Fast Models

Version 11.28

User Guide

Non-Confidential

Issue 00
100965_1128_00_en

Fast Models User Guide

This document (100965_1128_00_en) was issued on 2025-02-19. There might be a later issue at https://developer.arm.com/documentation/100965

The product version is 11.28.

See also: Proprietary Notice | Product and document information | Useful resources

Start Reading

If you prefer, you can skip to the start of the content.

Intended audience

This document is written for software developers using Fast Models to build and run custom platform models.

Inclusive language commitment

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

This document includes language that can be offensive. We will replace this language in a future issue of this document.

To report offensive language in this document, email terms@arm.com.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com.

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Contents

1. Introduction to Fast Models

This chapter provides a general introduction to Fast Models.

1.1 What is Fast Models?

The Fast Models product comprises a library of *Programmer's View* (PV) models and tools that enable partners to build, execute, and debug virtual platforms. Virtual platforms enable the development and validation of software without the need for target silicon. The same virtual platform can be used to represent the processor or processor subsystem in SoC validation.

Fast Models are delivered in two ways:

- As a portfolio of models of Arm® IP and tools to let you build a custom model of your exact system.
- As standalone models of complete Arm® platforms that run out-of-the-box to let you test your code on a generic system quickly. These pre-built platform models are built by Arm using Fast Models components and are called Fixed Virtual Platforms, or FVPs.

The benefits of using Fast Models include:

Develop code without hardware

Fast Models provides early access to Arm® IP, well ahead of silicon being available. Virtual platforms are suitable for OS bring-up and for driver, firmware, and application development. They provide an early development platform for new Arm® technology and accelerate time-to-market.

High performance

Fast Models uses *Code Translation* (CT) processor models, which translate Arm® instructions into the instruction set of the host dynamically, and cache translated blocks of code. This and other optimization techniques, for instance temporal decoupling and *Direct Memory Interface* (DMI), produce fast simulation speeds for generated platforms, between 20-200 MIPS on a typical workstation, enabling an OS to boot in tens of seconds.

Customize to model your exact system

Fast Models provides a portfolio of models that are flexible and can easily be customized using parameters to test different configurations. Using the System Canvas tool you can model your own IP and integrate it with existing model components.

You can also export components and subsystems from the Fast Models portfolio to SystemC for use in a SystemC environment. Such an exported component is called an *Exported Virtual Subsystem* (EVS). EVSs are compliant with SystemC TLM 2.0 specifications to provide compatibility with Accellera SystemC and a range of commercial simulation solutions.

Run standalone or debug using development tools

Generated platform models are equipped with *Component Architecture Debug Interface* (CADI). This allows them to be used standalone or with development tools such as Arm®

Development Studio or Arm® Keil® MDK, as well as providing an API for third party tool developers.

Test architecture compliance

Fast Models provides *Architecture Envelope Models* (AEMs) for Arm®v8-A, Arm®v9-A, Arm®v8-R, and Arm®v8-M. These are specialist architectural models that are used by Arm and by Arm® architecture licensees to validate that implementations are compliant with the architecture definition.

Trace and debug interfaces

Fast Models provides the *Model Trace Interface* (MTI) and CADI for trace and debug. These APIs enable you to write plug-ins to trace and debug software running on models. Fast Models also provides some pre-built MTI plug-ins, for example Tarmac Trace, that you can use to output trace information.

Build once, run anywhere

Since the same binary runs on the model, the target development hardware, and the final product, you only need to build it using the Arm® toolchain.

Host platform compatibility

Fast Models supports x86-64 host platforms running Linux or Microsoft Windows, and Arm® AArch64 hosts running Linux. For details, see 2.1 Requirements for Fast Models on page 17.

Related information

System Canvas GUI
LISA+ Language for Fast Models Reference Guide
Model Debugger

1.2 What does Fast Models consist of?

The Fast Models package contains the tools and model components that are needed to model a system. The tools and the portfolio of models are installed under separate directories, `FastModelsTools_n.n` and `FastModelsPortfolio_n.n` respectively, where `n.n` is the Fast Models version number.

Arm also supplies a wide range of pre-built *Fixed Virtual Platforms* (FVPs), including some free of charge FVPs, separately from the Fast Models package.

1.2.1 Fast Models tools

Fast Models tools enable you to create custom system models from the Fast Models portfolio of component models, and debug them.

System Generator or `simgen`

A backend tool that handles system model generation. System Generator can either be invoked from the System Canvas GUI, or by using the `simgen` command-line utility. System models that are created using System Generator can be used with other Arm® development tools, for example Arm® Development Studio or Model Debugger, or can be exported to SystemC for integration with proprietary models.

System Canvas or `sgcanvas`

A GUI design tool for developing new model components written in LISA+ and for creating and building system models. To launch System Canvas from the command line, type `sgcanvas`. The GUI displays the model as either LISA+ source code, or graphically, in a block diagram editor.

Iris Monitor

A browser-based GUI debugger for Fast Models that lets you debug and view the state of the Fast Models components in a system. It is the successor to Model Debugger.

Model Debugger

A symbolic debugger with a GUI that communicates with models using the Component Architecture Debug Interface (CADI). It enables you to launch a model or connect to a running model, and debug it. It has been replaced by Iris Monitor.

Model Shell

A command-line tool for launching simulations that are implemented as CADI libraries. It can also run a CADI debug server to enable CADI-enabled debuggers to connect to the model.

Note

Arm deprecates Model Shell. Instead, we recommend you use Integrated SIMulators (ISIMs), which are standalone executables that do not require Model Shell.

Related information

System Generator
System Canvas
Model Debugger
Model Shell

1.2.2 Fast Models portfolio

Fast Models portfolio is a library of component models of Arm® IP.

It includes the following:

- A collection of models and protocols, provided as LISA+ components. You can use them to create a system using the Fast Models tools. Ports and protocols are used for communication between components. Some models are of Arm® IP, while others are not. Examples of Arm® IP models include:
  - Processors, including models of all Arm® Cortex® and Neoverse™ processors, and architectural models, called AEMs.
  - Models of Arm® media IP such as GPUs, video processors, and display processors.

    - Peripherals, for instance Arm® CoreLink™ interconnect, interrupt controllers, and memory management units.

  Some models are abstract components that do not model specific Arm® IP, but are required by the software modeling environment. For example:

    - `PVBus` components to model bus communication between components.
    - Emulated I/O components to allow communication between the simulation and the host, such as a terminal, a visualization window, and an ethernet bridge.

- Platform model examples that show how to integrate the model components. They are supplied as project files, so must first be built using System Generator. Examples are provided for both standalone simulation and for SystemC export, and include:
    - Base Platform systems for Arm®v8-A, Arm®v8-R, and Arm®v9-A.
    - Systems based on Arm® Versatile™ Express development boards for Arm®v7-A and Arm®v7-R processors.
    - Systems based on MPS2 development boards for Arm®v6-M, Arm®v7-M, and Arm®v8-M processors.
- Accellera SystemC and TLM header files and libraries, which are required to build FVPs and the platform model examples.
- *Model Trace Interface* (MTI) plug-ins. MTI is the interface used by Fast Models to emit trace events during execution of a program, for example branches, exceptions, and cache hits and misses. Fast Models provides some pre-built MTI plug-ins that you can load into a model to capture trace data, without having to write your own plug-ins. For example:
    - `TarmacTrace` can trace all processor activity or a subset of it, for instance only branch instructions or memory accesses.
    - `GenericTrace` allows you to trace any of the MTI trace sources that the models can produce.

  Some trace plug-ins are provided in source form as programming examples.
- Some ELF images that you can run on models for evaluation purposes.
- Networking setup scripts to bridge network traffic from the simulation to the host machine's network.

1.2.3 Other Fast Models products

The following Fast Models products are available separately from the main Fast Models package:

Fixed Virtual Platforms (FVPs)

FVPs are models of Arm® platforms, including processors, memory, and peripherals. They are supplied as pre-built executables for Linux and Windows. Their composition is fixed, although you can configure their behavior using parameters.

Arm provides different types of FVP, based on the following platforms:

- Base Platform, for A-profile architectures.

- BaseR Platform for Arm®v8-R.
- Arm® Versatile™ Express development boards.
- Arm® MPS2 or Arm® MPS2+ platforms, for Cortex®-M series processors.

FVPs are available for all Cortex®-A, Cortex®-R, and Cortex®-M processors, and they support the CADI, MTI, and Iris interfaces, so can be used for debugging and for trace output.

The most commonly-used FVPs are supplied in a single package which is downloadable from Arm Developer, see Fixed Virtual Platforms.

Arm provides validated Linux and Android deliverables for the AEM Base Platform FVP and for the Foundation Platform. These are available on the Arm Development Platforms wiki on Arm Community.

To get started with Linux on Arm®v8-A FVPs, see FVPs on Arm Community.

Foundation Platform

A simple FVP that includes an AEM that supports both Arm®v8-A and Armv9-A architectures, that is suitable for running bare-metal applications and for booting Linux. It is available for Linux hosts only and can be downloaded free of charge from Fixed Virtual Platforms on Arm Developer. Registration and login are required.

System Guidance platforms

These FVPs include documentation to guide SoC design and a reference software stack that is validated on the FVP. They are also known as Reference Design FVPs. For more information, see Reference Design on Arm Developer.

Third party IP

A package that contains third party add-ons for Fast Models. These include some additional ELF images, including Dhrystone.

1.3 Fast Models glossary

This glossary defines some Arm-specific technical terms and acronyms that are used in the Fast Models documentation.

AMBA-PV

A set of classes and interfaces that model AMBA® buses. They are implemented as an extension to the TLM v2.0 standard.

See AMBA-PV extensions.

Architecture Envelope Model (AEM)

A fully-configurable, generic model of an Arm® architecture. It aims to expose software bugs by modeling the range of behavior that the architecture allows. Fast Models provides AEMs for Arm®v8-A, Arm®v8-R, Arm®v8-M, and Arm®v9-A. For example, AEMvACT models both Arm®v8-A and Arm®v9-A.

Auto-bridging

A Fast Models feature that SimGen uses to automatically convert between LISA+ protocols and their SystemC equivalents. It helps to automate the generation of SystemC wrappers for LISA+ subsystem models.

See 5.2 Auto-bridging on page 37.

Note

Auto-bridging is deprecated in Fast Models 11.27 and will be removed in a future release.

Base Platform

A range of example Fast Models platforms that can boot a full OS, including Linux and Android images that can be downloaded from Linaro. Base Platforms support the Arm®v8 and Arm®v9 architectures, replacing VE platforms, which support Arm®v7.

See Base Platform.

Component Architecture Debug Interface (CADI)

A legacy C++ debug interface that enables run control and inspection of models. It has been replaced by Iris.

See Introduction to the Component Architecture Debug Interface.

Code Translation (CT)

A technique that processor models use to enable fast execution of code. CT models translate code dynamically and cache translated code sequences to achieve fast simulation speeds.

Exported Virtual Subsystem (EVS)

A Fast Models component or subsystem that is built by SimGen from a LISA+ model description as a SystemC shared library. The library can be incorporated into a SystemC platform.

See 5.1 About SystemC Export with Multiple Instantiation on page 36.

Fast Models

High performance software models of components of Arm® SoCs, for example processors, system IP, and bus infrastructure. Fast Models components can be connected together and configured to build a platform model, for example an FVP.

Fixed Virtual Platform (FVP)

A pre-built platform model composed of Fast Models components. FVPs enable applications and operating systems to be written and debugged without the need for real hardware. FVPs are also referred to as *Fixed Virtual Prototypes*.

See Introduction to FVPs.

Foundation Model

See *Foundation Platform*.

Foundation Platform

A freely available, easy-to-use FVP for application developers that supports the Arm®v8-A and Arm®v9-A architectures. It can be downloaded from Fixed Virtual Platforms on Arm Developer, registration and login are required.

IMP DEF

Used in register descriptions in the *Fast Models Reference Guide* to indicate behavior that the architecture does not define. Short for *Implementation Defined*.

Integrated Simulator (ISIM)

A simulation executable generated by SimGen from a LISA+ model description. SimGen links the executable against a SystemC shared library.

See Building a SystemC ISIM target.

Iris

An interface for debugging and tracing model behavior. Iris is the replacement for CADI.

See Iris User Guide

Language for Instruction Set Architectures (LISA, LISA+)

LISA is a language that describes instruction set architectures. LISA+ is an extended form of LISA that supports peripheral modeling. LISA+ is used for creating and connecting model components. The Fast Models documentation does not always distinguish between the two terms, and sometimes uses *LISA* to mean both.

See LISA+ Language for Fast Models Reference Guide.

Microcontroller Prototyping System (MPS2)

Arm® Versatile™ Express V2M-MPS2 and V2M-MPS2+ are motherboards that enable software prototyping and development for Cortex®-M processors. The MPS2 FVP models a subset of the functionality of this hardware.

See MPS2.

Model Debugger

A Fast Models debugger that enables you to execute, connect to, and debug any CADI-compliant model. You can run Model Debugger using a GUI or from the command line.

See Model Debugger.

Model Shell

A command-line utility for configuring and running CADI-compliant models. Arm deprecates Model Shell. Use ISIM executables instead.

See Model Shell.

Model Trace Interface (MTI)

A trace interface that is used by Fast Models to expose real-time information from the model.

See Model Trace Interface Reference Manual.

Platform Model

A model of a development platform, for example an FVP.

Programmers' View (PV) Model

A high performance, functionally accurate model of a hardware platform. It can be used for booting an operating system and executing software, but not to provide hardware-accurate timing information.

PVBus

An abstract, programmers view model of the communication between components. *Bus masters* generate transactions over the PVBus and *bus slaves* fulfill them.

See PVBus components.

Real-Time System Model (RTSM)

An obsolete term for *Fixed Virtual Platform (FVP)*.

SimGen

An alternative name for *System Generator*.

Synchronous CADI (SCADI)

An interface that provides a subset of CADI functions to synchronously read and write registers and memory. You can only call SCADI functions from the model thread itself, rather than from a debugger thread. SCADI is typically used from within MTI or by peripheral components to access the model state and to perform run control.

See SCADI.

syncLevel

Each processor model has a syncLevel with four possible values. It determines when a synchronous watchpoint or an external peripheral breakpoint can stop the model, and the accuracy of the model state when it is stopped.

See syncLevel definitions.

System Canvas

An application that enables you to manage and build model systems using components. It has a block diagram editor for adding and connecting model components and setting parameters.

See System Canvas.

SystemC Virtual Platform (SVP)

A Fast Models platform that consists of components and subsystems that are individually exported to SystemC as a collection of multiple EVSs.

System Generator

A utility that generates a platform model by processing LISA files. You can run System Generator from the command line by invoking `simgen`, or from the System Canvas GUI. It is also referred to as SimGen.

See System Generator.

System Model

An alternative term for *Platform Model*.

Tarmac trace

A format for tracing the execution on code on an Arm® core. Fast Models includes a TarmacTrace plug-in that can consume and display tarmac trace.

See TarmacTrace.

Timing Annotation

A Fast Models feature that adds delays to transactions in the platform, enabling timing configuration for various operations, for instance branch prediction. It also supports setting Cycles Per Instruction (CPI) values not equal to one.

See 6. Timing annotation on page 100.

Versatile™ Express (VE)

A family of Arm® hardware development boards targeting the Arm®v7 architecture. The term is abbreviated to *VE* when used in names of FVPs, for example, FVP_VE_Cortex-A5x1. For Arm®v8 and Arm®v9 support, VE platform models have been replaced by Base Platform models.

Related information

Arm Glossary

1.4 Security assumptions for Fast Models

The threat model for Fast Models is very permissive. We make the following assumptions about how you use Fast Models with respect to security.

- We expect you not to run Fast Models with elevated privileges, for example as root on Linux or administrator on Windows.
- If you build your own virtual platform using the Fast Models portfolio and distribute it within your own environment or to third parties, the integrity of the platform is your responsibility. For example it is up to you to ensure that the platform is not inadvertently modified.
- Fast Models does not provide a sandbox environment. We expect code running on the model to be able to trivially interact with the host environment for ease of use or during development, for example through semihosting. It is your responsibility to verify the integrity and validity of any code you run on the model.
- Similarly, plug-ins that you create are not isolated in any way. It is your responsibility to verify the integrity and validity of plug-ins that you use with a model, or any other code that you integrate with the model.

2. Installing Fast Models

This chapter describes the system requirements for Fast Models and how to install and uninstall Fast Models.

2.1 Requirements for Fast Models

This section describes the host hardware and software requirements for using Fast Models.

Host machine

Architecture

x86-64 and Arm® AArch64 host platforms are supported.

Minimum specification

At least 2GB RAM, preferably 4GB.

2GHz Intel Core2Duo, or similar, that supports the MMX, SSE, SSE2, SSE3, and SSSE3 instruction sets.

Recommended specification

At least double the RAM of the platform you intend to simulate. For example, a simulated platform containing 8GB of DRAM should be run on a 16GB host machine.

Fast Models and associated FVPs benefit most from high single-threaded performance. For example, a high frequency (4-5GHz) Intel Core i9 or i7 or AMD Ryzen 9 or 7 host CPU gives a significant improvement, between 30-60%, over Intel Xeon cores (2-3 GHz).

Linux

Operating system

Red Hat Enterprise Linux 7 or 8 (for 64-bit architectures), Ubuntu 20.04 or 22.04 *Long Term Support* (LTS).

Shell

A shell compatible with `sh`, such as `bash` or `tcsh`.

Compiler

GCC 9.3.0 (x86-64 and Arm® AArch64 hosts), GCC 10.3.0 (x86-64 and Arm® AArch64 hosts), GCC 12.3.0 (x86-64 and Arm® AArch64 hosts).
The following table shows the supported GCC versions on a Linux x86 host:

Table 2-1: Supported GCC versions on Linux (x86 host)

OS	GCC versions supported
RHEL 7	GCC 9.3.0
RHEL 8	GCC 9.3.0, GCC 10.3.0, GCC 12.3.0
Ubuntu 20.04 LTS	GCC 9.3.0

OS	GCC versions supported
Ubuntu 22.04 LTS	GCC 10.3.0

The following table shows the supported GCC versions on a Linux Arm® AArch64 host:

Table 2-2: Supported GCC versions on Linux (Arm® AArch64 host)

OS	GCC versions supported
RHEL 8	GCC 9.3.0, GCC 10.3.0, GCC 12.3.0
Ubuntu 20.04 LTS	GCC 9.3.0
Ubuntu 22.04 LTS	GCC 10.3.0

Note

For full compatibility, it is highly recommended that all code that links against the Fast Models is compiled with C++14 support enabled. There are no known issues when linking non-C++14 code with the Fast Models. However, the compiler does not guarantee that the ABI is the same for both types of code. Compiling models with C++14 support disabled might cause data corruption or other issues when using them.

The following combinations of GCC and GNU binutils were used to build Fast Models libraries:

Table 2-3: GCC and binutils versions

GCC version	GNU binutils version
9.3.0	2.32
10.3.0	2.38
12.3.0	2.38

PDF Reader

Adobe does not support Adobe Reader on Linux. Arm recommends system provided equivalents, such as Evince, instead.

Microsoft Windows

Operating system

Microsoft Windows 10 64-bit.

Compiler

Microsoft Visual Studio 2019 version 16.11 or later.
The following Visual Studio components are required:

- Visual C++ ATL for x86 and x64.
- Windows SDK version 10.0.16299.0 or later.

PDF Reader

Adobe Reader 8 or higher.

Note

- To build models using Visual Studio requires you to install the Visual Studio redistributable package which contains the runtime libraries for Visual Studio. Fast Models does not provide these libraries. Download the libraries for Visual Studio free of charge from Microsoft, from https://www.microsoft.com/en-gb/download/details.aspx?id=48145.
- On Windows, Fast Models libraries are built with one of the following MSVC compiler options:
  - `/MD` for release builds
  - `/MDd` for debug builds

  Any objects or libraries that link against the Fast Models libraries must also be built with the same `/MD` or `/MDd` option.
- Fast Models does not support Express or Community editions of Visual Studio.

Licensing

Fast Models use either User-Based Licensing or FlexNet Licensing:

- Arm user-based licensing is only available to customers with a user-based licensing license. Documentation for user-based licensing is available at https://lm.arm.com. For assistance with user-based licensing issues, visit https://developer.arm.com/support and open a support case.
- For FlexNet Publisher node-locked or floating licensing, the latest version of the FlexNet software is available for download from License Server Management Software.

  Note

  - Set up a single `armlmd` license server. Spreading Fast Models license features over servers can cause feature denials.
  - To run `armlmd` and `lmgrd` on Linux, install these libraries:

    Red Hat

    `lsb`, `lsb-linux`

    Ubuntu

    `lsb`
  - If you use Microsoft Windows *Remote Desktop* (RDP) to access System Canvas or a simulation that it generated, your license type can restrict you:
    - Floating licenses require a license server, and have no RDP restrictions. Arm issues them on purchase.
    - Node locked licenses apply to specific workstations. Existing node locked licenses and evaluation licenses do not support running the product over RDP connections. Visit https://developer.arm.com/support for more information.

2.2 Installation

This section describes how to install the Fast Models package.

Before you begin

The Windows and Linux installers for Fast Models might be vulnerable to some permission-based attacks. For more information, see Installer vulnerabilities CVE-2022-43701, CVE-2022-43702, and CVE-2022-43703.

Procedure

1. Unpack the installation package, if necessary, and execute `./setup.sh` on Linux or run `Setup.exe` as administrator on Windows.
   To install the package without the need for user interaction or a GUI, use the `--i-accept-the-end-user-license-agreement` command-line option.

   Note

   Using this option means you have read and accepted the terms and conditions of the End User License Agreement for the product and version installed.

   This option can be followed by either or both of these options:

   --basepath <*path*>
   Set the base directory for the installation.

   --licpath <*path*>
   Set the location of the license file.

   On Linux, `setup.sh` displays a list of any missing prerequisite packages that must be installed before installation can continue.

   On Windows, the installer automatically defines the following environment variables:

   MAXCORE_HOME
   Points to the installation directory of the Fast Models Tools.

   PVLIB_HOME
   Points to the installation directory of the Fast Models Portfolio.

   SYSTEMC_HOME
   Points to the Accellera SystemC library installation directory. This package includes the SystemC and TLM header files and libraries that you need to build an EVS, FVP, or SVP.

   IRIS_HOME
   Points to the `%PVLIB_HOME%\Iris` directory, which contains Iris headers, examples, and the `iris.debug` Python module.

   PYTHONPATH
   To use the `iris.debug` Python module, the `PYTHONPATH` environment variable is updated to include `%IRIS_HOME%\Python`.

> **Note**
>
> On Windows, the Fast Models examples are installed in `%PVLIB_HOME%\examples\`. The installer makes a copy of them in `%USERPROFILE%\ARM\FastModelsPortfolio_%FM-VERSION%\examples\`. This copy allows you to save configuration changes to these examples without requiring administrator permissions.

2. On Linux, after the installation has completed, source the appropriate script for your shell to set up these environment variables. Ideally, include it for sourcing into the user environment on log-in:

   bash/sh

   ```
   . <install_directory>/FastModelTools_x.x/etc/setup_all.sh
   ```

   csh

   ```
   source <install_directory>/FastModelTools_x.x/etc/setup_all.csh
   ```

3. The Fast Models Arm Virtual Hardware (AVH) peripherals require the standard Python libraries. These libraries are provided in the Fast Models package. Before you run a platform model that uses the AVH peripherals, for example an MPS2 platform, set the `PYTHONHOME` environment variable to the location of these libraries.

   - On Linux:

     ```
     $PVLIB_HOME/lib/Linux64_<compiler>/python
     ```

   - On Windows:

     ```
     %PVLIB_HOME%\lib\Win64_<compiler>\<regime>\python
     ```

4. Optionally, download and install the Third-Party IP (TPIP) add-on package from Product Download Hub. It contains third party add-ons for Fast Models, including ELF images that you can run on the example platforms for evaluation purposes and the GDB Remote Connection plug-in.

Related information

GDBRemoteConnection

2.3 Uninstallation

On Linux, uninstall Fast Models Tools and Fast Models Portfolio by deleting the installation directories.

On Windows, uninstall Fast Models Tools and Fast Models Portfolio by selecting the **Uninstall** option for each product from the **Start > Settings > Apps > Apps & features** list.

2.4 Dependencies for Red Hat Enterprise Linux

Fast Models requires some packages that are part of Red Hat Enterprise Linux, which you might need to install.

Some packages might depend on other packages. If you install with the Add/Remove software GUI tool or the `yum` command line tool, these dependencies resolve automatically. If you install packages directly using the `rpm` command, you must resolve these dependencies manually.

To display the package containing a library file on your installation, enter:

```
rpm -qf library_file
```

For example, to list the package containing `/lib/tls/libc.so.6`, enter the following on the command line:

```
rpm -qf /lib/tls/libc.so.6
```

The following output indicates that the library is in version `2.3.2-95.37` of the `glibc` package:

```
glibc-2.3.2-95.37
```

Table 2-4: Dependencies for Red Hat Enterprise Linux

Package	Required for
`alsa-lib`	Fast Models virtual platforms
`gcc-toolset-10`	Fast Models tools
`gcc-toolset-10-libatomic-devel.x86_64`	Fast Models tools
`glibc`	Fast Models tools and virtual platforms
`glibc-devel`	Fast Models tools
`libgcc`	Fast Models tools and virtual platforms
`libstdc++`	Fast Models tools and virtual platforms
`libstdc++-devel`	Fast Models tools
`libXext`	Fast Models tools and virtual platforms
`libX11`	Fast Models tools and virtual platforms
`libXau`	Fast Models tools and virtual platforms
`libxcb`	Fast Models tools and virtual platforms
`libSM`	Fast Models tools and virtual platforms
`libICE`	Fast Models tools and virtual platforms
`libuuid`	Fast Models tools and virtual platforms
`libXcursor`	Fast Models tools and virtual platforms
`libXfixes`	Fast Models tools and virtual platforms
`libXrender`	Fast Models tools and virtual platforms
`libXft`	Fast Models tools and virtual platforms

Package	Required for
`libXrandr`	Fast Models tools and virtual platforms
`libXt`	Fast Models tools and virtual platforms
`make`	Fast Models tools
`telnet`	Fast Models virtual platforms
`xterm`	Fast Models virtual platforms

3. Building Fast Models

This chapter describes how to use Fast Models tools to build model components and platforms.

It describes the following types of model:

- Integrated SIMulators (ISIMs)
- Exported Virtual Subsystem (EVS) components and platforms

The example command lines shown use a Linux host and GCC 9.3. Other hosts and GCC versions are also supported, see 2.1 Requirements for Fast Models on page 17 for details.

Fast Models includes source code for a range of example platforms, under `$PVLIB_HOME/examples/`. For instructions on building and running them, see the Fast Models Reference Guide:

- For an ISIM, see Build and run an FVP example.
- For an EVS platform, see Build and run an EVS platform example.

3.1 Build targets

Models are built using a tool called System Generator, also called SimGen. To configure the build, SimGen uses a project file, with a `.sgproj` extension.

SimGen supports different build targets, which you specify either in the `.sgproj` file or in the System Canvas **Project Settings** dialog.

This chapter describes the following build targets:

Integrated SIMulator (ISIM)

A simulation executable generated by SimGen from a LISA+ model description. SimGen links the executable against a SystemC shared library.

The build target name in the `.sgproj` file for an ISIM is `TARGET_SYSTEMC_ISIM`.

Exported Virtual Subsystem (EVS)

An EVS can either be a component or a platform:

- An EVS component is a simulation shared library and associated C++ generated by SimGen from a LISA+ model description. You can incorporate the shared library into your own SystemC platform. SimGen links the shared library against a shared SystemC library but it can optionally be linked against a static SystemC library, by setting `USE_STATIC_SYSTEMC_LIB=1` in the `.sgproj` file.
- An EVS platform is an example SystemC platform which provides reference code to demonstrate building a simulation executable by instantiating an EVS component in a hand-coded SystemC main (`sc_main()`).

The build target name in the `.sgproj` file for an EVS is `TARGET_SYSTEMC`.

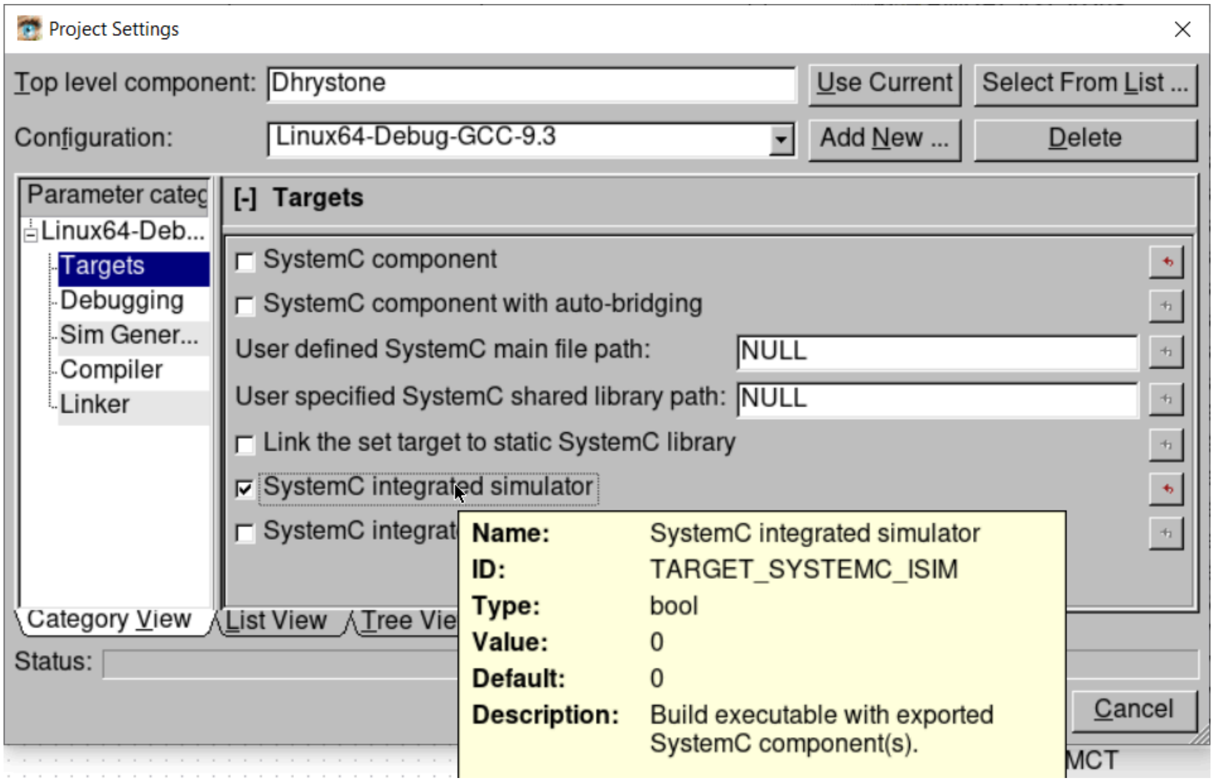
For more information about SimGen and System Canvas, see Fast Models Tools User Guide.

3.2 Building an ISIM

You can build an ISIM either entirely within System Canvas or by invoking SimGen on the command line.

Select ISIM as the build target in either of the following ways:

- In System Canvas, under **Project > Project Settings > Targets**, select **SystemC integrated simulator**:
- **Figure 3-1: Selecting an ISIM build target in System Canvas**



Project Settings

Top level component: Dhrystone

Configuration: Linux64-Debug-GCC-9.3

[-] Targets

□ SystemC component

□ SystemC component with auto-bridging

User defined SystemC main file path: NULL

User specified SystemC shared library path: NULL

□ Link the set target to static SystemC library

☑ SystemC integrated simulator

□ SystemC integrat

Name: SystemC integrated simulator
ID: TARGET_SYSTEMC_ISIM
Type: bool
Value: 0
Default: 0
Description: Build executable with exported SystemC component(s).

- If invoking SimGen directly, use this statement in the `.sgproj` file's active configuration:

```
TARGET_SYSTEMC_ISIM = "1";
```

There is no default build target, so you must specify one, or `simgen` returns an error.

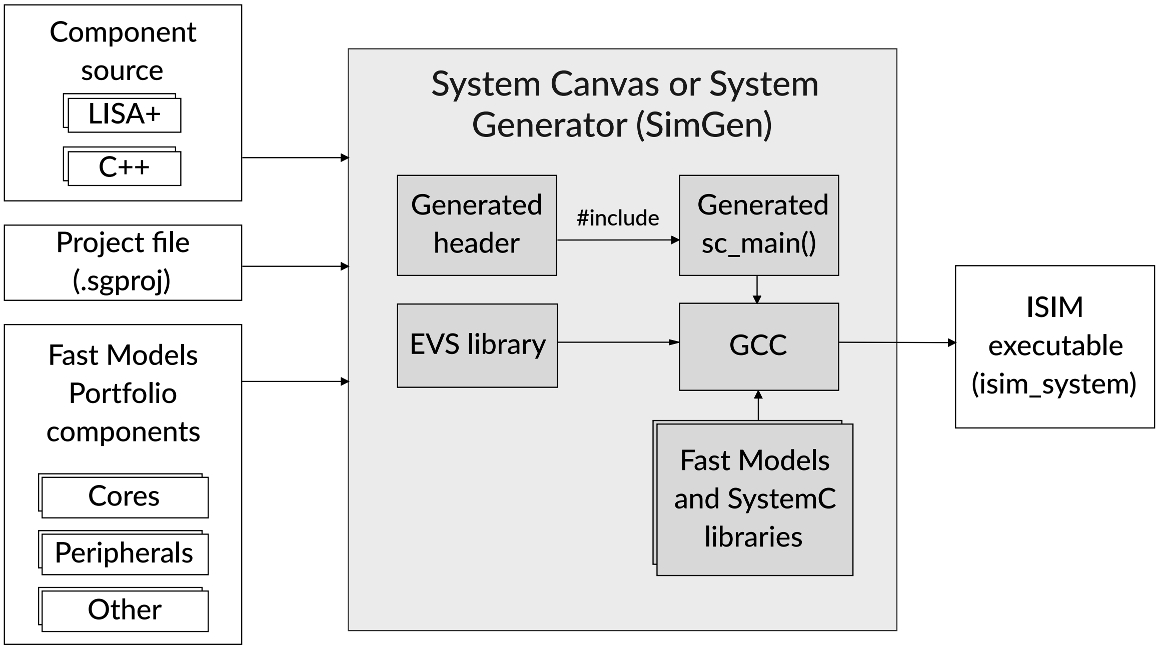
For more information about `.sgproj` files, see SimGen project (sgproj) file format in the Fast Models Tools User Guide or see the `.sgproj` files for the ISIM platform examples installed under `$PVLIB_HOME/examples/LISA/`.

Note

For information on how to build the ISIM platform examples, see Build and run an FVP example in the Fast Models Reference Guide.

The following diagram shows the process for building an ISIM. The shaded area represents the work that SimGen does for you:

Figure 3-2: Build process for an ISIM


Component source
LISA+
C++
Project file (.sgproj)
Fast Models Portfolio components
Cores
Peripherals
Other
System Canvas or System Generator (SimGen)
Generated header
#include
Generated sc_main()
EVS library
GCC
Fast Models and SystemC libraries
ISIM executable (isim_system)

SimGen takes as input the LISA+ or C++ source code for the platform and its components, and the `.sgproj` file.

It generates:

- An EVS library.
- The EVS header file, for example `./Linux64-Release-GCC-9.3/gen/scx_evs_<top_level_component>.h`, which defines the SystemC wrapper.
- A SystemC source file, `./Linux64-Release-GCC-9.3/gen/scx_main_system.cpp` which defines a default `sc_main()` function. This function is the entry point for the simulation. It initializes the simulation, constructs the SystemC wrapper, parses the command-line options, and starts the simulation.

SimGen then links the EVS library with the Fast Models and SystemC libraries, and outputs the executable, named `isim_system`.

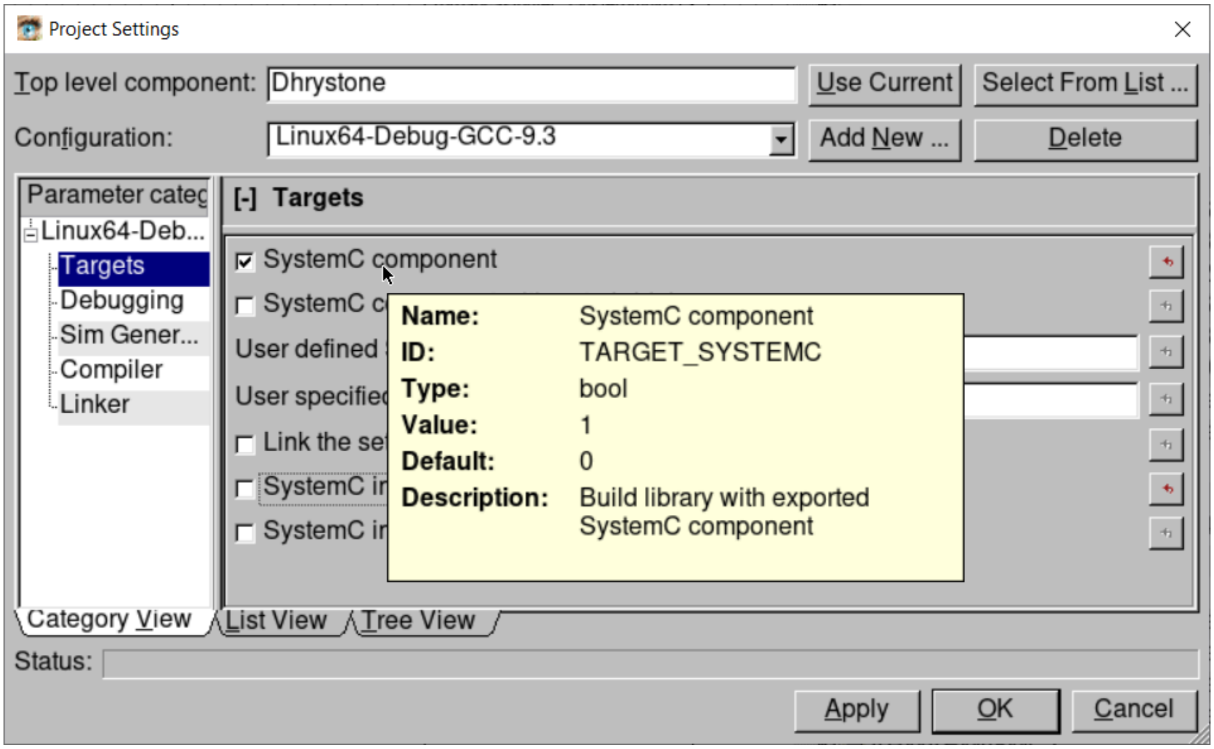
3.3 Building an EVS

SimGen builds an Exported Virtual Subsystem (EVS) as a single, shared library.

Select EVS as the build target in either of the following ways:

- In System Canvas, under **Project > Project Settings > Targets** , select **SystemC component**:

  Figure 3-3: Selecting an EVS build target in System Canvas



Project Settings

Top level component: Dhrystone

Configuration: Linux64-Debug-GCC-9.3

☑ SystemC component

Name: SystemC component
ID: TARGET_SYSTEMC
Type: bool
Value: 1
Default: 0
Description: Build library with exported SystemC component

Status:

- If invoking SimGen directly, use the following statement in the `.sgproj` file's active configuration:

  ```
  TARGET_SYSTEMC = "1";
  ```

  There is no default build target, so you must specify one, or `simgen` returns an error.

  For more information about `.sgproj` files, see SimGen project (sgproj) file format in the Fast Models Tools User Guide or see the example `.sgproj` files installed under `$PVLIB_HOME/examples/SystemCExport/EVS_Components/`.

When building the EVS, it must link against the SystemC library. Select the library to link against in one of the following ways:

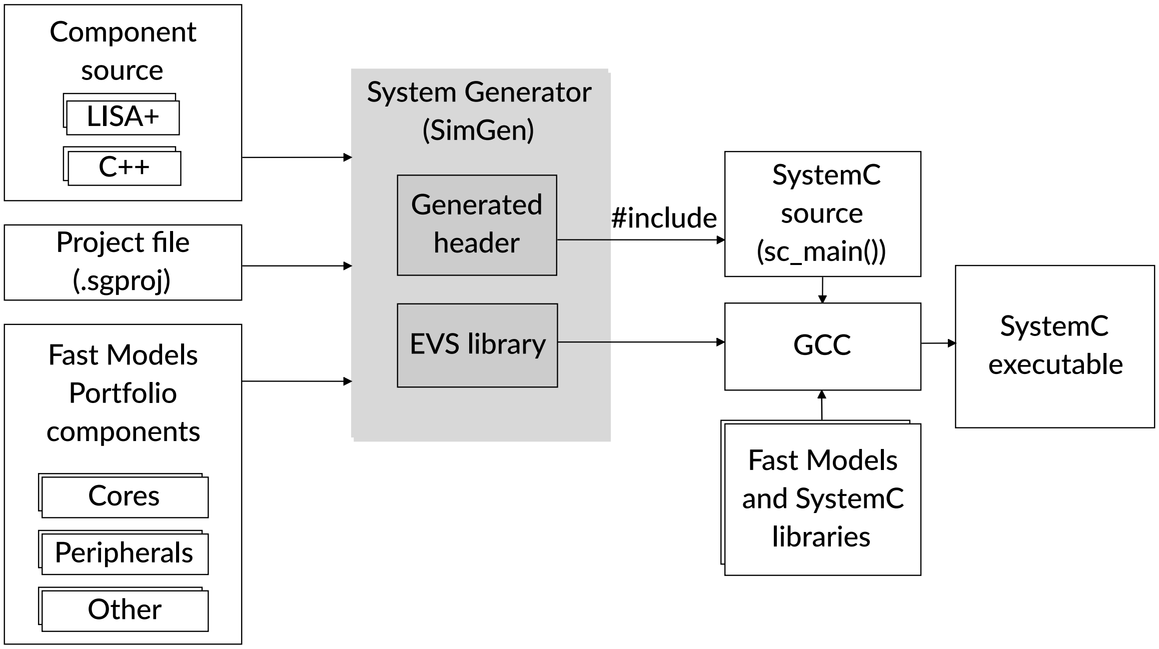
- Link against the dynamic SystemC shared library whose location is given by the `SYSTEMC_HOME` environment variable. This is the default.

- Link against a static SystemC shared library by setting the `.sgproj` configuration parameter `USE_STATIC_SYSTEMC_LIB`. By default, SimGen links against the library located in `SYSTEMC_HOME`.
- Link against either a static or dynamic SystemC library in a different location to `SYSTEMC_HOME`. To do this, set the `USER_SYSTEMC_DYNLIB` `.sgproj` configuration parameter to the full path of the library.

For more information, see 3.5 Linking against the SystemC library on page 30.

The following diagram shows the build process for an EVS. The shaded area represents SimGen and its output. The rest of the diagram is the responsibility of the user. Because the platform must provide its own `sc_main()`, unlike an ISIM, you cannot build it entirely within System Canvas:

Figure 3-4: Build process for an EVS



The Fast Models EVS platform examples are located in `$PVLIB_HOME/examples/SystemCExport/EVS_Platforms/`.

Their purpose is to demonstrate how to instantiate an EVS component, including:

- The generated header file to include, `./Linux64-Release-GCC-9.3/gen/scx_evs_<top_level_component>.h`
- The EVS shared library name to link against
- The component port names for the SystemC exported components

They are built using a Makefile whose target configuration has the pattern:

```
<release>_<compiler_version>_<arch_bits>
```

For instance, to build an example platform in release mode using a model library built with GCC 9.3, as a 64-bit binary, use this command:

```
make rel_gcc93_64
```

Or on Windows:

```
nmake rel_vs142_64
```

Note

For more information on how to build an EVS platform example, see Build and run an EVS platform example in the Fast Models Reference Guide.

3.4 Building an EVS on Windows

When building an EVS on Windows, there are a few extra considerations to be aware of.

- On Windows, SimGen generates a solution containing several project files. For example, the following command outputs `Dhrystone.sln`, where `Dhrystone` is the name of the top-level component, and the following project files, which must be built in the order in which they are listed here:

```
"%MAXCORE_HOME%"\bin\simgen -b -p EVS_Dhrystone_Cortex-A65x1.sgproj --configuration Win64-
Release-VC2019 …
```

Note

When invoking SimGen on Windows, you might need to use the `--devenv-path` option to specify the path to `devenv`.

1. `scx.vcxproj`. This project builds the static library `scx.lib` containing default implementations of the SystemC report handler, simulation controller, and scheduler.
2. `Dhrystone_scx_wrapper_Win64-Release-VC2019.vcxproj`. This project builds the dynamic library `Dhrystone-Win64-Release-VC2019.dll`, which is required when you launch the platform.
3. For an ISIM, an extra project is created, called `scx_isim_<top-component>_<config>.vcxproj`. This file is the SystemC project that creates the executable, `isim_system_<config>.exe`.

- On Windows, any SystemC code that instantiates an exported Fast Model must include `%PVLIB_HOME%\include\fmruntime\sg\IncludeMeFirst.h` before any other include files. If not, the compiler throws an error.

  This file is used to check the underlying Windows API version. It is automatically included in EVSs generated by SimGen, but SystemC models that use Fast Models libraries can be built without using SimGen.
- On Windows, Fast Models libraries are built with one of the following MSVC compiler options:
  - `/MD` for release builds.
  - `/MDd` for debug builds.

  Any objects or libraries that link against the Fast Models libraries must also be built with the same `/MD` or `/MDd` option.
- For a list of additional libraries that are needed when building the EVS, see `%PVLIB_HOME%\examples\SystemCExport\Common\nMakefile.common`.
- Use the `/vmg` compiler option to correctly compile source code for use with SystemC on Windows.

3.5 Linking against the SystemC library

To build an ISIM or EVS, you must have installed SystemC 2.3.4 and the ISIM or EVS must link against the SystemC library at build time.

When you install the main Fast Models package, you have the option of also installing Accellera SystemC 2.3.4. On Windows, the installer automatically sets the `SYSTEMC_HOME` environment variable to the location of the SystemC installation. On Linux, you need to run the appropriate setup script.

By default, SimGen uses the Accellera SystemC dynamic library that is located in `SYSTEMC_HOME`. You might want to use a library stored in a different location, for example if `SYSTEMC_HOME` is not available. You can do this in the following ways:

- In System Canvas, specify the absolute path to the SystemC library, including the library name, in the `User specified SystemC shared library path` field in **Project > Project Settings > Targets** .
- If invoking SimGen directly, provide the path and filename of the SystemC library to SimGen in either of the following ways:
  - Use the `.sgproj` configuration parameter `USER_SYSTEMC_DYNLIB`. For example, on Linux:

    ```
    config "Linux64-Release-GCC-9.3"
    {
        …
        USER_SYSTEMC_DYNLIB = "/path/to/libname_of_systemc.so";
    }
    ```

    On Windows:

    ```
    config "Win64-Release-VC2019"
    {
    ```

```
    …
    USER_SYSTEMC_DYNLIB = "C:\\path\\to\\import_library_name_of_systemc.lib";
}
```

- Use the SimGen command-line option `--override-config-parameter`. For example, on Linux:

```
--override-config-parameter USER_SYSTEMC_DYNLIB="/path/to/libname_of_systemc.so"
```

On Windows:

```
--override-config-parameter USER_SYSTEMC_DYNLIB="C:\\path\\to\
\import_library_name_of_systemc.lib"
```

- To link against a static SystemC library, set the `.sgproj` configuration parameter `USE_STATIC_SYSTEMC_LIB` to 1. The static library can either be located in the default location, `SYSTEMC_HOME`, or in a different location, defined by the parameter `USER_SYSTEMC_DYNLIB`.

  For a CADI library, whose build target is `TARGET_SYSTEMC_MAXVIEW`, `USE_STATIC_SYSTEMC_LIB` is set by default and cannot be unset.

Related information

SimGen command-line options

3.6 Libraries required to run the platform

When you run a platform model, some shared libraries must be present in the same directory as the model, or a diagnostic message is given and the model might fail to run.

If you build a model using SimGen, it can copy the required shared libraries into the location of the generated model. If you then copy the model elsewhere, then you must also copy these shared libraries to the new location.

The list of libraries and executables evolves over time and can vary depending on the IP included in the platform, but might include the following:

- `armlm-ipc` or `armlm-ipc.exe`
- `arm_singleton_registry.so` on Linux or `arm_singleton_registry.dll` on Windows

Note

This is the singleton registry library, which enables multiple simultaneous simulations on the same host platform. If it is not located in the same directory as the executable, set the `FASTSIM_SINGLETON_REGISTRY` environment variable to the full path of the library. If the library is not found, a warning is reported. In this case, a single simulation can still run, but multiple simultaneous simulations might lead to a crash.

- `libarmctmodel.so` or `armctmodel.dll`
- `libarmlm.so` or `armlm.dll`
- `libMAXCOREInitSimulationEngine.3.so` or `libMAXCOREInitSimulationEngine.3.dll`

- `libscxframework.so` or `scxframework.dll`
- `libSDL2-2.0.so.0.10.0` or `SDL2.dll`
- `newt.so` or `newt.dll`

Note

The libraries `libarmlm.so` or `armlm.dll` and `armlm-ipc` or `armlm-ipc.exe` are required by User Based Licensing (UBL). For more information, see the Knowledge Base Article How do I ensure my Fast Model works with User Based Licensing (UBL)?

3.7 Building an SVP

An SVP (SystemC Virtual Platform) is a platform model that consists of LISA+ components or subsystems that are individually exported to SystemC as multiple EVSs, using the Multiple Instantiation (MI) feature.

The build process for an SVP is the same as for an EVS platform, except you must build and link multiple EVS libraries.

SVPs can provide more flexibility than EVS platforms because components in an SVP can be replaced without the need to modify any LISA+ code.

For more information, see the SVP examples under `$PVLIB_HOME/examples/SystemCExport/SVP_Platforms/`.

4. Optimizing runtime performance of Fast Models

Fast Models platforms have many configuration options and parameters. The default ones are reasonable for most workloads on most platforms but there are a few you can change to make the model run as fast as possible.

4.1 Use a suitable host machine

The choice of processor and the amount of RAM available on the machine on which you run the Fast Model can significantly affect its performance.

The Fast Models simulation code runs primarily on a single thread. Arm FVPs and platforms built against the reference SystemC scheduler do not directly support multithreading. As a result, the single-threaded performance of the host machine matters much more than the number of processors it has. The choice of processor can influence model runtime by 1.5x, or more, so use the fastest possible single-threaded processor. As chip manufacturers make improvements, later generations of chips tend to be faster than previous ones.

The amount of memory that a Fast Model uses is unbounded. It allocates enough virtual memory to simulate the platform. If you add the `--stat` option to your FVP command line, on exit, as well as printing the performance statistics for the run, it prints the maximum amount of virtual memory that the platform used. Performance is greatly affected if this figure is greater than the physical memory available to the Fast Model process. We recommend host RAM to be at least 2.5x the amount of RAM the hosted workload uses.

4.2 Configure the model using options and parameters

Some command-line options and parameters should always be set to improve performance. Others should be configured depending on the workload.

These options are recommended:

- If you are targeting performance, always turn cache state modeling off. Running the model with cache state modeling on can slow down the model by more than an order of magnitude. See also Caches in PV models in the *Fast Models Reference Guide*.
- Platforms that support FastRAM always run faster with FastRAM enabled. For more information, see 7. FastRAM on page 131.
- Fast Models implements a large suite of trace sources. MTI trace plug-ins, for example TarmacTrace or GenericTrace, can subscribe to these trace sources, but this has an overhead as some trace fields, for example instruction disassembly, can be expensive to produce. For maximum speed, run the model with no trace plug-ins loaded.

- Some architectural features can be expensive to simulate and might not be required for development purposes. In such cases, a parameter might be present that treats the architectural feature as a NOP, which can improve performance. For example, to disable pointer authentication, use the `treat_PAC_as_NOP` parameter. To discover whether a feature can be treated as a NOP in this way, print a list of the available model parameters using the `-l` command-line option.
- Disable Memory Tagging Extension (MTE) if your workload allows it, or set `force_mte_tag_access_razwi_and_ignore_tag_checks` to true.
- To reduce overhead caused by the scheduler in a Fast Models simulation, use experimentation to tune the quantum and the minimum synchronization latency to your workload. In general, the longer the quantum, the faster the model runs, although this reduces latency which can result in lower performance. A PE can only see changes in the platform when it yields to the scheduler. So the quantum must be short enough that the PE does not spend time on work that will be superceded by other work happening in the platform. To configure the quantum and the minimum synchronization latency, use the options `--quantum` (`-Q` for short) and `--min-sync-latency` (`-M` for short).
- To significantly improve the performance of the Ethos models, set the `ethosu.extra_args="--fast"` parameter.
- If you are using a CPU AEM:
  - Disable the `check_memory_attributes` parameter.
  - Increase the `stage12_tlb_size` parameter to 1024.

Related information

FVP command-line options

4.3 Make the platform faster

If you can customize your platform, or are implementing your own components, use these techniques to make the platform run faster.

- For speed, it is essential that your platform uses DMI (Direct Memory Interface). Fast Models aggressively attempts to use DMI for all load and store operations. A DMI memory operation is often two orders of magnitude faster than a memory transaction that walks the bus to the memory device. If possible, all memory-like components should provide DMI to the Fast Model. Since DMI is so important, invalidation of DMI should be done carefully.
- The Fast Models portfolio contains an MMC component which is based on an old MMC standard and takes a lot of wall clock time to load a large file. If possible, use the Virtio model in the Fast Models portfolio for storage instead.

Related information

MMC component
VirtioP9Device

4.4 Make the workload faster

If possible, ensure your workload avoids busy loops.

Fast Models uses a cooperative scheduler so when a PE is running a busy loop, nothing else can run. The following techniques can avoid this problem:

- Busy loops waiting on time are faster if the GenericTimers and WFE/WFI are used instead. The Fast Model can advance time faster than real time if all PEs are in a WFI/WFE state, skipping over time when no instructions need to run.
- Busy loops waiting on DRAM memory to change should use the Exclusive loads and stores mechanism, so the core can go into WFE, yielding to the scheduler, and wait for the exclusive monitor to wake them.
- You should check busy loops that are part of peripheral initialization to make sure that the peripheral is modeled by the Fast Models platform. If Linux/Android has stalled for many minutes, it could be waiting on peripherals present in the OS device tree and memory map, but which are not modeled.

5. SystemC Export with Multiple Instantiation

This chapter describes the Fast Models SystemC Export feature with *Multiple Instantiation* (MI) support.

5.1 About SystemC Export with Multiple Instantiation

SystemC Export wraps the components of a SystemC-based virtual platform into an *Exported Virtual Subsystem* (EVS). *Multiple Instantiation* (MI) enables the generation and integration of multiple EVS instances into a single SystemC simulation.

SystemC Export with MI enables the generation of EVSs as first-class SystemC components:

- Capable of running any number of instances, alongside other EVSs.
- Providing one `sc_THREAD` per core component (that is, one `sc_THREAD` per core component in a cluster *Code Translation* (CT) model).

MI enables the generation and integration of multiple EVS instances into a virtual platform with SystemC as the single simulation domain. A single EVS can appear in multiple virtual platforms. Equally, multiple EVSs can combine to create a single platform. A platform that consists of multiple EVSs is called an SVP (SystemC Virtual Platform).

SystemC components (including Fast Models ones) can exchange data via the *Direct Memory Interface* (DMI) or normal (blocking) *Transaction Level Modeling* (TLM) transactions.

Fast Models supports SystemC 2.3.4, including integrated TLM 2.0.6. In this version, the TLM and SystemC headers are in the same place, and some filenames are different.

Before using SimGen to build a SystemC simulation, the environment variable `SYSTEMC_HOME` must be set to the directory containing the Accellera SystemC library installation.

When running a SystemC simulation, the following environment variables might be useful:

`SCX_EVS_VERBOSE`

Set to 1 to enable tracing of the default scheduler mapping implementation.

`FM_SCX_VERBOSITY_LEVEL`

Set to one of the following values to set the verbosity level for debug messages from the SystemC simulation:

0	None
100	Low
200	Medium
300	High

400	Full
500	Debug

Note

When loading an image on an EVS, you might see the following warning:

```
Warning: Base.cluster0.cpu0: Uncaught exception, thread terminated
In file: gen/scx_scheduler_mapping.cpp:523
In process: Base.thread_p_5 @ 0 s
```

This warning means that the image is attempting to run from DRAM, but this is access-controlled by the TZC_400 component. To disable security checking by the TZC_400, specify `-C Base.bp.secure_memory=false` when running the EVS.

Related information

Fast Models Reference Guide
Accellera Systems Initiative (ASI)
IEEE Std 1666-2005, SystemC Language Reference Manual, 31 March 2006
Accellera, TLM 2.0 Language Reference Manual, July 2009

5.2 Auto-bridging

Auto-bridging is a Fast Models feature that SimGen uses to automatically convert between LISA+ protocols and their SystemC equivalents. It helps to automate the generation of SystemC wrappers for LISA+ subsystem models.

Note

Auto-bridging is deprecated in Fast Models 11.27 and will be removed in a future release.

A *bridge* is a LISA component that converts transactions from one protocol to another. A wide variety of bridges are available to convert between LISA+ protocols and their SystemC equivalents. For example, `PVBus2AMBAPV` converts from PVBus to AMBA-PV protocols.

When auto-bridging is enabled, you do not need to manually add bridges to your LISA+ file. Auto-bridging causes SimGen to apply the protocol-to-bridge mappings that are defined in a configuration file to the LISA+ components and generate a single EVS component.

Enable auto-bridging by selecting both the `TARGET_SYSTEMC` and `TARGET_SYSTEMC_AUTO` build targets in the `.sgproj` file. Or, in System Canvas Project Settings, select both targets **SystemC component** and **SystemC component with auto-bridging**.

Use the `--bridge-conf-file` SimGen command-line option to select your own auto-bridging configuration file. Alternatively, edit the file `$PVLIB_HOME/etc/bridges_conf.json`, which SimGen uses if you do not specify this option. The syntax is:

```
    "<protocol_name>" : {
        "master" : {
            "name" : "<bridge_name>"
        },
        "slave"  : {
            "name" : "<bridge_name>"
        },
        "peer"   : {
            "name" : "<bridge_name>"
        }
    },
```

- SimGen ignores any bridges whose name is empty in the configuration file.
- Auto-bridging is not applied to any ports that are marked as internal in the LISA+ file.
- SimGen reports an error if auto-bridging is enabled and a top-level port in a LISA+ component uses a protocol that is not listed in the configuration file.
- SimGen reports an error if auto-bridging is enabled and it cannot find the configuration file.
- You do not need to specify bridges for the following LISA+ protocols. When ports that use these protocols are exported to SystemC, SimGen can automatically generate the TLM sockets for them, without the need for bridging:
    - `AudioControl`
    - `ClockRateControl`
    - `ClockSignal`

      Note

      Do not export the `ClockSignal` port if your intention is to drive a clock from an external SystemC source. The `ClockSignal` should be driven from the MasterClock in the Fast Model, not from the external SystemC source.
    - `CounterInterface`
    - `GICv3Comms`
    - `InstructionCount`
    - `KeyboardStatus`
    - `LCD`
    - `MouseStatus`
    - `PChannel`
    - `SystemCoherencyInterface`
    - `VECBProtocol`
    - `VirtualEthernet`

To access the generated TLM sockets from SystemC, you must `#include` the appropriate header files from under `$PVLIB_HOME/examples/SystemCExport/Common/Protocols/`.

5.3 SystemC Export generated ports

This section describes the generated ports and the associated port protocols.

Related information

About SystemC Export generated ports on page 136

5.3.1 Protocol definition

The ports of the top level Fast Models component, used to create SystemC ports, have protocols.

The behaviors in a Fast Models protocol definition must match exactly the functions in the SystemC port class. System Canvas does not check this for consistency, but the C++ compiler can find inconsistencies when compiling the generated SystemC component.

The set of functions and behaviors, their arguments, and their return value must be the same. The order of the functions and behaviors does not matter.

All behaviors in the Fast Models protocol must be slave behaviors. There is no corresponding concept of master behaviors.

The protocol definition also contains a properties section that contains the properties that describe the SystemC C++ classes that implement the corresponding ports on the SystemC side.

Related information

LISA+ Language for Fast Models Reference Guide

5.3.2 TLM 1.0 protocol for an exported SystemC component

Here is an example of a TLM 1.0 signal protocol.

```
protocol MySignalProtocol
{
  includes
  {
    #include <mySystemCClasses.h>
  }
  properties
  {
    sc_master_port_class_name = "my_signal_base<bool>";
    sc_slave_base_class_name = "my_slave_base<bool>";
    sc_slave_export_class_name = "my_slave_export<bool>";
  }
  slave behavior set_state(const bool & state);
}
```

5.3.3 TLM 2.0 bus protocol for an exported SystemC component

Here is an example of a TLM 2.0 bus protocol.

```
protocol MyProtocol
{
  includes
  {
    #include <mySystemCClasses.h>
  }
  properties
  {
    sc_master_base_class_name = "my_master_base";
    sc_master_socket_class_name = "my_master_socket<64>";
    sc_slave_base_class_name = "my_slave_base<64>";
    sc_slave_socket_class_name = "my_slave_socket<64>";
  }
  slave behavior read(uint32_t addr, uint32_t &data);
  slave behavior write(uint32_t addr, uint32_t data);
  master behavior invalidate_dmi(uint32_t addr);
}
```

This protocol enables declaring ports that have `read()` and `write()` functions. This protocol can declare master and slave ports.

5.3.4 Properties for TLM 1.0 based protocols

TLM 1.0 based protocols map to their SystemC counterparts using properties in the LISA protocol definition. The protocol description must set these properties.

`sc_master_port_class_name`

The sc_master_port_class_name property is the class name of the SystemC class that the generated SystemC component instantiates for master ports on the SystemC side. This class must implement the functions defined in the corresponding protocol, for example:

```
void my_master_port<bool>::set_state(bool state)
```

`sc_slave_base_class_name`

The sc_slave_base_class_name property is the class name of the SystemC class that the generated SystemC component specializes for slave ports on the SystemC side. This class must declare the functions defined in the corresponding protocol, for example:

```
void my_slave_base<bool>::set_state(const bool &state)
```

The SystemC component must define it to forward the protocol functions from the SystemC component to the Fast Models top level component corresponding port. It must also provide a constructor taking the argument:

```
const std::string &name
```

sc_slave_export_class_name

The `sc_slave_export_class_name` property is the class name of the SystemC class that the generated SystemC component instantiates for slave ports (exports) on the SystemC side. The component binds to the derived `sc_slave_base_class_name` SystemC class, and forwards calls from the SystemC side to the bound class.

AMBAPV Signal protocol in Fast Models

```
protocol AMBAPVSignal {

  includes {
    #include <amba_pv.h>
  }

  properties {
    description = "AMBA-PV signal protocol";
    sc_master_port_class_name = "amba_pv::signal_master_port<bool>";
    sc_slave_base_class_name = "amba_pv::signal_slave_port<bool>";
    sc_slave_export_class_name = "amba_pv::signal_slave_export<bool>";
  }
...
```

`sc_slave_export_class_name` and `sc_master_port_class_name` describe the type of the port instances in the SystemC domain.

`sc_slave_base_class_name` denotes the base class from which the SystemC component publicly derives.

AMBAPV Signal protocol in SystemC component class

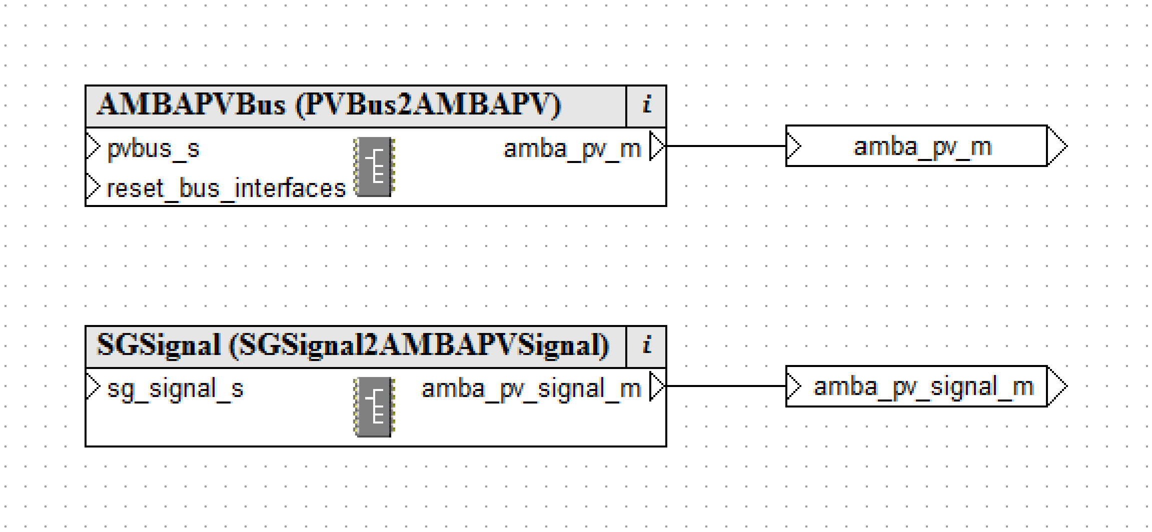
The SystemC module ports must use the corresponding names in the SystemC code.

```
class pv_dma: public sc_module,
              public amba_pv::signal_slave_base<bool> {

  /* Module ports */
  amba_pv::signal_master_port<bool> signal_out;
  amba_pv::signal_slave_export<bool> signal_in;
  ...
```

The SystemC port names must also match the Fast Models port names. For example, `signal_out` is the instance name for the master port in the Fast Models AMBAPVBus component and the SystemC port.

Figure 5-1: SGSignal component in System Canvas



5.3.5 Properties for TLM 2.0 based protocols

The TLM 2.0 protocol provides forward and backward paths for master and slave sockets. Protocols that use TLM 2.0 must specify properties in the protocol declaration.

`sc_master_socket_class_name`

This is the class name of the SystemC class that the generated SystemC component instantiates for master sockets on the SystemC side. The component binds to the derived `sc_master_base_class_name` SystemC class and forwards calls from:

- The bound class to SystemC (forward path).
- The SystemC side to the bound class (backward path).

`sc_master_base_class_name`

This is the class name of the SystemC class that the generated SystemC component specializes for master sockets on the SystemC side. This class must declare the master behavior functions defined in the corresponding protocol, for example:

```
my_master_base::invalidate_dmi(uint32_t addr)
```

The SystemC component must define it to forward the protocol functions from the SystemC component (backward path) to the System Generator top level component corresponding socket. It must also provide a constructor taking the argument:

```
const std::string &
```

`sc_slave_socket_class_name`

This is the class name of the SystemC class that the generated SystemC component instantiates for slave sockets on the SystemC side. The component binds to the derived `sc_slave_base_class_name` SystemC class and forwards calls from:

- The bound class to SystemC (backward path).
- The SystemC side to the bound class (forward path).

`sc_slave_base_class_name`

This is the class name of the SystemC class that the generated SystemC component specializes for slave sockets on the SystemC side. It must also provide a constructor taking the argument:

```
const std::string &
```

AMBAPV protocol in System Generator

```
protocol AMBAPVSignal {
  includes {
  #include <amba_pv.h>
  }

  properties {
    description = "AMBA-PV protocol";
    sc_master_base_class_name = "amba_pv::amba_pv_master_base";
    sc_master_socket_class_name = "amba_pv::amba_pv_master_socket<64>";
    sc_slave_base_class_name = "amba_pv::amba_pv_slave_base<64>";
    sc_slave_socket_class_name = "amba_pv::amba_pv_slave_socket<64>";
  }
```

AMBAPV protocol in SystemC component class

The SystemC module sockets must use the corresponding names in the SystemC code.

```
class pv_dma: public sc_module,
              public amba_pv::amba_pv_slave_base<64>,
              public amba_pv::amba_pv_master_base {

/* Module ports */
    amba_pv::amba_pv_slave_socket<64> amba_pv_s;
    amba_pv::amba_pv_master_socket<64> amba_pv_m;
    ...
}
```

5.4 SystemC Export API

This section describes the *SystemC eXport* (SCX) API provided by Fast Models *Exported Virtual Subsystems* (EVSs). Each description of a class or function includes the C++ declaration and the use constraints.

5.4.1 SystemC Export header file

To use the SystemC Export feature, an application must include the C++ header file `scx.h` at appropriate positions in the source code as required by the scope and linkage rules of C++.

The header file `$PVLIB_HOME/include/fmruntime/scx/scx.h` adds the namespace scx to the declarative region that includes it. This inclusion declares all definitions related to the SystemC Export feature of Fast Models within that region.

```
#include "scx.h"
```

5.4.2 scx::scx_initialize

This function initializes the simulation.

Initialize the simulation before constructing any exported subsystem.

```
void scx_initialize(const std::string &id,
                    scx_simcontrol_if *ctrl = scx_get_default_simcontrol());
```

`id`

an identifier for this simulation.

`ctrl`

a pointer to the simulation controller implementation. It defaults to the one provided with Arm® models.

Note

Arm recommends specifying a unique identifier across all simulations running on the same host.

5.4.3 scx::scx_set_single_evs

Sets the simulation engine to accept a single EVS only.

```
void scx_set_single_evs();
```

The EVS name will be stripped from CADI parameters.

Call this function immediately after calling `scx_initialize()`.

5.4.4 scx::scx_load_application

This function loads an application in the memory of an instance.

```
void scx_load_application(const std::string &instance,
                          const std::string &application);
```

`instance`

the name of the instance to load into. The parameter `instance` must start with an EVS instance name, or with `"*"` to load the application into the instance on all EVSs in the platform. To load the same application on all cores of an SMP processor, specify `"*"` for the core instead of its index, in parameter `instance`.

`application`

the application to load.

Note

The loading of the application happens at `start_of_simulation()` call-back, at the earliest.

5.4.5 scx::scx_load_application_all

This function loads an application in the memory of instances that execute software, across all EVSs in the platform.

```
void scx_load_application_all(const std::string &application);
```

`application`

the application to load.

Note

The loading of the application happens at `start_of_simulation()` call-back, at the earliest.

5.4.6 scx::scx_load_data

This function loads binary data in the memory of an instance at a memory address.

```
void scx_load_data(const std::string &instance,
                   const std::string &data,
                   const std::string &address);
```

instance

the name of the instance to load into. The parameter `instance` must start with an EVS instance name, or with `"*"` to load `data` into the instance on all EVSs in the platform. On an SMP processor, if `instance` specifies `"*"` for the core instead of its index, the binary data loads only on the first processor.

data

the filename of the binary data to load.

address

the memory address at which to load the data. The parameter `address` might start with a memory space specifier.

Note

The loading of the binary data happens at `start_of_simulation()` call-back, at the earliest.

5.4.7 scx::scx_load_data_all

This function loads binary data in the memory of instances that execute software, across all EVSs in the platform, at a memory address. On SMP processors, the data loads only on the first core.

```
void scx_load_data_all(const std::string &data,
                       const std::string &address);
```

data

the filename of the binary data to load.

address

the memory address at which to load the data. The parameter `address` might start with a memory space specifier.

Note

The loading of the binary data happens at `start_of_simulation()` call-back, at the earliest.

5.4.8 scx::scx_set_parameter

This function sets the value of a parameter in components present in EVSs or in plug-ins.

-
  ```
  bool scx_set_parameter(const std::string &name, const std::string &value);
  ```

-
  ```
  template<class T>
  ```

```
bool scx_set_parameter(const std::string &name, T value);
```

`name`

the name of the parameter to change. The parameter `name` must start with an EVS instance name for setting a parameter on this EVS, or with `"*"` for setting a parameter on all EVSs in the platform, or with a plug-in prefix (defaults to `"TRACE"`) for setting a plug-in parameter.

`value`

the value of the parameter.

This function returns `true` when the parameter exists, `false` otherwise.

Note

- Changes made to parameters within System Canvas take precedence over changes made with `scx_set_parameter()`.
- You can set parameters during the construction phase, and before the elaboration phase. Calls to `scx_set_parameter()` after the construction phase are ignored.
- You can change run-time parameters after the construction phase with the debug interface.
- Specify plug-ins before calling the platform parameter functions, so that the plug-ins load and their parameters are available. Any plug-in that is specified after the first call to any platform parameter function is ignored.

5.4.9 scx::scx_get_parameter

This function retrieves the value of a parameter from components present in EVSs or from plug-ins.

-
  ```
  bool scx_get_parameter(const std::string &name, std::string &value);
  ```

-
  ```
  template<class T>
  bool scx_get_parameter(const std::string &name, T &value);
  ```

-
  ```
  bool scx_get_parameter(const std::string &name, int &value);
  ```

-
  ```
  bool scx_get_parameter(const std::string &name, unsigned int &value);
  ```

-
  ```
  bool scx_get_parameter(const std::string &name, long &value);
  ```

-
  ```
  bool scx_get_parameter(const std::string &name, unsigned long &value);
  ```

-
  ```
  bool scx_get_parameter(const std::string &name, long long &value);
  ```

-
  ```
  bool scx_get_parameter(const std::string &name, unsigned long long &value);
  ```

-
  ```
  std::string scx_get_parameter(const std::string &name);
  ```

name

the name of the parameter to retrieve. The parameter `name` must start with an EVS instance name for retrieving an EVS parameter or with a plug-in prefix (defaults to `"TRACE"`) for retrieving a plug-in parameter.

value

a reference to the value of the parameter.

The `bool` forms of the function return `true` when the parameter exists, `false` otherwise. The `std::string` form returns the value of the parameter when it exists, empty string (`""`) otherwise.

Note

Specify plug-ins before calling the platform parameter functions, so that the plug-ins load and their parameters are available. Any plug-in that is specified after the first call to any platform parameter function is ignored.

5.4.10 scx::scx_get_parameter_list

This function retrieves a list of all parameters in all components present in all EVSs and from all plug-ins.

```
std::map<std::string, std::string> scx_get_parameter_list();
```

The parameter names start with an EVS instance name for EVS parameters or with a plug-in prefix (defaults to `"TRACE"`) for plug-in parameters.

Note

- Specify plug-ins before calling the platform parameter functions, so that the plug-ins load and their parameters are available. Any plug-in that is specified after the first call to any platform parameter function is ignored.
- If `scx_set_parameter()` is called after the simulation elaboration phase, the new value is not set in the model, although it is returned by `scx_get_parameter_list()`.

5.4.11 scx::scx_get_parameter_infos

Retrieve a list of descriptions for all parameters within the simulation.

```
std::map<std::string, std::string> scx_get_parameter_infos();
```

The list includes parameters for all components present in all EVSs and for all plug-ins.

The names of EVS parameters start with an EVS instance name and the names of plug-in parameters start with a plug-in prefix, which defaults to `TRACE`.

Note

Plug-ins must be specified before calling any of the platform parameter functions, otherwise plug-ins are not loaded and their parameters are not available. Any plug-in specified after the first call to any platform parameter function is ignored.

5.4.12 scx::scx_get_cadi_parameter_infos

Retrieve a vector of `CADIParameterInfo_t` objects for all the parameters in the simulation.

```
std::vector<eslapi::CADIParameterInfo_t> scx_get_cadi_parameter_infos();
```

Use this function to get CADI parameter objects with all the relevant fields present for all EVSs, external SystemC modules, and loaded plug-ins.

Note

Plug-ins must be specified before calling any of the platform parameter functions, otherwise plug-ins are not loaded and their parameters are not available. Any plug-in specified after the first call to any platform parameter function is ignored.

5.4.13 scx::scx_set_cpi_file

Sets the Cycles Per Instruction (CPI) file for CPI class functionality.

```
void scx_set_cpi_file(const std::string & cpi_file_path);
```

`cpi_file_path`

the path to the CPI file.

Use this function to activate the CPI class functionality.

Note

This function must be called before any call to a platform parameter function.

5.4.14 scx::scx_cpulimit

Sets the maximum number of CPU (User + System) seconds to run, excluding startup and shutdown.

```
void scx_cpulimit(double t);
```

`t`

the number of seconds to run. Defaults to unlimited.

5.4.15 scx::scx_timelimit

Sets the maximum number of seconds to run, excluding startup and shutdown.

```
void scx_timelimit(double t);
```

`t`

the number of seconds to run. Defaults to unlimited.

5.4.16 scx::scx_add_breakpoint

Set a breakpoint at a specific address.

```
void scx_add_breakpoint(std::string instance, uint64_t addr,
bool perthread, uint32_t threadid);
```

`instance`

Name of the core target instance.

`addr`

Address at which to set the breakpoint.

`perthread`

If true, the breakpoint only hits if `threadid` matches the current thread.

`threadid`

Thread ID for the breakpoint. Only used if `perthread` is true.

5.4.17 scx::scx_set_start_pc

Set the initial value of the PC register for a specific instance.

```
void scx_set_start_pc(std::string instance, uint64_t addr);
```

`instance`

Name of the core target instance.

`addr`

Start PC address.

5.4.18 scx::scx_dump

Set the details of a memory dump to be written to a file.

```
void scx_dump(std::string instance, std::string filename, std::string memSpace,
 uint64_t addr, uint64_t size);
```

`instance`

Name of the target instance to dump memory from.

`filename`

The path to the file to dump memory to.

`memSpace`

The name or ID of the memory space.

`addr`

Start address from which to dump.

`size`

Number of bytes of memory to dump.

5.4.19 scx::scx_load_params_file

Load parameter values from a configuration file.

```
void scx_load_params_file(const std::string& filename);
```

`filename`

The name of the configuration file to load.

Note

Plug-ins must be specified before calling any of the platform parameter functions, otherwise these plug-ins will not be loaded and their parameters will not be available.

5.4.20 scx::scx_list_instances

List all instances in the simulation.

```
void scx_list_instances(const std::string& filename = std::string());
```

`filename`

The path to the file to hold the output. The default is an empty string, which sends output to `std::cout`.

5.4.21 scx::scx_list_registers

List all simulation registers.

```
void scx_list_registers(const std::string& filename = std::string());
```

`filename`

The path to the file to hold the output. The default is an empty string, which sends output to `std::cout`.

5.4.22 scx::scx_check_registers

List all simulation registers and perform extra consistency checks on the CADI register API.

```
void scx_check_registers(const std::string& filename = std::string());
```

`filename`

The path to the file to hold the output. The default is an empty string, which sends output to `std::cout`.

5.4.23 scx::scx_list_memory

List all simulation memory.

```
void scx_list_memory(const std::string& filename = std::string());
```

`filename`

The path to the file to hold the output. The default is an empty string, which sends output to `std::cout`.

5.4.24 scx::scx_parse_and_configure

This function parses command-line options and configures the simulation accordingly.

```
void scx_parse_and_configure(int argc,
                             char *argv[],
                             const char *trailer = NULL,
                             bool sig_handler = true);
```

`argc`

the number of command-line options listed with `argv[]`.

argv

command-line options.

trailer

a string that follows the option list when printing the help message (`--help` option).

sig_handler

whether to enable signal handler function. `true` to enable (default), `false` to disable.

The application must pass the values of the options from function `sc_main()` as arguments to this function.

-a, --application

application to load, format: `-a [INST=]FILE`. For SMP cores: `-a INST*=FILE`.

-A, --iris-allow-remote

allow remote connections from another machine to the Iris server. Defaults to not allowed.

-b, --break

set a breakpoint, format: `-b [INST=]ADDRESS`. This option can be specified multiple times.

-C, --parameter

set a parameter, format: `-C INST.PARAM=VALUE`. This option can be specified multiple times.

--check-regs

the same as `--list-regs` but does more consistency checks on the CADI register API.

--cpi-file

use *`FILE`* to set Cycles Per Instruction (CPI) classes, format: `--cpi-file FILE`

--cpulimit

maximum number of CPU (User + System) seconds to run, excluding startup and shutdown, format: `--cpulimit NUM`. Defaults to unlimited.

--cyclelimit

number of cycles to run, ignored if the debug server has started, format: `--cyclelimit NUM`. Defaults to unlimited.

-D, --allow-debug-plugin

allow a plug-in to debug the simulation.

--data

raw data to load, format: `--data [INST=]FILE@[MEMSPACE:]ADDRESS`

--dump

dump a section of memory into *`FILE`*, format: `--dump [INST=]FILE@[MEMSPACE:]ADDRESS,SIZE`. This option can be specified multiple times.

--dump-params

dump the list of model parameters into a JSON file and exit.

-f, --config-file

load model parameters from configuration file *`FILE`*, format: `--config-file FILE`

-h, --help

print help message and exit.

--iris-connect

start an Iris server. Format: `--iris-connect CONSPEC` where `CONSPEC` is a structured string argument which can contain flags and parameters:

- An empty string is an error.
- `help` prints a list of supported connection types.

Command line options `--iris-server`, `--iris-allow-remote`, `--iris-port`, and `--iris-port-range` are ignored when using `--iris-connect`.

-i, --iris-log

Iris log level. This option can be specified multiple times, for example: `-ii` for log level 2.

-I, --iris-server

start an Iris server, allowing debuggers to connect to targets in the simulation.

--iris-port

set a specific port to use for the Iris server, format: `--iris-port PORT`

--iris-port-range

set the range of ports to scan when starting an Iris server. The first available port found is used, format: `--iris-port-range MIN:MAX`

-K, --keep-console

keep the console window open after completion. This option applies to Microsoft Windows only.

-l, --list-params

print the list of available model parameters and their default values to standard output and exit.

--list-set-params

save the list of model parameters for each component in the model and the values that are set to a file or print to standard output. Format: `--list-set-params FILE` or `--list-set-params -` to print to stdout.

-L, --cadi-log

log all CADI calls to XML log files.

--list-instances

print list of target instances to standard output.

--list-memory

print model memory information to standard output.

--list-regs

print model register information to standard output.

-o, --output

redirect parameters, memory and instance lists to output file *`FILE`*, format: `--output FILE`

-p, --print-port-number

print the TCP port number the CADI server is listening to.

-P, --prefix

prefix semihosting output with the name of the instance.

--plugin

plug-in to load, format: `--plugin [NAME=]FILE`

-q, --quiet

suppress informational output.

-R, --run

run the simulation immediately after the CADI server starts.

-S, --cadi-server

start a CADI server, allowing debuggers to connect to targets in the simulation.

--simlimit

maximum number of seconds to simulate, ignored if the debug server has started, format: `--simlimit NUM`. Defaults to unlimited.

--start

set initial PC to application start address, format: `--start [INST=]ADDRESS`

--stat

print run statistics on simulation exit.

-T, --timelimit

maximum number of seconds to run, excluding startup and shutdown, ignored if the debug server has started, format: `--timelimit NUM`. Defaults to unlimited.

--trace-plugin

deprecated, use `--plugin` instead.

This function treats all other command-line arguments as applications to load.

This function calls `std::exit(EXIT_SUCCESS)` to exit, for options `--list-params` and `--help`. It calls `std::exit(EXIT_FAILURE)` if there was an error in the parameter specification, or an invalid option was specified, or if the application or plug-in was not found.

5.4.25 scx::scx_register_synchronous_thread

This function registers a new thread in the simulation engine which is implicitly synchronized with the simulation thread.

```
void scx_register_synchronous_thread(std::thread::id thread_id);
```

thread_id

ID of the newly registered thread.

The caller must make sure that the simulation thread and the newly registered thread do not run concurrently.

Calling this function for a thread *x* completely disables the thread synchronization for thread *x*, that is, marshaling of function calls from the calling thread onto the simulation thread, for example Iris calls.

This function is useful for debugger threads which are blocking the simulation thread and which still want to issue Iris calls while the simulation thread is blocked.

5.4.26 scx::scx_get_error_count

This function returns the number of errors recorded by the simulation engine.

Note

The count includes internal errors recorded by the simulation engine, some of which are not reported as errors by `scx_report_handler`.

```
size_t scx_get_error_count();
```

5.4.27 scx::scx_get_exitcode_list

This function returns the list of exit codes that were logged by the simulation engine.

The returned list is a `std::vector` that contains the logged exit codes in order. Each entry in the list is a struct of type 5.4.28 scx::scx_exitcode_entry on page 56. The last entry is the most recent.

Note

- If no exit code was logged, the returned list is empty.
- This function only produces valid output after `sc_start()` has returned. It must not be called beforehand.

```
const scx_exitcode_list_t & scx_get_exitcode_list();
```

5.4.28 scx::scx_exitcode_entry

Represents an entry in the exit code list.

Note

The exit code list is returned by 5.4.27 scx::scx_get_exitcode_list on page 56.

```
struct scx_exitcode_entry
{
    scx_exitcode_entry(int exitcode_, std::string component_name_, std::string
 kind_, std::string reason_)
    : exitcode(exitcode_)
    , component_name(component_name_)
    , kind(kind_)
    , reason(reason_)
    {}

    int exitcode;
    std::string component_name;
    std::string kind;
    std::string reason;
};
```

exitcode

The exit code that was logged.

component_name

The name of the component that generated the exit code. This name is auto-generated by the simulation engine at the time of logging.

kind

The type of component that generated the exit code.

reason

Optional field that provides a human-readable string explaining why the exit code was logged. If this field is empty, then no reason was given and this field can be ignored.

5.4.29 scx::scx_start_cadi_server

This function specifies whether to start a CADI server.

```
void scx_start_cadi_server(bool start = true, bool run = true, bool debug = false);
```

`start`

`true` to start a CADI server, `false` otherwise.

`run`

`true` to run the simulation immediately after the CADI server has been started, `false` otherwise.

`debug`

`true` to enable debugging through a plug-in, `false` otherwise.

Starting a CADI server enables the attachment of a debugger to debug targets in the simulation.

When `debug` is set to `true`, the CADI server does not start, but a plug-in can implement an alternative debugging mechanism in place of it.

When `start` is set to `true`, it overrides `debug`.

Note

- A CADI server cannot start after simulation starts.
- You do not need to call this function if you have called `scx_parse_and_configure()` and parsed at most one of `-S` or `-D` into `sc_main()`.

5.4.30 scx::scx_enable_cadi_log

This function specifies whether to log all CADI calls to XML files.

```
void scx_enable_cadi_log(bool log = true);
```

`log`

`true` to log CADI calls, `false` otherwise.

Note

You cannot enable logging once simulation starts.

5.4.31 scx::scx_print_port_number

This function specifies whether to enable printing of the TCP port number that the CADI server is listening to.

```
void scx_print_port_number(bool print = true);
```

`print`

`true` to enable printing of the TCP port number, `false` otherwise.

Note

You cannot enable printing of the TCP port number once simulation starts.

5.4.32 scx::scx_print_statistics

This function specifies whether to enable printing of simulation statistics at the end of the simulation.

```
void scx_print_statistics(bool print = true);
```

print

true to enable printing of simulation statistics, false otherwise.

Note

- You cannot enable printing of statistics once simulation starts.
- The statistics include LISA reset() behavior run time and application load time. A long simulation run compensates for this.

5.4.33 scx::scx_register_cadi_target

Register a CADI target info and interface into the simulation.

```
void scx_register_cadi_target(eslapi::CADITargetInfo_t * info, eslapi::CAInterface *
 caif = NULL);
```

info

Points to an eslapi::CADITargetInfo_t structure describing this CADI target.

caif

Points to an eslapi::CAInterface of this CADI target.

Use this function to register a target into the simulation. The target is then accessible through a CADI debugger attached to the simulation.

Note

Registering a target must be perfomed before the end of elaboration.

5.4.34 scx::scx_unregister_cadi_target

Unregister a specific CADI target from the simulation.

```
void scx_unregister_cadi_target(const std::string &);
```

`name`

Instance name of this CADI target.

Use this function to unregister a target from the simulation. After calling this function, the target will not be accessible through a CADI debugger.

5.4.35 scx::scx_load_trace_plugin

Arm deprecates this function. Use `scx_load_plugin()` instead.

5.4.36 scx::scx_load_plugin

This function specifies a plug-in to load.

```
void scx_load_plugin(const std::string &file);
```

`file`

the file of the plug-in to load.

The plug-in loads at `end_of_elaboration()`, at the latest, or as soon as a platform parameter function is called.

Note

Specify plug-ins before calling the platform parameter functions, so that the plug-ins load and their parameters are available. Any plug-in that is specified after the first call to any platform parameter function is ignored.

5.4.37 scx::scx_get_global_interface

This function accesses the global interface.

```
eslapi::CAInterface *scx_get_global_interface();
```

The global interface allows access to all of the registered interfaces in the simulation.

5.4.38 scx::scx_enable_iris_server

Starts or stops the Iris server.

```
void scx_enable_iris_server(const std::string& connection_spec)
```

```
void scx_enable_iris_server(bool enable = true);
```

`connection_spec`

String that specifies the type of server to start and its parameters. For example:

- For a TCP server: `tcpserver,port=7100,endport=7163,allowRemote`
- For a UNIX domain socket connection: `socketfd=42`
- To stop a running Iris server, use an empty string.
- To display all supported connection types, specify `help`.

`enable`

`true` to start an Iris server (default), `false` to stop it.

Note

Starting the Iris server puts the simulation into a wait state, until a client connects to the server.

5.4.39 scx::scx_set_iris_server_port_range

Set the range of ports to scan. The Iris server uses the first available port found in the range.

```
void scx_set_iris_server_port_range(uint16_t port_min, uint16_t port_max);
```

`port_min`

the port number at the start of the range.

`port_max`

the port number at the end of the range.

Note

This function only takes effect if you call it before starting the Iris server.

Related information

scx::scx_enable_iris_server on page 60

5.4.40 scx::scx_get_iris_server_port

Return the Iris TCP port number that is assigned when the Iris server starts, or zero if the Iris server has not yet started.

```
uint16_t scx_get_iris_server_port();
```

5.4.41 scx::scx_set_iris_server_port

Set a specific port for the Iris server to listen on.

```
inline void scx_set_iris_server_port(uint16_t port)
```

`port`

The port number for the Iris server to listen on.

Note

This function only takes effect if you call it before starting the Iris server.

Related information

scx::scx_enable_iris_server on page 60

5.4.42 scx::scx_enable_iris_log

This function sets the Iris message log level.

```
void scx_enable_iris_log(unsigned level = 0);
```

`level`

the log level. The possible values are:

0

Logging is disabled. This is the default value.

1

Log messages use a compact, single-line format.

2

Log messages use a single-line, pseudo-JSON format.

3

Log messages use a more readable multi-line, pseudo-JSON format.

4

As 3 but also prints the U64JSON hex value of the message.

Note

An alternative way to set the Iris log level is to use the `IRIS_GLOBAL_INSTANCE_LOG_MESSAGES` environment variable.

5.4.43 scx::scx_get_iris_connection_interface

Return the `IrisConnectionInterface` for the simulation. This can be used to create and register `IrisInstance`s.

```
iris::IrisConnectionInterface *scx_get_iris_connection_interface();
```

5.4.44 scx::scx_evs_base

This class is the base class for EVSs. EVSs are the principal subsystems of the Fast Models SystemC Export feature.

```
class scx_evs_base {
  public:
    void load_application(const std::string &, const std::string &);
    void load_data(const std::string &, const std::string &, const std::string &);
    bool set_parameter(const std::string &, const std::string &);
    template<class T>
    bool set_parameter(const std::string &, T);
    bool get_parameter(const std::string &, std::string &) const;
    template<class T>
    bool get_parameter(const std::string &, T &) const;
    std::string get_parameter(const std::string &) const;
    std::map<std::string, std::string> get_parameter_list() const;
  protected:
    scx_evs_base(const std::string &, sg::ComponentFactory *);
    virtual ~scx_evs_base();
    void before_end_of_elaboration();
    void end_of_elaboration();
    void start_of_simulation();
    void end_of_simulation();
};
```

5.4.45 scx::load_application

This function loads an application in the memory of an instance.

```
void load_application(const std::string &instance, const std::string &application);
```

`instance`

the name of the instance to load into.

`application`

the application to load.

Note

The application loads at `start_of_simulation()`, at the earliest.

5.4.46 scx::load_data

This function loads raw data in the memory of an instance at a memory address.

```
void load_data(const std::string &instance,
               const std::string &data,
               const std::string &address);
```

`instance`

the name of the instance to load into.

`data`

the file name of the raw data to load.

`address`

the memory address at which to load the raw data. The parameter address might start with a memory space specifier.

Note

The raw data loads at `start_of_simulation()`, at the earliest.

5.4.47 scx::set_parameter

This function sets the value of a parameter from components present in the EVS.

-
  ```
  bool set_parameter(const std::string &name, const std::string &value);
  ```

-
  ```
  template<class T>
  bool set_parameter(const std::string &name, T value);
  ```

`name`

the name of the parameter to change.

`value`

the value of the parameter.

This function returns `true` when the parameter exists, `false` otherwise.

Note

- Changes made to parameters within System Canvas take precedence over changes made with `set_parameter()`.
- You can set parameters during the construction phase, and before the elaboration phase. Calls to `set_parameter()` after the construction phase are ignored.
- You can change run-time parameters after the construction phase with the debug interface.

5.4.48 scx::get_parameter

This function retrieves the value of a parameter from components present in the EVS.

-
  ```
  bool get_parameter(const std::string &name, std::string &value) const;
  ```

-
  ```
  template<class T>
  bool get_parameter(const std::string &name, T &value) const;
  ```

-
  ```
  std::string get_parameter(const std::string &name);
  ```

`name`

the name of the parameter to retrieve.

`value`

a reference to the value of the parameter.

The `bool` forms of the function return `true` when the parameter exists, `false` otherwise. The `std::string` form returns the value of the parameter when it exists, empty string (`""`) otherwise.

5.4.49 scx::get_parameter_list

This function retrieves a list of all parameters in all components present in the EVS.

```
std::map<std::string, std::string> get_parameter_list();
```

5.4.50 scx::scx_evs_base constructor

This function constructs an EVS.

```
scx_evs_base(const std::string &, sg::ComponentFactory *);
```

name

the name of the EVS instance.

factory

the sg::ComponentFactory to use to instantiate the corresponding LISA subsystem. The factory initializes within the generated derived class.

EVS instance names must be unique across the virtual platform. The EVS instance name initializes using the value passed as an argument to the constructor of the generated derived class.

5.4.51 scx::scx_evs_base destructor

This function destroys an EVS including the corresponding subsystem, and frees the associated resources.

```
~scx_evs_base();
```

5.4.52 scx::before_end_of_elaboration

This function calls the instantiate(), configure(), init(), interconnect(), and populateCADIMap() LISA behaviors of the corresponding exported subsystem.

```
void before_end_of_elaboration();
```

The generated derived class calls this function, after the SystemC simulation call-backs.

5.4.53 scx::end_of_elaboration

This function initializes the simulation framework.

```
void end_of_elaboration();
```

The generated derived class calls this function, after the SystemC simulation call-backs.

5.4.54 scx::start_of_simulation

This function calls the reset() LISA behaviors of the corresponding exported subsystem. It then loads applications.

```
void start_of_simulation();
```

The generated derived class calls this function, after the SystemC simulation call-backs.

5.4.55 scx::end_of_simulation

This function shuts down the simulation framework.

```
void end_of_simulation();
```

The generated derived class calls this function, after the SystemC simulation call-backs.

5.4.56 scx::scx_simcallback_if

This interface is the base class for simulation control call-backs.

```
class scx_simcallback_if {
  public:
    virtual void notify_running() = 0;
    virtual void notify_stopped() = 0;
    virtual void notify_debuggable() = 0;
    virtual void notify_idle() = 0;
   protected:
    virtual ~scx_simcallback_if() {
    }
};
```

The simulation framework implements this interface. The simulation controller uses the interface to notify the simulation framework of changes in the simulation state.

5.4.57 scx::notify_running

This function notifies the simulation framework that the simulation is running.

```
void notify_running();
```

The simulation controller calls this function to notify the simulation framework that the simulation is running. The simulation framework then notifies debuggers of the fact.

5.4.58 scx::notify_stopped

This function notifies the simulation framework that the simulation has stopped.

```
void notify_stopped();
```

The simulation controller calls this function to notify the simulation framework that the simulation has stopped. The simulation framework then notifies debuggers of the fact.

5.4.59 scx::notify_debuggable

This function notifies the simulation framework that the simulation is debuggable.

```
void notify_debuggable()
```

The simulation controller periodically calls this function to notify that the simulation is debuggable. This typically occurs while the simulation is stopped, to allow clients to process debug activity, for instance memory or breakpoint operations.

This version of the function does nothing.

5.4.60 scx::notify_idle

This function notifies the simulation framework that the simulation is idle.

```
void notify_idle();
```

The simulation controller periodically calls this function to notify the simulation framework that the simulation is idle, typically while the simulation is stopped, to allow clients to process background activity, for example, GUI events processing or redrawing.

5.4.61 scx::scx_simcallback_if destructor

Destructor.

```
~scx_simcallback_if();
```

This version of the function does not allow destruction of instances through the interface.

5.4.62 scx::scx_simcontrol_if

This is the simulation control interface.

```
class scx_simcontrol_if {
  public:
    virtual eslapi::CAInterface *get_scheduler() = 0;
    virtual scx_report_handler_if *get_report_handler() = 0;
    virtual void run() = 0;
    virtual void stop() = 0;
    virtual bool is_running() = 0;
    virtual void stop_acknowledge(sg::SchedulerRunnable *runnable) = 0;
    virtual void process_debuggable();
    virtual void notify_pending_debug();
    virtual void process_idle() = 0;
    virtual void shutdown() = 0;
    virtual void add_callback(scx_simcallback_if *callback_obj) = 0;
    virtual void remove_callback(scx_simcallback_if *callback_obj) = 0;
  protected:
    virtual ~scx_simcontrol_if();
};
```

The simulation controller, which interacts with the simulation framework, must implement this interface. The simulation framework uses this interface to access current implementations of the scheduler and report handler, as well as to request changes to the state of the simulation.

Unless otherwise stated, requests from this interface are asynchronous and can return immediately, whether the corresponding operation has completed or not. When the operation is complete, the corresponding notification must go to the simulation framework, which in turn notifies all connected debuggers to allow them to update their states.

Unless otherwise stated, an implementation of this interface must be thread-safe, that is it must not make assumptions about threads that issue requests.

The default implementation of the simulation controller provided with Fast Models is at: `$MAXCORE_HOME/lib/template/tpl_scx_simcontroller.{h,cpp}`.

5.4.63 scx::get_scheduler

This function returns a pointer to the implementation of the simulation scheduler.

```
eslapi::CAInterface *get_scheduler();
```

The simulation framework calls the `get_scheduler()` function to retrieve the scheduler implementation for the simulation at construction time.

Note

Implementations of this function need not be thread-safe.

5.4.64 scx::get_report_handler

This function returns a pointer to the current implementation of the report handler.

```
scx_report_handler_if *get_report_handler();
```

`scx_initialize()` calls the `get_report_handler()` function to retrieve the report handler implementation for the simulation at construction time.

Note

Implementations of this function need not be thread-safe.

5.4.65 scx::run

This function requests to run the simulation.

```
void run();
```

The simulation framework calls `run()` upon receipt of a CADI run request from a debugger.

5.4.66 scx::stop

This function requests to stop the simulation as soon as possible, that is at the next `wait()`.

```
void stop();
```

The simulation framework calls `stop()` upon receipt of a CADI stop request from a debugger, a component, or a breakpoint hit.

5.4.67 scx::is_running

This function returns whether the simulation is running.

```
bool is_running();
```

The return value is `true` when the simulation is running, `false` when it is paused or stopped.

The simulation framework calls `is_running()` upon receipt of a CADI run state request from a debugger.

5.4.68 scx::stop_acknowledge

This function blocks the simulation while the simulation is stopped.

```
void stop_acknowledge(sg::SchedulerRunnable *runnable);
```

`runnable`

a pointer to the scheduler thread calling `stop_acknowledge()`.

The scheduler thread calls this function to effectively stop the simulation, as a side effect of calling `stop()` to request that the simulation stop.

An implementation of this function must call `clearStopRequest()` on `runnable` (when not `NULL`).

5.4.69 scx::process_debuggable

This function processes debug activity while the simulation is at a debuggable point.

```
void process_debuggable()
```

This function is called by the scheduler thread whenever the simulation is at a debuggable point, to enable debug activity to be processed.

An implementation of this function might simply call `scx_simcallback_if::notify_debuggable()` on all registered clients.

This version of the function does nothing.

5.4.70 scx::notify_pending_debug

Notifies the simulation controller that debug requests are pending and need processing as soon as possible while the simulation is stopped.

```
virtual void notify_pending_debug()
```

An implementation of this behavior might simply call `scx_simcontrol::process_debuggable()` on all registered clients, while the simulation is stopped in `scx_simcontrol::stop_acknowledge()`.

5.4.71 scx::process_idle

This function processes idle activity while the simulation is stopped.

```
void process_idle();
```

The scheduler thread calls this function whenever idle to enable the processing of idle activity.

An implementation of this function might simply call `scx_simcallback_if::notify_idle()` on all registered clients.

5.4.72 scx::shutdown

This function requests to stop the simulation.

```
void shutdown();
```

The simulation framework calls `shutdown()` to notify itself that it wants the simulation to stop. Once the simulation has shut down it cannot run again.

Note

There are no call-backs associated with `shutdown()`.

5.4.73 scx::add_callback

This function registers call-backs with the simulation controller.

```
void add_callback(scx_simcallback_if *callback_obj);
```

`callback_obj`

a pointer to the object whose member functions serve as call-backs.

Clients call this function to register with the simulation controller a call-back object that handles notifications from the simulation.

5.4.74 scx::remove_callback

This function removes call-backs from the simulation controller.

```
void remove_callback(scx_simcallback_if *callback_obj);
```

`callback_obj`

a pointer to the object to remove.

Clients call this function to unregister a call-back object from the simulation controller.

5.4.75 scx::scx_simcontrol_if destructor

Destructor.

```
~scx_simcontrol_if();
```

This version of the function does not allow destruction of instances through the interface.

5.4.76 scx::scx_get_default_simcontrol

This function returns a pointer to the default implementation of the simulation controller provided with Fast Models.

```
scx_simcontrol_if *scx_get_default_simcontrol();
```

5.4.77 scx::scx_get_curr_simcontrol

Return a pointer to the current simulation controller implementation.

```
extern scx_simcontrol_if * scx_get_curr_simcontrol();
```

5.4.78 scx::scx_report_handler_if

This interface is the report handler interface.

```
class scx_report_handler_if {
  public:
    virtual void set_verbosity_level(int verbosity) = 0;
    virtual int get_verbosity_level() const = 0;
    virtual void report_info(const char *id,
                             const char *file,
                             int line,
                             const char *fmt, ...) = 0;
    virtual void report_info_verb(int verbosity,
                                   const char *id,
                                   const char *file,
                                   int line,
                                   const char *fmt, ...) = 0;
    virtual void report_warning(const char *id,
                                const char *file,
                                int line,
                                const char *fmt, ...) = 0;
    virtual void report_error(const char *id,
                              const char *file,
                              int line,
                              const char *fmt, ...) = 0;
    virtual void report_fatal(const char *id,
                              const char *file,
                              int line,
                              const char *fmt, ...) = 0;
  protected:
    virtual ~scx_report_handler_if() {
    }
};
```

This interface provides run-time reporting facilities, similar to the ones provided by SystemC. It has the additional ability to specify a format string in the same way as the `std::vprintf()` function, and associated variable arguments, for the report message.

The Fast Models simulation framework for SystemC Export uses this interface to report various messages at run-time.

The default implementation of the report handler provided with Fast Models is in: `$MAXCORE_HOME/lib/template/tpl_scx_report.cpp`.

Related information

IEEE Std 1666-2005, SystemC Language Reference Manual, 31 March 2006

5.4.79 scx::scx_get_default_report_handler

This function returns a pointer to the default implementation of the report handler provided with Fast Models.

```
scx_report_handler_if *scx_get_default_report_handler();
```

5.4.80 scx::scx_get_curr_report_handler

This function returns a pointer to the current implementation of the report handler.

```
scx_report_handler_if *scx_get_curr_report_handler();
```

5.4.81 scx::scx_sync

This function adds a future synchronization point.

```
void scx_sync(double sync_time);
```

`sync_time`

the time of the future synchronization point relative to the current simulated time, in seconds.

SystemC components call this function to hint to the scheduler when a system synchronization point will occur.

The scheduler uses this information to determine the quantum sizes of threads.

Threads that have run their quantum are unaffected; all other threads (including the current thread) run to the `sync_time` synchronization point.

Calling `scx_sync()` again adds another synchronization point.

Synchronization points automatically vanish when the simulation time passes.

Warning

Arm deprecates this function. Use IEEE 1666 SystemC 2011 `sc_core::sc_prim_channel::async_request_update()` instead.

5.4.82 scx::scx_set_min_sync_latency

This function sets the minimum synchronization latency for this scheduler.

```
void scx_set_min_sync_latency(double t);
void scx_set_min_sync_latency(sg::ticks_t t);
```

`t`

the minimum synchronization latency. Measured in seconds.

The minimum synchronization latency helps to ensure that sufficient simulated time has passed between two synchronization points for synchronization to be efficient.

A small latency increases accuracy but decreases simulation speed.

A large latency decreases accuracy but increases simulation speed.

The scheduler uses this information to compute the next synchronization point as returned by `sg::SchedulerInterfaceForComponents::getNextSyncPoint()`.

Related information

scx::scx_get_min_sync_latency on page 75

5.4.83 scx::scx_get_min_sync_latency

This function returns the minimum synchronization latency, measured in seconds, for this scheduler.

```
double scx_get_min_sync_latency();
sg::ticks_t scx_get_min_sync_latency(sg::Tag<sg::ticks_t> *);
```

Related information

scx::scx_set_min_sync_latency on page 75

5.4.84 scx::scx_simlimit

This function sets the maximum number of seconds to simulate.

```
void scx_simlimit(double t);
```

t

the number of seconds to simulate. Defaults to unlimited.

5.4.85 scx::scx_create_default_scheduler_mapping

This function returns a pointer to a new instance of the default implementation of the scheduler mapping that is provided with Fast Models.

```
sg::SchedulerInterfaceForComponents *
 scx_create_default_scheduler_mapping(scx_simcontrol_if * simcontrol);
```

simcontrol

pointer to an existing simulation controller. When this is `NULL`, this function returns `NULL`.

5.4.86 scx::scx_get_curr_scheduler_mapping

This function returns a pointer to the current implementation of the scheduler mapping interface.

```
sg::SchedulerInterfaceForComponents * scx_get_curr_scheduler_mapping();
```

5.5 Scheduler API

The Fast Models Scheduler API makes modeling components and systems accessible in different environments, with or without a built-in scheduler. Examples are a SystemC environment or a standalone simulator.

The Fast Models Scheduler API is a C++ interface consisting of a set of abstract base classes. The header file that defines them is `$PVLIB_HOME/include/fmruntime/sg/SGSchedulerInterfaceForComponents.h`. This header file depends on other header files under `$PVLIB_HOME/include`.

All Scheduler API constructs are in the namespace `sg`.

The interface decouples the modeling components from the scheduler implementation. The parts of the Scheduler API that the modeling components use are for the scheduler or scheduler adapter to implement. The parts that the scheduler or scheduler adapter use are for the modeling components to implement.

class SchedulerInterfaceForComponents

The scheduler (or an adapter to the scheduler) must implement an instance of this interface class for Fast Models components to work. Fast Models components use this interface to talk to the scheduler, for example, to create threads and timers. This class is the main part of the interface.

`class SchedulerThread`

An abstract Fast Models thread class, which `createThread()` creates instances of. For example, CT core models use this class. The scheduler implements it. Threads have co-routine semantics.

`class SchedulerRunnable`

The counterpart of the `SchedulerThread` class. The modeling components, which contain the thread functionality, implement it.

`class ThreadSignal`

A class of event that threads can wait on. It has `wait()` and `notify()` but no timing functions. The scheduler implements it.

`class Timer`

An abstract interface for one-shot or continuous timed events, which `createTimer()` creates instances of. The scheduler implements it.

`class TimerCallback`

The counterpart of the `Timer` class. The modeling components, which contain the functionality for the timer callback, implement it. Arm deprecates this class.

`class SchedulerCallback`

A callback function class. The modeling components, which use `addCallback()` (asynchronous callbacks), implement it.

`class FrequencySource`

An abstract interface class that provides a frequency in Hz. The modeling components implement it. The scheduler uses it to determine the time intervals for timed events. Arm deprecates this class.

`class FrequencyObserver`

An abstract interface class for observing a `FrequencySource` and changes to the frequency value. The scheduler implements it for objects that have access to a `FrequencySource` (`Timer` and `SchedulerThread`). Arm deprecates this class.

`class SchedulerObject`

The base class for all scheduler interface objects, which provides `getName()`.

5.5.1 Accessing SchedulerInterfaceForComponents from a modeling component

This topic shows ways of accessing the SchedulerInterfaceForComponents interface from a LISA, C ++, and SystemC component.

LISA component

```
includes
{
 #include "sg/SGSchedulerInterfaceForComponents.h"
 #include "sg/SGComponentRegistry.h"
}

behavior init
```

```
{
 sg::SchedulerInterfaceForComponents *scheduler =
   sg::obtainComponentInterfacePointer<sg::SchedulerInterfaceForComponents>
     (getGlobalInterface(), "scheduler");
}
```

C++ component

C++ components have an `sg::SimulationContext` pointer passed into their constructor.

```
#include "sg/SGSchedulerInterfaceForComponents.h"
#include "sg/SGComponentRegistry.h"

sg::SchedulerInterfaceForComponents *scheduler =
  sg::obtainComponentInterfacePointer<sg::SchedulerInterfaceForComponents>
    (simulationContext->getGlobalInterface(), "scheduler");
```

SystemC component

```
#include "sg/SGSchedulerInterfaceForComponents.h"
#include "sg/SGComponentRegistry.h"

sg::SchedulerInterfaceForComponents *scheduler =
  sg::obtainComponentInterfacePointer<sg::SchedulerInterfaceForComponents>
    (scx::scx_get_global_interface(), "scheduler");
```

5.5.2 Intended mapping of the Scheduler API onto SystemC/TLM

How Scheduler API functionality might map onto SystemC functionality.

`sg::SchedulerInterfaceForComponents::wait(time)`

Call `sc_core::wait(time)` and handle all pending asynchronous events that are scheduled with `sg::SchedulerInterfaceForComponents::addCallback()` before waiting.

`sg::SchedulerInterfaceForComponents::wait(sg::ThreadSignal)`

Call `sc_core::wait(sc_event)` on the `sc_event` in `sg::ThreadSignal` and handle all pending asynchronous events that are scheduled with `sg::SchedulerInterfaceForComponents::addCallback()` before waiting.

`sg::SchedulerInterfaceForComponents::getCurrentSimulatedTime()`

Return the current SystemC scheduler time in seconds as in `sc_core::sc_time_stamp().to_seconds()`.

`sg::SchedulerInterfaceForComponents::addCallback(), removeCallback()`

SystemC has no way to trigger simulation events from alien (non-SystemC) host threads in a thread-safe way: buffer and handle these asynchronous events in all regularly re-occurring scheduler events. Handling regular simulation `wait()` and `timerCallback()`calls is sufficient.

`sg::SchedulerInterfaceForComponents::stopRequest(), stopAcknowledge()`

Pause and resume the SystemC scheduler. This function is out of scope of SystemC/TLM functionality, but in practice every debuggable SystemC implementation has ways to pause and resume the scheduler. Do not confuse these functions with `sc_core::sc_stop()`, which exits the SystemC simulation loop. They work with the `sg::SchedulerRunnable` instances and the `scx::scx_simcontrol_if` interface.

`sg::SchedulerInterfaceForComponents::createThread(), createThreadSignal(), createTimer()`

Map these functions onto SystemC threads created with `sc_spawn()` and `sc_event`s. You can create and destroy `sg::SchedulerThread`, `sg::ThreadSignal`, and `sg::Timer` objects during elaboration, and delete them at runtime, unlike their SystemC counterparts. This process requires careful mapping. For example, consider what happens when you remove a waited-for `sc_event`.

`sg::ThreadSignal`

Map onto `sc_event`, which is notifiable and waitable.

`sg::SchedulerThread`

Map onto a SystemC thread that was spawned with `sc_core::sc_spawn()`. The thread function can call `sg::SchedulerThread::threadProc()`.

`sg::QuantumKeeper`

Map onto the `tlm_quantumkeeper` utility class because the semantics of these classes are similar. Arm deprecates this class.

`sg::Timer`

Map onto a SystemC thread that, after the timer is `set()`, issues calls to the call-backs in the intervals (according to the `set()` interval).

5.5.3 sg::SchedulerInterfaceForComponents class

The modeling components use this interface class, which gives access to all other parts of the Scheduler API, directly or indirectly. The scheduler must implement this class.

```
// Main scheduler interface class
class sg::SchedulerInterfaceForComponents
{
public:
    static eslapi::if_name_t IFNAME() { return
 "sg.SchedulerInterfaceForComponents"; }
    static eslapi::if_rev_t IFREVISION() { return 1; }
    virtual eslapi::CAInterface * ObtainInterface(eslapi::if_name_t,
 eslapi::if_rev_t, eslapi::if_rev_t *) = 0;
    virtual sg::Timer * createTimer(const char *, sg::TimerCallback *) = 0;
    virtual sg::SchedulerThread * createThread(const char *, sg::SchedulerRunnable
 *) = 0;
    virtual sg::SchedulerThread * currentThread();
    virtual sg::ThreadSignal * createThreadSignal(const char *) = 0;
    virtual void wait(sg::ticks_t);
    virtual void wait(sg::ThreadSignal *) = 0;
    virtual void setGlobalQuantum(sg::ticks_t);
    virtual sg::ticks_t getGlobalQuantum(sg::Tag<sg::ticks_t> *);
    virtual double getGlobalQuantum();
    virtual void setMinSyncLatency(sg::ticks_t);
    virtual sg::ticks_t getMinSyncLatency(sg::Tag<sg::ticks_t> *);
    virtual double getMinSyncLatency();
    virtual void addSynchronisationPoint(sg::ticks_t);
    virtual sg::ticks_t getNextSyncPoint(sg::Tag<sg::ticks_t> *);
    virtual double getNextSyncPoint();
    virtual void getNextSyncRange(sg::ticks_t &, sg::ticks_t &);
    virtual void getNextSyncRange(double&, double&);
    virtual void addCallback(sg::SchedulerCallback *) = 0;
    virtual void removeCallback(sg::SchedulerCallback *) = 0;
    virtual sg::ticks_t getCurrentSimulatedTime(sg::Tag<sg::ticks_t> *);
    virtual double getCurrentSimulatedTime();
    virtual double getSimulatedTimeResolution();
    virtual void setSimulatedTimeResolution(double resolution);
    virtual void stopRequest() = 0;
    virtual void stopAcknowledge(sg::SchedulerRunnable *) = 0;
};
```

Note

Pass a null pointer to the extra `Tag<>` argument in `getGlobalQuantum()`, `getMinSyncLatency()`, `getNextSyncPoint()`, and `getCurrentSimulatedTime()`.

Arm deprecates these API functions:

```
virtual void wait(sg::ticks_t, sg::FrequencySource *)
virtual void setGlobalQuantum(sg::ticks_t, sg::FrequencySource *)
virtual void setMinSyncLatency(sg::ticks_t, sg::FrequencySource *)
virtual void addSynchronisationPoint(sg::ticks_t, sg::FrequencySource *)
```

Arm deprecates classes `sg::FrequencySource` and `sg::FrequencyObserver`. Modeling components must not use these classes to directly communicate with the Scheduler API. Use the `sg::Time` class instead.

Modeling components use this interface to create threads, asynchronous and timed events, system synchronization points, and to request a simulation stop. Examples of components that access this interface are:

- CT core models.
- Timer peripherals.
- Peripheral components with timing or that indicate system synchronization points.
- Peripheral components that can stop the simulation for certain conditions (external breakpoints).
- GUI components.

Passive components that do not interact with the scheduler (and that do not need explicit scheduling) usually do not access this interface.

Related information

Accessing SchedulerInterfaceForComponents from a modeling component on page 77

5.5.3.1 eslapi::CAInterface and eslapi::ObtainInterface

The `CAInterface` base class and the `ObtainInterface()` function make the interface discoverable at runtime through a runtime mechanism. All interfaces in Fast Models that must be discoverable at runtime derive from `CAInterface`.

The functions `IFNAME()`, `IFREVISION()`, and `ObtainInterface()` belong to the base class `eslapi::CAInterface`. `IFNAME()` and `IFREVISION()` return static information (name and revision) about the interface (not the interface implementation). An implementation of the interface cannot re-implement these functions. To access this interface, code must pass these two values to the `ObtainInterface()` function to acquire the `SchedulerInterfaceForComponents`.

Use `ObtainInterface()` to access the interfaces that the scheduler provides. As a minimum requirement, the implementation of `ObtainInterface()` must provide the `SchedulerInterfaceForComponents` interface itself and also the `eslapi::CAInterface` interface. The easiest way to provide these interfaces to use the class `eslapi::CAInterfaceRegistry` and register these two interfaces and forward all `ObtainInterface()` calls to this registry. See the default implementation of the Scheduler API over SystemC for an example.

Note

`CAInterface` and `ObtainInterface()` are not part of the scheduler functionality but rather of the simulation infrastructure. The information here is what is necessary to understand and implement `ObtainInterface()`. For more details on the `eslapi::CAInterface` class, see the header file `$PVLIB_HOME/include/fmruntime/eslapi/CAInterface.h`.

5.5.3.2 sg::SchedulerInterfaceForComponents::addCallback

This method schedules a callback in the simulation thread. `AsyncSignal` uses it.

```
virtual void addCallback(SchedulerCallback *callback)=0;
```

callback

Callback object to call. If `callback` is `NULL`, the call has no effect.

Any host thread can call this method. It is thread safe. It is always the simulation thread (host thread which runs the simulation) that calls the callback function (`callback->schedulerCallback()`). The scheduler calls the callback function when it can respond to the `addCallback()` function.

Multiple callbacks might be pending. The scheduler can call them in any order. Do not call `addCallback()` or `removeCallback()` from a callback function.

Callbacks automatically vanish once called. Removing them deliberately is not necessary unless they become invalid, for example on the destruction of the object implementing the callback function.

Related information

sg::SchedulerInterfaceForComponents::removeCallback on page 85

5.5.3.3 sg::SchedulerInterfaceForComponents::addSynchronisationPoint

This method adds synchronization points.

```
virtual void addSynchronisationPoint(ticks_t ticks);
```

ticks

Simulated time for synchronization relative to the current simulated time, in ticks relative to simulated time resolution.

Modeling components can call this function to hint to the scheduler when a potentially useful system synchronization point will occur. The scheduler uses this information to determine the quantum sizes of threads.

Calling this function again adds another synchronization point.

Synchronization points automatically vanish when reached.

5.5.3.4 sg::SchedulerInterfaceForComponents::createThread

CT core models and modeling components call this method to create threads. This method returns an object implementing `SchedulerThread`. (Not `NULL` except when `runnable` is `NULL`.)

```
virtual SchedulerThread *createThread(const char *name, SchedulerRunnable
 *runnable)=0;
```

name

Instance name of the thread. Ideally, the hierarchical name of the component that owns the thread is included in the name. If `name` is `NULL`, it receives the name '`(anonymous thread)`'. The function makes a copy of `name`.

runnable

Object that implements the `SchedulerRunnable` interface. This object is the one that contains the actual thread functionality. The returned thread uses this interface to communicate with the thread implementation in the modeling component. If `runnable` is `NULL`, the call returns `NULL`, which has no effect.

Having created the thread, start it with a call to `SchedulerThread::start()`.

Destroying the returned object with the `SchedulerThread` destructor might not kill the thread.

Related information

sg::SchedulerInterfaceForComponents::currentThread on page 83

sg::SchedulerRunnable class on page 88
sg::SchedulerThread class on page 92
sg::SchedulerThread::destructor on page 92
sg::SchedulerThread::start on page 93

5.5.3.5 sg::SchedulerInterfaceForComponents::createThreadSignal

CT core models use this method to create thread signals. A thread signal is a nonschedulable event that threads wait for. Giving the signal schedules all waiting threads to run.

```
virtual ThreadSignal* createThreadSignal(const char* name)=0;
```

`name`

Instance name of the thread. Ideally, the hierarchical name of the component that owns the thread is included in the name. If `name` is `NULL`, it receives the name '`(anonymous thread signal)`'. The function makes a copy of `name`.

Destroying the returned object while threads are waiting for it leaves the threads unscheduled.

5.5.3.6 sg::SchedulerInterfaceForComponents::createTimer

Modeling components call this method to create objects of class `Timer`. They use timers to trigger events in the future (one-shot or repeating events).

```
virtual Timer* createTimer(const char* name, TimerCallback* callback)=0;
```

5.5.3.7 sg::SchedulerInterfaceForComponents::currentThread

This method returns the currently running scheduler thread, which `createThread()` created, or null if not in any `threadProc()` call.

```
virtual SchedulerThread* currentThread();
```

Related information

sg::SchedulerInterfaceForComponents::createThread on page 82

5.5.3.8 sg::SchedulerInterfaceForComponents::getCurrentSimulatedTime

This method returns the simulated time in `ticks` relative to simulated time resolution, since the creation of the scheduler. `ClockDivider` and `MasterClock(ClockSignalProtocol::currentTicks())` use it.

```
virtual ticks_t getCurrentSimulatedTime(Tag<ticks_t>*);
```

This clock accurately reflects the time on the last timer callback invocation or the last return from `SchedulerThread::wait()`, whichever was last. The return values monotonically increase over (real or simulated) time.

5.5.3.9 sg::SchedulerInterfaceForComponents::getGlobalQuantum

This method returns the global quantum in `ticks` relative to simulated time resolution.

```
virtual ticks_t getGlobalQuantum(Tag<ticks_t>*);
```

Related information

sg::SchedulerInterfaceForComponents::setGlobalQuantum on page 85

5.5.3.10 sg::SchedulerInterfaceForComponents::getMinSyncLatency

This method returns the minimum synchronization latency in ticks relative to simulated time resolution.

```
virtual ticks_t getMinSyncLatency(Tag<ticks_t>*);
```

Related information

sg::SchedulerInterfaceForComponents::setMinSyncLatency on page 85

5.5.3.11 sg::SchedulerInterfaceForComponents::getNextSyncPoint

This method returns the next synchronization point relative to the current simulated time. The next synchronization point is expressed in `ticks` relative to simulated time resolution.

```
virtual ticks_t getNextSyncPoint(Tag<ticks_t>*);
```

Modeling components can call this function for a hint about when a potentially useful system synchronization point will occur. Core threads use this information to determine when to synchronize.

5.5.3.12 sg::SchedulerInterfaceForComponents::getSimulatedTimeResolution

This method returns the simulated time resolution in seconds.

```
virtual double getSimulatedTimeResolution();
```

5.5.3.13 sg::SchedulerInterfaceForComponents::removeCallback

This method removes all callbacks that are scheduled using `addCallback()` for this callback object. `AsyncSignal` uses it.

```
virtual void removeCallback(SchedulerCallback *callback)=0;
```

`callback`

The callback object to remove. If `callback` is `NULL`, an unknown callback object, or a called callback, then the call has no effect.

Any host thread can call this method. It is thread safe.

The scheduler will not call the specified callback after this function returns. It can, however, call it while execution control is inside this function.

Callbacks automatically vanish after being called. Removing them deliberately is not necessary unless they become invalid, for example on the destruction of the object implementing the callback function.

Related information

sg::SchedulerInterfaceForComponents::addCallback on page 81

5.5.3.14 sg::SchedulerInterfaceForComponents::setGlobalQuantum

This method sets the global quantum.

```
virtual void setGlobalQuantum(ticks_t ticks);
```

`ticks`

Global quantum value, relative to simulated time resolution. The global quantum is the maximum time that a thread can run ahead of simulation time.

All threads must synchronize on timing points that are multiples of the global quantum.

Related information

sg::SchedulerInterfaceForComponents::getGlobalQuantum on page 84

5.5.3.15 sg::SchedulerInterfaceForComponents::setMinSyncLatency

This method sets the minimum synchronization latency.

```
virtual void setMinSyncLatency(ticks_t ticks);
```

`ticks`

Minimum synchronization latency value, relative to simulated time resolution.

The minimum synchronization latency helps to ensure that sufficient simulated time has passed between two synchronization points for synchronization to be efficient. A small latency increases accuracy but decreases simulation speed. A large latency decreases accuracy but increases simulation speed.

The scheduler uses this information to set the minimum synchronization latency of threads with `sg::SchedulerRunnable::setThreadProperty()`, and to compute the next synchronization point as returned by `getNextSyncPoint()`.

Related information

sg::SchedulerInterfaceForComponents::getMinSyncLatency on page 84

5.5.3.16 sg::SchedulerInterfaceForComponents::setSimulatedTimeResolution

This method sets the simulated time resolution in seconds.

```
virtual void setSimulatedTimeResolution(double resolution);
```

`resolution`

Simulated time resolution in seconds.

Setting simulated time resolution after the start of the simulation or after setting timers is not possible.

5.5.3.17 sg::SchedulerInterfaceForComponents::stopAcknowledge

This function blocks the simulation thread until being told to resume.

```
virtual void stopAcknowledge(SchedulerRunnable *runnable)=0;
```

`runnable`

Pointer to the runnable instance that called this function, or `NULL` when not called from a runnable instance. If not `NULL` this function calls `runnable->clearStopRequest()` once it is safe to do so (with respect to non-simulation host threads).

CT core models call this function from within the simulation thread in response to a call to `stopRequest()` or spontaneously (for example, breakpoint hit, debugger stop). The call must always be from the simulation thread. The scheduler must block inside this function. The function must return when the simulation is to resume.

The scheduler usually implements a thread-safe mechanism in this function that allows blocking and resuming of the simulation thread from another host thread (usually the debugger thread).

Calling this function from a nonsimulation host thread is wrong by design and is forbidden.

This function must clear the stop request that led to calling this function by calling `runnable->clearStopRequest()`.

This function must have no effects other than blocking the simulation thread.

5.5.3.18 sg::SchedulerInterfaceForComponents::stopRequest

This function requests the simulation of the whole system to stop (pause).

```
virtual void stopRequest()=0;
```

You can call this function from any host thread, whether the simulation is running or not. The function returns immediately, possibly before the simulation stops. This function will not block the caller until the simulation stops. The simulation stops as soon as possible, depending on the `syncLevel` of the threads in the system. The simulation calls the function `stopAcknowledge()`, which blocks the simulation thread to pause the simulation. This function must not call `stopAcknowledge()` directly. It must only set up the simulation to stop at the next sync point, defined by the `syncLevels` in the system. Reset this state with `stopAcknowledge()`, which calls `SchedulerRunnable::clearStopRequest()`.

Debuggers and modeling components such as CT cores and peripherals use this function to stop the simulation from within the simulation thread (for example for external breakpoints) and also asynchronously from the debugger thread. Calling this function again (from any host thread) before `stopAcknowledge()` has reset the stop request, using `SchedulerRunnable::clearStopRequest()` is harmless. The simulation only stops once.

Note

The simulation can stop (that is, call `stopAcknowledge()`) spontaneously without a previous `stopRequest()`. This stop happens for example when a modeling component hits a breakpoint. A `stopRequest()` is sufficient, but not necessary, to stop the simulation.

The scheduler implementation of this function is to forward this `stopRequest()` to the running runnable object, but only for `stopRequest()` calls from the simulation thread. When the runnable object accepts the `stopRequest()` (`SchedulerRunnable::stopRequest()` returns `true`), the scheduler need do nothing more because the runnable object will respond with a `stopAcknowledge()` call. If the runnable object did not accept the `stopRequest()`

(`SchedulerRunnable::stopRequest()` returns `false`) or if this function call is outside of the context of a runnable object (for example, from a call-back function) or from a non-simulation host thread, then the scheduler is responsible for handling the `stopRequest()` itself by calling `stopAcknowledge()` as soon as possible.

The stop handling mechanism should not change the scheduling order or model behavior (non-intrusive debugging).

Related information

sg::SchedulerRunnable::stopRequest on page 91

5.5.3.19 sg::SchedulerInterfaceForComponents::wait(ThreadSignal)

This method waits on a thread signal.

```
virtual void wait(ThreadSignal* threadSignal)=0;
```

`threadSignal`

Thread signal object to wait for. A call with `threadSignal` of `NULL` is valid, but has no effect.

`wait()` blocks the current thread until it receives `ThreadSignal::notify()`. This function returns when the calling thread can continue to run.

Only call this method from within a `SchedulerRunnable::threadProc()` context. Calling this method from outside of a `threadProc()` context is valid, but has no effect.

5.5.3.20 sg::SchedulerInterfaceForComponents::wait(ticks_t)

This method blocks the running thread and runs other threads for a specified time.

```
virtual void wait(ticks_t ticks);
```

`ticks`

Time to wait for, in `timebase` units. `ticks` can be `0`.

Only call this method from within a `SchedulerRunnable::threadProc()` context. Calls from outside of a `threadProc()` context are valid, but have no effect.

This method blocks a thread for a time while the other threads run. It returns when the calling thread is to continue, at the co-routine switching point. Typically, a thread calls `wait(ticks)` in its loop when it completes `ticks` ticks of work. `ticks` is a “quantum”.

5.5.4 sg::SchedulerRunnable class

This class is a thread interface on the runnable side. The modeling components create and implement `SchedulerRunnable` objects and pass a pointer to a `SchedulerRunnable` interface to `SchedulerInterfaceForComponents::createThread()`. The scheduler uses this interface to run the thread.

Related information

sg::SchedulerInterfaceForComponents::createThread on page 82

5.5.4.1 sg::SchedulerRunnable::breakQuantum

This function breaks the quantum. Arm deprecates this function.

5.5.4.2 sg::SchedulerRunnable::clearStopRequest

This function clears stop request flags.

```
void clearStopRequest();
```

Only `SchedulerInterfaceForComponents::stopAcknowledge()` calls this function, so calls are always from the simulation thread.

Related information

sg::SchedulerRunnable::stopRequest on page 91

5.5.4.3 sg::SchedulerRunnable::getName

This function returns the name of the instance that owns the object.

```
const char *getName() const;
```

By convention, this is the name that `createThread()` received. `SchedulerRunnable` inherits this function from `sg::SchedulerObject`.

5.5.4.4 sg::SchedulerRunnable::setThreadProperty, sg::SchedulerRunnable::getThreadProperty

These functions set and get thread properties.

```
bool setThreadProperty(ThreadProperty property, uint64_t value);
bool getThreadProperty(ThreadProperty property, uint64_t &valueOut);
```

Scheduler-configures-runnable properties

`TP_BREAK_QUANTUM`

Arm deprecates this property.

`SchedulerInterfaceForComponents::getNextSyncPoint()` gives the next quantum size.

`TP_DEFAULT_QUANTUM_SIZE`

Arm deprecates this property. Use `SchedulerInterfaceForComponents::set/getGlobalQuantum()`.

`TP_COMPILER_LATENCY`

`set`

Compiler latency, the maximum interval in which generated straight-line code checks for signals and the end of the quantum.

`get`

Compiler latency.

`default`

1024 instructions.

`TP_MIN_SYNC_LATENCY`

`set`

Synchronization latency, the minimum interval in which generated straight-line code inserts synchronization points.

`get`

Synchronization latency.

`default`

64 instructions.

`TP_MIN_SYNC_LEVEL`

`set`

`syncLevel` to at least *`N`* (0-3).

`get`

Minimum `syncLevel`.

`default`

`min_sync_level` CADI parameter and the `syncLevel`* registers also determine the `syncLevel`. If nothing else is set, the default is 0 (`SL_OFF`).

`TP_LOCAL_TIME`

`set`

Local time of temporally decoupled thread.

`get`

Current local time.

`TP_LOCAL_QUANTUM`

`set`

Local quantum of temporally decoupled thread.

`get`

Current local quantum.

Note

The temporally decoupled thread usually retrieves the local quantum by calling `SchedulerInterfaceForComponents::getNextSyncPoint()`.

Runnable-configures-scheduler properties

`TP_STACK_SIZE`

`set`

Return `false` and ignore the value. Not for a scheduler to call.

`get`

Intended stack size for the thread in bytes. If this field returns `false` or a low value, this field uses the default stack size that the scheduler determines. Not all schedulers use this field. If a scheduler supports setting the stack size, it requests this field from `SchedulerInterfaceForComponents::createThread()` or `SchedulerThread::start()`. Is to return a constant value.

`default`

2MB.

Schedulers need not use all fields, and runnable objects need not provide all fields. If a runnable object does not support a property or value, it must return `false`.

Related information

sg::SchedulerRunnable::breakQuantum on page 89

5.5.4.5 sg::SchedulerRunnable::stopRequest

This function requests the simulation of the whole system to stop (pause) as soon as possible by setting a request flag. This might be to inspect a runnable, for example to pause at an instruction boundary to inspect a processor component with a debugger.

```
bool stopRequest();
```

You can call this function from any host thread, whether the simulation is running or not. The function returns immediately, before the simulation stops. This function will not block the caller until the simulation stops. The simulation stops as soon as possible, depending on the `syncLevel` of the runnable.The simulation calls the function `SchedulerInterfaceForComponents::stopAcknowledge()`, which blocks the simulation thread to pause the simulation. The function must not call `stopAcknowledge()` directly but only set up a state such that the simulation stops at the next sync point, defined by the `syncLevel` of this runnable. Reset this state with `stopAcknowledge()`, which calls `clearStopRequest()`.

Modeling components use this function to stop the simulation from within the simulation thread (for example for external breakpoints) and also asynchronously from from the debugger thread. Calling this function again (from any host thread) before `stopAcknowledge()` has reset the stop request using `clearStopRequest()` is harmless. The simulation only stops once.

Returns `true` when the runnable accepts the stop request and will stop later. Returns `false` when the runnable does not accept the stop request. In this case, the scheduler must stop the simulation when the runnable returns control to the scheduler (for example, by use of `wait()`).

Related information

sg::SchedulerRunnable::clearStopRequest on page 89

5.5.4.6 sg::SchedulerRunnable::threadProc

This is the main thread function, the thread entry point.

```
void threadProc();
```

When `threadProc()` returns, the thread no longer runs and this `SchedulerThread` instance will not call `threadProc()` again. The thread usually does not return from this function while the thread is running.

`threadProc()` is to call `SchedulerInterfaceForComponents::wait(0, ...)` after completing initialization. `threadProc()` is to call `SchedulerInterfaceForComponents::wait(t>=0, ...)` after completing `t` ticks worth of work.

Do not create/destroy any other threads or scheduler objects within the context of this function.

5.5.5 sg::SchedulerThread class

This class is a thread interface on the thread instance/scheduler side. The `SchedulerInterfaceForComponents::createThread()` function creates the `SchedulerThread` objects. Modeling components use this interface to talk to the scheduler.

Related information

sg::SchedulerInterfaceForComponents::createThread on page 82

5.5.5.1 sg::SchedulerThread::destructor

This method destroys `SchedulerThread` objects.

```
~SchedulerThread();
```

This destructor kills threads if the underlying scheduler implementation supports it. Killing threads without their cooperation is unclean because it might leak resources. To end a thread cleanly, signal the thread to return from its `threadProc()` function, for example by using an exception that is caught in `threadProc()`. Destroying this object before calling `start()` must not start the thread. Destroying this object after calling `start()` might kill the thread immediately or leave it running until it returns from its `threadProc()`.

`SchedulerThread` inherits this method from `sg::SchedulerObject`.

Related information

sg::SchedulerInterfaceForComponents::createThread on page 82

5.5.5.2 sg::SchedulerThread::getName

This method returns the name of the instance that owns the object.

```
const char *getName() const;
```

This is the name that `createThread()` received.

`SchedulerThread` inherits this method from `sg::SchedulerObject`.

5.5.5.3 sg::SchedulerThread::setFrequency

This method sets the frequency source to be the parent clock for the thread. Arm deprecates this function.

5.5.5.4 sg::SchedulerThread::start

This method starts the thread.

```
void start();
```

This method calls the `threadProc()` function immediately, which must call `wait(0, ...)` after initialization in order for `start()` to return. `start()` only runs the `threadProc()` of the associated thread and no other threads. Calling `start()` on a running thread has no effect. Calling `start()` on a terminated thread (`threadProc()` returned) has no effect.

Note

The modeling component counterpart of the `sg::SchedulerThread` class is `sg::SchedulerRunnable`. Runnable objects must call `sg::QuantumKeeper::sync()` regularly to pass execution control on to other threads.

Related information

sg::SchedulerInterfaceForComponents::createThread on page 82

5.5.6 sg::ThreadSignal class

This section describes the `ThreadSignal` class. It represents a nonschedulable event on which threads can wait. When the event is signaled, all waiting threads can run.

5.5.6.1 sg::ThreadSignal::destructor

This method destroys `ThreadSignal` objects, thread signals.

```
~ThreadSignal();
```

Destroying these objects while threads are waiting for them leaves the threads unscheduled.

5.5.6.2 sg::ThreadSignal::notify

This method notifies the system of the event, waking up any waiting threads.

```
void notify();
```

`SchedulerRunnable::threadProc()` can call this method, but calls can come from outside of `threadProc()`. Calling this method when no thread is waiting for the signal is valid, but has no effect.

5.5.6.3 sg::ThreadSignal::getName

This method returns the name of the instance that owns the object.

```
const char *getName() const;
```

This is the name that `createThreadSignal()` received.

`ThreadSignal` inherits this method from `sg::SchedulerObject`.

5.5.7 sg::Timer class

This section describes the `Timer` interface class. The `SchedulerInterfaceForComponents::createTimer()` method creates `Timer` objects.

5.5.7.1 sg::Timer::cancel

This method unsets the timer so that it does not fire.

```
void cancel();
```

If the timer is not set, this method has no effect.

5.5.7.2 sg::Timer::destructor

This method destroys `Timer` objects.

```
~Timer();
```

The timer must not call `TimerCallback::timerCallback()` after the destruction of this object.

5.5.7.3 sg::Timer::getName

This method returns the name of the instance that owns the object.

```
const char *getName() const;
```

This is the name that `createTimer()` received.

`Timer` inherits this method from `sg::SchedulerObject`.

5.5.7.4 sg::Timer::isSet

This method returns `true` if the timer is set and queued for call-back, otherwise `false`.

```
bool isSet();
```

This method has no side effects.

5.5.7.5 sg::Timer::remaining

This method requests the remaining number of ticks relative to simulated time resolution until a timer makes a signal.

```
ticks_t remaining();
```

This method returns 0 if there are no ticks remaining or if the timer is not set.

This method has no side effects.

5.5.7.6 sg::Timer::set

This method sets a timer to make a signal.

```
bool set(ticks_t ticks);
```

ticks

the number of ticks after which the timer is to make a signal.

The signal that this method makes is a call to the user call-back function. If the return value *t* is 0, the timer does not repeat, otherwise it repeats after *t* ticks. The latest `set()` overrides the previous one.

This method returns `false` if `ticks` is too big to schedule the timer.

5.5.7.7 sg::Timer::setFrequency

This method sets the frequency source clock for the timer. Arm deprecates this function. Simulated time is relative to global time resolution. See `SchedulerInterfaceForComponents::getSimulatedTimeResolution()` and `SchedulerInterfaceForComponents::setSimulatedTimeResolution()`.

5.5.8 sg::TimerCallback class

This section describes the `TimerCallback` base class. This interface does not allow object destruction.

5.5.8.1 sg::TimerCallback::getName

This method returns the name of the instance that owns the object.

```
const char *getName() const;
```

Conventionally, this is the name that `createTimer()` received.

`TimerCallback` inherits this method from `sg::SchedulerObject`.

5.5.8.2 sg::TimerCallback::timerCallback

The `createTimer()` method receives a `timerCallback` instance. This `timerCallback()` method is called whenever the timer expires. This method returns a value *t*. If *t* is `0`, the timer does not repeat, otherwise it is to call `timerCallback()` again after *t* ticks.

```
ticks_t timerCallback();
```

5.5.9 sg::FrequencySource class

`FrequencySource` objects provide clock frequencies, and notify frequency observers of frequency changes. This interface does not allow object destruction. Arm deprecates this class. Simulated time is relative to global time resolution. See `SchedulerInterfaceForComponents::getSimulatedTimeResolution()` and `SchedulerInterfaceForComponents::setSimulatedTimeResolution()`.

5.5.10 sg::FrequencyObserver class

`FrequencySource` instances notify `FrequencyObserver` instances of `FrequencySource` instance changes. This interface does not allow object destruction. Arm deprecates this class. Simulated time is relative to global time resolution. See `SchedulerInterfaceForComponents::getSimulatedTimeResolution()` and `SchedulerInterfaceForComponents::setSimulatedTimeResolution()`.

5.5.11 sg::SchedulerObject class

This section describes the `SchedulerObject` class. It is the base class for scheduler objects and interfaces. This interface does not allow object destruction.

5.5.11.1 sg::SchedulerObject::getName

This method returns the name of the instance that implements the object or interface. The intended use is debugging.

```
const char *getName() const;
```

Although Arm does not guarantee this name to be unique or hierarchical, Arm recommends including or using the hierarchical component name. The caller must not free/delete the returned string. This object owns the string. The pointer is valid as long as the object implementing this interface exists. If the caller cannot track the lifetime of this object and wants to remember the name, it must copy it.

5.5.12 sg::scx_create_default_scheduler_mapping

This function returns a pointer to a new instance of the default implementation of the scheduler mapping provided with Fast Models.

```
sg::SchedulerInterfaceForComponents
 *scx_create_default_scheduler_mapping(scx_simcontrol_if *simcontrol);
```

simcontrol

a pointer to an existing simulation controller. If this is `NULL`, this function returns `NULL`.

5.5.13 sg::scx_get_curr_scheduler_mapping

This function returns a pointer to the scheduler mapping interface.

```
sg::SchedulerInterfaceForComponents *scx_get_curr_scheduler_mapping();
```

5.6 SystemC Export limitations

This section describes the limitations of the current release of SystemC Export.

The *Exported Virtual Subsystems* (EVSs) are deliberately not time or cycle accurate, although they are accurate on a functional level.

5.6.1 SystemC Export limitation on reentrancy

Processor models, and the CCI400, MMU_400, and MMU_500 component models support reentrancy.

Reentrancy occurs when a component in an EVS issues a blocking transaction to a SystemC peripheral that in turn generates another blocking transaction back into the same component. This generation might come directly or indirectly from a call to `wait()` or by another SystemC peripheral.

Virtual platforms including EVSs that comprise a processor model do support such reentrancy.

For models that do not support reentrancy, the virtual platform might show unpredictable behavior because of racing within the EVS component.

5.6.2 SystemC Export limitation on calling wait()

Arm only supports calling `wait()` on bus transactions.

When a SystemC peripheral must really issue a `wait()` in reaction to a signal that is changing, buffer the signal in the bridge between the EVS and SystemC. On the next activation of the bridge, set the signal with the thread context of the EVS.

Note

The EVS runs in a temporally decoupled mode using a time quantum. *Transaction Level Modeling* (TLM) 2.0 targets using the Loosely-Timed coding style do not call `wait()`.

5.6.3 SystemC Export limitation on code translation support for external memory

EVS core components use code translation for speed. Not enabling *Direct Memory Interface* (DMI) reduces performance.

The core components in EVSs use code translation for high simulation speed. Therefore they fetch data from external memory to translate it into host machine code. Changing the memory contents outside of the scope of the core makes the data inconsistent.

Enable DMI accesses to instruction memory to avoid dramatic performance reductions. Otherwise, EVSs:

- Model all accesses.
- Perform multiple spurious transactions.
- Translate code per instruction not per block of instructions.

5.6.4 SystemC Export limitation on Fast Models versions for MI platforms

SystemC Export with *Multiple Instantiation* (MI) supports virtual platforms with multiple EVSs made with the same version of Fast Models. Integrating EVSs from different versions of Fast Models might result in unpredictable behavior.

6. Timing annotation

This chapter describes timing annotation, which enables you to perform high-level performance estimation on Fast Models.

Fast Models are Programmers View (PV) models that are targeted at software development. They sacrifice timing accuracy to achieve fast simulation execution speeds. By default, each instruction takes a single simulator clock cycle, with no delays for memory accesses.

Timing annotation enables you to perform more accurate performance estimation on SystemC-based models with minimal simulation performance impact. You can use it to show performance trends and to identify test cases for further analysis on approximately timed or cycle-accurate models.

Timing annotation is always enabled for Fast Models platforms.

You can configure the following aspects of timing annotation:

- The time that processors take to execute instructions. This can be modeled in either of the following ways:
  - As an average *Cycles Per Instruction* (CPI) value, using the `cpi_mul` and `cpi_div` model parameters.
  - By assigning CPI values to different instruction classes, using CPI files.
- Branch predictor type and misprediction latency. For details, see BranchPrediction in the *Fast Models Reference Guide*
- Instruction and data prefetching.
- Cache and TLB latency.
- Latency caused by pipeline stalls. For details, see PipelineModel in the *Fast Models Reference Guide*.

6.1 CPI files

Cycles Per Instruction (CPI) files define classes of instructions and assign CPI values to them. CPI files give a more accurate estimate of the number of cycles required to run a program on the model.

Arm does not provide CPI files, only some pre-defined CPI instruction classes which can help you to create your own CPI files. To create a CPI file for a specific CPU:

1. Create a set of mappings between the instruction encodings for the instruction set and a set of instruction classes or groups of classes. Arm provides pre-defined instruction classes and groups for the A32, T32, and A64 instruction sets in `$PVLIB_HOME/etc/CPIPredefines/`. You can include these pre-defined instruction classes in your CPI files, or you can define your own classes.

2. Create a file to map these instruction classes to CPI values. This is the CPI file. Calculate the CPI values to use based on observations from a cycle accurate model, or see the Arm® Software Optimization Guides, which are available on Arm Developer.

Note

- An alternative to using CPI files is to use the `cpi_mul` and `cpi_div` parameters on a core in the model. These parameters are integers that represent a CPI multiplication or division factor for all instructions. They can also be used together to represent non-integer values. For example, use `cpi_mul` = 5, `cpi_div` = 4 for a CPI of 1.25.
- To calculate values for `cpi_mul` and `cpi_div`, experiment with running a workload on a cycle accurate simulation to choose values that give the most accurate results.
- If a CPI file is present, it overrides the `cpi_mul` and `cpi_div` parameters.
- If you do not set these parameters and do not specify a CPI file, a CPI value of 1.0 is used for all instructions.

A CPI file can support multiple instruction sets, including `A64`, `A32`, and `T32`. It can also support multiple processor types, including pre-defined and user-defined types.

Specify a CPI file when launching a platform model by using the `--cpi-file` command-line parameter, for example:

```
./isim_system … --cpi-file CPI_file.txt --stat
```

The `--stat` parameter displays timing statistics on simulation exit.

Alternatively, specify a CPI file in your SystemC code by calling the function 5.4.13 scx::scx_set_cpi_file on page 49.

`CPIValidator` is a command-line tool provided in `$MAXCORE_HOME/bin/` to help you create valid CPI files. Use the `--help` switch to list the available options. For example, the following command parses and builds the evaluation tree for `CPI_file.txt`, and prints it in plain text to a file called `CPIEvaluationTree.txt`:

```
$MAXCORE_HOME/bin/CPIValidator --input-file CPI_file.txt --output-file CPIEvaluationTree.txt
```

Related information

CPI file syntax on page 101
BNF specification for CPI files on page 107

6.2 CPI file syntax

CPI files are plain text files that contain a series of statements, one per line. Lines that begin with a `#` character are ignored.

In the following syntax definitions, square brackets `[]` enclose optional attributes. An ellipsis ... indicates attributes that can be repeated.

The valid statements in a CPI file are:

`DefineCpi`

Defines the CPI value to use for an instruction class or group. The syntax is:

```
DefineCpi class_or_group ISet=iset [CpuType=cputype] Cpi=cpi
```

where:

`class_or_group`

The name of an instruction class or group. This name can contain wildcards.

A decoded instruction is matched against all `DefineCpi` statements in the order they appear in the CPI file from top to bottom. The first instruction class match is used and all following statements are ignored.

`ISet=`*`iset`*

Specifies which instruction set this CPI value refers to. This parameter is one of `A32`, `A64`, `Thumb`, or `T2EE`, or use the `*` character to specify all instruction sets.

`CpuType=`*`cputype`*

Specifies which Arm® processor type this CPI value refers to. This parameter can be a user-defined type, or one of the following pre-defined types:

- `ARM_Cortex-A12`
- `ARM_Cortex-A17`
- `ARM_Cortex-A15`
- `ARM_Cortex-A7`
- `ARM_Cortex-A5MP`
- `ARM_Cortex-M4`
- `ARM_Cortex-M7`
- `ARM_Cortex-A57`
- `ARM_Cortex-A72`
- `ARM_Cortex-A53`
- `ARM_Cortex-R7`
- `ARM_Cortex-R5`
- `ARM_Cortex-A9MP`

- `ARM_Cortex-A9UP`
- `ARM_Cortex-A8`
- `ARM_Cortex-R4`
- `ARM_Cortex-M3`
- `ARM_Cortex-M0+`
- `ARM_Cortex-M0`

Use the `*` character to specify any processor type. Specifying no `CpuType` is equivalent to specifying `CpuType=*`.

`Cpi=`*`cpi`*

The CPI value to assign to this instruction class or group.

For example:

```
DefineCpi Load_instructions ISet=A64 CpuType=ARM_Cortex-A53 Cpi=2.15
```

`DefineClass`

Defines an instruction class. The syntax is:

```
DefineClass class Mask=mask Value=value [ProhibitedMask=pmask
ProhibitedValue=pvalue …] ISet=iset [CpuType=cputype]
```

where:

`class`

The name of the instruction class to define. It must be unique in the CPI file. It can be used in a subsequent `DefineCpi` statement.

`Mask=`*`mask`*

A bitmask to apply to an instruction encoding before comparing the result with the `Value` attribute. This parameter identifies which bits in the encoding are relevant for comparing with `Value`.

For example, the value `0000xxxx1xxx100x` is represented as `Mask=0xF08E Value=0x0088`.

`Value=`*`value`*

The binary value to compare with the instruction encodings. A match indicates that the instruction belongs to this class, unless the encoding also matches the `ProhibitedValue`.

`ProhibitedMask=`*`pmask`*

A bitmask to apply to an instruction encoding before comparing the result with the `ProhibitedValue` attribute. It identifies which bits in the encoding are relevant for comparing with `ProhibitedValue`.

`ProhibitedValue=`*`pvalue`*

The binary value to compare with the instruction encodings. A match indicates that the instruction does not belong to this class.

ISet=*iset*

Specifies which instruction set this class refers to. See `DefineCpi` for the possible values.

CpuType=*cputype*

Specifies which Arm® processor type this class refers to. See `DefineCpi` for the possible values.

Note

A `DefineClass` statement must include a single `Mask` and `Value` attribute pair, but can include any number of `ProhibitedMask` and `ProhibitedValue` attribute pairs.

For example:

```
DefineClass Media_instructions Mask=0x0E000010 Value=0x06000010
 ProhibitedMask=0xF0000000 ProhibitedValue=0xF0000000 ISet=A32
```

DefineGroup

Defines a group of instruction classes. The syntax is:

```
DefineGroup group Classes=class[,class,…] ISet=iset [CpuType=cputype]
[Mix=mix[,mix,…]]
```

where:

group

The name of the group to define. It must be unique in the CPI file. It can be used in a subsequent `DefineCpi` statement.

Classes=*class*[,*class*,…]

A comma-separated list of instruction classes that belong to this group.

ISet=*iset*

Specifies which instruction set this group refers to. See `DefineCpi` for the possible values.

CpuType=*cputype*

Specifies which Arm® processor type this group refers to. See `DefineCpi` for the possible values.

Mix=*mix*[,*mix*,…]

A comma-separated list of mixin names that cause additional instruction groups and classes to be automatically defined.

For example:

```
DefineGroup Divide_instructions Classes=SDIV,UDIV CpuType=ARM_Cortex-A73
 ISet=A32
```

`DefineMixIn`

Defines a single mask/value pair and suffix that can optionally be used in `DefineGroup` statements to automatically define new instruction groups and classes. Applying a mixin to a group causes a new instruction group or class to be defined for every instruction group or class that is included in the group, and also for the group itself. The names of these newly-defined groups and classes is the original group or class name followed by an underscore character, then the mixin suffix.

The syntax is:

`DefineMixIn` *`mix`* `Mask=`*`mask`* `Value=`*`value`* `Suffix=`*`suffix`*

where:

`mix`

The name of the mixin to define. It must be unique in the CPI file. It can be used in subsequent `DefineGroup` statements.

`Mask=`*`mask`*

A bitmask to apply to an instruction encoding before comparing the result with the `Value` attribute.

`Value=`*`value`*

The binary value to compare with the instruction encodings. A match indicates that the instruction belongs to this group or class.

`Suffix=`*`suffix`*

After applying a mixin to a group, this suffix is appended to the names of the automatically-defined groups and classes.

In the following example, the `DefineGroup` statement defines `my_group`, but also automatically defines `my_group_AL` and `my_class_AL`:

```
DefineMixIn my_mixin Mask=0xF0000000 Value=0xE0000000 Suffix=AL
…
DefineClass my_class Mask=0x0FF00000 Value=0x03000000 ISet=A32
DefineGroup my_group Classes=my_class ISet=A32 Mix=my_mixin
```

`DefineCpuType`

Defines a processor type. The syntax is:

`DefineCpuType` *`cputype`* `ISets=`*`iset`*`[,`*`iset`*`,…]`

where:

`cputype`

The name of the processor type to define. It must be unique in the CPI file. It can be used in subsequent `DefineCpi`, `DefineClass`, `DefineGroup`, and `MapCpu` statements.

`ISets=`*`iset`*`[,`*`iset`*`,…]`

A comma-separated list of instruction sets that this processor type supports. See `DefineCpi` for the possible values.

For example:

```
DefineCpuType ARM_Cortex-A73 ISets=*
```

`MapCpu`

Maps a CPU instance by name to a CPU type. The syntax is:

```
MapCpu cpuinstance ToCpuType=cputype
```

where:

`cpuinstance`

The name of the CPU instance to map to a processor type. It can contain wildcards.

`ToCpuType=`*`cputype`*

The processor type to map the CPU instance onto. See the list of `CpuType`s in `DefineCpi` for the possible values.

For example:

```
MapCpu FVP_Base_AEMvA_AEMvA.cluster0.cpu0 ToCpuType ARM_Cortex-A73
```

`Defaults`

Defines the default CPI value to be used for instructions that do not match any class or group. This statement is optional and can occur more than once in the CPI file. The syntax is:

```
Defaults ISet=iset [CpuType=cputype] Cpi=cpi
```

where:

`ISet=`*`iset`*

Specifies which instruction set this value refers to. See `DefineCpi` for the possible values.

`CpuType=`*`cputype`*

Specifies which Arm® processor type this value refers to. See `DefineCpi` for the possible values.

`Cpi=`*`cpi`*

The default CPI value for the specified instruction set and processor type.

For example:

```
Defaults ISet=* CpuType=* Cpi=0.82
```

`Include`

Includes a supplementary CPI file at this point in the file. This is equivalent to the `#include` preprocessor directive in C. The evaluation of the `FilePath` attribute is to first treat it as an absolute path, then as a relative path, and finally as relative to the `PVLIB_HOME` environment variable. The syntax is:

```
Include FilePath=path
```

For example:

```
Include FilePath=etc/CPIPredefines/ARMv8A_A32_Mnemonics.txt
```

6.3 BNF specification for CPI files

CPI files have the following BNF specification:

```
                <CPIFile> ::= <Statements>
             <Statements> ::= <Statement> <Statements>
                            | <Statement>
              <Statement> ::= <Comment>
                            | <DefineCpiStatement>
                            | <DefaultsStatement>
                            | <DefineCpuTypeStatement>
                            | <MapCpuStatement>
                            | <DefineClassStatement>
                            | <DefineGroupStatement>
                            | <IncludeStatement>
                            | <DefineMixInStatement>
      <DefineCpiStatement ::= "DefineCpi" <InstructionClassOrGroup>
<DefineCpiAttributes> <EOL>
      <DefaultsStatement> ::= "Defaults" <DefineCpiAttributes> <EOL>
  <DefineCpuTypeStatement ::= "DefineCpuType" <UserCpuType>
<DefineCpuTypeAttributes> <EOL>
         <MapCpuStatement ::= "MapCpu" <CpuInstance> <MapCpuAttributes> <EOL>
    <DefineClassStatement ::= "DefineClass" <InstructionClass>
<DefineClassAttributes> <EOL>
    <DefineGroupStatement ::= "DefineGroup" <InstructionGroup>
<DefineGroupAttributes> <EOL>
       <IncludeStatement> ::= "Include" <IncludeAttributes> <EOL>
   <DefineMixInStatement> ::= "DefineMixIn" <MixInType> <DefineMixInAttributes>
<EOL>
    <DefineCpiAttributes> ::= <DefineCpiAttribute> <DefineCpiAttributes>
                            | <DefineCpiAttribute>
     <DefineCpiAttribute> ::= <ISetAttribute>       { Mandatory }
                            | <CpuTypeAttribute>    { Optional }
                            | <CpiAttribute>        { Mandatory }
          <ISetAttribute> ::= "ISet" "=" <ISetOrStar>
             <ISetOrStar> ::= <ISet> | "*"
                   <ISet> ::= "A32" | "A64" | "Thumb" | "T2EE"
       <CpuTypeAttribute> ::= "CpuType" "=" <CpuType>
                <CpuType> ::= "ARM_Cortex-A12" | "ARM_Cortex-A17"
                            | "ARM_Cortex-A15" | "ARM_Cortex-A7"
                            | "ARM_Cortex-A5MP" | "ARM_Cortex-M4"
                            | "ARM_Cortex-M7" | "ARM_Cortex-A57"
                            | "ARM_Cortex-A72" | "ARM_Cortex-A53"
                            | "ARM_Cortex-R7" | "ARM_CortexR5"
                            | "ARM_Cortex-A9MP" | "ARM_Cortex-A9UP"
                            | "ARM_Cortex-A8" | "ARM_Cortex-R4"
                            | "ARM_Cortex-M3" | "ARM_Cortex-M0+"
                            | "ARM_Cortex-M0" | <UserCpuType> | "*"
           <CpiAttribute> ::= "Cpi" "=" <Cpi>
<DefineCpuTypeAttributes> ::= <ISetsAttribute>
         <ISetsAttribute> ::= "ISets" "=" <ISetsOrStar>
            <ISetsOrStar> ::= <ISets> | "*"
                  <ISets> ::= <ISet> "," <ISets> | <ISet>
       <MapCpuAttributes> ::= <ToCpuTypeAttribute>
     <ToCpuTypeAttribute> ::= "ToCpuType" "=" <CpuType>
  <DefineClassAttributes> ::= <DefineClassAttribute> <DefineClassAttributes>
                            | <DefineClassAttribute>
   <DefineClassAttribute> ::= <MaskAttribute>               { Mandatory }
                            | <ValueAttribute>              { Mandatory }
```

```
                             | <ProhibitedPairsAttribute>   { Optional }
                             | <ISetAttribute>              { Mandatory }
                             | <CpuTypeAttribute>           { Optional }
           <MaskAttribute> ::= "Mask" "=" <Mask>
          <ValueAttribute> ::= "Value" "=" <Value>
<ProhibitedPairsAttribute> ::= <ProhibitedPairAttribute> <ProhibitedPairsAttribute>
                             | <ProhibitedPairAttribute>
  <ProhibitedPairAttribute> ::= <ProhibitedMaskAttribute> <ProhibitedValueAttribute>
 <ProhibitedMaskAttribute> ::= "ProhibitedMask" "=" <Mask>
<ProhibitedValueAttribute> ::= "ProhibitedValue" "=" <Value>
   <DefineGroupAttributes> ::= <DefineGroupAttribute> <DefineGroupAttributes>
                             | <DefineGroupAttribute>
    <DefineGroupAttribute> ::= <ClassesAttribute>          { Mandatory }
                             | <ISetAttribute>             { Mandatory }
                             | <CpuTypeAttribute>          { Optional }
                             | <MixAttribute>              { Optional }
        <ClassesAttribute> ::= "Classes" "=" <InstructionClassOrGroups>
        <MixAttribute>     ::= "Mix" "=" <MixInTypes>
<InstructionClassOrGroups> ::= <InstructionClassOrGroup> ","
 <InstructionClassOrGroups>
      <instructionClasses> ::= <InstructionClass>
 <InstructionClassOrGroup> ::= <InstructionClass>
                             | <InstructionGroup>
              <MixInTypes> ::= <MixInType> "," <MixInTypes>
               <MixInType> ::= <Symbol>
       <IncludeAttributes> ::= <FilePathAttribute>
       <FilePathAttribute> ::= "FilePath" "=" <FilePath>
   <DefineMixInAttributes> ::= <DefineMixInAttribute> <DefineClassAttributes>
    <DefineMixInAttribute> ::= <MaskAttribute>
                             | <ValueAttribute>
                             | <SuffixAttribute>
         <SuffixAttribute> ::= "Suffix" "=" <String>
                <FilePath> ::= <String>
        <InstructionClass> ::= <Symbol>
        <InstructionGroup> ::= <Symbol>
             <UserCpuType> ::= <Symbol>
             <CpuInstance> ::= <QuotedString>       { Supports use of wild cards }
                     <Cpi> ::= <Double>
                    <Mask> ::= <UnsignedInteger>
                   <Value> ::= <UnsignedInteger>
```

6.4 Instruction and data prefetching

Arm® Cortex®-A series processors implement prefetching instructions and data into caches to improve the cache hit rate and improve performance. Fast Models supports prefetching instructions and data independently, by using model parameters.

6.4.1 Configuring instruction prefetching

Configure instruction cache prefetching by using the following cluster-level parameters.

`icache-prefetch_enabled`

> `true` to enable simulation of instruction cache prefetching, `false` otherwise. Defaults to `false`.
>
> The execution of a branch instruction causes the model to prefetch instructions from the memory region starting at the branch target address into a number of sequential cache lines. If `true`, the following extra parameters are available:

`icache-prefetch_level`

Specifies the zero-indexed cache level into which instructions are prefetched. Defaults to 0, which means L1.

`icache-nprefetch`

Specifies the number of additional, sequential instruction cache lines to prefetch. Defaults to 1.

Note

These parameters only have an effect when cache state modeling is enabled, which is controlled by the model parameter `icache-state_modelled` or `cache_state_modelled`.

Example

The following command line enables instruction cache prefetching and prints `WAYPOINT` trace events to the console. A `WAYPOINT` is a point at which instruction execution by the processor might change the program flow.

```
./FVP_Base_AEMvA …
-C cache_state_modelled=1 \
-C cluster0.icache-prefetch_enabled=1 \
--plugin $PVLIB_HOME/plugins/Linux64_GCC-9.3/GenericTrace.so \
-C TRACE.GenericTrace.trace-sources=WAYPOINT
```

Related information

Loading a plug-in

6.4.2 Configuring data prefetching

The purpose of data prefetch modeling is to make the contents of the data cache more closely resemble those on a system with a hardware prefetcher. A default data prefetcher is supplied, which is relatively configurable. It is not intended to match any specific processor.

To run the model with data prefetch modeling enabled, using the default data prefetcher with default parameters, use the following parameters:

```
-C cache_state_modelled=true --plugin "<<internal><DataPrefetch>>" -C cluster0.dcache-
prefetch_enabled=1
```

When the model exits, it reports how many prefetches were issued and how many cache hits on recently-prefetched data were detected. The performance impact is about 10% compared to running with cache state modeling enabled.

By default, a data prefetch plug-in attaches to all processors and clusters in a system, and maintains independent internal state for each processor. To change this, for example if you want a different number of tracked streams on big and LITTLE cores, load the plug-in twice and pass a different `.cluster` parameter to each instance, for example:

```
--plugin "DP_BIG=<<internal><DataPrefetch>>" --plugin "DP_LITTLE=<<internal><DataPrefetch>>" \
 -C DataPrefetch.DP_BIG.cluster=0 -C DataPrefetch.DP_LITTLE.cluster=1 \
 -C DataPrefetch.DP_BIG.lfb_entries=16 -C DataPrefetch.DP_LITTLE.lfb_entries=4
```

The names `DP_BIG` and `DP_LITTLE` are examples. They can be any names you choose.

The example prefetcher is a basic stride-detecting prefetcher, but relatively configurable using the following parameters:

Table 6-1: Parameters for the example prefetcher

Parameter	Description
`history_length`	Length of history to maintain.
`history_threshold`	Number of misses to allow in history before issuing a prefetch.
`lfb_entries`	Number of access streams to track.
`mbs_expire`	Number of non-hitting loads to allow before the prefetcher stops tracking a potential access stream.
`pf_count`	Number of prefetch streams available.
`pf_tracker_count`	Number of prefetches tracked.
`pf_initial_number`	Initial number of prefetches to issue for a new stream.
`prefetch_all_levels`	Prefetch to all cache levels rather than just the lowest level.

An *access stream* is created whenever a load is made to an address that is not within three cache lines of a previously observed load. This might overwrite a previously created access stream. When a consistent stride has been observed, that is, when addresses *N*, *N*`+delta`, *N*`+2*delta` are seen, a prefetch stream is allocated with stride `delta` and a lifetime of `pf_initial_number`.

Prefetches are issued in a round-robin fashion from active prefetch streams (the lifetime goes down by one each time a prefetch is issued) whenever there have been fewer than `history_threshold` cache misses among the last `history_length` loads. The rationale is that if lots of cache hits are occurring, there should be available bandwidth on the memory interface to be used by prefetching.

Issued prefetches are tracked in a circular list of size `pf_tracker_count`, and if the prefetcher sees a load to an address in this circular list, it increments the lifetime of the prefetch stream that issued the successful prefetch.

Note

Prefetches are to *physical* addresses, and as a result, a prefetch stream expires when it reaches the end of a 4KB region.

6.5 Configuring cache and TLB latency

You can configure latency for different cache operations for Cortex®-A processor models by setting model parameters.

The following parameters are available:

- Read access latency for L1 D-cache, L1 I-cache, or L2 cache. For example `dcache-read_access_latency`.
- Separate latencies for read hits and misses in L1 D-cache, L1 I-cache, or L2 cache. For example `dcache-hit_latency` and `dcache-miss_latency`. The total latency for a read access is the sum of the read access latency and the hit or miss latency.
- Write access latency for L1 D-cache or L2 cache. For example `dcache-write_access_latency`.
- Latency for cache maintenance operations for L1 D-cache, L1 I-cache, or L2 cache. For example `dcache-maintenance_latency`.
- Latency for snoop accesses that perform a data transfer for L1 D-cache or L2 cache. For example `dcache-snoop_data_transfer_latency`.
- Latency for snoop accesses that are issued by L2 cache. For example `l2cache-snoop_issue_latency`.
- TLB and page table walk latencies. For example `tlb_latency`.

Note

- These parameters only take effect when cache state modeling is enabled. This is controlled using parameters, for example `dcache-state_modelled` and `icache-state_modelled`.
- All of these latency values are measured in clock ticks.
- For reads and writes, latency can be specified per access, for example `dcache-read_access_latency`, or per byte, for example `dcache-read_latency`. If both parameters are set, the per-access value takes precedence over the per-byte value.

6.6 Timing annotation tutorial

This tutorial shows how to use the Cycles Per Instruction (CPI) specification and branch prediction modeling features with a Fast Models example platform model, and how to measure their impact on code execution time. The commands shown are for Linux, although the process is the same on Windows.

6.6.1 Setting up the environment

This tutorial runs some example applications on the `EVS_Base_Cortex-A75` example virtual platform to show different timing annotation features.

6.6.1.1 Prerequisites

To use timing annotation, you require the following:

- A SystemC-integrated virtual platform, for instance an ISIM or an EVS platform.
- An application that enables caches.
- A way of calculating the execution of time of individual instructions.
- A way of determining the total execution time of the simulation.
- A way of calculating the average Cycles Per Instruction (CPI) value for the simulation.

6.6.1.2 Building the EVS_Base_Cortex-A75 example platform

The `EVS_Base_Cortex-A75` platform includes a single EVS that is connected to SystemC components that model a timer, and an application memory component that supports individual configuration of read and write latencies.

About this task

The platform is not provided pre-built in the Fast Models Portfolio installation, so you must first build it, for example:

```
cd $PVLIB_HOME/examples/SystemCExport/EVS_Platforms/EVS_Base/Build_Cortex-A75/
make rel_gcc93_64
```

6.6.1.3 Calculating the execution time of an instruction

The `INST` MTI trace source displays every instruction that is executed while running a program. It also displays the current simulation time after an instruction has completed executing.

The number of ticks an instruction takes to execute is the difference between the times of two consecutive instructions. The default is one tick (on the core) for each instruction. With the default clock speed of 100MHz, this gives a default execution time for an instruction of 10000 picoseconds. Any changes to latency due to branch mispredictions, memory accesses, or CPI specifications can be observed by comparison with this value.

This tutorial uses the `INST` trace source to measure the time it takes to execute an instruction. To generate trace, it uses the `GenericTrace` plug-in. This plug-in allows you to output any number of MTI trace sources to a text file.

Use the following extra parameters when launching the model to collect the `INST` trace source:

```
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-9.3/GenericTrace.so \
-C TRACE.GenericTrace.trace-sources=INST \
-C TRACE.GenericTrace.trace-file=/path/to/trace/file.txt
```

The trace that is produced for the first two instructions might look like this:

```
INST: PC=0x0000000080000000 OPCODE=0x58001241 SIZE=0x04 MODE=EL3h ISET=AArch64
PADDR=0x0000000080000000 NSDESC=0x00 PADDR2=0x0000000080000000 NSDESC2=0x00 NS=0x00
ITSTATE=0x00 INST_COUNT=0x0000000000000001 LOCAL_TIME=0x0000000000001388
CURRENT_TIME=0x0000000000001388 CORE_NUM=0x00 DISASS="LDR      x1,{pc}+0x248 ;
0x80000248"

INST: PC=0x0000000080000004 OPCODE=0xd518c001 SIZE=0x04 MODE=EL3h ISET=AArch64
PADDR=0x0000000080000004 NSDESC=0x00 PADDR2=0x0000000080000004 NSDESC2=0x00 NS=0x00
ITSTATE=0x00 INST_COUNT=0x0000000000000002 LOCAL_TIME=0x0000000000003a98
CURRENT_TIME=0x0000000000003a98 CORE_NUM=0x00 DISASS="MSR      VBAR_EL1,x1"
```

The `CURRENT_TIME` value for the first instruction is `0x1388`, or 5000ps. This value shows that the instruction took 0.5 ticks to execute. Timing annotation has halved the execution time of this instruction.

The difference between the `CURRENT_TIME` values of the two instructions is `0x2710`, or 10000 picoseconds. This value shows that the second instruction took one tick to execute.

Related information

MTI trace sources on page 123

6.6.1.4 Displaying the total execution time of the simulation

You can use MTI trace to calculate the execution time of individual instructions, but to determine the overall simulation time, use the command-line option `--stat` instead.

This option causes the model to print performance statistics to the terminal on exiting. The statistics include `Simulated time`, which is the total simulation time in seconds. For example:

```
--- Base statistics: ----------------------------------------------------------
Simulated time                          : 0.001206s
User time                               : 0.276000s
System time                             : 0.136000s
Wall time                               : 0.700834s
Performance index                       : 0.00
Base.cluster0.cpu0                      : 0.42 MIPS (172289 Inst)
-------------------------------------------------------------------------------
```

Note

The MIPS value is based on the host system time, not the simulated time.

This tutorial uses the `--stat` option to compare the model's performance in different timing annotation configurations.

6.6.1.5 Calculating the average CPI value

Calculate the average CPI value for the simulation by using the instruction count and the simulated time value, as displayed by the `--stat` option.

Use the following formula:

```
average_cpi = simulated_time_in_picoseconds / (10000 * instruction_count)
```

This example calculates an average CPI value of 0.69999:

```
average_cpi = (0.001206 * 10^12) / (10000 * 172289) = 0.69999
```

6.6.2 Modeling Cycles Per Instruction (CPI)

This section demonstrates how to precisely model the simulated time per instruction by using the CPI timing annotation feature.

6.6.2.1 CPI parameters

You can specify a single CPI value for all instructions that execute within a cluster. This value is referred to as a fixed CPI value. Alternatively, use a custom CPI file to define individual CPI values for specific instructions. Use a fixed CPI value instead of a CPI file when precise per-instruction modeling is not required.

When running a simulation with either of these options, you can calculate the average CPI value using the formula that is shown in 6.6.1.5 Calculating the average CPI value on page 113.

Note

You can combine the CPI specification with other timing annotation features. Therefore, the average CPI value that you observe can be different from the fixed CPI value that you specify.

6.6.2.2 Specifying a fixed CPI value

Specify a fixed CPI value by using the per-cluster model parameters `cpi_mul` and `cpi_div`. If you do not set these parameters and do not specify a CPI file, a CPI value of 1.0 is used for all instructions.

These parameters are integers that represent a CPI multiplication or division factor that is applied to all instructions during execution within that cluster.

They can be used together to represent non-integer values. For example, use `cpi_mul` = 5, `cpi_div` = 4 for a CPI of 1.25.

The fixed CPI value is used in a way that *core_clock_period * fixed_cpi_value* is rounded to the nearest picosecond.

Related information

Running the example with a fixed CPI value on page 120

6.6.2.3 Example CPI file

CPI files can be large because they have to cover multiple encodings for many of the instructions that are included. Various predefined encodings are provided under `$PVLIB_HOME/etc/CPIPredefines/` that can help you to create CPI files. This tutorial does not use predefined encodings.

The following example defines CPI values for the instructions `ADRP`, `ADR`, `ADD`, `CMP`, `ORR`, `LDP`, `STR`, branches, exception generating instructions, and system instructions. It defines a default CPI value of 0.75 for all other instructions. It applies to the A64 instruction set, and does not restrict the values to a specific core.

Note

These CPI values are for demonstration purposes only. They are arbitrary and are not representative of any Arm® processor.

```
# -------------------
# Instruction classes
# -------------------
## PC-relative addressing
DefineClass ADRP                   Mask=0x9F000000 Value=0x90000000 ISet=A64
DefineClass ADR                    Mask=0x9F000000 Value=0x10000000 ISet=A64
## Arithmetic
DefineClass ADD_ext_reg            Mask=0x7FE00000 Value=0x0B200000 ISet=A64
DefineClass ADD_sft_reg            Mask=0x7F200000 Value=0x0B000000 ISet=A64
DefineClass ADD_imm                Mask=0x7F000000 Value=0x11000000 ISet=A64
DefineClass CMP_ext_reg            Mask=0x7FE0001F Value=0x6B20001F ISet=A64
DefineClass CMP_sft_reg            Mask=0x7F20001F Value=0x6B00001F ISet=A64
DefineClass CMP_imm                Mask=0x7F00001F Value=0x7100001F ISet=A64
## Logical
DefineClass ORR_sft_reg            Mask=0x7F200000 Value=0x2A000000 ISet=A64
DefineClass ORR_imm                Mask=0x7F800000 Value=0x32000000 ISet=A64
## Branches, exception generating and system instructions
DefineClass B_gen_except_sys       Mask=0x1C000000 Value=0x14000000 ISet=A64
## Load register pair
DefineClass LDP_post_idx           Mask=0x7FC00000 Value=0x28C00000 ISet=A64
DefineClass LDP_pre_idx            Mask=0x7FC00000 Value=0x29C00000 ISet=A64
DefineClass LDP_sgn_off            Mask=0x7FC00000 Value=0x29400000 ISet=A64
## Store register
DefineClass STR_reg                Mask=0xBFE00C00 Value=0xB8200000 ISet=A64
DefineClass STR_imm_post_idx       Mask=0xBFE00C00 Value=0xB8000400 ISet=A64
DefineClass STR_imm_pre_idx        Mask=0xBFE00C00 Value=0xB8000C00 ISet=A64
DefineClass STR_imm_usg_off        Mask=0xBFC00000 Value=0xB9000000 ISet=A64
# ------------------
# Instruction groups
# ------------------
DefineGroup PC_rel_addr_instr      Classes=ADRP,ADR                              ISet=A64
DefineGroup ADD_instr              Classes=ADD_ext_reg,ADD_sft_reg,ADD_imm       ISet=A64
DefineGroup CMP_instr              Classes=CMP_ext_reg,CMP_sft_reg,CMP_imm       ISet=A64
DefineGroup ORR_instr              Classes=ORR_sft_reg,ORR_imm                   ISet=A64
DefineGroup B_gen_except_sys_instr Classes=B_gen_except_sys                      ISet=A64
DefineGroup LDP_instr              Classes=LDP_post_idx,LDP_pre_idx,LDP_sgn_off ISet=A64
DefineGroup STR_instr
 Classes=STR_reg,STR_imm_post_idx,STR_imm_pre_idx,STR_imm_usg_off ISet=A64
# ----------
# CPI values
# ----------
DefineCpi   PC_rel_addr_instr      ISet=A64 Cpi=0.25
DefineCpi   ADD_instr              ISet=A64 Cpi=0.50
DefineCpi   CMP_instr              ISet=A64 Cpi=0.75
DefineCpi   ORR_instr              ISet=A64 Cpi=0.50
DefineCpi   B_gen_except_sys_instr ISet=A64 Cpi=1.00
DefineCpi   LDP_instr              ISet=A64 Cpi=2.00
DefineCpi   STR_instr              ISet=A64 Cpi=1.00
# --------
# Defaults
# --------
Defaults ISet=* Cpi=0.75
```

Related information

CPI file syntax on page 101

6.6.2.4 Defining CPI values in a CPI file

To define CPI values in a CPI file, use the following procedure for each instruction or set of instructions:

Procedure

1. Create an instruction class for each encoding of an instruction or set of instructions by using the `DefineClass` keyword.
2. Group instruction classes by using the `DefineGroup` keyword.
3. Set a CPI value for each instruction class or group of classes by using the `DefineCpi` keyword.

Results

The encodings for each instruction in the A64 instruction set are provided by the Arm®v8-A Architecture Reference Manual, chapter 4. It also describes groups of instructions that share encodings. You can use these encodings to define the `Mask` and `Value` fields in the CPI file.

The `Mask` field must cover all bits that are fixed in the encoding of an instruction. The `Value` field must specify the value of these bits. For example, chapter 4 of the Arm®v8-A Architecture Reference Manual defines a set of instructions called *PC-rel. addressing*. In the example CPI file, the following statements specify a common CPI value for these instructions:

```
DefineClass ADRP Mask=0x9F000000 Value=0x90000000 ISet=A64
DefineClass ADR Mask=0x9F000000 Value=0x10000000  ISet=A64
DefineGroup PC_rel_addr_instr Classes=ADRP,ADR    ISet=A64
DefineCpi PC_rel_addr_instr ISet=A64 Cpi=0.25
```

For both instruction classes, the `Mask` value has bit[31] set to `0b1` and bits [28:24] set to `0b11111`. As shown in the reference manual, a value of `0b10000` for bits [28:24] identifies the instruction as being `ADR` or `ADRP`. Therefore, both `Value` fields set bits [28:24] to `0b10000`. Bit[31] distinguishes between `ADR` and `ADRP`, so bit[31] in the `Value` field for `ADR` is set to `0b0` and to `0b1` for `ADRP`.

This specification allows the model to specify a CPI value of 0.25 for the `PC_rel_addr_instr` group of instructions. A similar process has been followed to determine the `Mask` and `Value` fields for the other instructions in the CPI file example.

Related information

CPI file syntax on page 101
Arm Architecture Reference Manual for A-profile architecture

6.6.2.5 Validating a CPI file

To validate CPI files, use the `CPIValidator` tool. You can find this tool in a Fast Models Tools installation under `$MAXCORE_HOME/bin/`. The tool can detect missing or incompatible instruction groups and classes, but cannot validate the encodings themselves.

For example, if you remove the `DefineClass` statement for the `B_gen_except_sys` instruction class, and validate the example CPI file by using the following command:

```
CPIValidator --input-file /path/to/custom_cpi.txt --output-file cpi_evaluation.txt
```

the tool produces the following output:

```
ERROR: Instruction Class 'B_gen_except_sys' has no definition, when Instruction Set
 is 'A64' and the CPU Type is 'Default ARM Core'.
ERROR: Processing error in file /path/to/custom_cpi.txt
```

Using the tool with the complete CPI file produces the following output:

```
Core Performance Profile: Default ARM Core
--------------------------------------------------------------------------------
Instruction Set: A32 Default Cpi:0.75
Instruction Set: A64 Default Cpi:0.75
    (0x1c000000|0x14000000) Cpi:1 Name:B_gen_except_sys
    (0x7f000000|0x11000000) Cpi:0.5 Name:ADD_imm
    (0x7f00001f|0x7100001f) Cpi:0.75 Name:CMP_imm
    (0x7f200000|0x0b000000) Cpi:0.5 Name:ADD_sft_reg
    (0x7f200000|0x2a000000) Cpi:0.5 Name:ORR_sft_reg
    (0x7f20001f|0x6b00001f) Cpi:0.75 Name:CMP_sft_reg
    (0x7f800000|0x32000000) Cpi:0.5 Name:ORR_imm
    (0x7fc00000|0x28c00000) Cpi:2 Name:LDP_post_idx
    (0x7fc00000|0x29400000) Cpi:2 Name:LDP_sgn_off
    (0x7fc00000|0x29c00000) Cpi:2 Name:LDP_pre_idx
    (0x7fe00000|0x0b200000) Cpi:0.5 Name:ADD_ext_reg
    (0x7fe0001f|0x6b20001f) Cpi:0.75 Name:CMP_ext_reg
    (0x9f000000|0x10000000) Cpi:0.25 Name:ADR
    (0x9f000000|0x90000000) Cpi:0.25 Name:ADRP
    (0xbfc00000|0xb9000000) Cpi:1 Name:STR_imm_usg_off
    (0xbfe00c00|0xb8000400) Cpi:1 Name:STR_imm_post_idx
    (0xbfe00c00|0xb8000c00) Cpi:1 Name:STR_imm_pre_idx
    (0xbfe00c00|0xb8200000) Cpi:1 Name:STR_reg
Instruction Set: Thumb Default Cpi:0.75
Instruction Set: T2EE Default Cpi:0.75
```

6.6.2.6 CPI class example program

This example program is designed to show the effect of the CPI values specified in the example CPI file.

The example consists of two source files, `main.c` and `asm_func.s`.

`main.c` contains the following code:

```
#include <stdio.h>
#include <string.h>

extern void asm_cpi(volatile int *value0, volatile int *value2);

volatile int values[2] = {1, 2};

int main(void) {
 asm_cpi(&values[0], &values[1]);
 return 0;
}
```

`asm_func.s` defines an embedded assembly language function `asm_cpi()` which uses instructions with defined CPI values:

```
 .section asm_func, "ax"
 .global  asm_cpi
 .type    asm_cpi, "function"
asm_cpi:
 ldp  w1, w2, [x0]
 cmp  w1, w2
 b.gt skip
 orr  w1, w1, w2
 str  w1, [x0]
skip:
 ret
```

This sequence of instructions checks if the second value in a two-element array pointed to by the address in `x0` is greater than the first value. If so, it performs a bitwise OR operation using the two values, storing the result as the new first value. The rest of this section examines this sequence by running this code on a platform model with the following CPI configurations:

- Using the default CPI value.
- Using the custom CPI file that was described earlier in the tutorial.
- Using a fixed CPI value.

Build the example using Arm Compiler for Embedded:

```
armclang -c --target=aarch64-arm-none-eabi -mcpu=cortex-a75 main.c -o main.o
armclang -c --target=aarch64-arm-none-eabi -mcpu=cortex-a75 asm_func.s -o asm_func.o
armlink main.o asm_func.o -o ta_cpi.axf
```

The name of the executable image used in these examples is `ta_cpi.axf`.

6.6.2.7 Running the example with the default CPI value

If you do not specify any CPI parameters, a default CPI value of 1.00 is used. This value establishes a baseline to compare with the other CPI configurations.

To use the default CPI value of 1.00, launch the model using the following command:

```
$PVLIB_HOME/examples/SystemCExport/EVS_Platforms/EVS_Base/Build_Cortex-A75/EVS_Base_Cortex-A75.x
 \
-C Base.bp.secure_memory=0 \
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-9.3/GenericTrace.so \
-C TRACE.GenericTrace.trace-sources=INST \
-C TRACE.GenericTrace.trace-file=trace.txt \
-a $PVLIB_HOME/images/ta_cpi.axf \
--stat
```

In the trace file that the `GenericTrace` plug-in produces, find the instruction at address `0x800005a4`. The trace for this instruction and the one before it is as follows:

```
INST: PC=0x00000000800005a0 OPCODE=0x910003fd SIZE=0x04 MODE=EL1h ISET=AArch64
PADDR=0x00000000800005a0 NSDESC=0x01 PADDR2=0x00000000800005a0 NSDESC2=0x01 NS=0x01
ITSTATE=0x00 INST_COUNT=0x000000000000b7bc LOCAL_TIME=0x0000000000007530
CURRENT_TIME=0x000000001c091fc0 CORE_NUM=0x00 DISASS="MOV      x29,sp"

INST: PC=0x00000000800005a4 OPCODE=0x90000020 SIZE=0x04 MODE=EL1h ISET=AArch64
PADDR=0x00000000800005a4 NSDESC=0x01 PADDR2=0x00000000800005a4 NSDESC2=0x01 NS=0x01
ITSTATE=0x00 INST_COUNT=0x000000000000b7bd LOCAL_TIME=0x0000000000009c40
CURRENT_TIME=0x000000001c0946d0 CORE_NUM=0x00 DISASS="ADRP     x0,{pc}+0x4000 ; 0x800045a4"
```

Using the `CURRENT_TIME` values, it can be observed that the instruction took 10000ps or 1 tick to complete, which shows the default CPI value of 1.00 is being used. You can verify that all other instructions are also using the default CPI value by examining the trace.

6.6.2.8 Running the example with a custom CPI file

To use the custom CPI file, launch the model using the following command:

```
$PVLIB_HOME/examples/SystemCExport/EVS_Platforms/EVS_Base/Build_Cortex-A75/EVS_Base_Cortex-A75.x
 \
-C Base.bp.secure_memory=0 \
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-9.3/GenericTrace.so \
-C TRACE.GenericTrace.trace-sources=INST \
-C TRACE.GenericTrace.trace-file=trace.txt \
-a $PVLIB_HOME/images/ta_cpi.axf \
--cpi-file $PVLIB_HOME/images/source/ta_cpi/custom_cpi.txt \
--stat
```

Using the trace output that the `GenericTrace` plug-in produces for the 10 instructions starting at address `0x800005a4`, and the `--stat` output, the following information can be obtained for the embedded assembly code sequence in the example program:

Table 6-2: CPI values for embedded assembly instructions

Address	Instruction	Simulated time (ps)	CPI value observed
`0x800005a4`	`ADRP x0,{pc}+0x4000`	2500	0.25

Address	Instruction	Simulated time (ps)	CPI value observed
`0x800005a8`	`ADD x0,x0,#0x9f0`	5000	0.50
`0x800005ac`	`ADD x1,x0,#4`	5000	0.50
`0x800005b0`	`BL {pc}+0x4294`	10000	1.00
`0x80004844`	`LDP w1,w2,[x0,#0]`	20000	2.00
`0x80004848`	`CMP w1,w2`	7500	0.75
`0x8000484c`	`B.GT {pc}+0xc`	10000	1.00
`0x80004850`	`ORR w1,w1,w2`	5000	0.50
`0x80004854`	`STR w1,[x0,#0]`	10000	1.00
`0x80004858`	`RET`	10000	1.00

This table shows that the CPI values that are defined in the example CPI file have been applied to the appropriate instructions.

The following information can be obtained for the simulation as a whole:

Table 6-3: Statistics for the whole simulation

Total number of instructions	Overall simulated time in seconds	Average CPI value
47701	0.000362	0.75889

Note

The average CPI value being close to the default CPI value specified in the CPI file does not signify anything by itself. To draw any conclusions from it, further analysis on the distribution of instructions would be required.

6.6.2.9 Running the example with a fixed CPI value

The average CPI value that was observed when running the example program with the custom CPI file is approximately 0.75889. Fractionally, the exact value is 36200/47701.

This fraction can be applied to the simulation by using the `cpi_mul` and `cpi_div` model parameters as follows:

```
$PVLIB_HOME/examples/SystemCExport/EVS_Platforms/EVS_Base/Build_Cortex-A75/EVS_Base_Cortex-A75.x
 \
-C Base.bp.secure_memory=0 \
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-9.3/GenericTrace.so \
-C TRACE.GenericTrace.trace-sources=INST \
-C TRACE.GenericTrace.trace-file=trace.txt \
-C Base.cluster0.cpi_mul=36200 \
-C Base.cluster0.cpi_div=47701 \
-a $PVLIB_HOME/images/ta_cpi.axf \
--stat
```

For each instruction, a simulated time of 7589ps or 0.7589 ticks can be observed using the `GenericTrace` plugin. The `--stat` output is as follows and shows the same simulated time value as that obtained using the custom CPI file:

```
--- Base statistics: ----------------------------------------------------------
Simulated time                       : 0.000362s
User time                            : 0.171601s
System time                          : 0.015601s
Wall time                            : 0.196000s
Performance index                    : 0.00
Base.cluster0.cpu0                   : 0.25 MIPS (47701 Inst)
```

In this case, because the same application was run with the custom CPI file and with the average CPI value, an approximation of the average CPI value shows the same overall simulated time. However, the average CPI value for one application is not necessarily an accurate approximation of the average CPI value for a different application.

For example, running the branch prediction example application, described in the next section, clearly shows this difference. Specifying a branch misprediction latency increases the overall simulated time, and therefore gives a different average CPI value to the fixed CPI value that was specified. Using the custom CPI file produces a more accurate average CPI value for the branch prediction example.

Table 6-4: CPI values for simulation with branch prediction latency

Branch prediction example CPI configuration	Overall simulated time in seconds	Average CPI value
Using the average CPI value that was observed in the CPI class example program.	0.001726	1.00754
Using the custom CPI file.	0.001945	1.13538

Related information

Branch prediction example program on page 125

6.6.3 Modeling branch prediction

This section demonstrates various techniques for measuring the effectiveness of different branch prediction algorithms.

6.6.3.1 Branch predictor types and parameters

The `BranchPrediction` plug-in allows you to select the branch prediction algorithm to use, the type of statistics to collect, and the misprediction latency.

The plug-in parameters that are used in this tutorial are as follows:

Table 6-5: `BranchPrediction` plug-in parameters

Plug-in parameter	Purpose in this example	Values that are used in this example
`predictor-type`	Comparing the impact of different branch prediction algorithms.	• `FixedDirectionPredictor` • `BiModalPredictor` • `GSharePredictor` • `CortexA53Predictor`
`mispredict-latency`	Simulating the additional latency due to a pipeline flush that is caused by a branch misprediction.	11. This value is the minimum pipeline flush length for a Cortex-A75 processor.
`bpstat-pathfilename`	Providing statistics about the branch prediction behavior, to determine per-branch and overall predictor accuracy.	`stats.txt`

The different predictor types that are used in this example behave as follows:

`FixedDirectionPredictor`

Always predicts branches as `TAKEN`.

`BiModalPredictor`

Uses a 2-bit state machine to classify branches as one of `STRONGLY_NOT_TAKEN`, `WEAKLY_NOT_TAKEN`, `WEAKLY_TAKEN`, or `STRONGLY_TAKEN`, and predicts accordingly. Tracks up to 512 individual branch instructions by address.

`GSharePredictor`

Uses the history of the eight most recently executed branch instructions to classify a set of branch instructions, based on the instruction address, as one of `STRONGLY_NOT_TAKEN`, `WEAKLY_NOT_TAKEN`, `WEAKLY_TAKEN`, or `STRONGLY_TAKEN`, and predicts accordingly. Unlike the `BiModalPredictor`, it is not limited to a specific number of branch instruction addresses, but it is less precise than `BiModalPredictor`.

`CortexA53Predictor`

Implements the Cortex®-A53 branch prediction algorithm.

Related information

BranchPrediction

6.6.3.2 Generating branch misprediction statistics

There are two ways to trace branch mispredictions when running an application:

- Use the statistics that are produced by the `BranchPrediction` plug-in to get an overall picture, without context about the execution order.
- Load the `BranchPrediction` plug-in and use the MTI trace sources `INST`, `BRANCH_MISPREDICT`, and `WAYPOINT` to see branch misprediction details for individual instructions in execution order.

6.6.3.2.1 BranchPrediction plug-in statistics

The statistics feature of the `BranchPrediction` plug-in provides overall and per-branch statistics, which are saved to a file when the model exits. You can specify the filename and location using the `bpstat-pathfilename` parameter.

The overall branch prediction statistics are described in the following table:

Table 6-6: Overall statistics

Statistic	Description	Example
`Processor Core`	Name of the core to which the branch prediction plug-in was connected.	`ARM_Cortex-A75`
`Cluster instance`	The cluster number in the processor.	0
`Core instance`	The core number in the cluster.	0
`Mispredict Latency`	The branch misprediction latency as specified using the `mispredict-latency` parameter.	11
`Image executed`	The name of the application file that was executed.	`ta_brpred.axf`
`PredictorType`	The branch prediction algorithm as specified using the `predictor-type` parameter.	`FixedDirectionPredictor`
`Total branch calls`	The total number of times all branch instructions were executed.	37434
`Total Mispredictions`	The total number of mispredictions for all executed branch instructions.	5106
`Average prediction accuracy`	The fraction of all branch instructions that were correctly predicted.	0.8636
`Conditional Branches`	The total number of unique conditional branch instructions. This figure does not include the instructions `CBZ` and `CBNZ`.	123
`Total unique branch instructions`	The total number of unique conditional and unconditional branch instructions.	300

The following table shows the `BranchPrediction` plug-in statistics for each unique branch instruction. They can be used to analyze how a given branch prediction algorithm behaves with a particular type of branch instruction. The branch prediction example program uses this information to determine how effectively the different branch prediction algorithms predict different types of branches.

Table 6-7: Per-branch statistics

Statistic	Description	Example
`PC Addr`	The address of the branch instruction.	`0x8000062c`
`Calls`	The total number of times the branch was called.	2100
`Mispredict`	The total number of times the branch was mispredicted.	260
`Accuracy`	The fraction of calls to the branch instruction that were correctly predicted.	0.87619

Related information

Branch prediction example program on page 125
Branch predictor types and parameters on page 121

6.6.3.2.2 MTI trace sources

INST, BRANCH_MISPREDICT, and WAYPOINT are trace sources that can be used in combination to get useful information about branch mispredictions.

Whenever the BranchPrediction plug-in makes a branch misprediction, the BRANCH_MISPREDICT trace source prints the address of the branch instruction that was mispredicted. This address can be compared with the address from the corresponding INST trace event to determine the exact branch instruction involved. The number of BRANCH_MISPREDICT entries for a given branch address at the end of the simulation matches the Mispredict count for that address that is shown in the BranchPrediction plug-in statistics file.

The WAYPOINT trace source prints an event whenever an effective branch operation takes place. This event includes the address of the branch instruction, the target address of the branch, whether the branch is conditional, and whether it was taken. This trace source requires instruction prefetching to be enabled. Combined with a BRANCH_MISPREDICT trace event, it can be used to determine whether a branch was mispredicted as TAKEN or NOT_TAKEN.

To collect trace from these sources, run the model with the GenericTrace and BranchPrediction plug-ins. For example:

```
$PVLIB_HOME/examples/SystemCExport/EVS_Platforms/EVS_Base/Build_Cortex-A75/EVS_Base_Cortex-A75.x
 \
-C Base.bp.secure_memory=0 \
-C Base.cache_state_modelled=1 \
-C Base.cluster0.icache-prefetch_enabled=1 \
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-9.3/BranchPrediction.so \
-C BranchPrediction.BranchPrediction.predictor-type=FixedDirectionPredictor \
-C BranchPrediction.BranchPrediction.mispredict-latency=11 \
-C BranchPrediction.BranchPrediction.bpstat-pathfilename=stats.txt \
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-9.3/GenericTrace.so \
-C TRACE.GenericTrace.trace-sources=INST,BRANCH_MISPREDICT,WAYPOINT \
-C TRACE.GenericTrace.trace-file=trace.txt \
-a $PVLIB_HOME/images/ta_brpred.axf \
--stat
```

Related information

Calculating the execution time of an instruction on page 112

6.6.3.2.3 Example trace for a branch misprediction

The following example trace is for a branch misprediction with a misprediction latency of 11 ticks:

```
INST: PC=0x0000000080000628 OPCODE=0x7100655f SIZE=0x04 MODE=EL1h ISET=AArch64
PADDR=0x0000000080000628 NSDESC=0x01 PADDR2=0x0000000080000628 NSDESC2=0x01 NS=0x01
ITSTATE=0x00 INST_COUNT=0x000000000001080b LOCAL_TIME=0x000000000003f7a0
CURRENT_TIME=0x000000002eab53a0 CORE_NUM=0x00 DISASS="CMP      w10,#0x19"

INST: PC=0x000000008000062c OPCODE=0x54000168 SIZE=0x04 MODE=EL1h ISET=AArch64
PADDR=0x000000008000062c NSDESC=0x01 PADDR2=0x000000008000062c NSDESC2=0x01 NS=0x01
ITSTATE=0x00 INST_COUNT=0x000000000001080c LOCAL_TIME=0x0000000000041eb0
CURRENT_TIME=0x000000002eab7ab0 CORE_NUM=0x00 DISASS="B.HI     {pc}+0x2c ;
 0x80000658"

WAYPOINT: PC=0x000000008000062c ISET=AArch64 TARGET=0x0000000080000658
```

```
TARGET_ISET=AArch64 TAKEN=N IS_COND=Y CORE_NUM=0x00

BRANCH_MISPREDICT: PC=0x000000008000062c

INST: PC=0x0000000080000630 OPCODE=0x7100151f SIZE=0x04 MODE=EL1h ISET=AArch64
PADDR=0x0000000080000630 NSDESC=0x01 PADDR2=0x0000000080000630 NSDESC2=0x01 NS=0x01
ITSTATE=0x00 INST_COUNT=0x000000000001080d LOCAL_TIME=0x000000000005f370
CURRENT_TIME=0x000000002ead4f70 CORE_NUM=0x00 DISASS="CMP      w8,#5"
```

The following information can be gathered from this trace:

- The branch instruction at address 0x8000062c was mispredicted, as shown by the BRANCH_MISPREDICT trace event.
- The branch was conditional, and was incorrectly predicted as TAKEN, as shown by the TAKEN=N field in the WAYPOINT trace event. The PC field value from this source must correspond to the PC field value from the BRANCH_MISPREDICT source.
- As a result of the misprediction, the instruction following the branch instruction took 120,000 picoseconds, or 12 ticks to complete. The misprediction latency was defined as 11 ticks, so the instruction would have taken only 1 tick to complete if the branch had been predicted correctly. The execution time is the difference between:
  - The CURRENT_TIME value for the INST trace before the BRANCH_MISPREDICT trace.
  - The CURRENT_TIME value for the INST trace after the BRANCH_MISPREDICT trace.

  The branch instruction itself took 10,000 picoseconds, or one tick to complete. This is important, as it shows that the misprediction latency is added to the instruction after the mispredicted branch instruction, not to the branch instruction itself. The execution time is the difference between the CURRENT_TIME values for the INST traces corresponding to the branch instruction and the instruction before.

The rest of this tutorial uses these techniques to compare the different branch prediction algorithms.

6.6.3.3 Branch prediction example program

This example is designed to use various types of branch operations that can take place during the execution of a program.

These operations are:

- A branch to skip a loop after a fixed number of iterations has completed.
- A branch to skip a code sequence, depending on the value of a variable.
- A branch to skip a code sequence, which can only be executed a limited number of times consecutively, if a previous branch was taken.
- A branch for a condition that is always true if the conditions for two previous branches were true.
- A branch for a condition that is always true if the conditions for two previous branches were false.

The code operation is trivial. It looks for acronyms within the following constant string, and loops over this operation a set number of times:

```
Timing annotation can be used with an SVP, an EVS, or an ISIM.
```

The following code shows the branch operations of interest:

```
#define MAX_LENGTH 5
#define LOOP_COUNT 20
…
// A: loop not entered 1/LOOP_COUNT times
for(j = 0; j < LOOP_COUNT; j++) {
 printf("Starting iteration #%d\n", j);
 blockCount = 0;
 c = 0;
 resetOnly(&acronymLength, acronym);
 // B: loop not entered 1/length times
 for(i = 0; i < length; i++) {
  c = string[i];
  // C: condition true
  // (number_of_block_letters)/(total_characters_in_string) times
  if (c >= 'A' && c <= 'Z') {
   blockCount++;
   // D: condition true up to MAX_LENGTH times consecutively
   if (acronymLength < MAX_LENGTH) {
    acronym[acronymLength] = c;
   }
   // E: condition true up to MAX_LENGTH+1 times consecutively
   if (acronymLength <= MAX_LENGTH) {
    acronymLength++;
   }
  }
  else {
   // F: condition true if E was true then C was false
   if (acronymLength > 1 && acronymLength <= MAX_LENGTH) {
    printAndReset(&acronymLength, acronym);
   }
   // G: condition true if E was false then C was false
   else if (acronymLength != 0) {
    resetOnly(&acronymLength, acronym);
   }
  }
 }
}
```

The branch instructions that are assembled for the conditions A to G in this code snippet can be examined using branch prediction statistics and trace sources.

The conditions are described in the following table. The branch behavior column describes the relationship between the condition and the associated branch instruction.

Table 6-8: Branch behavior for each condition

Condition	Description	Compiled instruction	Branch behavior
A	Outer loop for processing string `LOOP_COUNT` times. Loop not entered `1/LOOP_COUNT` times.	`B.NE 0x800005f4` at address `0x80000698`.	Backwards branch. Taken to start of loop if more iterations remain.

Condition	Description	Compiled instruction	Branch behavior
B	Inner loop for iterating through characters in the string.	`B.NE 0x80000618` at address `0x8000068c`.	Backwards branch. Taken to start of loop if more iterations remain.
C	Condition true if the character being processed is upper case.	`B.HI 0x80000658` at address `0x8000062c`.	Forwards branch. Taken if the condition is false. Skips code that handles upper case characters.
D	Condition true up to `MAX_LENGTH` times consecutively.	`B.GE 0x80000644` at address `0x80000634`.	Forwards branch. Taken if the condition is false. Skips code that appends a letter to an acronym.
E	Condition true up to `MAX_LENGTH+1` times consecutively.	`B.GT 0x80000684` at address `0x80000648`.	Forwards branch. Taken if the condition is false. Skips code that increments the acronym length.
F	Condition true if E was true, after which C was false.	`B.HI 0x80000674` at address `0x80000660`.	Forwards branch. Never taken if the condition was true, that is, branch E was not taken and then branch C was taken. Skips the code to print a completed acronym.
G	Condition true if E was false, after which C was false.	`CBZ w8,0x80000684` at address `0x80000674`.	Forwards branch. Never taken if the condition was true, that is, branch E was taken then branch C was taken. Skips the code to clear the saved acronym.

6.6.3.4 Running the simulation

To generate trace and statistics for comparing the performance of the different branch predictors, run the simulation with the `BranchPrediction` plug-in parameters shown here.

For example, to use the `FixedDirectionPredictor`, launch the model using the following command, where `ta_brpred.axf` is the name of the executable image and `EVS_Base_Cortex-A75.x` is the platform executable:

```
$PVLIB_HOME/examples/SystemCExport/EVS_Platforms/EVS_Base/Build_Cortex-A75/EVS_Base_Cortex-A75.x
 \
-C Base.bp.secure_memory=0 \
-C Base.cache_state_modelled=1 \
-C Base.cluster0.icache-prefetch_enabled=1 \
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-9.3/BranchPrediction.so \
-C BranchPrediction.BranchPrediction.predictor-type=FixedDirectionPredictor \
-C BranchPrediction.BranchPrediction.mispredict-latency=11 \
-C BranchPrediction.BranchPrediction.bpstat-pathfilename=stats.txt \
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-9.3/GenericTrace.so \
-C TRACE.GenericTrace.trace-sources=INST,BRANCH_MISPREDICT,WAYPOINT \
-C TRACE.GenericTrace.trace-file=trace.txt \
-a $PVLIB_HOME/images/ta_brpred.axf \
--stat
```

The program prints the following output to the terminal:

```
Looking for acronyms of maximum length 5 in the string:
Timing annotation can be used with an SVP, an EVS, or an ISIM.

Starting iteration #0
SVP
EVS
ISIM
…
Starting iteration #19
SVP
EVS
ISIM

Info: /OSCI/SystemC: Simulation stopped by user.

--- Base statistics: ----------------------------------------------------------
Simulated time                          : 0.002275s
User time                               : 0.343203s
System time                             : 0.202801s
Wall time                               : 0.642064s
Performance index                       : 0.00
Base.cluster0.cpu0                      : 0.31 MIPS (      171308 Inst)
-------------------------------------------------------------------------------
```

You can now analyze the end of simulation statistics, the branch prediction statistics file `stats.txt`, and the MTI trace file `trace.txt`, that are generated for each branch predictor type.

Related information

Branch predictor types and parameters on page 121

6.6.3.5 Comparison of branch predictor types

Statistics about the accuracy of the different branch predictors for the various types of branch instructions can now be compared.

These statistics are shown in the following table:

Table 6-9: Comparison of branch predictor accuracy

		Branch instruction						
Branch predictor	**Statistic**	**A**	**B**	**C**	**D**	**E**	**F**	**G**
All	Calls	20	2100	2100	260	260	1840	1800
	`TAKEN`	19	2080	1840	0	0	1800	1800
	`NOT_TAKEN`	1	20	260	260	260	40	0
`FixedDirectionPredictor`	Mispredictions	1	20	260	260	260	40	0
	Mispredicted as `TAKEN`	1	20	280	260	260	40	0
	Mispredicted as `NOT_TAKEN`	0	0	0	0	0	0	0
	Accuracy (%)	95*	99*	88*	0	0	98*	100*
`BiModalPredictor`	Mispredictions	1	20	341	1	1	40	0
	Mispredicted as `TAKEN`	1	20	220	1	1	40	0
	Mispredicted as `NOT_TAKEN`	0	0	121	0	0	0	0

		Branch instruction						
Branch predictor	**Statistic**	**A**	**B**	**C**	**D**	**E**	**F**	**G**
	Accuracy (%)	95*	99*	84	100*	100*	98*	100*
`GSharePredictor`	Mispredictions	1	20	279	241	241	40	0
	Mispredicted as `TAKEN`	1	20	260	241	241	40	0
	Mispredicted as `NOT_TAKEN`	0	0	19	0	0	0	0
	Accuracy (%)	95*	99*	87	7	7	98*	100*
`CortexA53Predictor`	Mispredictions	1	23	324	2	1	49	0
	Mispredicted as `TAKEN`	1	20	221	2	1	40	0
	Mispredicted as `NOT_TAKEN`	0	3	103	0	0	9	0
	Accuracy (%)	95*	99*	85	99	100*	97	100*

The accuracy figures have been rounded to the nearest percentage. For each branch instruction type, A to G, the entry for the best accuracy is shown with an asterisk. As expected, different branch prediction algorithms are better suited to different types of branch instructions.

With the `FixedDirectionPredictor`, all branches are predicted as `TAKEN`, so the accuracy is equal to the percentage of calls to that branch that were `TAKEN`.

With the `BiModalPredictor` and `GSharePredictor` algorithms, only the random branch C was mispredicted both as `TAKEN` and `NOT_TAKEN`. With the other systematic branches, the misprediction was always in one direction. The result is different for the more complex algorithm of the `CortexA53Predictor`, which has mispredictions in both directions for systematic branches as well.

The `BiModalPredictor` is able to store the history of individual branches, and is therefore most accurate with predicting branches with a deterministic ratio between the number of times they are `TAKEN` and `NOT_TAKEN`. This accuracy can be seen with branches A, B, D, and E. With a more random branch, such as C, which depends entirely on the contents of a user-defined string, relying on the history of the branch proves ineffective.

Interestingly, the `GSharePredictor` appears to be highly inaccurate at predicting branches D and E. These branches are `NOT_TAKEN` a fixed number of times consecutively. However, since there are calls to many other branches between consecutive calls to these branches, the `GSharePredictor`'s global history is not able to use the specific outcome of these branches to update their prediction values effectively.

Overall, the `BiModalPredictor` and the `CortexA53Predictor` have predicted these branch instructions most accurately, as shown in the following table:

Table 6-10: Overall branch predictor accuracy

Predictor type	Overall accuracy (%)
`FixedDirectionPredictor`	86
`BiModalPredictor`	98
`GSharePredictor`	86
`CortexA53Predictor`	98

6.6.3.6 Impact of branch misprediction on simulation time

You can directly observe the impact of mispredictions on the overall simulation time, as shown in the `--stat` output after the model exits.

The simulated execution times with the different branch predictors are shown in the following table.

Note

The execution times also include the impact of branch mispredictions that occur in other parts of the code, as well as in the startup and shutdown sequences.

Table 6-11: Overall simulation time for each predictor type

Predictor type	Simulation time with `mispredict-latency=11`	Simulation time with `mispredict-latency=0`
`FixedDirectionPredictor`	0.002275s	0.001713s
`BiModalPredictor`	0.001805s	0.001713s
`GSharePredictor`	0.002289s	0.001713s
`CortexA53Predictor`	0.001806s	0.001713s

7. FastRAM

FastRAM is a bus optimization for Fast Models that can bring significant speed improvements to platform models.

7.1 Introducing FastRAM, a bus optimization for Fast Models

FastRAM is a fast interface to simulated RAM which allows platform models to avoid using bus models for most transactions.

FastRAM uses a cache of DMI pointers, each of which points to 64MB. This memory is tightly coupled to the Fast Models bus masters and models of IP that are bus masters. When FastRAM is enabled, accesses by Fast Models bus masters to platform RAM components do not use the PVBus or TLM bus models. Accesses to other platform components and areas of RAM for which FastRAM has not been enabled work as normal.

FastRAM can give significant speed improvements to large and complex platform models which can spend a lot of time in the bus models. It can particularly benefit SystemC platforms that use TLM, and multi-threaded platforms.

Most, but not all, platform models can safely use FastRAM. For conditions that can prevent its use, see 7.5 FastRAM limitations on page 134.

The behavior of platform models is functionally equivalent whether FastRAM is enabled or disabled. However, modeling bus transactions in a platform can lead to scheduling changes, so the overall flow of execution by components in a platform might not be identical.

7.2 How to enable FastRAM

Enable FastRAM by launching the platform model with the command-line parameter `--fast-ram <config_file>`.

The configuration file is an ASCII file located in the current working directory of the simulation that specifies:

- One or more physical address ranges to enable for FastRAM.
- Details of any address aliasing for the enabled ranges.
- Which bus masters to enable to use FastRAM.

7.3 FastRAM configuration file syntax

Each line in the configuration file starts with a single character option followed by the required arguments, separated with whitespace.

The following options are available:

T

Enable FastRAM trace on stdout from this point in the file.

Q

Disable FastRAM trace on stdout from this point in the file.

Note

The position of the `T` and `Q` options in the file is significant:

- To enable FastRAM trace during the entire initialization and runtime, start the file with `T` and do not use `Q`.
- To enable FastRAM trace during runtime but not initialization, end the file with `T` and do not use `Q`.
- To enable FastRAM trace during the initialization only, start the file with `T` and end the file with `Q`.
- To enable FastRAM trace during specific parts of the initialization, use one or more pairs of `T` and `Q` within the file.

S

Optimize FastRAM for single-threaded simulations.
In Fast Models 11.28, this option is deprecated. If used, FastRAM outputs a warning and ignores it. It is unnecessary because FastRAM automatically detects and optimises for a single-threaded context.

N

Disable MTE support with FastRAM if the platform has enabled the MTE feature. Enabled by default. See 7.5 FastRAM limitations on page 134 for requirements on the tag store.

F

Disable atomic memory operations through FastRAM. Enabled by default.

M <*`string`*> | ALL

Identify the bus masters to use FastRAM. You can select either masters whose id contains <*`string`*> or all masters. This option can be specified multiple times. For example, to enable FastRAM use by all masters with `A57` or `R52` in their id, specify:

```
M A57
M R52
```

Note

If the argument to `M` is not `ALL` and trace is enabled, then the ids of all masters are shown on the console with a message stating whether the master is enabled for FastRAM or not. To find the list of masters, use `M foo` then use the list to select the masters required.

+ <*base*> <*size*>

Add the physical address range <*base*> to <*base*>+<*size*>.

- <*base*> <*size*>

Remove the physical address range <*base*> to <*base*>+<*size*>.

= <*base-a*> <*base-b*> <*size*>

Alias a physical address range.

# <text>

Comment.

Note

All addresses and sizes must be 64MB-aligned (`0x4000000`) hexadecimal.

7.4 FastRAM configuration file example

This example FastRAM configuration file is written for a Base Platform FVP.

It does the following:

- Uses the `T` option at the start of the file to enable FastRAM trace output from the start of the FastRAM initialization.
- Enables FastRAM for the address range `0x08_00000000-0xff_ffffffff`.
- Defines `0x00_80000000-0x00_ffffffff` as an alias for the range `0x08_00000000-0x08_7fffffff`.
- Uses the `Q` option at the end of the file to disable FastRAM trace output at the end of the FastRAM initialization.

```
# FastRAM config file for FVP Base
T
M ALL
+ 800000000 F800000000
= 80000000 800000000 80000000
Q
```

If FastRAM has been successfully enabled, it prints the following output:

```
FastRAM: CONSTRUCTED
FastRAM: Address space size = 40 bits
```

```
FastRAM: Slab size = 64 Mb
FastRAM: Page size = 4 kb
FastRAM: Singleton size = 147 kb
FastRAM: Number of monitors = 16
FastRAM: Enable ALL masters
FastRAM: Add range 0x08_00000000...ff_ffffffff
FastRAM: Add range 0x00_80000000...00_ffffffff
FastRAM: Alias range 0x00_80000000...00_ffffffff <=> 0x08_00000000...08_7fffffff
```

7.5 FastRAM limitations

FastRAM can be used with most, but not all, platform models.

It can be used in a platform in which all of the following conditions are true:

- The platform contains one or more very frequently accessed RAM components that are a whole multiple of 64MB in size.
- These RAM components are always mapped to the same static physical range as seen by the bus masters that frequently access the RAM.
- The physical ranges used to access the RAM components by the bus masters can include aliased regions.
- The RAM components and the buses to them always give back DMI and for a given physical address always give back exactly the same DMI pointer and never invalidate DMI.
- Either all the bus masters in the platform use FastRAM for the configured physical ranges or you can identify the subset of masters that can use it by name. See 7.3 FastRAM configuration file syntax on page 131 for how to find the list of bus masters.
- All the bus masters that use FastRAM use the same physical address map to access the RAM components.
- If the RAM components internally allocate memory that is a whole multiple of 64MB, then FastRAM can be used with RAM instances that are accessed by:
  - Bus masters that are enabled to use FastRAM.
  - Bus masters that are not, or cannot, be enabled to use FastRAM.
- If the RAM components internally allocate memory that is not a whole multiple of 64MB, for example the RAMDevice LISA component, then FastRAM can only be used with RAM instances that are accessed by masters that are enabled to use FastRAM.

It cannot be used in a platform if any of the following conditions are true:

- Cache state modeling is enabled.
- The physical address map used by the bus masters to access the RAM is dynamic and can change at run time.
- The set of bus masters that will use FastRAM cannot be identified. See 7.3 FastRAM configuration file syntax on page 131 for how to find the list of bus masters.
- There is System IP between the bus masters and the RAM that needs to provide functionality other than a global monitor. However, a CCI or CCN with cache state modeling disabled is allowed.

- The platform RAM is mapped to an address greater than or equal to `0x100_0000_0000`.
- The expected functionality of the platform depends on being able to invalidate DMI. FastRAM ignores DMI invalidations other than what is required internally to support exclusives and RevokeReadOnWrite behavior.

To enable MTE support through FastRAM:

- The platform must use a single system-wide tag store that returns DMI that can be safely extrapolated to 64MB.
- If the platform uses a CI700, or a similar model, where each SN-F has its own tag store, these must be disabled. See the `bypass_tag_cache` parameter on the model. Also, the platform must have a PVMetaDataController close to the DRAM.
- FastRAM does not support platforms that use tag carveout.

Appendix A SystemC Export generated ports

This appendix describes Fast Models SystemC Export generated ports.

A.1 About SystemC Export generated ports

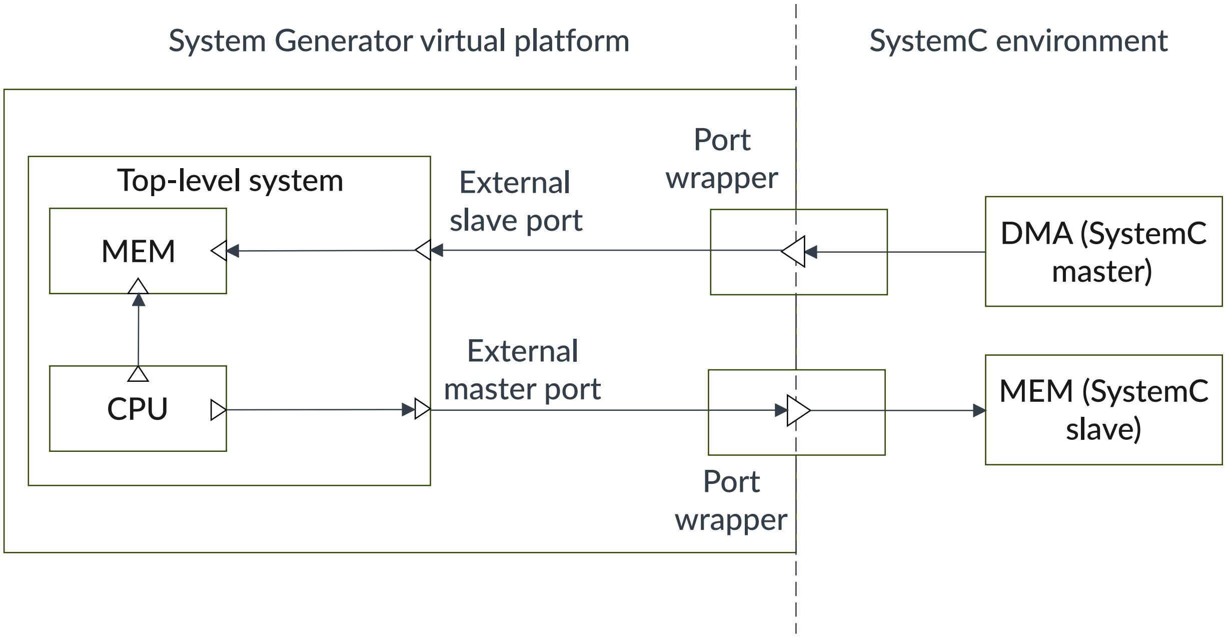
The generated SystemC component must have SystemC ports to communicate with the SystemC world. The SystemC Export feature automatically generates these ports from the Fast Models ports of the top-level component.

Caution

Although it is possible to export your own protocols, Arm strongly recommends using the AMBA-PV protocols provided and bridge from these in SystemC, if needed.

The SystemC export feature automatically generates port wrappers that bind the SystemC domain to the Fast Models virtual platform.

Figure A-1: Port wrappers connect Fast Models and SystemC components



Each master port in the Fast Models top level component results in a master port on the SystemC side. Each slave port in the Fast Models top level component results in a slave port (export) on the SystemC side.

For Fast Models to instantiate and use the ports, it requires protocol definitions that:

- Correspond to the equivalent SystemC port classes.
- Refer to the name of these SystemC port classes.

This effectively describes the mapping from Fast Models port types (protocols) to SystemC port types (port classes).

Related information

Fast Models Reference Guide

Proprietary Notice

This document is protected by copyright and other related rights and the use or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm Limited ("Arm"). No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether the subject matter of this document infringes any third party patents.

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm’s view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Reference by Arm to any third party’s products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Product and document information

Read the information in these sections to understand the release status of the product and documentation, and the conventions used in the Arm documents.

Product status

All products and Services provided by Arm require deliverables to be prepared and made available at different levels of completeness. The information in this document indicates the appropriate level of completeness for the associated deliverables.

Product completeness status

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

Revision history

These sections can help you understand how the document has changed over time.

Document release information

The Document history table gives the issue number and the released date for each released issue of this document.

Document history

Issue	Date	Confidentiality	Change
1128-00	19 February 2025	Non-Confidential	Update for v11.28.
1127-00	16 September 2024	Non-Confidential	Update for v11.27.
1126-00	19 June 2024	Non-Confidential	Update for v11.26.
1125-00	13 March 2024	Non-Confidential	Update for v11.25.
1124-00	6 December 2023	Non-Confidential	Update for v11.24.
1123-00	13 September 2023	Non-Confidential	Update for v11.23.

Issue	Date	Confidentiality	Change
1122-00	14 June 2023	Non-Confidential	Update for v11.22.
1121-00	22 March 2023	Non-Confidential	Update for v11.21.
1120-00	7 December 2022	Non-Confidential	Update for v11.20.
1119-00	14 September 2022	Non-Confidential	Update for v11.19.
1118-00	15 June 2022	Non-Confidential	Update for v11.18.
1117-00	16 February 2022	Non-Confidential	Update for v11.17.
1116-00	6 October 2021	Non-Confidential	Update for v11.16.
1115-00	29 June 2021	Non-Confidential	Update for v11.15.
1114-00	17 March 2021	Non-Confidential	Update for v11.14.
1113-00	9 December 2020	Non-Confidential	Update for v11.13.
1112-00	22 September 2020	Non-Confidential	Update for v11.12.
1111-00	9 June 2020	Non-Confidential	Update for v11.11.
1110-00	12 March 2020	Non-Confidential	Update for v11.10.
1109-00	28 November 2019	Non-Confidential	Update for v11.9.
1108-01	3 October 2019	Non-Confidential	Update for v11.8.1.
1108-00	5 September 2019	Non-Confidential	Update for v11.8.
1107-00	17 May 2019	Non-Confidential	Update for v11.7.

Issue	Date	Confidentiality	Change
1106-00	26 February 2019	Non-Confidential	Update for v11.6.
1105-00	23 November 2018	Non-Confidential	Update for v11.5.
1104-01	17 August 2018	Non-Confidential	Update for v11.4.2.
1104-00	22 June 2018	Non-Confidential	Update for v11.4.
1103-00	23 February 2018	Non-Confidential	Update for v11.3.
1102-00	17 November 2017	Non-Confidential	Update for v11.2.
1101-00	31 August 2017	Non-Confidential	Update for v11.1.
1100-00	31 May 2017	Non-Confidential	Update for v11.0. Document numbering scheme has changed.

For information about the functional changes to Fast Models, see the Fast Models Release Notes.

Conventions

The following subsections describe conventions used in Arm documents.

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: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.

Convention	Use
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<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	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.

Caution

We recommend the following. If you do not follow these recommendations your system might not work.

Warning

Your system requires the following. If you do not follow these requirements your system will not work.

Danger

You are at risk of causing permanent damage to your system or your equipment, or of harming yourself.

Note

This information is important and needs your attention.

Tip

This information might help you perform a task in an easier, better, or faster way.

Remember

This information reminds you of something important relating to the current content.

Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

Table 1: Arm publications

Document name	Document ID	Licensee only
Arm® Architecture Models	-	No
Download FlexNet Publisher	-	No
Fast Models Reference Guide	100964	No
Fixed Virtual Platforms	-	No
Fast Models Fixed Virtual Platforms Reference Guide	100966	No
Fast Models Tools User Guide	109415	No
How do I ensure my Fast Model works with User Based Licensing (UBL)?	ka005524	No
Iris User Guide	101196	No
IrisSupportLib Reference Guide	101319	No
LISA+ Language for Fast Models Reference Guide	101092	No
Open Source Software and Platforms wiki on Arm® Community	-	No
Product Download Hub	-	No
User-based Licensing User Guide	102516	No

Table 2: Arm publications

Document name	Document ID	Licensee only
Arm® Architecture Reference Manual for A-profile architecture	DDI 0487	Non-Confidential

Table 3: Other publications

Document ID	Organization	Document name
-	Accellera Systems Initiative	*Accellera Systems Initiative (ASI)*
-	Android Open Source Project	*Android Partitions*
IEEE 1666-2005	IEEE Standards association	*IEEE Standard SystemC(R) Language Reference Manual*
-	Microsoft	*Microsoft Visual C++ Redistributable*