::SimpleMemConfig simple_mem_params;
simple_mem_params.delay = 1;
simple_mem_params.ram_block = &ram_block;
simple_mem_params.busWidthBits = 64;

ARM::Models::SimpleMem simple_mem("RAM", simple_mem_params);
ARM::Models::BasicUART uart("UART", std::cout, "UART: ");

// BUS & its Mappings
ARM::Models::SimpleBus bus("Interconnect", 1, 1, 64);
bus.addMap(0, 0, 0xFFFFFFFF); // Main Memory
```

Port bindings

The test bench specifies all TLM port bindings. Signal bindings required for the system are also specified in this area. Specify any additional TLM port bindings and signal bindings in this area:

```
// Core Main Memory Port Bindings
        core.iSkt_AXI3_Master_PORT0->bind(bus.tSkt);
        core.directIskt_AXI3_Master_PORT0.bind(simple_mem.directTskt);

// Core Low Latency Peripheral Port Bindings
        core.iSkt_AXI3_Master_PERI->bind(uart.tSkt);
        core.directIskt_AXI3_Master_PERI.bind(uart.directTskt);

// Bus iSkt[0] connected to Main Memory
        bus.iSkt[0]->bind(simple_mem.tSkt);

// Clock
        core.clk(cpu_clk);
        simple_mem.clock.bind(cpu_clk);
        uart.clock.bind(cpu_clk);
        bus.clock.bind(cpu_clk);
```

For information about signal bindings, clock bindings, performance data generation, command line arguments, and simulation calls, use the instructions in *4.3.1 Modifying the test bench for pin-level models* on page 4-29.

Chapter 5
Troubleshooting

This chapter provides solutions for problems that may occur when working with CPAKs.

It contains the following sections:

5.1 carbon_sc_multiwrite_signal.h build error

Incorrectly set `CARBON_HOME` environment variable results in fatal error.

Cause

When using a version of the Cycle Model Studio runtime other than the one included in the CPAK, the environment variable `CARBON_HOME` defines the runtime location. When unset (this is the default for the CPAK), the Cycle Model Studio runtime included in the CPAK is used.

If `CARBON_HOME` is set, it must be set to a Cycle Model Studio runtime installation, or full version of Cycle Model Studio, that is Version 11 or later. Otherwise, an error similar to the following may occur when building the CPAK:

```
../MODELS/CortexR52_2CPU/gcc483/SystemC/libCortexR52.systemc.h:19:48: fatal error: carbon/
carbon_sc_multiwrite_signal.h: No such file or directory
#include "carbon/carbon_sc_multiwrite_signal.h"

compilation terminated.
make: *** [system_test.o] Error 1
```

Solution

Either:

- Source the location of a Version 11.0 or later Cycle Model Studio runtime or Cycle Model Studio.
- Unset `CARBON_HOME` to use the version of the Cycle Model Studio runtime included in the CPAK.

5.2 Unrecognized command line option

Running an unsupported GCC version results in the build error `unrecognized command line option "-std=c++11"`.

Cause

Using an unsupported GCC version may result in a build error similar to the following:

```
cc1plus: error: unrecognized command line option "-std=c++11"
make: *** [system_test.o] Error 1
```

Solution

Check your GCC version against the supported versions listed in *1.2 System requirements and prerequisites* on page 1-11. In the following example, GCC 4.4.7 is unsupported and GCC 4.8.3 (a supported version) is sourced:

```
$ gcc --version
gcc (GCC) 4.4.7 20120313 (Red Hat 4.4.7-11)
Copyright (C) 2010 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

$ source /o/Linux64/etc/setup.sh
$ gcc --version
gcc (GCC) 4.8.3
Copyright (C) 2013 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

5.3 Licensing errors

A valid Arm Cycle Model runtime license is required to run CPAK simulations.

Cause

If a valid license is not available, an error similar to the following may occur when running the test bench simulation:

```
Checkout of CM_ARM_Runtime : No feature match. - checkout failed (nowait): Cannot find
license file.
The license files (or license server system network addresses) attempted are
listed below.  Use LM_LICENSE_FILE to use a different license file,
or contact your software provider for a license file.
Feature:       CM_ARM_Runtime
Filename:      /license
License path:  /license:
```

Solution

- Ensure that the environment variable `ARMLMD_LICENSE_FILE` is set to your license server; for example, `export ARMLMD_LICENSE_FILE=`*`port@host`*.
- Contact Arm Techical Support (support-esl@arm.com).

5.4 Bytes requested error when specifying application

`Number of bytes requested` error message occurs when starting a simulation and specifying an application.

Cause

When launching a simulation with a specified application, use the `-a` or `--application` flag only with TLM-based CPAKs. Using these flags with pin-level CPAKs results in an error similar to the following:

```
$ Number of bytes requested for write (1) does not match numbers of bytes reportedly
written(0)
```

Solution

Specify the application according to the instructions in *4.2 Application loading* on page 4-28.