



Fast Models

Version 1.0

Tutorials

Non-Confidential

Copyright © 2023–2025 Arm Limited (or its affiliates).
All rights reserved.

Issue 04

107644_0100_04_en



Fast Models Tutorials

This document is Non-Confidential.

Copyright © 2023–2025 Arm Limited (or its affiliates). All rights reserved.

This document is protected by copyright and other intellectual property rights.

Arm only permits use of this document if you have reviewed and accepted [Arm's Proprietary Notice](#) found at the end of this document.

This document (107644_0100_04_en) was issued on 2025-11-19. There might be a later issue at <https://developer.arm.com/documentation/107644>

The product version is 1.0.

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 who are constructing, building, and running Fast Models system 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. Constructing a simple Fast Model.....	5
1.1 Constructing the simple model.....	5
1.2 Code to run on the simple model.....	10
1.3 Compiling the code to run on the simple model.....	11
1.4 Running the code on the simple model.....	12
2. Constructing a dual core Fast Model.....	13
2.1 Constructing the dual core model.....	13
2.2 Code to run on the dual core model.....	15
2.3 Compiling the code to run on the dual core model.....	17
2.4 Running the code on the dual core model.....	18
3. Using Arm Development Studio to debug Fast Models systems.....	19
3.1 Introduction to Arm Development Studio.....	19
3.2 Import the FVP into Arm Development Studio.....	19
3.3 Create a debug configuration and launch the FVP.....	23
3.4 Step through the application.....	24
3.5 View the contents of memory.....	25
3.6 Useful information.....	27
4. Toggle trace using ToggleMTIPlugin.....	28
4.1 Introduction to ToggleMTIPlugin.....	28
4.2 Installation and setup.....	28
4.3 How to use ToggleMTIPlugin.....	29
4.4 Toggle trace using a debugger.....	29
4.5 Toggle trace using HLT instructions.....	33
5. Generating MTI trace from a LISA component.....	36
5.1 What is MTI?.....	36
5.2 Setting up the trace source in the LISA component.....	36
5.3 Generating trace data in the LISA component.....	37
5.4 What is in the example and what does it do?.....	38
5.5 Run the GeneratingTraceFromLISA example.....	39

6. Dynamically driving signals with SignalDriver.....	40
6.1 What is the SignalDriver component and what is it designed to do?.....	40
6.2 How is SignalDriver implemented?.....	40
6.3 Using the parameter to change the signal.....	41
6.4 Using the register to change the signal.....	42
6.5 Using the bus to change the signal.....	43
6.6 SignalDriver example.....	44
7. Connecting multiple clusters with PVCoherentInterconnect.....	49
7.1 What is the PVCoherentInterconnect component and what is it designed to do?.....	49
7.2 How is PVCoherentInterconnect implemented?.....	49
7.3 Example system using the interconnect.....	51
7.4 Code to run on the example model.....	53
7.5 Compiling the code to run on the example model.....	60
7.6 Running the code and testing coherency on the example model.....	60
7.7 Using Model Trace to show coherency.....	65
8. Timing annotation.....	71
8.1 Prerequisites.....	71
8.2 Build the FVP_Base_Cortex-A57 example platform.....	71
8.3 Calculate the execution time of an instruction using the default CPI value.....	71
8.4 Display the total execution time of the simulation.....	72
8.5 Calculate the average CPI value.....	73
8.6 Run the example with a non-default CPI value.....	73
8.7 Additional timing annotation features.....	74
Proprietary notice.....	76
Product and document information.....	78
Product status.....	78
Revision history.....	78
Conventions.....	79
Useful resources.....	81

1. Constructing a simple Fast Model

In this tutorial, we will build a simple model containing a single-core ARMCortexR52CT component. Then we will write hello world and run it on the model.

1.1 Constructing the simple model

Model generation uses Arm tools for construction and compilation of the model.

Before you begin

The two tools we will use are:

- A canvas tool to place components and connect them. It is called System Canvas.
- A generator tool to take the files generated by the canvas and build them into a runnable model. It is called simgen.

The simgen tool is callable from within the canvas tool. We shall be using this method for model generation.

Both these tools may be downloaded from the Arm website as part of the Arm Fast Models package.

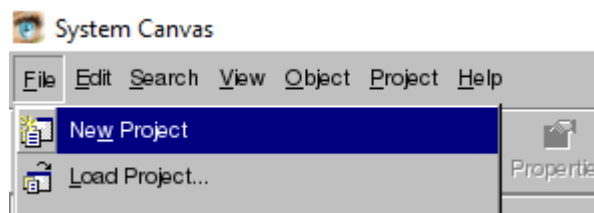


These are concise instructions for constructing a model using System Canvas. Refer to the [Fast Models Tools User Guide](#) if the location of a particular menu item or step is unclear.

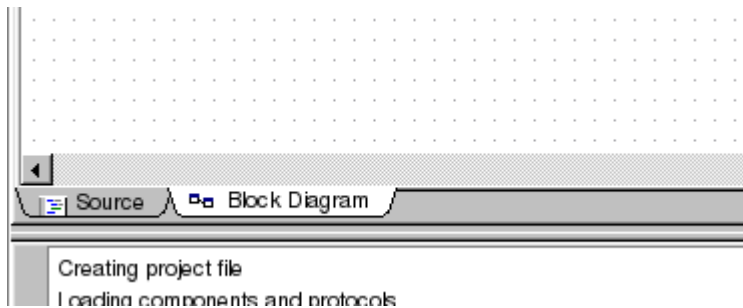
Procedure

1. Start System Canvas, either on the command line by typing `sgcanvas` in the terminal in Linux (assuming the executable is on your path) or using the Start Menu in Windows.
2. Click **File > New Project** and give a name to your project, for example `r52example`. Accept the default name for the top component and LISA file. Make sure you are saving the project and files in a location you have write access to.

Figure 1-1: Create new Project

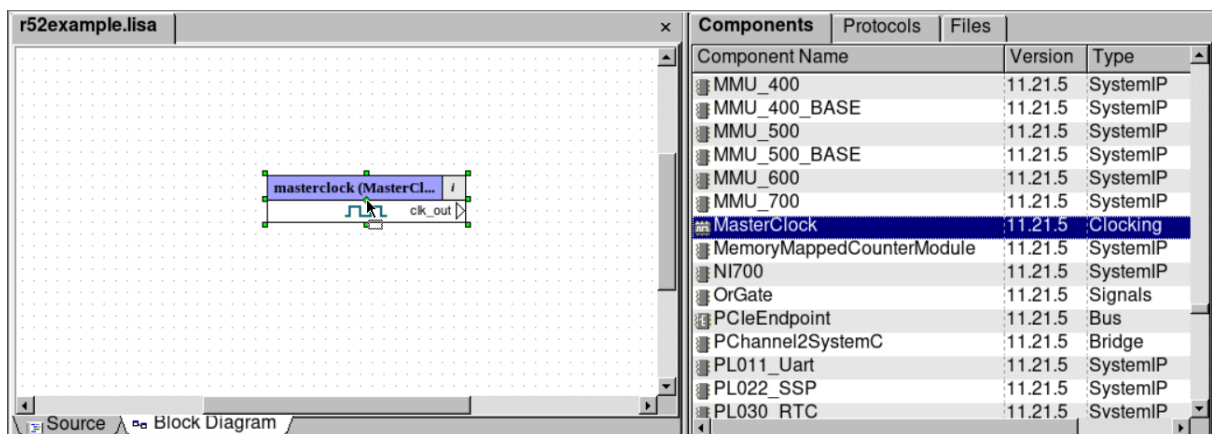


3. Click on the **Block Diagram** tab.

Figure 1-2: Click Block Diagram tab

4. Add the following components to the canvas by clicking and dragging them from the Component list on the right onto the canvas:

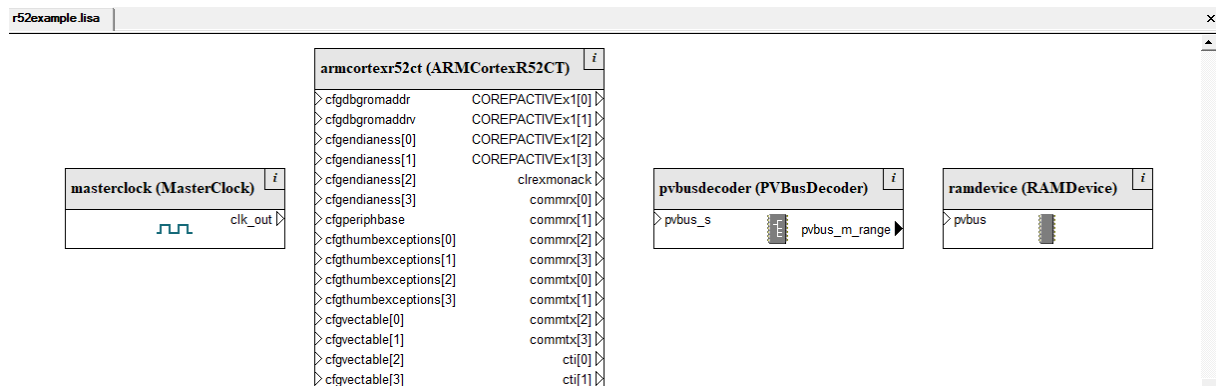
- MasterClock
- ARMCortexR52CT
- PVBusDecoder
- RAMDevice

Figure 1-3: Drag components**Note**

The ARMCortexR52CT component has a `num_cores` parameter which determines the number of cores in the cluster for the component. As we want a single-core cluster, we can leave it set to the default value of 1.

The **Block Diagram** view shows the components:

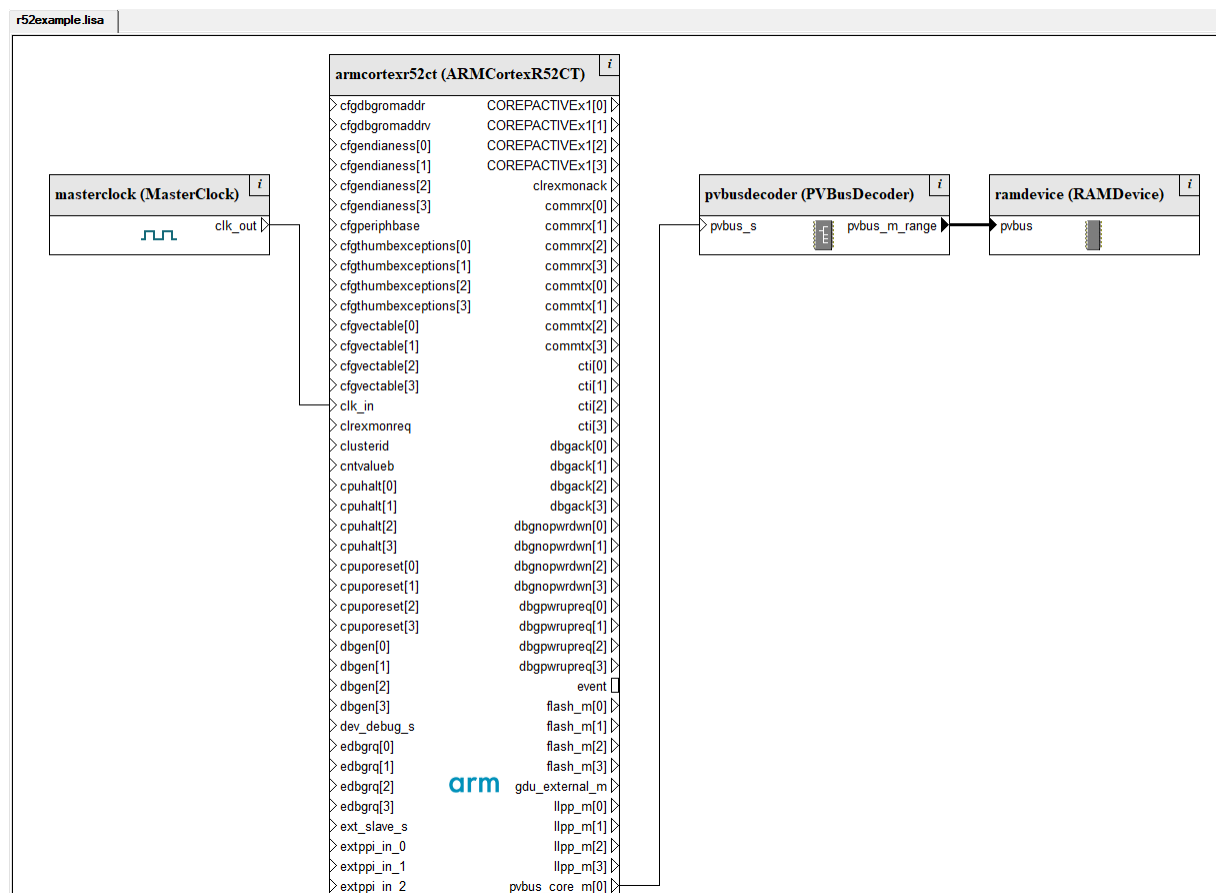
Figure 1-4: All components in Canvas



5. Connect the components using the **Connect** tool on the tool bar:

- Connect MasterClock `clk_out` to ARMCortexR52CT `clk_in`
- Connect ARMCortexR52CT `pvbus_core_m[0]` to PVBusDecoder `pvbus_s`
- Connect PVBusDecoder `pvbus_m_range` to RAMDevice `pvbus`

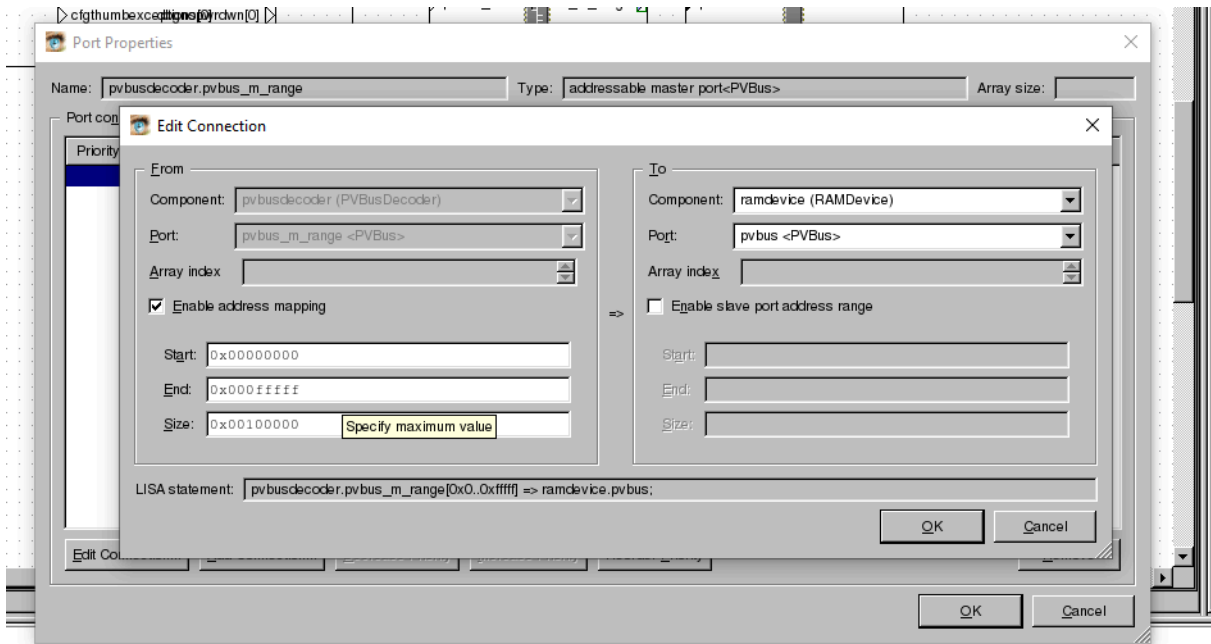
Figure 1-5: Connected system



6. Map the PVBUSDecoder to your RAMDevice:

- Right click on the `pvbust_m_range` port on the PVBUSDecoder to get to **Object Properties**
- Click **Edit Connection** on the singular connection in the **Port Properties** dialog box
- Select the **Enable address mapping** checkbox and give the connection a start address of `0x0` and an end address of `0x0FFFFFF` (1MB mapped to RAM)

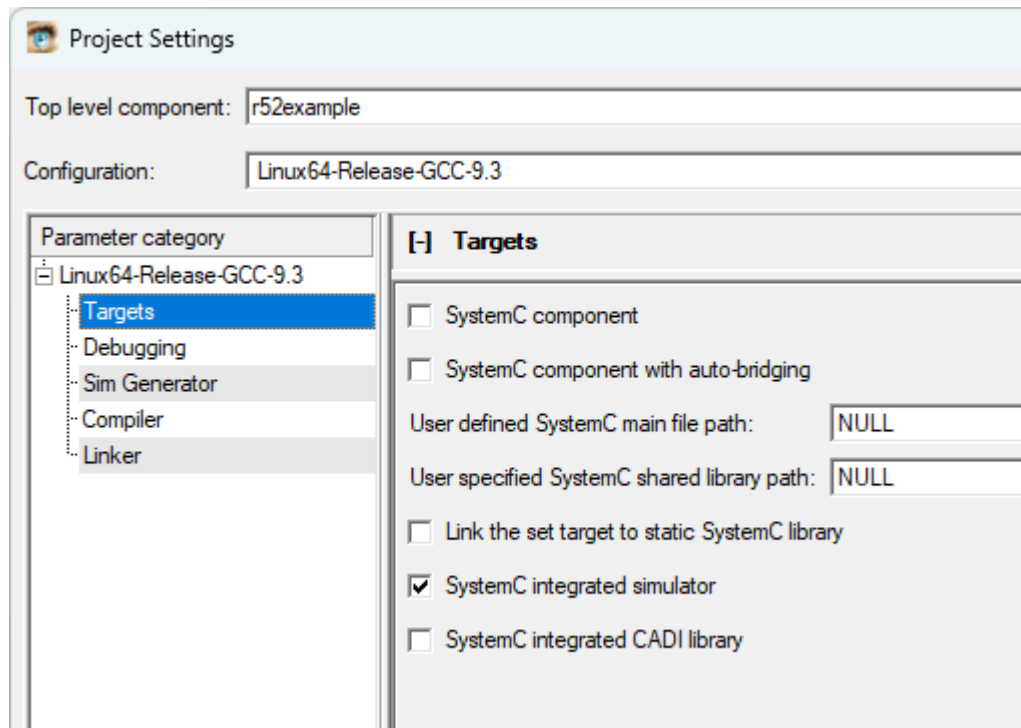
Figure 1-6: Mapping the PVBUSDecoder



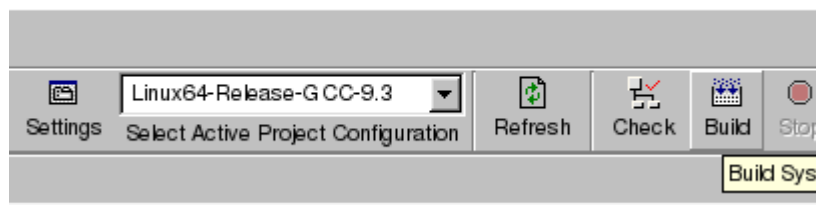
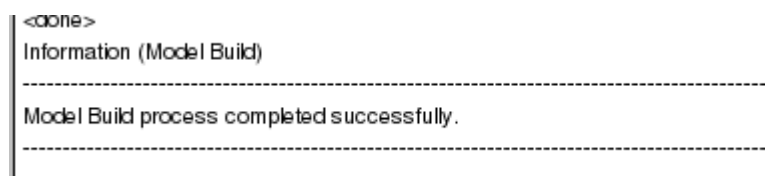
At this point if you click on the **Source** tab you will see:

```
// This file was generated by System Generator Canvas
// -----
component r52example
{
  composition
  {
    pvbusdecoder : PVBUSDecoder();
    armcortexr52ct : ARMCortexR52CT();
    ramdevice : RAMDevice();
    masterclock : MasterClock();
  }
  connection
  {
    masterclock.clk_out => armcortexr52ct.clk_in;
    armcortexr52ct.pvbust_core_m[0] => pvbusdecoder.pvbust_s;
    pvbusdecoder.pvbust_m_range[0x0..0xffff] => ramdevice.pvbust;
  }
}
```

7. In **Settings** select the **Configuration** and set the Target to be a SystemC Integrated Simulator. Close the **Settings** dialog box. Then select **File > Save All** to save the modified settings.

Figure 1-7: Set your target

8. Build the model by clicking the **Build** button. If the build is successful, the log view should display the text `Model Build process completed successfully`. The model is called `isim_system` and is created in a subdirectory named after the build configuration (for example `<build_dir>/Linux64-Release-GCC-9.3/isim_system`).

Figure 1-8: Build the model**Figure 1-9: Successful build**

1.2 Code to run on the simple model

In this section, we will write some code to run on the model we have just built.

Procedure

1. Open your preferred text editor
2. Start with:

```
.section BOOT, "ax"  
.align 3
```

This names the section `BOOT` and uses the flags `"ax"` to indicate that it is all memory resident during execution (see [ELF image SHF_ALLOC attribute](#)) and executable (see [ELF image SHF_EXECINSTR attribute](#))

In addition the alignment of this section is 8 bytes (2^3 bytes).

3. Add some defines:

```
.equ SH_TRAP_INST_A32,      0x123456  
.equ SYS_WRITE0,           0x4  
.equ SYS_EXIT,             0x18
```

This defines 3 constants. In order:

- The SVC instruction magic value to call Semihost functionality
- The Semihost string write call magic value
- The Semihost exit call magic value

Semihost is a special feature that permits a core to call to either the simulation host or debug host to access some features provided by the host such as file or console access. It is accessed by setting register `R0` to a magic value for the particular service call, register `R1` to the argument and then executing the SVC instruction (which is normally used for triggering exceptions) with a magic value so that the host traps the breakpoint rather than the core.

By using Semihost we do not need to add UARTs or a CLCD, or program a driver for them, to have output. This can aid the rapid bring-up of software.

4. Add section definitions:

```
.global start  
.type start, "function"  
start:
```

The last line puts the label `start` in the code. The two previous lines define it as a global so it is visible externally, and declare it as a function. This means `start` can be used as the entry point for this code block.

5. Add register initialization:

```
// Clear registers  
// -----  
  
MOV      R0, #0
```

```
MOV    R1, #0
```

This example zeros the registers before it uses them. This is not required because on a Fast Model, registers are set to zero at reset. It is done to encourage good practice, because on real hardware, after reset, the registers are set to an unknown value and cause X-propagation if they are not set to a known value.

Some cores have an input signal to set all architectural registers to a known value, after reset, to aid with this issue.

6. Add the main print code:

```
// -----
// Main of your bare-metal software
// -----
main:
    LDR    R1, =print_hello
    BL     semihost_print

    // Semihosting exit
    MOV    R0, #SYS_EXIT
    SVC    #SH_TRAP_INST_A32

// -----
// Semihost call
// -----
semihost_print:
    MOV    R0, #SYS_WRITE0
    SVC    #SH_TRAP_INST_A32
    BX     LR

// -----
// String literals
// -----
print_hello:
    .string "Hello world from the Cortex-R52 Fast Model!\n"
```

We load register R1 with the address of the string held at label `print_hello`. Then we branch to `semihost_print` which loads the register R0 with the `sys_write0` value then executes a `SVC` with the Semihost magic value. This prints the string pointed to by R1 to the console of the host. We then branch to `exit` which executes a semihost `sys_exit` which, if the host supports it, will halt simulation.

7. Save the code to the file `startup.s`.

1.3 Compiling the code to run on the simple model

After we have written the assembly code, we need to compile it into an ELF (Executable and Linker Format) image.

Procedure

1. Ensure that Arm Compiler for Embedded is in your `$PATH`.
2. Compile `startup.s` into an object file `startup.o`:

```
armclang -c --target=arm-arm-none-eabi -march=armv8r startup.S
```

3. Link the object file `startup.o` into an ELF object that can be run, specifying the location in memory it will load from and what the entry point is:

```
armlink --ro-base=0x8000 startup.o -o image.axf --entry=start
```

1.4 Running the code on the simple model

Running the code on the model you built is simple. The model is an executable called `isim_system` and takes the ELF image created as an argument.

Procedure

- On Linux, if you are using GCC-9.3, the command line is:
`./Linux64-Release-GCC-9.3/isim_system -a image.axf`
- On Microsoft Windows, the command line is:
`.\Win64-Release-VC2019\isim_sytem.exe -a image.axf`

Results

You will then see the following output:

```
Hello world from the Cortex-R52 Fast Model!  
Info: /OSCI/SystemC: Simulation stopped by user.
```

2. Constructing a dual core Fast Model

In this tutorial, we will build a dual core model containing an ARMCortexA57CT component with its `NUM_CORES` parameter set to 2 and write and then run software on it.

2.1 Constructing the dual core model

Model generation uses Arm tools for construction and compilation of the model.

Before you begin

The two tools we will use are:

- A canvas tool to place components and connect them. It is called System Canvas.
- A generator tool to take the files generated by the canvas and build them into a runnable model. It is called simgen.

The simgen tool is callable from within the canvas tool. We shall be using this method for model generation.

Both these tools may be downloaded from the Arm website as part of the Arm Fast Models package.

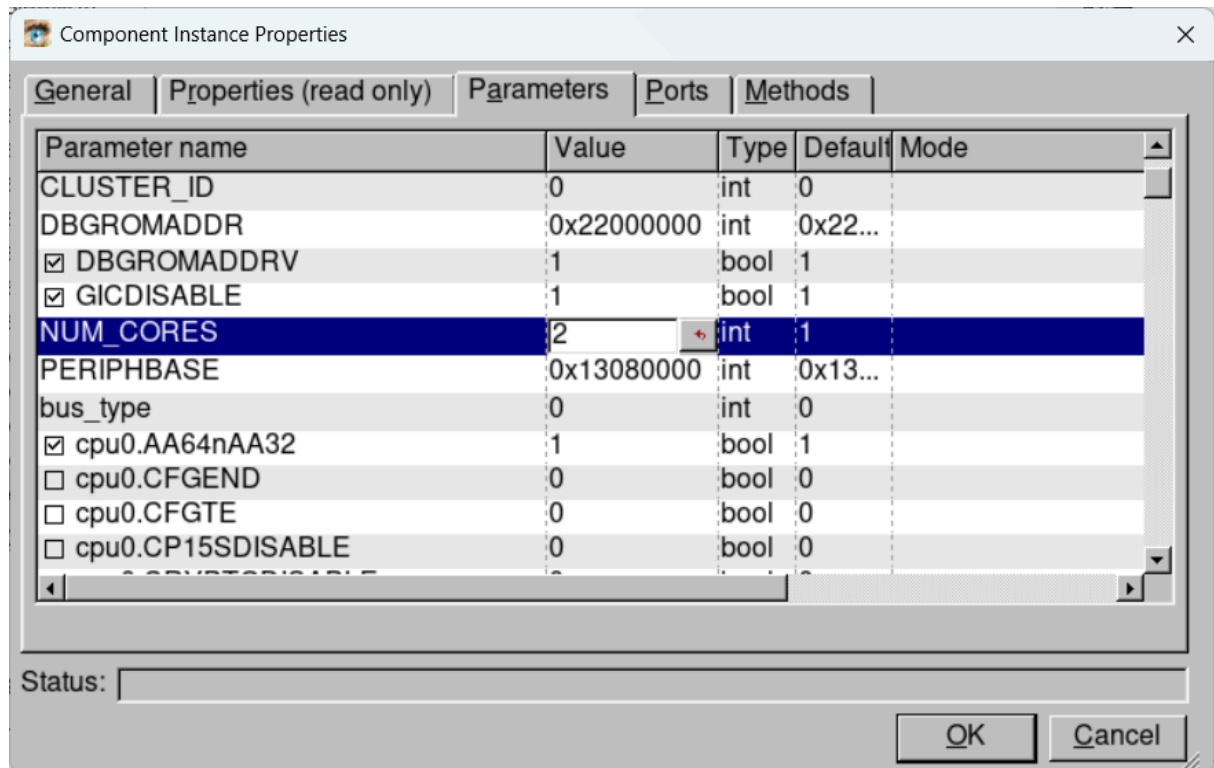


Note

These are concise instructions for constructing a model using System Canvas. Refer to the [Fast Models Tools User Guide](#) if the location of a particular menu item or step is unclear.

Procedure

1. Start System Canvas, either on the command line by typing `sgcanvas` in the terminal in Linux, assuming the executable is on your path, or using the Start Menu in Windows.
2. Click **File**→**New Project** and give a name to your project, for example `a57example`. Accept the default name for the top component and LISA file. Make sure you are saving the project and files in a location you have write access to.
3. Click the **Block Diagram** tab.
4. Add the following components to the canvas by clicking and dragging the components from the **Component** list on the right onto the canvas:
 - MasterClock
 - ARMCortexA57CT
 - RAMDevice
5. Select the ARMCortexA57CT component, then click the **Properties** button.
6. Click the **Parameters** tab, then scroll down to the `NUM_CORES` parameter. Change its value to 2, then click **OK**, then **Save**.

Figure 2-1: Set the NUM_CORES parameter

7. Connect the components using the **Connect** tool on the tool bar:
 - Connect MasterClock `clk_out` to ARMCortexA57CT `clk_in`
 - Connect ARMCortexA57CT `pbus_m0` to RAMDevice `pbus`

At this point if you click on the **Source** tab you will see:

```
// This file was generated by System Generator Canvas
// -----
component a57example
{
  composition
  {
    ramdevice : RAMDevice();
    masterclock : MasterClock();
    armcortexa57ct : ARMCortexA57CT("NUM_CORES"=2);
  }
  connection
  {
    masterclock.clk_out => armcortexa57ct.clk_in;
    armcortexa57ct.pbus_m0 => ramdevice.pbus;
  }
}
```

8. Click the **Settings** button and select the configuration and set the Target to be a systemc Integrated Simulator.
9. Build the model by clicking the **Build** button.

Results

If the build is successful, the log view displays the text `Model Build process completed successfully`. The generated model is called `isim_system` and is created in a subdirectory of the location of the project file, named after the build configuration, for example `<proj_file_dir>/Linux64-Release-GCC-9.3/isim_system`.

2.2 Code to run on the dual core model

In this section, we will write some code to run on the model we have just built.

Procedure

1. Open your preferred text editor
2. Start with:

```
.section BOOT,"ax"  
.align 3
```

This names the section `BOOT` and uses the `"ax"` flags to indicate that it is all memory resident and executable, see [ELF image attributes SHF_ALLOC and SHF_EXECINSTR](#).

In addition the alignment of this section is 8 bytes (2^3 bytes).

3. Add some defines:

```
.equ SH_TRAP_INST_A64,      0xF000  
.equ SYS_WRITE0,            0x4  
.equ SYS_EXIT,              0x18
```

This defines 3 constants. In order:

- The HLT instruction magic value to call Semihost functionality
- The Semihost string write call magic value
- The Semihost exit call magic value

Semihost is a special feature that permits a core to call to either the simulation host or debug host to access some features provided by the host such as file or console access. It is accessed by setting register `X0` to a magic value for the particular service call, register `X1` to the argument and then executing the HLT instruction (which is normally a halting breakpoint) with a magic value so that the host traps the breakpoint rather than the core.

By using Semihost we do not need to add UARTs or a CLCD or program a driver to have output. This can aid the rapid bring-up of software.

4. Add section definitions:

```
.global start64  
.type start64, @function  
start64:
```

The last line puts the label `start64` in the code. The two previous lines define it as a global so is visible externally and declare it as a function. This means `start64` can be used as the entry point for this code block.

5. Add register initialization:

```
// Clear registers
// -----

MOV    x0, #0
MOV    x1, #0
MOV    x2, #0
```

This example zeros the registers before it uses them. This is not required on a Fast Model, because registers are set to zero at reset. It is done to encourage good practice on real hardware, where after reset, the registers are set to an unknown value and cause X-propagation if they are not set to a known value.

Some cores have an input signal to set all architectural registers to a known value, after reset, to aid with this issue.

6. Add code to determine which core we are running on:
When the CPU starts, all cores start running the same code immediately. Add code to determine which core we are running on:

```
// Which core am I
// -----

MRS    x0, MPIDR_EL1
AND     x0, x0, #0xFF           // Mask off to leave Aff0 - this assumes
                                // a pre v8.4 processor
CBZ     x0, primary             // If core 0, run the primary core code
B       secondary               // Else, run secondary cores code
```

First the `MPIDR_EL1` register is read. This contains the affinity information of the core the code is running on. The bottom byte of the affinity holds the core number (in a pre-v8.4A core). We extract the bottom byte and compare with zero. If it is zero we are the primary core otherwise we are one of the secondary cores.

If we are a secondary core we jump to the code at label `secondary` otherwise we continue.

7. Add the main print code:

```
// -----
// Primary core
// -----
primary:
    LDR x1, =print_msg0
    BL semihost_printf

    // Semihosting exit
    MOV    w0, #SYS_EXIT
    HLT    #SH_TRAP_INST_A64

1:
    WFI
    B      1b

// -----
```



```

// Secondary core
// -----
secondary:
    LDR x1, =print_msg1
    BL semihost_print

    // Semihosting exit
    MOV     w0, #SYS_EXIT
    HLT     #SH_TRAP_INST_A64

2:

    WFI
    B       2b

// -----
// Semihost call
// -----
semihost_print:
    MOV     w0, #SYS_WRITE0
    HLT     #SH_TRAP_INST_A64
    RET

// -----
// String literals
// -----
print_msg0:
    .string "Hello from core 0!\n"

print_msg1:
    .string "Hello from core 1!\n"

```

If we are the primary core then execution continues at the `primary:` label.

We load register X1 with the address of the string held at label `print_msg0`. Then we branch to `semihost_print` which loads the bottom word of register X0 with the `SYS_WRITE0` value then execute a `HLT` with the Semihost magic value. This prints the string pointed to by X1 to the console of the host. We then execute `RET` which returns to the instruction after the branch.

After that we execute a Semihost `SYS_EXIT` which, if the host supports it, halts simulation. Otherwise the core executes the following 2 instructions and sits in a WFI loop.

The secondary core(s) branch to `secondary:` and execute a similar code sequence to the primary core except the address of string label `print_msg1` is loaded and so a slightly different string is printed.

8. Save the code to the file `startup.s`.

2.3 Compiling the code to run on the dual core model

After we have written the assembly code, we need to compile it into an ELF (Executable and Linkable Format) image.

Procedure

1. Ensure that Arm Compiler for Embedded is in your `$PATH`. For installation instructions, see [System requirements and installation](#)

2. Compile `startup.s` into an object file `startup.o`:

```
armclang -c --target=aarch64-arm-none-eabi -march=armv8.1-a startup.S
```
3. Link the object file `startup.o` into an ELF object that can be run, specifying the location in memory it will load to and what the entry point is:

```
armlink --ro-base=0x80000000 startup.o -o image.axf --entry=start64
```

2.4 Running the code on the dual core model

Running the code on the model you built is simple. The model is an executable called `isim_system` and takes the ELF image created as an argument.

Procedure

- On Linux, if you are using GCC-9.3, the command line is:

```
./Linux64-Release-GCC-9.3/isim_system -a image.axf
```
- On Microsoft Windows, the command line is:

```
.\Win64-Release-VC2019\isim_sytem.exe -a image.axf
```

Results

You will then see the following output:

```
Hello from core 1!
Hello from core 0!

Info: /OSCI/SystemC: Simulation stopped by user.
```

The order of the print (in this case core 1 before core 0) is not significant. It only depends on which core reached the critical section of calling Semihost first. In addition, Fast Models is single threaded and each core in a multi-core simulation executes a number of instructions in a block before execution continues on another core. Here execution started on core 1 and on reaching the WFI, execution continued on core 0. This sequential execution of instruction blocks is architecturally valid.

3. Using Arm Development Studio to debug Fast Models systems

In this chapter, we will import the Dual Core model into Arm® Development Studio, then debug it using Arm Debugger.

3.1 Introduction to Arm Development Studio

Arm Development Studio is a suite of software development tools for bare-metal and Linux-based Arm systems.

It includes:

- An Eclipse-based Integrated Development Environment (IDE).
- Arm Compiler toolchain.
- Arm Debugger.
- Arm Streamline and Graphics Analyzer performance analysis tools.
- Some Fixed Virtual Platforms (FVPs).

You can import additional FVPs into Arm Development Studio. The Arm Debugger can connect to the built-in and imported FVPs using the Iris debug interface.

In this tutorial, we will:

1. Import the Dual Core FVP which we built previously into Arm Development Studio.
2. Create a debug configuration to customize the debugging session with the FVP.
3. Launch the FVP in Arm Development Studio.
4. Start the Arm Debugger and connect it to the FVP, using the Iris debug interface.
5. Debug software running on the FVP.
6. Give links to more information about Arm Development Studio.

3.2 Import the FVP into Arm Development Studio

First, import the FVP into the Development Studio configuration database, then configure a connection to it.

Before you begin

This tutorial assumes the following:

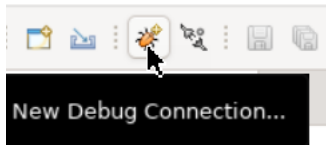
- You have installed Arm Development Studio and have a license to use it. If not, download it from [Arm Development Studio](#).

- You have built the FVP and executable image from the [Constructing a Dual Core Fast Model](#) tutorial.

Procedure

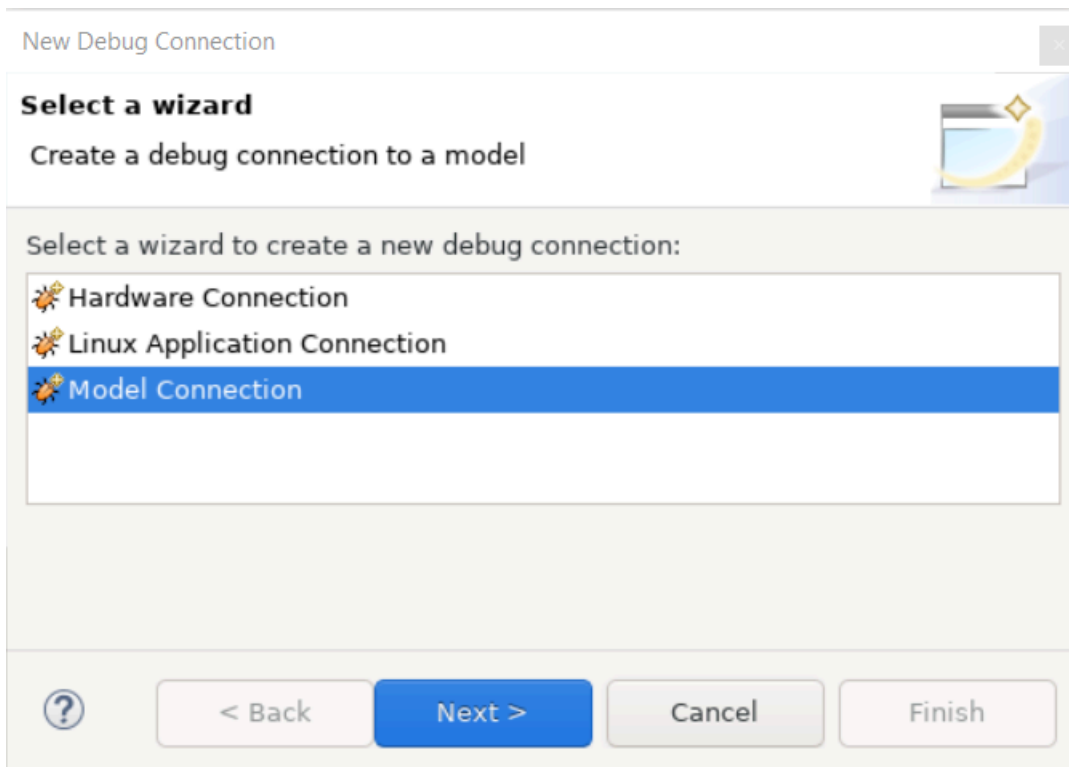
1. Start Arm Development Studio.
2. Click **New Debug Connection...**:

Figure 3-1: New Debug Connection button

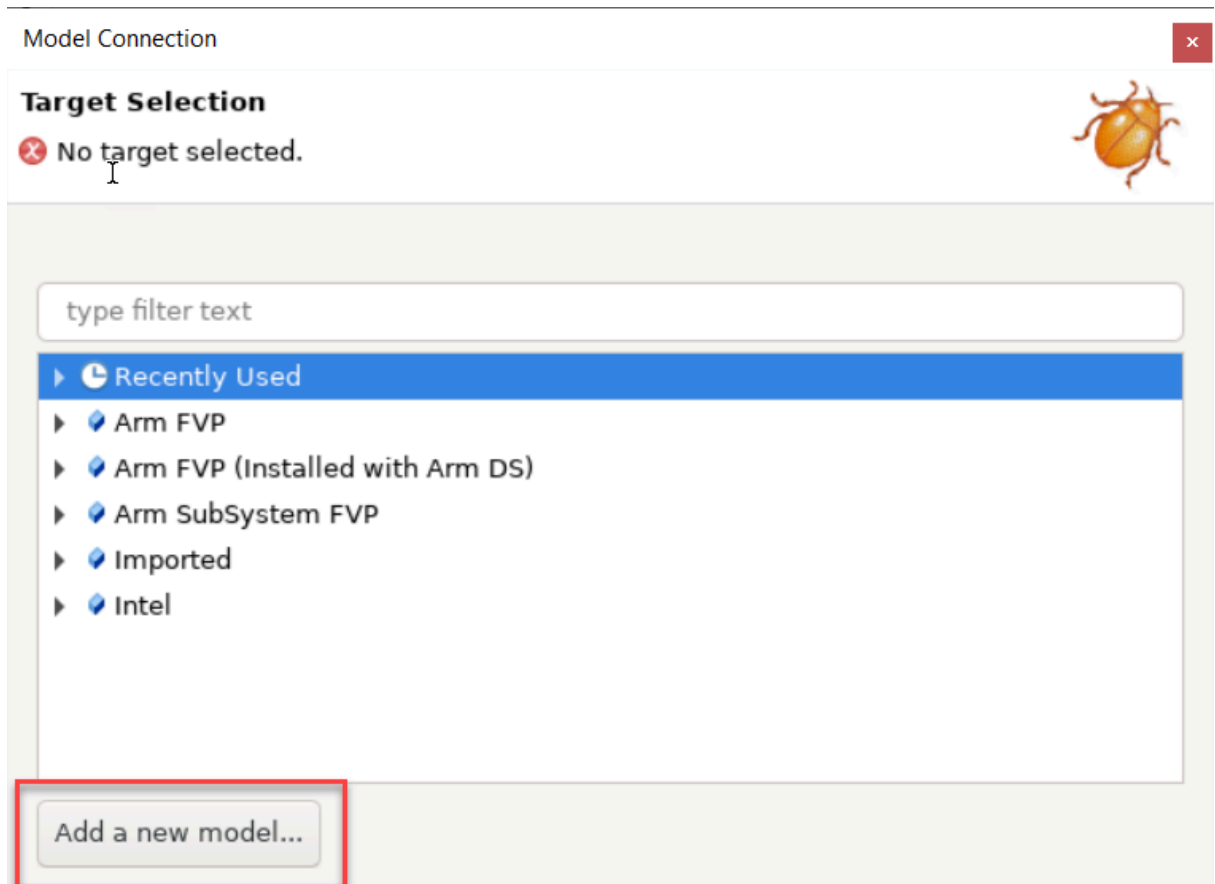


3. In the **New Debug Connection** dialog box, select **Model Connection**, then click **Next**:

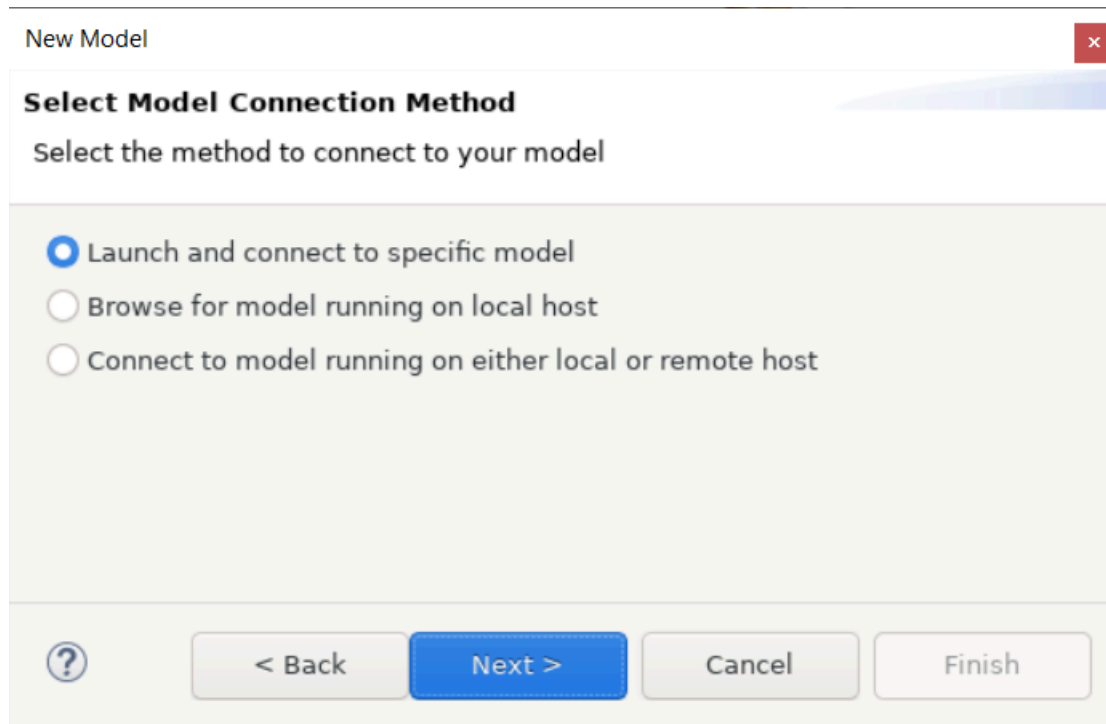
Figure 3-2: New Debug Connection dialog



4. In the **Debug Connection** dialog box, give the debug connection a name, for example **a57example**, then click **Next**.
5. In the **Target Selection** dialog box, click **Add a new model...**:

Figure 3-3: Model Connection dialog

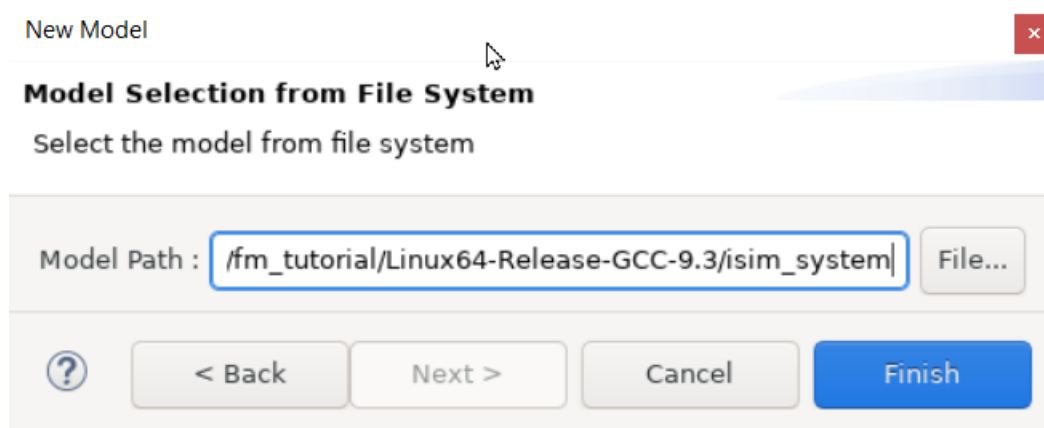
6. In the **Select Model Connection Method** dialog box, select the **Launch and connect to specific model** radio button:

Figure 3-4: New Model dialog

Note that you can only select the other options in this dialog box if you had previously launched the FVP on a local or a remote host using the `--iris-connect` option.

Click **Next**.

7. In the **Model Selection from File System** dialog box, navigate to the FVP called `isim_system` that we built in the [Constructing a Dual Core Fast Model](#) tutorial, then click **Finish**:

Figure 3-5: Model Selection from File System

Arm Development Studio imports the FVP so that it appears in the list of available model connections.

- Click **Finish**. Arm Development Studio displays the **Edit configuration and launch** dialog box.

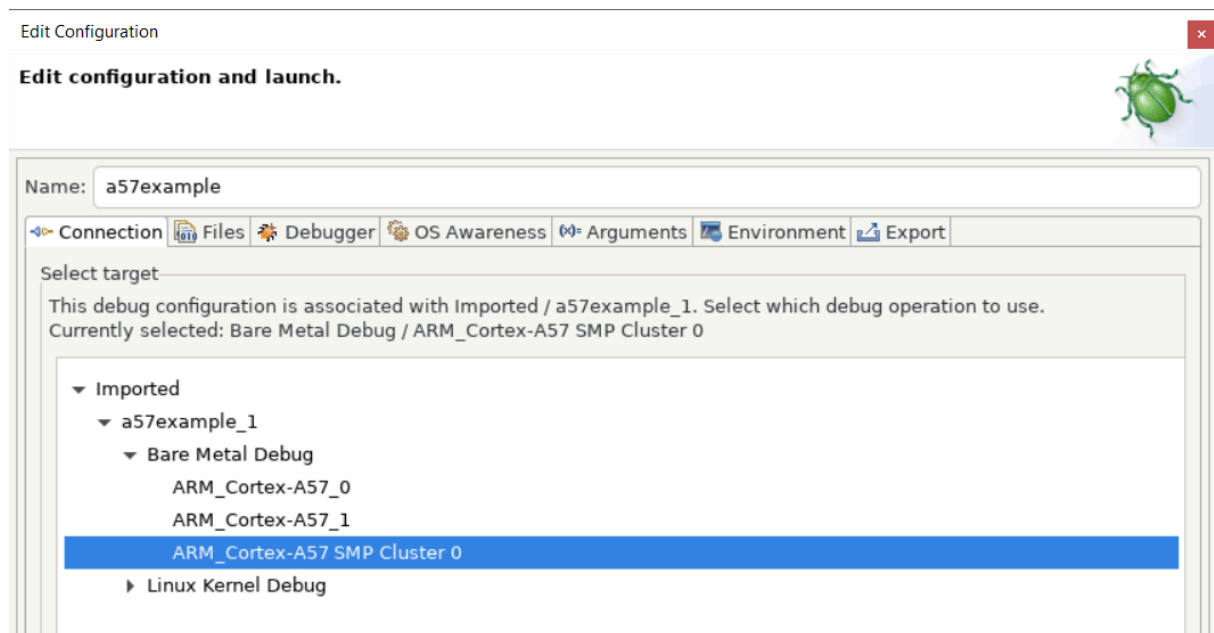
3.3 Create a debug configuration and launch the FVP

Next, use the **Edit configuration and launch** dialog box to configure the debugging session for the FVP that we have just imported.

Procedure

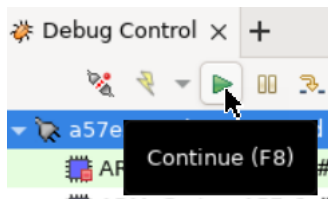
- In the **Connection** tab, select Cluster 0 as the target for the debugger to connect to:

Figure 3-6: Edit configuration and launch



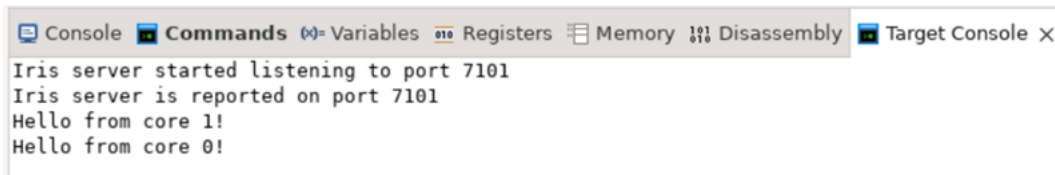
- In the **Files** tab, navigate to the application image file from the Dual Core tutorial, `image.axf`. Arm Development Studio will download this file from the host and run it on the FVP.
- In the **Debugger** tab, select the **Debug from entry point** radio button, then click **Debug**. The Iris server starts running, the FVP is launched, and the debugger connects to the FVP. Execution stops at the image entry point.
- Click **Continue** in the **Debug Control**:

Figure 3-7: Continue button



Results

The **Target Console** displays the output from the application:

Figure 3-8: Target Console output

3.4 Step through the application

Now that we have created a debug configuration, we can debug the application and step through the source code.

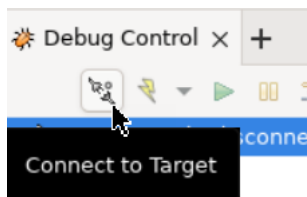
Procedure

1. To perform source-level debugging, we must first rebuild `image.axf` to include debug symbols because by default, the Arm Compiler, `armclang` does not produce them. Run the same `armclang` and `armlink` commands as before to rebuild the image, but add the `-g` option to `armclang`:

```
armclang -c -g --target=aarch64-arm-none-eabi -march=armv8.1-a startup.s  
armlink --ro-base=0x80000000 startup.o -o image.axf --entry=start64
```

Note that Arm Debugger is compatible with DWARF format version 4, which is the version produced by the `-g` option.

2. In the **Debug Control**, click **Connect to Target**:

Figure 3-9: Connect to Target button

3. Arm Development Studio now displays the source file `startup.s` in the Editor view, highlighting in green the line at which execution stopped:

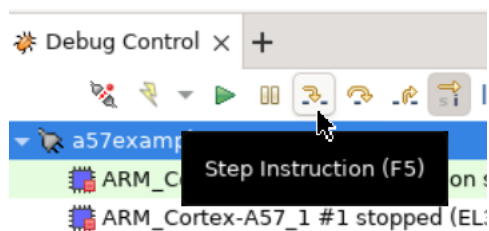
Figure 3-10: Editor view

```

1.section B00T,"ax"
2.align 3
3.equ SH_TRAP_INST_A64, 0xF000
4.equ SYS_WRITE0, 0x4
5.equ SYS_EXIT, 0x18
6.global start64
7.type start64, @function
8.start64:
9// Clear registers
10// -----
11
12MOV x0, #0
13MOV x1, #0
14MOV x2, #0
15// Which core am I
16// -----

```

4. Use the different run control icons in the **Debug Control**, for example, **Step Instruction** to step through the source code:

Figure 3-11: Step instruction button

3.5 View the contents of memory

Arm Development Studio offers many views to show different types of input and output from the debug session.

About this task

For a list of all available views, see [Perspectives and Views](#) in *Arm Development Studio User Guide*. For example:

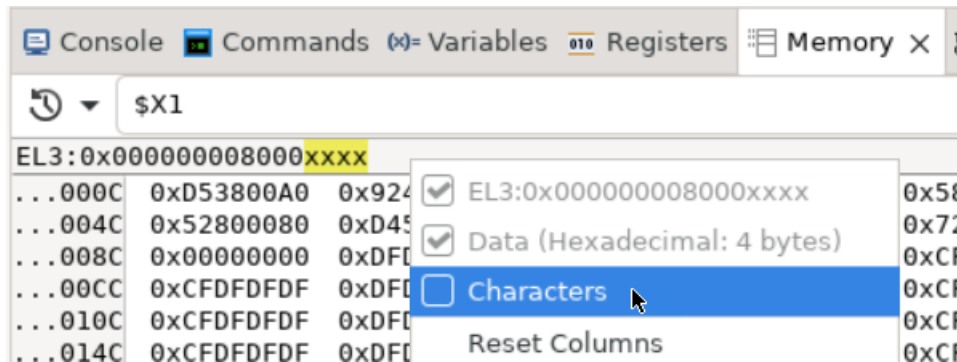
- The **Commands** view displays messages output by the debugger. You can also use it to send commands to the debugger. For example, in the **Command** field, type `stepi` then click **submit**, to step the code by one instruction. For a full list of commands, see [Arm Debugger commands listed in alphabetical order](#).
- The **Memory** view displays the contents of memory on the target.

Procedure

1. If you have disconnected from the target, reconnect to it. Click **Continue**. Execution halts at the breakpoint.
2. Click on the **Memory** tab and type `$x1` in the **Address** field.

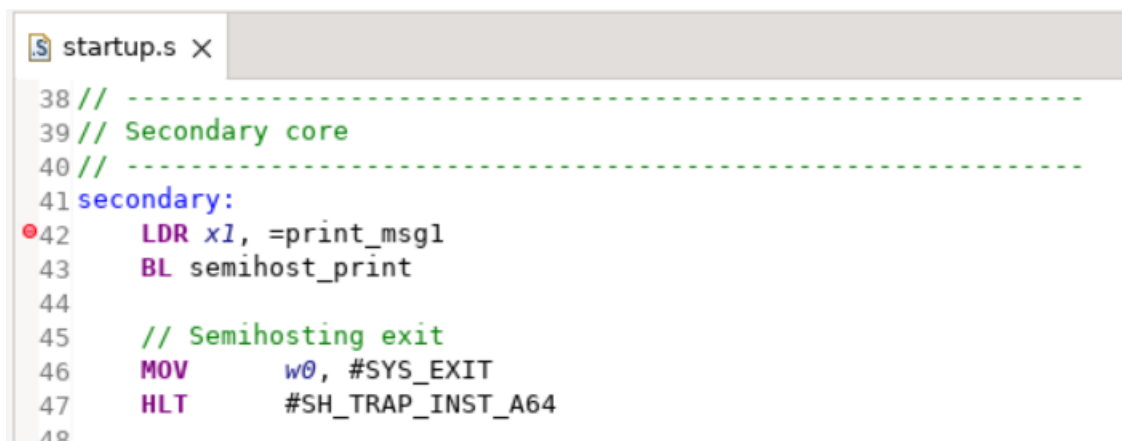
- Right click in the column heading and select the **Characters** checkbox to display the **Characters** column, then press **Enter**:

Figure 3-12: Adding the Characters column



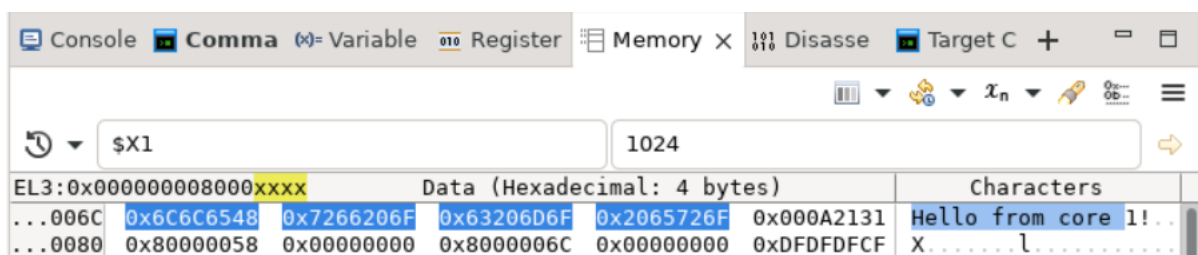
- Add a breakpoint to the line following the label `secondary` in the source code by double clicking in the column containing the line number:

Figure 3-13: Add a breakpoint



- Press **F5** to step a single instruction. The **Characters** column now shows the contents of the address in `x1`.
- Highlight the string `Hello from core 1!` and the memory view highlights the hexadecimal values of the ASCII characters stored in memory:

Figure 3-14: Memory view



3.6 Useful information

For more information about using FVPs in Arm Development Studio, see the *Arm Development Studio Getting Started Guide*.

- [Using FVPs with Arm Development Studio](#).
- [Connect to new or custom models](#).

4. Toggle trace using ToggleMTIPlugin

In this chapter, we will show two ways of using ToggleMTIPlugin to turn trace generation on or off during a simulation.

4.1 Introduction to ToggleMTIPlugin

Tracing an entire simulation session can generate a huge amount of data, which can make it difficult to find what you are looking for and can significantly impact performance. You can avoid these problems in different ways.

For example:

- Some trace plug-ins have parameters to start and stop trace at a specific instruction count, or to restrict trace to a subset of trace sources.
- Some trace plug-ins allow you to load an additional plug-in called ToggleMTIPlugin to restrict trace to the specific parts of the code you are interested in. This is the method we will demonstrate in this tutorial.

For more information about ToggleMTIPlugin, see [ToggleMTIPlugin](#) in *Fast Models Reference Guide*.

To complete this tutorial, you need:

- An installation of Fast Models. To toggle trace using a debugger, we will use Iris Monitor, which is an Iris-enabled debugger that is included in Fast Models version 11.27 or later.
- To toggle trace using HLT instructions you need a compiler to rebuild the source code. See [Requirements for Fast Models](#) in *Fast Models User Guide* for a list of supported operating systems and compilers.
- An image to run on the FVP. This tutorial uses one of the sample images included in the Fast Models package in the directory `$PVLIB_HOME/images/`.

4.2 Installation and setup

To demonstrate ToggleMTIPlugin we first need to build a Fixed Virtual Platform (FVP).

Procedure

1. Build one of the Fast Models example FVPs to run the image on. The following command builds the FVP_Base_Cortex-A57 example, creating an executable FVP called `isim_system`:

```
cd $PVLIB_HOME/examples/LISA/FVP_Base/Build_Cortex-A57
simgen -b -p FVP_Base_Cortex-A57.sgproj --configuration Linux64-Release-GCC-9.3
```

2. Launch `isim_system`, loading the `brot_ve_64.axf` image. Also load the TarmacTrace plug-in:

```
./Linux64-Release-GCC-9.3/isim_system -C bp.secure_memory=false \
-a $PVLIB_HOME/images/brot_ve_64.axf \
--plugin $PVLIB_HOME/plugins/Linux64_GCC-9.3/TarmacTrace.so
```

By default, the TarmacTrace plug-in traces all instructions and also some other events throughout the simulation. Notice how slowly the simulation runs, and the large amount of trace data that it outputs to the console.

3. Press Ctrl+C to stop the execution of the model.

Next steps

Next, we will repeat this command, except we will also load ToggleMTIPlugin to turn TarmacTrace on and off at specific points in the code.

4.3 How to use ToggleMTIPlugin

ToggleMTIPlugin can be used in one of two ways in a simulation session. Use the `TRACE.ToggleMTIPlugin.use_hlt` parameter to select which one to use.

It can have one of the following values:

TRACE.ToggleMTIPlugin.use_hlt=0

This method toggles trace using a parameter `TRACE.ToggleMTIPlugin.disable_mti_runtime` that you can set at runtime using a debugger or Python script. It turns trace on when you set it to 0 or off when you set it to 1.

TRACE.ToggleMTIPlugin.use_hlt=1

This method toggles trace using special `Hlt` instructions inserted into the image being traced. To use this method you must be able to modify and rebuild the source code.

4.4 Toggle trace using a debugger

To demonstrate this method, we will use the Iris Monitor debugger.

Procedure

1. Run the FVP using the same command that we used previously, but add these extra parameters:

```
--plugin $PVLIB_HOME/plugins/Linux64_GCC-9.3/ToggleMTIPlugin.so
```

Loads ToggleMTIPlugin. It must be the last plug-in to be loaded in the command.

```
-C TRACE.ToggleMTIPlugin.use_hlt=0
```

Selects the debugger method of toggling trace.

```
-C TRACE.ToggleMTIPlugin.disable_mti_from_start=1
```

Disables trace from the start of the simulation. We will enable it when execution reaches the region of interest.

```
-C cluster0.NUM_CORES=1
```

Model uses a single core, instead of the default, which is 4 cores.

-I

Starts the Iris server, which enables Iris Monitor to connect to the simulation.

The full command is:

```
./isim_system -C bp.secure_memory=false \
-a $PVLIB_HOME/images/brot_ve_64.axf \
--plugin $PVLIB_HOME/plugins/Linux64_GCC-9.3/TarmacTrace.so \
--plugin $PVLIB_HOME/plugins/Linux64_GCC-9.3/ToggleMTIPlugin.so \
-C TRACE.ToggleMTIPlugin.use_hlt=0 \
-C TRACE.ToggleMTIPlugin.disable_mti_from_start=1 \
-C cluster0.NUM_CORES=1 \
-I
```

The simulation starts running and displays a visualization window. It shows an instruction count of zero because it is waiting for a debugger to connect to the Iris server.

- Launch Iris Monitor, for example by typing `irismonitor` in the terminal on Linux, assuming it is on your path, or using the Start Menu on Windows. Iris Monitor is installed in `$MAXCORE_HOME/bin/IrisMonitor/bin/`.
Iris Monitor automatically connects to the running simulation and prints an IP address to the terminal, by default `http://127.0.0.1:8080`.
- Paste the IP address shown in the terminal into the address bar of a web browser.
If the connection is successful, the browser displays the [Iris Monitor GUI](#), with a default layout. The views of interest are:
 - The Simulation status and control area displays a filtered list of instances in the simulation. By default it only shows instances that support software execution, in this example `cluster0.cpu0`.
 - The **Disassembly** view shows the instruction disassembly for the selected instance.
- In the **Disassembly** view, set breakpoints on the instructions where you want tracing to start and stop by clicking in the gutter beside the **Address** column. Breakpoints are indicated by red dots:

Figure 4-1: Set breakpoints

DISASSEMBLY □ □ □ □ ×				
Memory Space	Address	Size	Mode	<input type="checkbox"/> Track
Secure Monitor	0x80000000	100	Auto	Loa
Address	Op Code	Disassembly		
● 0x80000000	d503201f	NOP		
0x80000004	9400000c	BL 0x80000034		
0x80000008	a9bf7bf5	STP x21,x30,[sp,#-0x10]!		
0x8000000C	a9bf0fe2	STP x2,x3,[sp,#-0x10]!		
0x80000010	a9bf07e0	STP x0,x1,[sp,#-0x10]!		
● 0x80000014	94000013	BL 0x80000060		
0x80000018	a8c107e0	LDP x0,x1,[sp],#0x10		

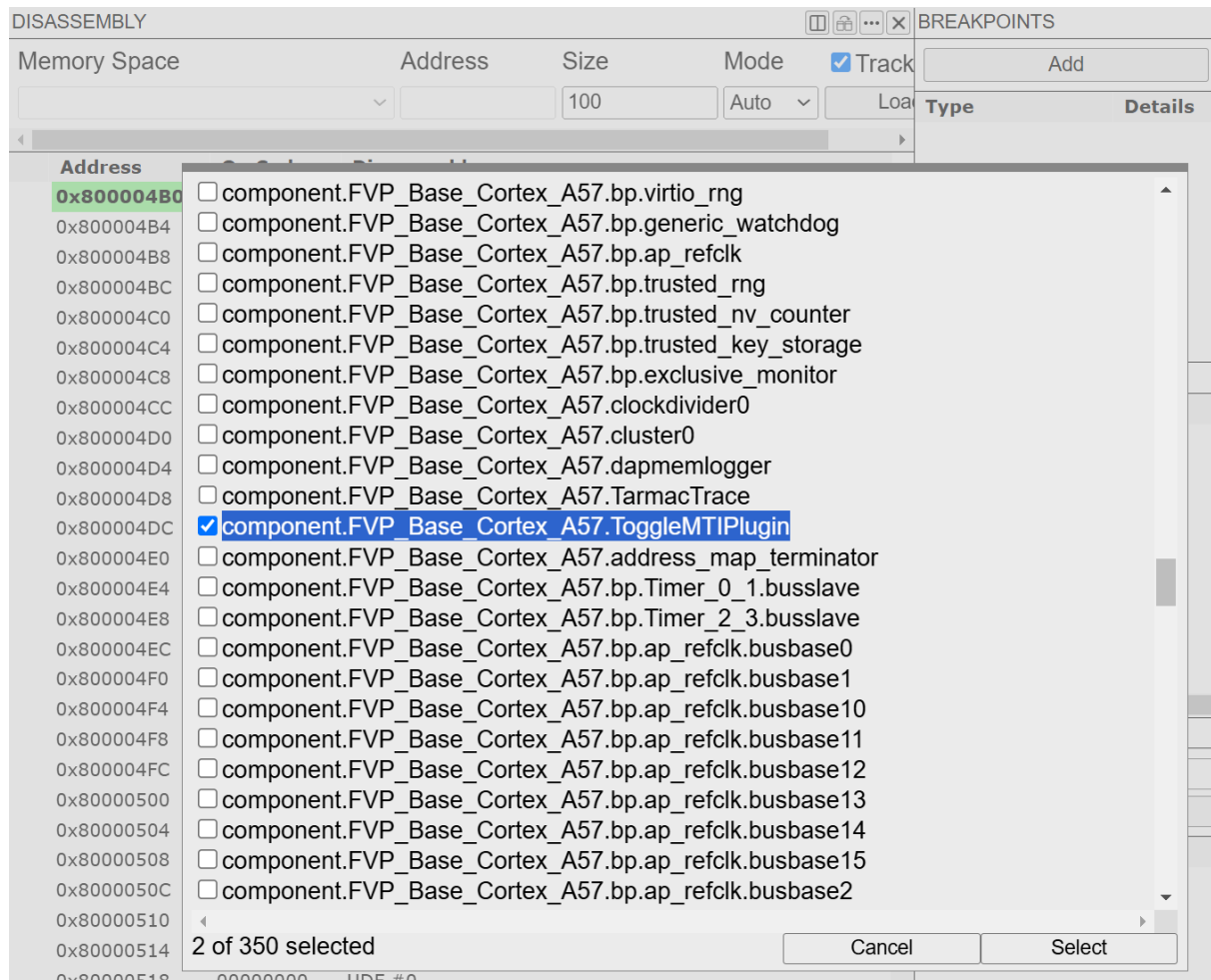
5. Click **Run**.

The simulation starts running, and stops at the first breakpoint. Because we disabled trace from the start of the simulation, the terminal does not yet show any trace output.

The following steps use ToggleMTIPlugin to enable trace output from this point in the simulation.

6. In the Simulation status and control area, click **Change**.
7. Select the checkbox for the `component.FVP_Base_Cortex_A57.ToggleMTIPlugin` instance, then click **Select**:

Figure 4-2: Select targets

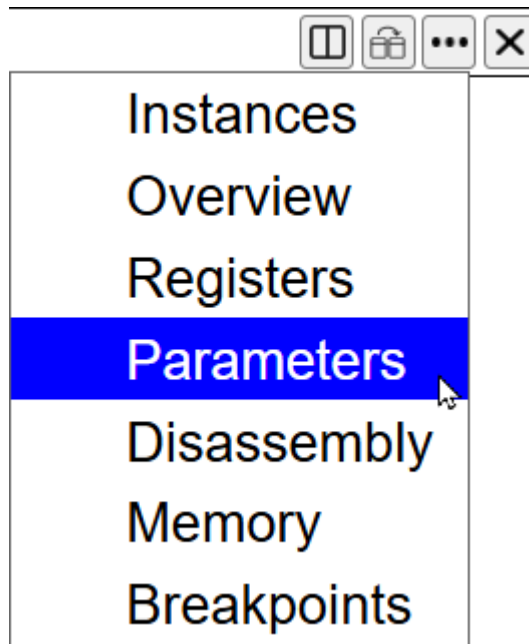


8. ToggleMTIPlugin appears in the instance selection area. Click it to select it.

- 9.



Click the **Switch View** button in any of the views, then select **Parameters**:

Figure 4-3: Switch to Parameters view

10. To enable trace capture:

- a. Double click in the **Value** column for the `disable_mti_runtime` parameter. Unlike the other parameters in this list which can only be set at initialization time, this parameter can be changed when the model is running.
- b. Set the value to zero:

Figure 4-4: disable_mti_runtime parameter

PARAMETERS			
Name	Value	Modifiable	Type
hlt_imm16	0x000000000000F000	init-time	int
use_hlt	0x0	init-time	bool
diagnostics	0x0	init-time	bool
disable_mti_runtime	<input type="text" value="0x0"/>	run-time	bool
disable_mti_from_st...	0x1	init-time	bool

- c. Press Enter to save the new value and close the editor.
11. In the instance selection area, select the `cluster0.cpu0` instance again and click **Run**.
12. Execution restarts then stops at the second breakpoint. The terminal now shows TarmacTrace output:

Figure 4-5: Trace output

```

In process: FVP_Base_Cortex_A57.thread_p_24 @ 0 s
0 clk R cpsr 000003cd
1 clk IT (3) 80000000 d503201f 0 EL3h_s : NOP
2 clk IT (4) 80000004 9400000c 0 EL3h_s : BL      0x80000034
2 clk R X30 0000000080000008
3 clk IT (5) 80000034 580000a0 0 EL3h_s : LDR      x0,0x80000048
3 clk MR8 80000048:000080000048 00000000_90000000
3 clk R X0 0000000090000000
4 clk IT (6) 80000038 9100001f 0 EL3h_s : ADD      sp,x0,#0
4 clk R SP_EL3 0000000090000000
5 clk IT (7) 8000003c 97ffffff3 0 EL3h_s : BL      0x80000008
5 clk R X30 0000000080000040
6 clk IT (8) 80000008 a9bf7bf5 0 EL3h_s : STP      x21,x30,[sp,#-0x10]!
6 clk MW8 8ffffff0:00008ffffff0 00000000_00000000
6 clk MW8 8ffffff8:00008ffffff8 00000000_80000040
6 clk R SP_EL3 000000008FFFFFFF0
7 clk IT (9) 8000000c a9bf0fe2 0 EL3h_s : STP      x2,x3,[sp,#-0x10]!
7 clk MW8 8ffffffe0:00008ffffffe0 00000000_00000000
7 clk MW8 8ffffffe8:00008ffffffe8 00000000_00000000
7 clk R SP_EL3 000000008FFFFFFE0
8 clk IT (10) 80000010 a9bf07e0 0 EL3h_s : STP      x0,x1,[sp,#-0x10]!
8 clk MW8 8ffffffd0:00008ffffffd0 00000000_90000000
8 clk MW8 8ffffffd8:00008ffffffd8 00000000_00000000
8 clk R SP_EL3 000000008FFFFFFD0
8 clk CADI E simulation_stopped

```

13. To turn trace off again, select ToggleMTIPlugin in the instance selection area, then in the **Parameters** view set the `disable_mti_runtime` parameter back to 0x1.
14. Click **Run** again to resume the simulation.

4.5 Toggle trace using HLT instructions

This method of toggling trace does not require a debugger but does require you to modify and rebuild the source code. We will demonstrate it using the model and image that we built in the Dual Core tutorial.

Procedure

1. In your source code editor, edit `startup.s` by inserting a pair of `HLT #0x5` instructions to enclose the code we want to trace.



Note

The value `#0x5` is chosen randomly. You can use any integer value, but you must use the same value for the `TRACE.ToggleMTIPlugin.hlt_imm16` plug-in parameter and the `trace_special_hlt_imm16` CPU parameter. These parameters are described later in this tutorial.

For example:

```
// Which core am I
// -----
    HLT      #0x5                                // Toggle trace on
    MRS x0, MPIDR_EL1
    AND x0, x0, #0xFF // Mask off to leave Aff0 - this assumes
    // a pre v8.4 processor
    HLT      #0x5                                // Toggle trace off
```

2. Save the file and rebuild the image using the following commands:

```
armclang -c -g --target=aarch64-arm-none-eabi -march=armv8.1-a startup.s
armlink --ro-base=0x80000000 startup.o -o image.axf --entry=start64
```

3. Launch the model, loading the image, the TarmacTrace plug-in, and ToggleMTIPlugin. Use these ToggleMTIPlugin parameters:

-C TRACE.ToggleMTIPlugin.use_hlt=1

Selects the `HLT` method of toggling trace.

-C TRACE.ToggleMTIPlugin.disable_mti_from_start=1

Disables trace from the start of the simulation.

-C TRACE.ToggleMTIPlugin.hlt_imm16=0x5

Sets the immediate value to use in `HLT` instructions to toggle trace.

You also need to set the following parameters on each CPU so that they ignore the special `HLT` instructions:

-C armcortexa57ct.cpu0.enable_trace_special_hlt_imm16=1

Enables the use of parameter `trace_special_hlt_imm16`.

-C armcortexa57ct.cpu0.trace_special_hlt_imm16=5

This value must match the immediate value you set in the `hlt_imm16` plug-in parameter.



Note

The prefixes for the CPU parameters vary depending on the platform. To see the full list of parameters and prefixes, run your platform with the `--list-params` option.

The full command is:

```
./Linux64-Release-GCC-9.3/isim_system -a image.axf \
--plugin $PVLIB_HOME/plugins/Linux64_GCC-9.3/TarmacTrace.so \
--plugin $PVLIB_HOME/plugins/Linux64_GCC-9.3/ToggleMTIPlugin.so \
-C TRACE.ToggleMTIPlugin.use_hlt=1 \
-C TRACE.ToggleMTIPlugin.disable_mti_from_start=1 \
-C TRACE.ToggleMTIPlugin.hlt_imm16=5 \
-C armcortexa57ct.cpu0.enable_trace_special_hlt_imm16=1 \
-C armcortexa57ct.cpu0.trace_special_hlt_imm16=5 \
-C armcortexa57ct.cpu1.enable_trace_special_hlt_imm16=1 \
-C armcortexa57ct.cpu1.trace_special_hlt_imm16=5
```

Results

In the terminal, you should see trace output limited to the selected range of instructions:

Figure 4-6: Trace output

```
500000000000 ps cpu1 IT (5) 80000010 d53800a0 O EL3h_s : MRS      x0,MPIDR_EL1
500000000000 ps cpu1 R X0 0000000080000001
600000000000 ps cpu1 IT (6) 80000014 92401c00 O EL3h_s : AND      x0,x0,#0xff
600000000000 ps cpu1 R X0 0000000000000001
700000000000 ps cpu1 IT (7) 80000018 d44000a0 O EL3h_s : HLT      #5
Hello from core 1!
700000000000 ps cpu0 R cpsr 000003cd
500000000000 ps cpu0 IT (5) 80000010 d53800a0 O EL3h_s : MRS      x0,MPIDR_EL1
500000000000 ps cpu0 R X0 0000000080000000
600000000000 ps cpu0 IT (6) 80000014 92401c00 O EL3h_s : AND      x0,x0,#0xff
600000000000 ps cpu0 R X0 0000000000000000
700000000000 ps cpu0 IT (7) 80000018 d44000a0 O EL3h_s : HLT      #5
Hello from core 0!
2 clk CADI E simulation_stopped

Info: /OSCI/SystemC: Simulation stopped by user.
```

5. Generating MTI trace from a LISA component

In this tutorial, we will describe what code is required within a LISA component to expose and generate MTI trace.

This tutorial is based on the example component and platform provided with the Fast Models installation in `$PVLIB_HOME/examples/LISAPlus/GeneratingTraceFromLISA/`.

5.1 What is MTI?

MTI stands for Model Trace Interface. It is the interface which the Fast Models use to expose trace information to the user. Lots of models provided by Arm expose MTI trace. However, these are mostly extern components and the mechanism of exposing these traces is not visible.

Generating and exposing trace information by a component is just half of the MTI story. This trace data must be consumed, typically by an MTI plug-in.

5.2 Setting up the trace source in the LISA component

Before you can generate trace data, you must create and configure the trace source.

Procedure

1. We must define a pointer of type `sg::EventSource` in the `resources` section of the LISA file and create the object of this type in `init()`. `sg::EventSource` is a templated class which uses its template parameters to define the type of data for each field of this trace source. In this example the trace source has two integer fields and a string field:

```
resources
{
    // declare EventSource pointer with the required number of arguments and
    types
    sg::EventSource<uint64_t, uint32_t, const char*>* write_trace;
    char tracestring[50];
}

behaviour init()
{
    composition.init();

    // create EventSource object
    write_trace = new sg::EventSource<uint64_t, uint32_t, const char*>();
```

2. After having created the `sg::EventSource` object, we will configure it by firstly providing a name and description:

```
write_trace->setName( "LISA_TRACE_WRITE" );
write_trace->setDescription( "Example trace source which displays the properties
of a transaction" );
```

3. The next step is to add fields to the trace object. Each field includes:

- Name
- Description
- Type
- Size

This is the prototype for the `AddField()` function:

```
/* provide field information
AddField(const char *name,
         const char *description,
         MTI::EventFieldType::Type type,
         MTI::EventFieldType::Size size,
         MTI::EventFieldType::Size max_size=0);
*/
```

You must add each field to the trace object in the order of its template parameters, in our example, int, int, string:

```
write_trace->AddField( "Address", "Address of the transaction",
                      MTI::EventFieldType::MTI_UNSIGNED_INT,
                      sizeof(uint64_t) );
write_trace->AddField( "Value", "Value written",
                      MTI::EventFieldType::MTI_UNSIGNED_INT,
                      sizeof(uint32_t) );
write_trace->AddField( "String", "String containing trace data",
                      MTI::EventFieldType::MTI_STRING,
                      0,
                      50 ); // 50 character long string
```

4. The last step in setting up the trace source is to register it with the simulation engine. This is done by calling the `addTraceSource()` function, passing in our configured `sg::EventSource` object:

```
// register trace source with the simulation engine
addTraceSource( write_trace );
```

5.3 Generating trace data in the LISA component

After the trace source has been configured and registered in the `init()` behaviour, you can generate trace information in any behaviour inside your LISA component.

Generating trace data is done by calling the `fire()` function on the `sg::EventSource` object. The arguments to `fire()` are the values of each trace field in the order they were defined. In our example, int, int, string:

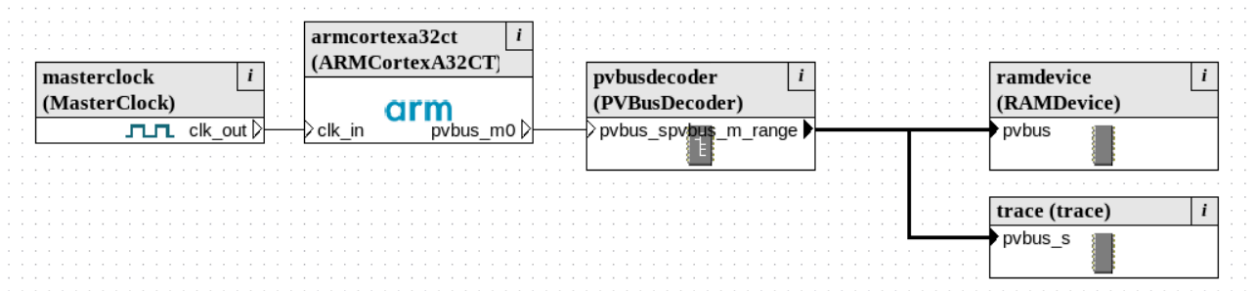
```
//trigger trace event
write_trace->fire( addr, value, tracestring);
```

5.4 What is in the example and what does it do?

An example component and platform showing the concepts described in this tutorial is provided with the Fast Models installation in `$PVLIB_HOME/examples/LISAPlus/GeneratingTraceFromLISA/`.

When the example project `$PVLIB_HOME/examples/LISAPlus/GeneratingTraceFromLISA/Build_Cortex-A32/GeneratingTraceFromLISA.sgproj` is loaded into System Canvas, it looks as follows:

Figure 5-1: Platform displayed in System Canvas



This platform consists of the following components:

- MasterClock
- Core
- PVBusDecoder
- RAMDevice
- trace component

The trace component defines and configures a trace source, `write_trace`, as described earlier in this tutorial. It also has a `PVBusSlave` component to provide a PVBus slave bus interface. The implementation of the `PVDevice` port's `write()` behavior does the following:

- Extracts the address and data to be written from the `pv::WriteTransaction` object.
- Creates a string containing the extracted data.
- Uses the `fire()` function to trigger the trace event. It uses all three variables.

```
behavior write(pv::WriteTransaction tx):pv::Tx_Result
{
    uint32_t addr = tx.getAddress();
    uint32_t value = tx.getData32();

    // generate string
    sprintf(tracestring, "Address= 0x%016x  data=0x%08x", addr, value );

    //trigger trace event
    write_trace->fire( addr, value, tracestring);

    return tx.writeComplete();
}
```

5.5 Run the GeneratingTraceFromLISA example

Before you can run the example, first you need to build the platform.

To do this, either use System Canvas or run SimGen directly, using the following command:

```
cd $PVLIB_HOME/examples/LISAPlus/GeneratingTraceFromLISA/Build_Cortex-A32
simgen -b -p GeneratingTraceFromLISA.sgproj --configuration Linux64-Release-GCC-9.3
```

The elf image to load onto the core, `test.axf`, is already provided, but there is a build script, `build.sh` provided if you want to rebuild it using `armasm` and `armlink`.

The test application performs three tasks:

1. Semihosted print
2. Store to the `trace` LISA component
3. Semihosted exit

The final step is to run the example. To capture the generated trace, we will use the `GenericTrace` MTI plugin:

```
./Linux64-Release-GCC-9.3/isim_system \
-a ../app/test.axf \
--plugin $PVLIB_HOME/plugins/Linux64_GCC-9.3/GenericTrace.so \
-C TRACE.GenericTrace.trace-sources=LISA_TRACE_WRITE
```

The example should produce the following output to stdout of the terminal:

```
GenericTrace: model version is 1.0
Storing 0xcafebabe to trace peripheral (0x100000000)
LISA_TRACE_WRITE: Address=0x0000000010000000 Value=0xcafebabe String="Address=
0x0000000010000000 data=0xcafebabe"
Info: /OSCI/SystemC: Simulation stopped by user.
```

As can be seen, the trace component triggers the `LISA_TRACE_WRITE` event when the store is made.

6. Dynamically driving signals with SignalDriver

In this tutorial, we will discuss the `signalDriver` component, the reasons for using it, how to use it, and introduce an example platform which illustrates its use within a platform.

6.1 What is the SignalDriver component and what is it designed to do?

`signalDriver` is a component written in LISA+ that primarily has a master signal port, `signal_out`. The state of this signal port is controlled by a parameter, register, or bus access.

When added to the platform, `signalDriver` can aid debugging by allowing users to interact with signals without having to program up the real peripherals to drive the signals.

In addition to driving the master signal port, the `signalDriver` component also has a trace source, `SIGNAL`. This trace source fires every time the signal is driven and contains fields for the signal value and whether the change was made by a parameter, register, or bus access.

6.2 How is SignalDriver implemented?

The main purpose of the component is to drive the `signal_out` master port.

```
master port<Signal> signal_out;
```

The driving of the port is done in a common behavior, `drive_signal()`. This is used irrespective of which input triggered the signal change:

```
behavior drive_signal(sg::Signal::State new_value,
                    Source source): void
{
    // update local representation of signal value
    signalState = new_value;

    // only drive the signal and generate trace when initialisation has completed
    if (init_complete == true)
    {
        // drive signal_out based on signalState
        signal_out.setValue(signalState);
    }
}
```


`drive_signal()` only drives the signal after initialisation and elaboration have occurred. Therefore, calling `signal_out.setValue()` is gated by the `init_complete` flag. This flag is set in the `reset()` behavior and the initial driving of the signal is done from there:



Note

```
behavior reset(int level)
{
    composition.reset(level);

    // flag initialisation/elaboration has completed
    init_complete=true;

    // set the initial state of the signal,
    // its value will be taken from the parameter
    drive_signal(signalState, PARAM);
}
```

`drive_signal()` takes as its arguments the new signal value and an enum specifying the source of the change, either parameter, register, or bus:

```
// enum for what caused the signal to change
enum Source
{
    PARAM,
    REG,
    BUS
};
```

In addition to driving the signal, `drive_signal()` also triggers the trace source. The trace source takes both the new signal value and an enum to identify the source of the change:

```
// generate trace information
trace->fire(signalState, source);
```

6.3 Using the parameter to change the signal

`SignalDriver` exposes a parameter to change the signals.

The parameter has the following properties:

type (bool)

Set to `True` or `1` to drive the signal set. Set to `False` or `0` to drive the signal clear.

default (false)

The default value is `false`, driving the signal clear.

runtime (true)

The parameter can be changed during runtime to modify the signal.

```
PARAMETER { description("Drive signal_out port with this parameter value"),
             default(false),
```

```

        type(bool),
        runtime(true),
        write_function(param_write),
        read_function(param_read)
    } param_input;

```

When the parameter is written, the `param_write()` behavior is called. This converts the `int64_t*` parameter value to `sg::Signal::State` and forwards it to `drive_signal()`, setting the source to enum `PARAM`:

```

behavior param_write(uint32_t id,
                    const int64_t *data) : AccessFuncResult
{
    drive_signal(*data? sg::Signal::Set : sg::Signal::Clear, PARAM);
    return ACCESS_FUNC_OK;
}

```

When the parameter is read, the `param_read()` behavior is called. This converts the `sg::Signal::State` signal value to `int64_t*` and returns this as a successful parameter read to the debugger:

```

behavior param_read(uint32_t id,
                   int64_t *data) : AccessFuncResult
{
    //parameter reading data from the signalState bool
    *data = (signalState == sg::Signal::Set) ? 1 : 0;
    return ACCESS_FUNC_OK;
}

```

6.4 Using the register to change the signal

`SignalDriver` exposes a register to change the signals.

This register is a single bit in size. When set to 1 the signal is driven `set`, and when set to 0 the signal is driven `clear`:

```

REGISTER { bitwidth(1),
           type(bool),
           write_function(reg_write),
           read_function(reg_read)
        } reg_input;

```



If the debugger shows that the registers contain more than 1 bit, only `bit[0]` is used, all other bits are ignored. For example `0b10` would drive the signal `clear` not `set` as `bit[0]` is 0.

When the register is written, the `reg_write()` behavior is called. This first extracts the value of `bit[0]` from the data passed by the debugger, converts this to `sg::Signal::State`, and forwards it to `drive_signal()`, setting the source enum to `REG`:

```
behavior reg_write(uint32_t reg_id,
                  const uint64_t *data,
                  bool side_effects) : AccessFuncResult
{
    // *data is a uint64_t even though reg_input is defined as a single-bit register
    // only bit 0 of *data contains valid data, the rest is uninitialised
    // therefore extract bit0 from *data
    reg_input = ((*data & 1) == 1)? (1): (0);

    drive_signal(reg_input? sg::Signal::Set : sg::Signal::Clear, REG);

    return ACCESS_FUNC_OK;
}
```

When the register is read, the `reg_read()` behavior is called. This converts the `sg::Signal::State` signal value to `uint64_t*` and returns this as a successful register read to the debugger:

```
behavior reg_read(uint32_t reg_id,
                 uint64_t *data,
                 bool side_effects) : AccessFuncResult
{
    //register reading the data from the signalState bool
    *data = (signalState == sg::Signal::Set) ? 1 : 0;
    return ACCESS_FUNC_OK;
}
```

6.5 Using the bus to change the signal

`SignalDriver` uses a `PVBusSlave` component to handle bus transactions on its `PVBus` slave port and exposes these transactions on the `PVBusSlave`'s device port.

The device port accepts any size and any address of transaction and handles the data as follows:

Write

Any non-zero data value drives the signal `set`, otherwise the signal is driven `clear`.

Read

1 is returned if the signal is `set`, and 0 returned if the signal is `clear`.



Note

A write of `0b00100000` drives the signal `set` even though bit 0 is zero.

The `write()` behavior extracts the data from the transaction object, `tx`, using the appropriate `getData` function based on the access size of the transaction. This is then cast to a `bool` and

converted to `sg::Signal::State` and finally forwarded to `drive_signal()`, setting the source enum to `BUS`:

```
behavior write(pv::WriteTransaction tx):pv::Tx_Result
{
    //data of value 0 means signal is clear, data of value > 0 means signal is set
    switch (tx.getAccessWidth())
    {
        case pv::ACCESS_8_BITS :
            drive_signal(~(bool)tx.getData8() ? sg::Signal::Set : sg::Signal::Clear, BUS);
            break;
        case pv::ACCESS_16_BITS :
            drive_signal(~(bool)tx.getData16() ? sg::Signal::Set : sg::Signal::Clear,
BUS);
            break;
        case pv::ACCESS_32_BITS :
            drive_signal(~(bool)tx.getData32() ? sg::Signal::Set : sg::Signal::Clear,
BUS);
            break;
        case pv::ACCESS_64_BITS :
            drive_signal(~(bool)tx.getData64() ? sg::Signal::Set : sg::Signal::Clear,
BUS);
            break;
        default:
            std::cout << "Access width is invalid" << std::endl;
    }
    return tx.writeComplete();
}
```

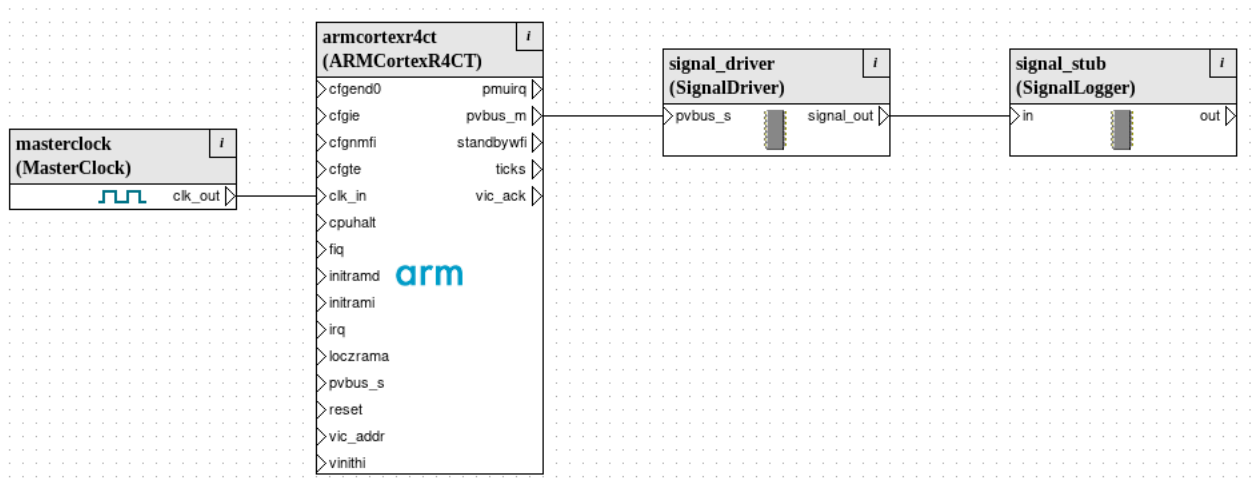
The `read()` behavior converts the `sg::Signal::State` to an `int` and sets the return value of the transaction to this value:

```
behavior read(pv::ReadTransaction tx):pv::Tx_Result
{
    switch (tx.getAccessWidth())
    {
        case pv::ACCESS_8_BITS :
            return tx.setReturnData8( (signalState == sg::Signal::Set) ? 1 : 0 );
            break;
        case pv::ACCESS_16_BITS :
            return tx.setReturnData16( (signalState == sg::Signal::Set) ? 1 : 0 );
            break;
        case pv::ACCESS_32_BITS :
            return tx.setReturnData32( (signalState == sg::Signal::Set) ? 1 : 0 );
            break;
        case pv::ACCESS_64_BITS :
            return tx.setReturnData64( (signalState == sg::Signal::Set) ? 1 : 0 );
            break;
        default:
            return tx.readComplete();
    }
}
```

6.6 SignalDriver example

To illustrate `signalDriver` working in a platform, an example is provided in `$PVLIB_HOME/examples/LISAPlus/SignalDriver/`.

When this example platform is loaded into System Canvas, it looks as follows:

Figure 6-1: Platform seen in System Canvas

In this platform we have:

- MasterClock
- Core
- SignalDriver
- SignalLogger

In the example, the `signalLogger` component is used to consume the signal driven by `signalDriver` and generate MTI trace data when it receives the updated signal.

Using the example

1. To run the example, first you need to build the platform. To do this either use System Canvas or run `simgen` directly:

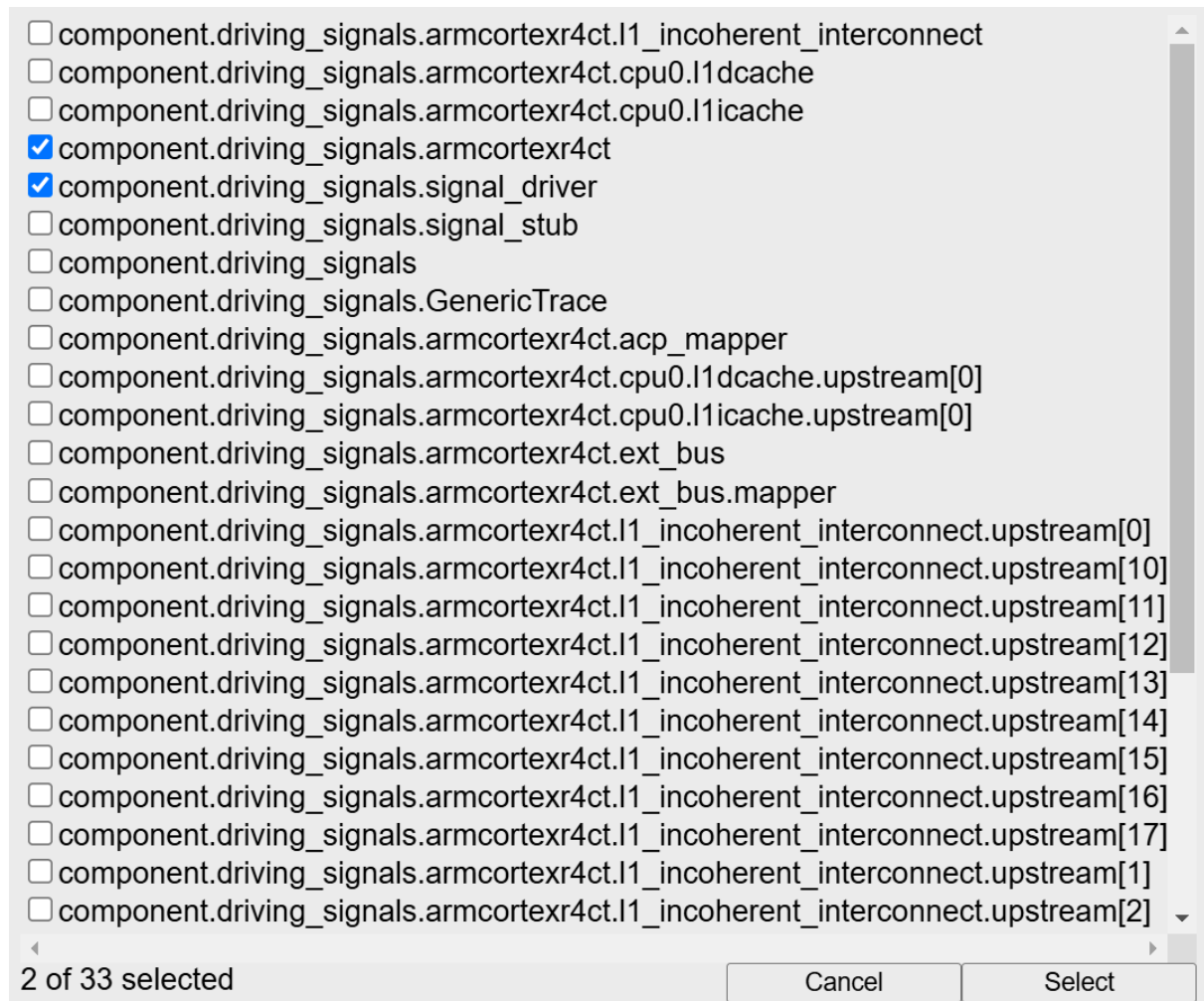
```
cd $PVLIB_HOME/examples/LISAPlus/SignalDriver
simgen -b -p SignalDriver.sproj --configuration Linux64-Release-GCC-9.3
```

No test image is required for this example, all interaction will be done using an attached debugger, in this case [Iris Monitor](#).

2. The next step is to run the example. To capture the generated trace, we will use the `GenericTrace` MTI plugin. We also use the `-I` option to start the Iris server, which enables Iris Monitor to connect to the model:

```
./Linux64-Release-GCC-9.3/isim_system \
--plugin $PVLIB_HOME/plugins/Linux64_GCC-9.3/GenericTrace.so \
-C TRACE.GenericTrace.trace-sources=SIGAL \
-I
```

3. With the model running, start Iris Monitor, which connects to the running model. In the Iris Monitor Simulation status and control area, click **Change**. Select the checkboxes for both the core and the `signalDriver` component:

Figure 6-2: Iris Monitor connect dialog

- In the instance selection area of Iris Monitor, select the `signalDriver` component. The current state of the signal can be seen in the **Registers** view. The default setting is for the signal to be driven `sg::Signal::clear` and therefore the register value is `0x0`:

Figure 6-3: Iris Monitor register view

REGISTERS			
Name	Value	Size	Access
Default			
reg_input	0x0	1	rw

- Now we will change the register value and observe the signal changes. Set the register value to `0x1`:

Figure 6-4: Iris Monitor register view

REGISTERS			
Name	Value	Size	Access
Default			
reg_input	0x1	1	rw

On the terminal we can see that:

- signalDriver is driving the signal set and this change occurred due to changing the register.
- The signalLogger stub component confirmed the new value of the signal:

```
signal_stub.SIGNAL: Value=Set
signal_driver.SIGNAL: Value=Set Source=Register
```

- Similar behavior can be seen when using the parameter.



Note

If the **Parameters** view is not shown in Iris Monitor, click the **Switch View**



button in another view, then select **Parameters**.

Viewing the signalDriver's parameter, we can see that it is set, matching the current state of the signal:

Figure 6-5: Iris Monitor parameter view

PARAMETERS			
Name	Value	Modifiable	Type
param_input	0x1	run-time	bool

Set the value to 0x0 to see this reflected in the traces:

```
signal_stub.SIGNAL: Value=Clear
signal_driver.SIGNAL: Value=Clear Source=Parameter
```

- Lastly, we can test the bus behavior by selecting the armcortexr4ct core instance in the instance selection area. In the **Memory** view, write 1 to any address:

Figure 6-6: Iris Monitor memory view

MEMORY ⏏ ⏏ ⋮ ✕				
Memory ▾	0x0	128	1 ▾	
Load				
Address				
00000000	01	01	01	
00000010	01	01	01	
00000020	01	01	01	
00000030	01	01	01	
00000040	01	01	01	
00000050	01	01	01	

Writing to an address updates the signal and also every address read back reflects the signal state:

```
signal_stub.SIGNAL: Value=Set
signal_driver.SIGNAL: Value=Set Source=Bus
```



SignalDriver does not use the address of a transaction received on its slave port. Therefore any address written can be used and all addresses read return the same value.

7. Connecting multiple clusters with PVCoherentInterconnect

In this tutorial, we will discuss the `PVCoherentInterconnect` component, explain the reasons for using it, look at its internals, and develop an example system to illustrate its use within a system.

7.1 What is the PVCoherentInterconnect component and what is it designed to do?

`PVCoherentInterconnect` is a component written in LISA+ that is designed to be a generic coherent interconnect.

It serves as an alternative to the models of Arm's coherent interconnects IP, such as the CCLs and the CMNs. Unlike these IP models, `PVCoherentInterconnect` allows up to 128 managers to be connected. It also does not require any software configuration to enable coherency.

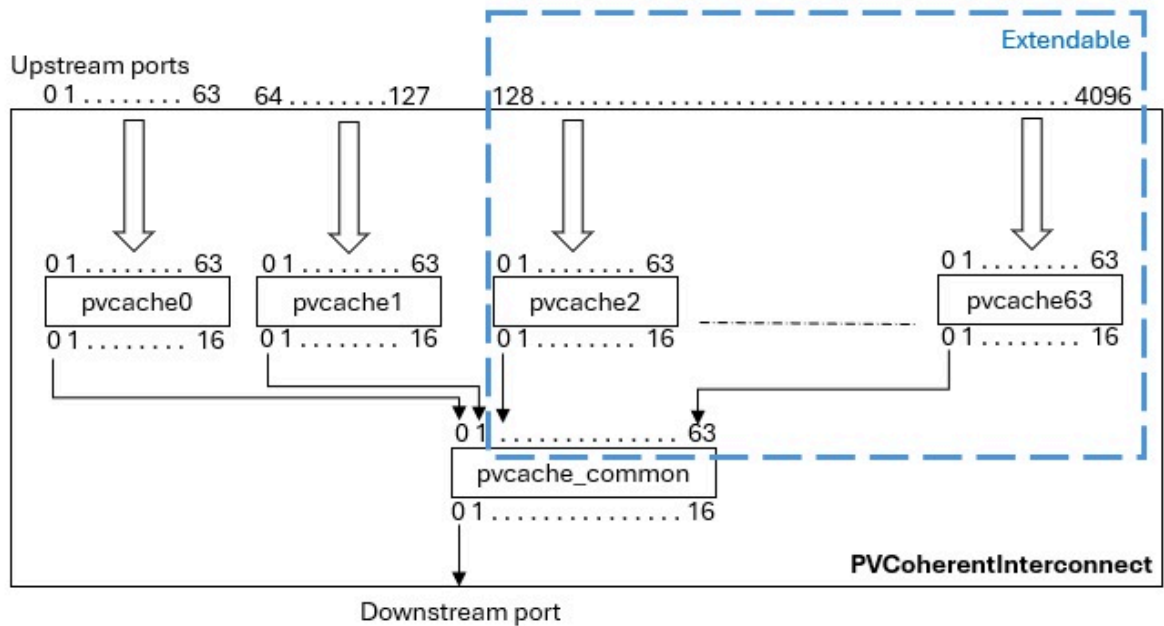
In addition, it can be modified to extend the number of clusters that can be connected to 4096. Details about its implementation and the way to extend it are presented in the next section.

7.2 How is PVCoherentInterconnect implemented?

`PVCache64` is a component that has both caching functionality and interconnect functionality. It can connect up to 64 clusters on its upstream ports and it has 16 downstream ports. In addition, multiple `PVCache64` components can be chained together.

`PVCoherentInterconnect` is a wrapper that for simplicity suppresses the caching functionality and related parameters in `PVCache64` components. It also connects two `PVCache64` components so that the maximum number of clusters that they can connect is increased to 128. Therefore, it has 128 upstream ports that can be connected to different clusters. It also has 1 downstream port that handles all the bus traffic.

The following diagram shows the structure of the `PVCoherentInterconnect` component.

Figure 7-1: PVCoherentInterconnect structure

We see that the two internal `pvcache64` components, `pvcache0` and `pvcache1`, are connected together using a single common downstream `pvcache64` component, `pvcache_common`. The downstream port 0, which handles bus traffic, of both these components is connected to the upstream ports on `pvcache_common`.

The external upstream ports of the `PVCoherentInterconnect` map to the internal components `pvcache0` and `pvcache1` in sequence. This snippet from the LISA file for the `PVCoherentInterconnect` captures this:

```
self.upstream[0] => pvcache0.upstream[0];
self.upstream[1] => pvcache0.upstream[1];
...
self.upstream[63] => pvcache0.upstream[63];

self.upstream[64] => pvcache1.upstream[0];
self.upstream[65] => pvcache1.upstream[1];
...
self.upstream[127] => pvcache1.upstream[63];
```

The single external downstream port of `pvcache0` maps to the downstream port 0 of the internal `pvcache_common` component:

```
pvcache_common.downstream[0] => self.downstream;
```

The diagram also shows the extendability of the `PVCoherentInterconnect` component, enclosed in the blue dotted rectangle. Additional internal `PVCache64` components can be instantiated and their upstream and downstream ports connected as depicted. Note that the number of external upstream ports of `PVCoherentInterconnect` needs to be updated accordingly.

The LISA file for `PVCoherentInterconnect` is available in the Fast Models package at `$PVLIB_HOME/LISA/PVCoherentInterconnect.lisa`.

7.3 Example system using the interconnect

An example system is provided in `$PVLIB_HOME/examples/LISAPlus/PVCoherentInterconnect/` to show `PVCoherentInterconnect` in action.

Procedure

1. Open System Canvas and load the example project.
2. Click on the **Source** tab.

It shows the following LISA code:

```
// This file was generated by System Generator Canvas
// -----
component PVCoherentInterconnectExample
{
  composition
  {
    ramdevice : RAMDevice("fill1"=0x0,"fill2"=0x0);
    pvcoherentinterconnect : PVCoherentInterconnect();
    armcortexa72ct : ARM Cortex A72CT("CLUSTER ID"=0,
                                     "NUM_CORES"=1,
                                     "dcache-state_modelled"=true,
                                     "icache-state_modelled"=true);
    armcortexa53ct : ARM Cortex A53CT("CLUSTER ID"=1,
                                     "NUM_CORES"=1,
                                     "dcache-state_modelled"=true,
                                     "icache-state_modelled"=true);
    masterclock : MasterClock();
  }
  connection
  {
    masterclock.clk_out => armcortexa72ct.clk_in;
    masterclock.clk_out => armcortexa53ct.clk_in;
    pvcoherentinterconnect.downstream => ramdevice.pvbus;
    armcortexa72ct.pvbus_m0 => pvcoherentinterconnect.upstream[0];
    armcortexa53ct.pvbus_m0 => pvcoherentinterconnect.upstream[127];
    armcortexa72ct.event => armcortexa53ct.event;
  }
}
```

This code adds the following components in our example system:

- MasterClock
- ARM Cortex A72CT
- ARM Cortex A53CT
- RAMDevice

- `PVCoherentInterconnect`

For the purposes of our example, we keep it simple by having two single-core clusters, `ARMCortexA72CT` and `ARMCortexA53CT`, connected using `PVCoherentInterconnect`. Henceforth, we will refer to these clusters as `A72CT` and `A53CT` for brevity.

The code initializes the following parameters for these clusters:

CLUSTER_ID

This value populates the MPIDR register at runtime. This value is used later in our example code to differentiate between the two clusters. Having a different cluster id is also essential for the exclusive monitors to work correctly in the modelling system.

NUM_CORES

This value specifies the number of cores in the cluster. Since we want single-core clusters, we set this value to 1.

dcache-state_modelled and icache-state_modelled

These parameters are false by default, that is, the caches are not modelled by default to improve the simulation performance. However, for the purposes of our example, we need to enable the cache state modelling to be able to show coherency being handled by the interconnect.

We then also add in a `RAMDevice` component, which models RAM memory.

The example has the following connections:

- `MasterClock` provides the clock source and its `clk_out` is connected to the `clk_in` of both the clusters.
 - PVBus manager port `pdbus_m0` of the first cluster, `A72CT`, is connected to the first upstream port (index 0) of the interconnect.
 - PVBus manager port `pdbus_m0` of the second cluster, `A53CT`, is connected to the 128th upstream port (index 127) of the interconnect. Since the `PVCoherentInterconnect` supports 128 clusters, this is the maximum index value at which a cluster can be connected.
 - On the downstream side of the interconnect, the single downstream port is connected to the `RAMDevice`.
 - The `event` ports of the two clusters are connected together. This is because we use `sev` (send event) and `wfe` (wait for event) instructions in our example code to control the sequence of operation of the two clusters, so we can show the coherency being handled.
3. To run the example, first you need to build the system. To do this, either use System Canvas or run the `simgen` command directly on the command line:

```
cd $PVLIB_HOME/examples/LISAPlus/PVCoherentInterconnect
simgen -b -p PVCoherentInterconnectExample.sgproj --configuration Linux64-Release-GCC-9.3
```

7.4 Code to run on the example model

In this section, we will write some code to run on the model we just built.

This example code is also available at `$PVLIB_HOME/examples/LISAPlus/PVCoherentInterconnect/startup.s` for convenience.

Procedure

1. Open your preferred text editor.
2. Start with:

```
.section BOOT, "ax"  
.align 3
```

This names the section `BOOT` and uses the `"ax"` flags to indicate that it is all memory-resident and executable, see [ELF image attributes SHF_ALLOC and SHF_EXECINSTR](#).

In addition the alignment of this section is 8 bytes (2^3 bytes).

3. Add some defines:

```
.equ SH_TRAP_INST_A64,      0xF000  
.equ SYS_WRITE0,           0x4  
.equ SYS_EXIT,             0x18
```

This defines 3 constants. In order:

- The HLT instruction special value to invoke semihosting functionality.
- The semihosting write call value to print a NULL-terminated string.
- The semihosting exit call value.

Semihosting is a feature that permits a core to call to either the simulation host or the debug host to access some features provided by the host such as file or console access.

It is accessed by:

- Setting register X0 to the value for the particular service call.
- Setting register X1 to the argument.
- Executing the HLT instruction, which is normally a halting breakpoint, with a special value so that the host rather than the core traps the breakpoint.

By using semihosting, we do not need to add UARTs or a CLCD or program a driver to have output. This can aid the rapid bring-up of software.

4. Add some more defines:

```
.equ RO_BASE, 0x80000000  
.equ RW_BASE, 0x80200000
```

These constants are defined to match the image structure.

The linker assumes a default memory map for the image if the scatter file is not explicitly specified. See [Scatter-loading images with a simple memory map](#).

We will use the linker options `--ro-base` and `--rw-base` to specify the base addresses of the two different execution regions. The `RO_BASE` and `RW_BASE` defines here are set to match them.

5. And yet more defines:

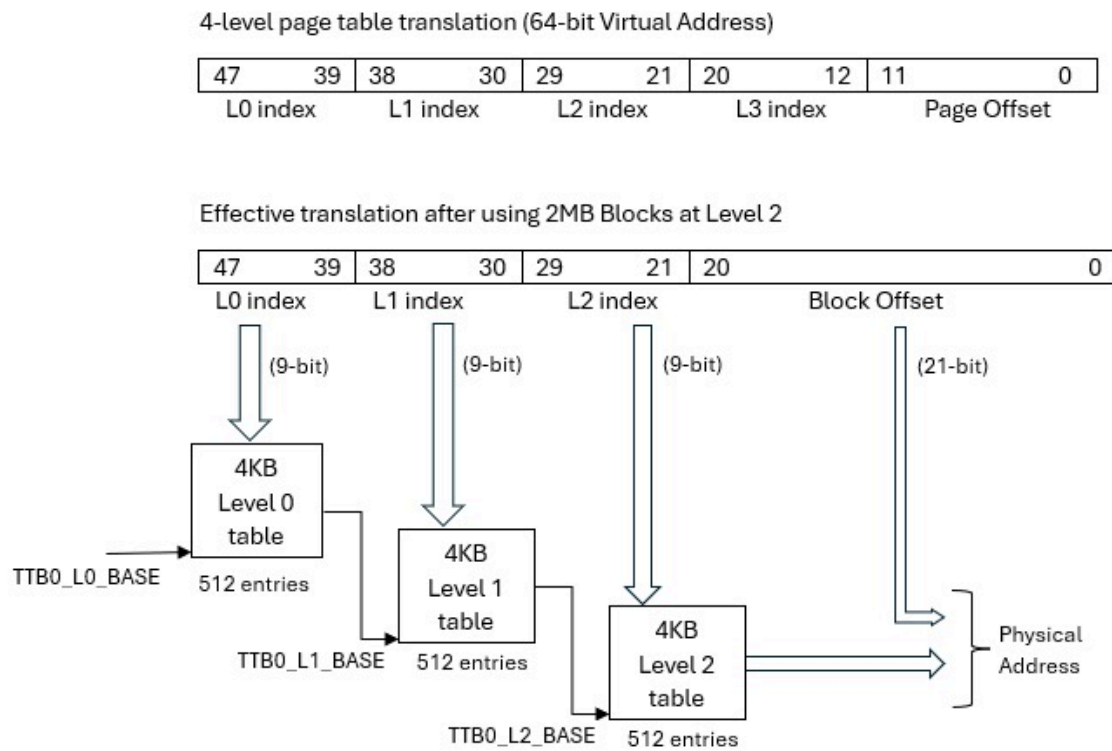
```
.equ TTB0_L0_BASE,      0x80001000
.equ TTB0_L1_BASE,      0x80002000
.equ TTB0_L2_BASE,      0x80003000
```

These constants point to the base addresses of Level 0 (L0), Level 1 (L1) and Level 2 (L2) page tables respectively. We will just need single page tables at L1 and L2.

For our example purposes, we are defining EL3 translation tables using 4 KB granularity and mapping 64-bit Virtual Address (VA) space to 32-bit Physical Address (PA) space.

This implies a 4-level page table scheme. However, to simplify, we will be using the entries in the L2 page table to point to a block of 2 MB, instead of further pointing to L3 page tables.

Also, we are flat-mapping the addresses so that a particular VA maps to the same address in the PA space.

Figure 7-2: Address translation scheme

Since our example code is very small, a single 2 MB block should suffice. However, to experiment with a shared read/write location to show coherency between clusters, we will be using another 2 MB block to keep that separate. So, in essence, we need one entry each in the L0 and L1 page tables, and two entries in the L2 page table pointing to the two separate 2 MB blocks.

If you look back at the `RO_BASE` and `RW_BASE` definitions introduced earlier, you would notice that they have been defined to be 2 MB apart to fit this address translation scheme.

6. Add section definitions:

```
.global start64
.type start64, @function
start64:
```

The last line puts the label `start64` in the code. The two previous lines define it as a global so it is visible externally and declare it as a function. This means `start64` can be used as the entry point for this code block.

7. Add code to determine which cluster we are running on.

Since they are single-core clusters, we do not need to further distinguish between the cores within the cluster for our example purposes.

When the system model starts, all clusters start running the same code immediately. Add code to determine which cluster we are running on:

```
// Which cluster am I in
// -----
MRS      x0, MPIDR_EL1
UBFX     x19, x0, #8, #8           // Extract Cluster ID Affl field
                                           // (bits 8 to 15) and store it in x19.
                                           // This assumes a pre v8.4 processor
CBZ      x19, first                // If cluster 0, run the code for core
                                           // in the first cluster
B        second                   // Else, run the code for core
                                           // in the second cluster
```

8. Create the print helper function, which uses semihosting to print the value:

```
// -----
// Print value using semihosting call
// -----
print_value:
    CMP x19, #0
    BNE p2
p1:
    LDR x3, =cluster0_print_msg
    LDR x4, =cluster0_print_value
    B common
p2:
    LDR x3, =cluster1_print_msg
    LDR x4, =cluster1_print_value
common:
    ADD w0, w0, #'0'           // Value to be printed is passed in w0.
                                // Add '0' to convert the digit to ascii
    STRB w0, [x4]              // Store it in the correct memory location

    MOV x1, x3                  // Set up semihosting call to print message
    MOV w0, #SYS_WRITE0
    HLT #SH_TRAP_INST_A64

    MOV x1, x4                  // Set up semihosting call to print value
    MOV w0, #SYS_WRITE0
    HLT #SH_TRAP_INST_A64

    RET

// -----
// String literals
// -----
cluster0_print_msg:
    .string "Printing from cluster0, value is "
cluster0_print_value:
    .string "y\n"              // String to print the value for cluster 0.
                                // Ascii value pertaining to the digit is updated
                                // here, so it can be printed as a string using
                                // the semihosting write call
cluster1_print_msg:
    .string "Printing from cluster1, value is "
cluster1_print_value:
    .string "z\n"              // String to print the value for cluster 1
```

9. Create a helper function to configure the MMU and caches in each cluster.

Since we are defining this function after the string literals, we need to align it so it starts on an 8-byte boundary:

```
.align 3
// -----
// Setup page tables and configure the MMU and caches
// -----
configure_mmu_and_caches:
// -----
// Invalidate caches and TLBs
// -----

//Invalidate Caches          // Cortex A72 and A53 have caches
// invalidated at reset. This step
// might be needed for other cores.

TLBI ALLE3                  // Invalidate EL3 related TLBs.

// -----
// Setup L1 and L2 page tables
// -----
LDR x1, =TTB0_L0_BASE      // Set TTBR0
MSR TTBR0_EL3, x1

MOV x1, #0xff44            // Set memory region attributes in MAIR register
MOVK x1, #4, LSL #16       // Attribute0 = 0x44 = Normal, Inner/Outer
// Non-Cacheable, Attribute1 = 0xff = Normal,
// Inner/Outer WriteBack Read/Write Allocate
MSR MAIR_EL3, x1          // Attribute2 = 0x04 = Device-nGnRE

LDR x1, =0x00000000000002500 // Setup TCR_EL3
MSR TCR_EL3, x1           // We are using 4k granularity, mapping 64-bit
// VA space to 32-bit PA space
ISB                       // And (outer-shareable policy, Inner and Outer
// WBWA Normal memory for Inner and Outer
// Regions) for translation tables

LDR x20, =TTB0_L0_BASE     // Setup L0, L1 and L2 page tables
LDR x21, =TTB0_L1_BASE
LDR x22, =TTB0_L2_BASE

LDR x4, =RO_BASE           // Create the single entry in L0 table
UBFX x23, x4, #39, #9      // Index into L0 table
ORR x1, x21, #0x3          // Setup the page table descriptor to point
// to L1 table base address(x21) and OR the
// PAGE attribute bits
STR x1, [x20, x23, lsl #3] // Write the L1 page table descriptor into
// the L0 table(x20) at the correct index

LDR x4, =RO_BASE           // Create the single entry in L1 table
UBFX x23, x4, #30, #9      // Index into L1 table (1GB per region)
ORR x1, x22, #0x3          // Setup the page table descriptor to point
// to L2 table base address(x22) and OR the
// PAGE attribute bits
STR x1, [x21, x23, lsl #3] // Write the L2 page table descriptor into
// the L1 table(x21) at the correct index

LDR x4, =RO_BASE           // Write L2 table entry corresponding to RO_BASE
UBFX x2, x4, #21, #9       // Index into L2 table (each entry covers
// 2MB) corresponding to RO_BASE
BIC x4, x4, #((1<<21)-1) // Create it as a BLOCK entry
LDR x1, =0xF25             // Configure bits in the page table
// descriptor for the current mapping as:-
// Block entry, Index 1 into MAIR for
// memory region type, Non-secure,
// AP=0 for ReadWrite Permissions,
// Inner Shareable, Access Flag = 1
// and Non-Global = 1

ORR x1, x1, x4
```

```

    STR x1, [x22, x2, lsl #3]          // Store the entry into the L2 page table

    LDR x4, =RW_BASE                  // Write L2 table entry corresponding to RW_BASE
    UBFX x2, x4, #21, #9              // Index into L2 table corresponding
                                      // to RW_BASE

    BIC x4, x4, #((1<<21)-1)
    LDR x1, =0xF25                    // Create it as a BLOCK entry
    ORR x1, x1, x4
    STR x1, [x22, x2, lsl #3]          // Store the entry into the L2 page table

    DSB ISH                           // Issue a barrier to ensure all table entry
                                      // writes are complete

    // -----
    // Enable the MMU
    // -----
    MRS x1, SCTLR_EL3
    ORR x1, x1, (1<<0)                // #SCTLR_ELx_M
    BIC x1, x1, (1<<1)                // #SCTLR_ELx_A // Disable alignment fault
checking.
    MSR SCTLR_EL3, x1
    ISB

    // -----
    // Enable the caches
    // -----
    MRS x1, SCTLR_EL3
    ORR x1, x1, (1<<2)                // #SCTLR_ELx_C
    ORR x1, x1, (1<<12)               // #SCTLR_ELx_I
    MSR SCTLR_EL3, x1
    ISB

    RET

```

10. Now that we have the helper functions in place, we can get back to our main example code.

Recollect that we determined which cluster we belong to, and branched off to `first` or `second` label depending on it. Now is the time to define the code for those labels:

```

// -----
// Core in the first cluster
// -----
first:
    BL configure_mmu_and_caches

    SEV
    WFE                               // Additional WFE needed to clear out the event
                                      // register set as a result of the configuration
                                      // done in configure_mmu_and_caches

    WFE

    // Set x8 to the address of a random memory location in RW_BASE section
    // that is shared with second cluster
    LDR x1, =RW_BASE
    ADD x8, x1, #0x654

    // Read the shared memory location and print value
    LDRB w0, [x8]
    BL print_value

    SEV
    WFE                               // Additional WFE needed to clear out the event
                                      // register set because of the HLT instruction
                                      // used for the semihosting call

    WFE

    // Write to the shared memory location
    MOV w2, #6

```

```

        STRB w2, [x8]
        LDRB w0, [x8]      // Read back the value and print it
        BL print_value

        SEV

1:      B 1b

// -----
// Core in the second cluster
// -----
second:
        WFE

        BL configure_mmu_and_caches

        SEV
        WFE                // Additional WFE needed to clear out the event
                           // register set as a result of the configuration
                           // done in configure_mmu_and_caches

        WFE

        // Set x8 to the address of the shared memory location
        LDR x1, =RW_BASE
        ADD x8, x1, #0x654

        // Read the shared memory location and print value
        LDRB w0, [x8]
        BL print_value

        SEV
        WFE                // Additional WFE needed to clear out the event
                           // register set because of the HLT instruction
                           // used for the semihosting call

        WFE

        // Read the shared memory location again and print value
        LDRB w0, [x8]
        BL print_value

        MOV w0, #SYS_EXIT      // Semihosting exit call
        HLT #SH_TRAP_INST_A64

2:      B 2b

```

Notice that we use the `sev` (Send Event) and `wfe` (Wait for Event) instructions to sequence the operations between the cores and avoid parallel operation of the cores.

So, the sequence of operations is:

- a. First cluster configures its MMU and Caches.
- b. Second cluster configures its MMU and Caches.
- c. First cluster reads the shared memory location.
- d. Second cluster reads the shared memory location.
- e. First cluster writes to the shared memory location.
- f. Second cluster reads the shared memory location.
- g. Second cluster terminates the application using the semihosting exit call.

One issue with the `WFE` instruction is that it returns immediately if the event register of the core is set. The event register gets set for multiple reasons as mentioned in the Architecture Reference Manual. One such case is the core entering halting debug state. This happens because of the `HLT` instruction in the semihosting call. To account for this, we need to use an additional `WFE` instruction after the `print_value()` call that uses semihosting, to really wait on the event. This is also the case after the call to `configure_mmu_and_caches()`. We need an additional `WFE` to make the core truly wait on the event sent from the other cluster.

7.5 Compiling the code to run on the example model

After we have written the assembly code, we need to compile it into an ELF (Executable and Linkable Format) image.

Procedure

1. Ensure that Arm Compiler for Embedded is in your `$PATH`. For installation instructions, see [System requirements and installation](#)
2. Compile `startup.s` into an object file `startup.o`:

```
armclang -c --target=aarch64-arm-none-eabi -march=armv8-a startup.S
```

3. Link the object file `startup.o` into an ELF object that can be run. Specify the entry point and the locations of the two executable regions (`ro-base` and `rw-base`) in the default linker scatter file as explained in the previous section:

```
armlink --ro-base=0x80000000 --rw-base=0x80200000 startup.o -o image.axf --entry=start64
```

7.6 Running the code and testing coherency on the example model

We will run the model in a few different ways to show coherency in action. In the next section, we will use model traces to show coherency being handled.

Run the model standalone

Running the code on the model we built is simple. The model is an executable called `isim_system` and takes the ELF image created as an argument.

- On Linux, if you are using GCC-9.3:

```
cd $PVLIB_HOME/examples/LISAPlus/PVCoherentInterconnect
./Linux64-Release-GCC-9.3/isim_system -a armcortexa72ct.cpu0=image.axf -a armcortexa53ct.cpu0=image.axf
```

- On Microsoft Windows:

```
cd %PVLIB_HOME%\examples\LISAPlus\PVCoherentInterconnect
.\Win64-Release-VC2019\isim_system.exe -a armcortexa72ct.cpu0=image.axf -a
armcortexa53ct.cpu0=image.axf
```



We use the format `INST=FILE` for each `-a` option to specify that the image be loaded on both clusters.

You will see the following output:

```
Printing from cluster0, value is 0
Printing from cluster1, value is 0
Printing from cluster0, value is 6
Printing from cluster1, value is 6

Info: /OSCI/SystemC: Simulation stopped by user.
```

The output shows that the clusters execute in turn in the sequence specified in the code using `sev` and `wfe` instructions.

After setting up the page table and the caches and MMU on each cluster, the core in `cluster0` first reads the byte value from the shared memory location. Since we have specified a `RAMDevice` component in our model definition with fill values of zero, the initial values of the memory locations are guaranteed to be zero:

```
ramdevice : RAMDevice("fill1"=0x0,"fill2"=0x0);
```

The core in `cluster1` then reads the value from the same location and reads a value of zero correctly. After that, the core in `cluster0` writes a value of 6 and reads it back and prints the value. And later, `cluster1` reads the shared memory location again and finds the new value 6.

In the next section, we will show that the coherency has been maintained without the new value itself being written all the way to memory (`RAMDevice`).

Run the model with Iris Monitor

Iris Monitor is a browser-based debugger for Fast Models which uses the Iris interface to connect to models. It is included with the Fast Models package.

1. From the terminal, launch the model using the command line shown previously, but this time, add the `-i` option to start the Iris server. This enables Iris Monitor to connect to the model.
 - On Linux, if you are using GCC-9.3:

```
./Linux64-Release-GCC-9.3/isim_system -a armcortexa72ct.cpu0=image.axf -a
armcortexa53ct.cpu0=image.axf -i
```

- On Microsoft Windows:

```
.\.Win64-Release-VC2019\isim_system.exe -a armcortexa72ct.cpu0=image.axf -a  
armcortexa53ct.cpu0=image.axf -I
```

- From another terminal, launch Iris Monitor, for example by typing `irismonitor` on Linux, assuming it is on your path, or using the Start Menu on Windows. Iris Monitor is installed in `$MAXCORE_HOME/bin/IrisMonitor/bin/`.

Iris Monitor automatically connects to the model and prints an IP address to the terminal, by default `http://127.0.0.1:8080`.

- Paste the IP address shown in the terminal into the address bar of a web browser. If the connection is successful, the browser displays the [Iris Monitor GUI](#).

The **Disassembly** view shows the application image has been loaded into memory at the address we specified in the `armlink` command (`0x80000000`) for the two cores to execute:

Figure 7-3: Iris Monitor Disassembly view

DISASSEMBLY			
Memory Space	Address	Size	Mode
<input type="text" value="0x80000000"/>	<input type="text" value="0x80000000"/>	<input type="text" value="100"/>	<input type="text" value="Auto"/>
Address	Op Code	Disassembly	
0x80000000	d53800a0	MRS x0,MPIDR_EL1	
0x80000004	d3483c13	UBFX x19,x0,#8,#8	
0x80000008	b4000a53	CBZ x19,0x80000150	
0x8000000C	14000062	B 0x80000194	
0x80000010	f100027f	CMP x19,#0	
0x80000014	54000081	B.NE 0x80000024	
0x80000018	58000e03	LDR x3,0x800001d8	

- Click **Run** in the Iris Monitor Simulation status and control area. Our example code, being very simple, completes in an instant and the simulation stops. You can see the following output in the terminal that the model was launched from, showing that each cluster sees the updated value of 6 for the shared memory location:

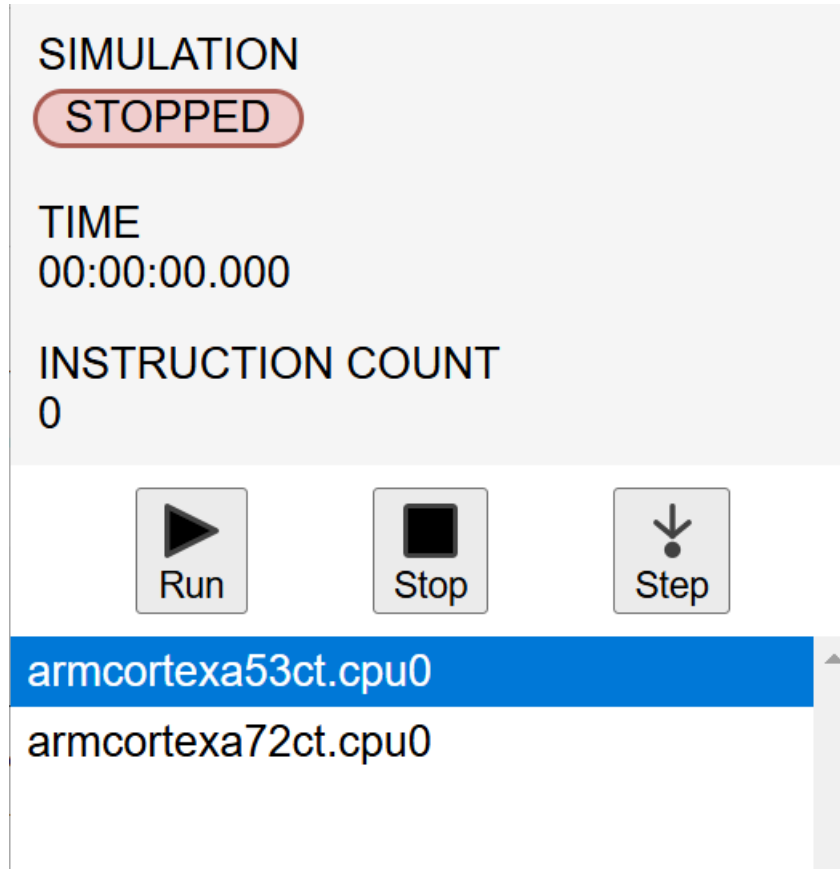
Figure 7-4: Terminal output

```
Printing from cluster0, value is 0  
Printing from cluster1, value is 0  
Printing from cluster0, value is 6  
Printing from cluster1, value is 6
```

- We will next inspect `RAMDevice` and look for the memory location at the physical address `0x80200654`.

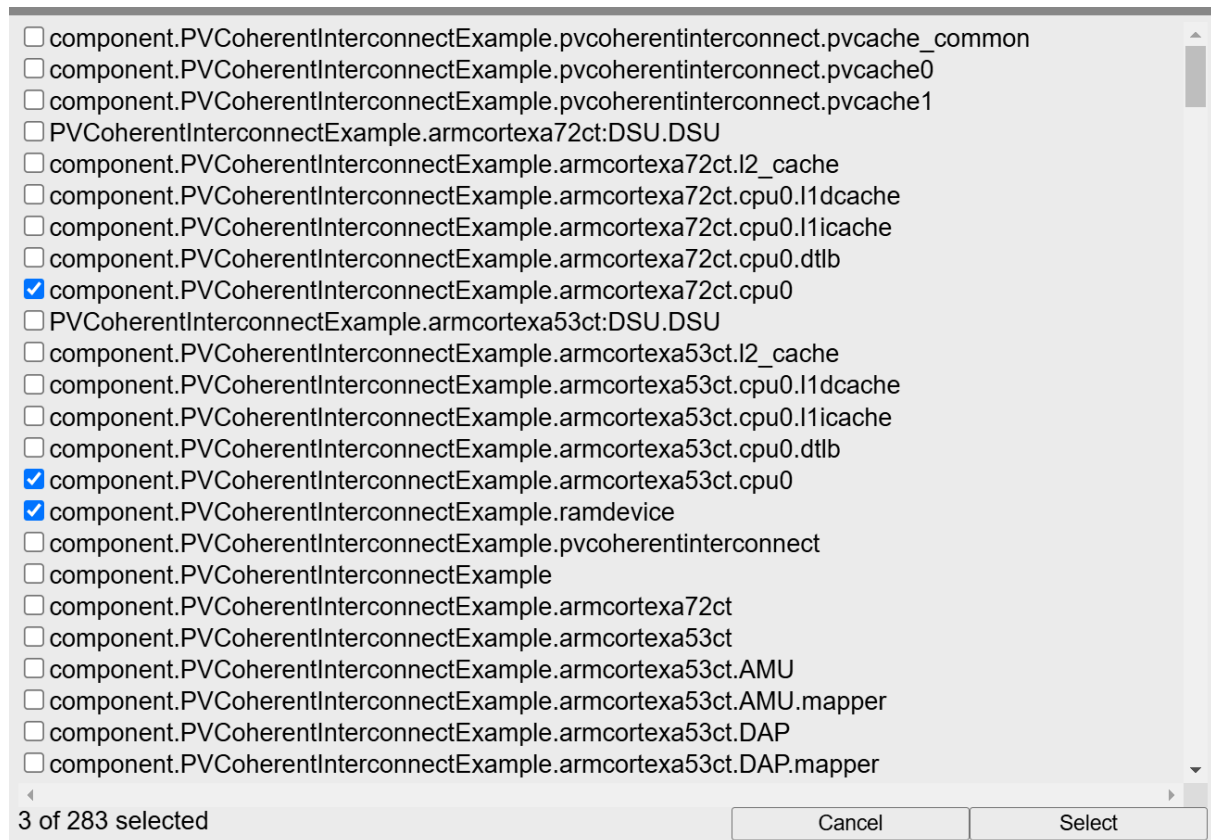
The instance selection area of Iris Monitor shows a filtered list of the instances in the model. By default, it only lists instances that support software execution, in this example, `armcortexa53ct.cpu0` and `armcortexa72ct.cpu0`:

Figure 7-5: Iris Monitor instance selection area



Click **Change**.

6. Select the checkbox for the `component.PVCoherentInterconnectExample.ramdevice` instance, then click **Select**:

Figure 7-6: Iris Monitor instance list

The Iris Monitor instance selection area now shows `ramdevice`, `armcortexa53ct.cpu0`, and `armcortexa72ct.cpu0`.

7. Select `ramdevice` by clicking on it.
8. In the Memory view, enter `0x80200654` in the **Address** field. You can see that the physical memory location still contains `0` and is not yet updated with the new value of `6`. This proves that inter-cluster coherency between the two clusters is being handled properly by the `PVCoherentInterconnect` component.

Figure 7-7: RAM contents

MEMORY				
<div> <div>ram_contents ▾</div> <div>0x80200654</div> <div>128</div> <div>4 ▾</div> </div> <div>Load</div>				
Address				
80200654	00000000	00000000	00000000	
80200664	00000000	00000000	00000000	
80200674	00000000	00000000	00000000	
80200684	00000000	00000000	00000000	
80200694	00000000	00000000	00000000	
802006A4	00000000	00000000	00000000	

7.7 Using Model Trace to show coherency

Refer to the Fast Models tutorial [Toggle trace using ToggleMTIPlugin](#), and the section [Toggle trace using HLT instructions](#).

We will use the HLT-based method to restrict trace generation to the accesses from each cluster to the shared memory location.

Procedure

1. Update the code to insert HLT #0x5 instructions around the accesses to the shared memory.

Code in the `first` and `second` labels with this change is shown below:

```
// -----
// Core in the first cluster
// -----
first:
    BL configure_mmu_and_caches

    SEV
    WFE                                // Additional WFE needed to clear out the event
                                      // register set as a result of the configuration
                                      // done in configure_mmu_and_caches
    WFE

    // Set x8 to the address of a random memory location in RW_BASE section
    // that is shared with second cluster
    LDR x1, =RW_BASE
    ADD x8, x1, #0x654

    // Read the shared memory location and print value
    HLT #0x5
    LDRB w0, [x8]
    HLT #0x5
    BL print_value
```

```

SEV
WFE                                     // Additional WFE needed to clear out the event
                                     // register set because of the HLT instruction
                                     // used for the semihosting call

WFE

// Write to the shared memory location
MOV w2, #6
HLT #0x5
STRB w2, [x8]
LDRB w0, [x8]      // Read back the value and print it
HLT #0x5
BL print_value

SEV

1:
  B 1b

// -----
// Core in the second cluster
// -----
second:
  WFE

  BL configure_mmu_and_caches

  SEV
  WFE                                     // Additional WFE needed to clear out the event
                                     // register set as a result of the configuration
                                     // done in configure_mmu_and_caches

  WFE

  // Set x8 to the address of the shared memory location
  LDR x1, =RW_BASE
  ADD x8, x1, #0x654

  // Read the shared memory location and print value
  HLT #0x5
  LDRB w0, [x8]
  HLT #0x5
  BL print_value

  SEV
  WFE                                     // Additional WFE needed to clear out the event
                                     // register set because of the HLT instruction
                                     // used for the semihosting call

  WFE

  // Read the shared memory location again and print value
  HLT #0x5
  LDRB w0, [x8]
  HLT #0x5
  BL print_value

  MOV w0, #SYS_EXIT      // Semihosting exit call
  HLT #SH_TRAP_INST_A64

2:
  B 2b

```

2. Compile and generate the ELF image again:

```
armclang -c --target=aarch64-arm-none-eabi -march=armv8-a startup.S
```

```
armlink --ro-base=0x80000000 --rw-base=0x80200000 startup.o -o image.axf --entry=start64
```

3. Run the model with parameters to generate trace using ToggleMTIPlugin and HLT instruction-based trace toggling.

The full command is:

```
./Linux64-Release-GCC-9.3/isim system -a armcortexa72ct.cpu0=image.axf -a
armcortexa53ct.cpu0=image.axf \
--plugin $PVLIB_HOME/plugins/Linux64_GCC-9.3/GenericTrace.so \
--plugin $PVLIB_HOME/plugins/Linux64_GCC-9.3/ToggleMTIPlugin.so \
-C TRACE.GenericTrace.trace-sources="*"
-C TRACE.ToggleMTIPlugin.use_hlt=1 \
-C TRACE.ToggleMTIPlugin.disable_mti_from_start=1 \
-C TRACE.ToggleMTIPlugin.hlt_imm16=5 \
-C armcortexa72ct.cpu0.enable_trace_special_hlt_imm16=1 \
-C armcortexa72ct.cpu0.trace_special_hlt_imm16=5 \
-C armcortexa53ct.cpu0.enable_trace_special_hlt_imm16=1 \
-C armcortexa53ct.cpu0.trace_special_hlt_imm16=5
> trace_output.log
```

Notice that we are using a different plugin, `GenericTrace.so` instead of `TarmacTrace.so`. This is because the `GenericTrace` plugin gathers the trace from all the components in the system, while the `TarmacTrace` plugin only gathers the trace generated from the cores.

`GenericTrace` also takes an argument `TRACE.GenericTrace.trace-sources` to filter the trace to a particular set of trace sources. Here, we are gathering trace from all the sources at the point of the access. Hence, we use `"*"` for the argument.

The amount of trace generated is still significant. So, we redirect the output to a file so we can inspect it later.

4. Open the trace output and search for 80200654 to locate the transactions corresponding to the shared address.

Since the trace output is very detailed, just the key trace points are highlighted here to keep it simple. Further traces around these points can be checked further to look at the complete sequence.

- The first access in our code sequence is a read from the first cluster (A72CT).

Accordingly, the first matching line in the log shows the access sequence from `cpu0` in `armcortexa72ct` cluster in the log. Since this is the first time this address is being accessed, there is a TLB MISS:

```
armcortexa72ct.cpu0.UTLB.MMU_TLB_MISS: CORE_NUM=0x00 SIDE=Data
VADDR=0x0000000080200654 ASID=0x00000000 VMID=0x00000000 NS=Secure Hyp=N
REGIME_EL=EL3
```

Further trace logs (which are not shared here for brevity) indicate the page tables being fetched and read, as a result.

Then, the mapping gets created. Caches (11dcache and 12_cache) are then checked, and they indicate a miss on the translated Physical Address:

```
armcortexa72ct.cpu0.MMU_TRANS: CORE_NUM=0x00 SIDE=Data ASID=0x0000
VMID=0x00 VADDR=0x00000000080200654_Hyp=N nG=Non-Global
NSDESC=NonSecure PADDR=0x00000000080200654 PAGESIZE=0x15 MEMTYPE=Normal
OUTERCACHE_TYPE=WriteBack OUTERCACHE_RA=Y OUTERCACHE_WA=Y
OUTERCACHE_TRANSIENT=N INNERCACHE_TYPE=WriteBack INNERCACHE_RA=Y
INNERCACHE_WA=Y INNERCACHE_TRANSIENT=N SH=InnerShareable
STAGE1_PERM=0x00000077 STAGE2_PERM=0x77 XS=N PAS=NS MPAM_SP=0x00
MPAM_PMG=0x00 MPAM_PARTID=0x0000 MECID=0x0000
armcortexa72ct.cpu0.l1dcache.upstream[0].DMI_BUS_SLAVE:
DMI_OPERATION=SET STORAGE ADDRESS=0x00000000080200000 SIZE=0x00001000
STORAGE=0x0000000000000000
armcortexa72ct.cpu0.l1dcache.CACHE_READ_MISS: IS_SHARED=SHARED
IS_PRELOAD=NOT_PRELOAD MASTER_ID=0x00000000 LATENCY=0x0000000000000000
armcortexa72ct.l2_cache.upstream[0].DMI_BUS_SLAVE: DMI_OPERATION=SET STORAGE
ADDRESS=0x00000000080200000 SIZE=0x00001000 STORAGE=0x0000000000000000
armcortexa72ct.l2_cache.CACHE_READ_MISS: IS_SHARED=SHARED
IS_PRELOAD=NOT_PRELOAD MASTER_ID=0x00000000 LATENCY=0x0000000000000000
```

This results in an external bus access from the cluster:

[illegible]

A complete cache line of size 64 bytes is fetched from memory (`ramdevice`) starting from address `0x0000000080200640`, which includes our shared memory location. Here we see that the coherency operation for this transaction is indicated by `ACE=ReadShared`. We also see `pycoherentinterconnect` in action.

Further trace after this shows the cache lines being filled and the value being passed to the core.

- The second access is a read from the second cluster (A53CT).

Here, since this is the first access from the core in this cluster, we see the similar sequence of:

[illegible]

- The fourth access is a read back from the first cluster (A72CT).

We see that the access causes a TLB hit and a cache hit as well, and the transaction is completely satisfied from the cache without initiating any external bus transaction. We see that the read value is the same as the previous write (`data=0x06:1`):

```
armcortexa72ct.cpu0.l1dcache.CACHE_READ_HIT: IS_SHARED=SHARED
IS_PRELOAD=NOT_PRELOAD MASTER_ID=0x00000000 LATENCY=0x0000000000000000
armcortexa72ct.cpu0.l1dcache.entry after read: entry_index=0x00000032
entry_begin_address=0x0000000080200640 ns=ns snoop=N ace_operation=ReadOnce
unique=Y dirty=Y trans_begin_address=0x0000000080200654 data=0x06:1
pas=non secure
```

- The fifth access is a read from the second cluster (A53CT).

Here, we see that there is a TLB hit. However, it misses in the cache because the caches in the A53CT cluster have been invalidated to maintain coherency as part of the third access (write from the other cluster, A72CT). Hence, it initiates an external bus transaction to fetch the data. The bus transaction is satisfied by the other cluster (A72CT) as a snoop request `snoop=y` and the actual memory in `ramdevice` is not accessed. Looking closely at the following trace snippet shows that the data value is indeed 6 and not 0:

[illegible]

The above steps show the coherency being handled properly by `pvcCoherentInterconnect`.

8. Timing annotation

This tutorial shows how to use the Cycles Per Instruction (CPI) specification modeling feature with a Fast Models example platform model, and how to measure its impact on code execution time. The commands shown are for Linux, although the process is the same on Windows.

8.1 Prerequisites

This tutorial uses the following:

- A SystemC-integrated virtual platform, for instance an ISIM or an EVS platform. This tutorial uses the FVP_Base_Cortex-A57 example virtual platform.
- An application to run on the platform. This tutorial uses one of the sample images included in the Fast Models package in the `$PVLIB_HOME/images/` directory.
- A way of calculating the execution 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.

8.2 Build the FVP_Base_Cortex-A57 example platform

The FVP_Base_Cortex-A57 platform is provided as a source code example in the Fast Models Portfolio, so we must first build it.

For example:

```
cd $PVLIB_HOME/examples/LISA/FVP_Base/Build_Cortex-A57
simgen -p FVP_Base_Cortex-A57.sgproj -b
```

8.3 Calculate the execution time of an instruction using the default CPI value

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. This example uses the `INST` trace source to measure the time it takes to execute an instruction.

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 100 MHz, 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.

To generate trace, load the `GenericTrace` plug-in. This plug-in allows you to output any number of MTI trace sources to a file. We also need an application to run on the platform. We will use the Dhrystone image, which is included in the Fast Models package in the `$PVLIB_HOME/images/` directory.

If you run the model without using the `cpi_mul` or `cpi_div` model parameters, a default CPI value of 1.00 is used. This value establishes a baseline to compare the other CPI configurations against.

Launch the platform with some extra parameters to load the `GenericTrace` plug-in and collect the `INST` trace source, see the last 3 lines in the following command:

```
echo 10000 | ./Linux64-Release-GCC-9.3/isim_system -C bp.secure_memory=0 \
-a $PVLIB_HOME/images/dhrystone_v8.axf \
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-9.3/GenericTrace.so \
-C TRACE.GenericTrace.trace-sources=INST \
-C TRACE.GenericTrace.trace-file=trace.log
```

In the trace file that the `GenericTrace` plug-in produces, the first 2 lines are as follows:

```
cpu3.INST: DEBUG STATE=N PC=0x0000000080001098 OPCODE=0xd51e115f SIZE=0x04
MODE=EL3h ISET=AArch64 PADDR=0x0000000080001098 NSDESC=0x00 PAS=S
PADDR2=0x0000000080001098 NSDESC2=0x00 PAS=S NS=0x00 SECURITY STATE=S
ITSTATE=0x00 INST_COUNT=0x0000000000000001 LOCAL_TIME=0x00000000000002710
CURRENT_TIME=0x00000000000002710 CORE_NUM=0x03 DISASS="MSR      CPTR_EL3,xzr"

cpu3.INST: DEBUG STATE=N PC=0x000000008000109c OPCODE=0xd53800a0 SIZE=0x04
MODE=EL3h ISET=AArch64 PADDR=0x000000008000109c NSDESC=0x00 PAS=S
PADDR2=0x000000008000109c NSDESC2=0x00 PAS=S NS=0x00 SECURITY STATE=S
ITSTATE=0x00 INST_COUNT=0x0000000000000002 LOCAL_TIME=0x00000000000004e20
CURRENT_TIME=0x00000000000004e20 CORE_NUM=0x03 DISASS="MRS      x0,MPIDR_EL1"
```

The `CURRENT_TIME` value of the first instruction is `0x2710`, or 10000 ps. The difference between the two instructions is also `0x2710`. This shows that the both instructions took one 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.

8.4 Display the total execution time of the simulation

To determine the overall simulation time, use the command-line option `--stat`.

Launch the platform again, using the `--stat` option:

```
echo 10000 | ./Linux64-Release-GCC-9.3/isim_system -C bp.secure_memory=0 -a
$PVLIB_HOME/images/dhrystone_v8.axf --stat
```

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:

```
--- FVP_Base_Cortex_A57 statistics: -----
Simulated time           : 0.045701s
User time at initialisation : 0.247442s
```



```

User time                : 0.036818s
System time at initialisation : 0.331253s
System time              : 0.013061s
Wall time at initialisation : 1.140045s
Wall time                 : 0.068660s
Performance index         : 0.67
cluster0.cpu0             : 66.63 MIPS ( 4574531 Inst)
cluster0.cpu1             : 0.00 MIPS ( 8 Inst)
cluster0.cpu2             : 0.00 MIPS ( 8 Inst)
cluster0.cpu3             : 0.00 MIPS ( 8 Inst)
Total                     : 66.63 MIPS ( 4574555 Inst)
Memory highwater mark at initialisation : 0x198b6000 bytes ( 0.399 GB )
Memory highwater mark     : 0x19f83000 bytes ( 0.406 GB )
-----

```

**Note**

The MIPS values are based on the host system time, not the simulated time.

8.5 Calculate 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 0.999:

```
average_cpi = (0.045701 * 10^12) / (10000 * 4574555) = 0.999
```

8.6 Run the example with a non-default CPI value

This section demonstrates how to model the simulated time per instruction by using the CPI timing annotation feature.

You can specify a CPI value for all instructions that execute within a cluster by using the per-cluster model parameters `cpi_mul` and `cpi_div`. If you do not set these parameters, 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 CPI value is used in a way that $\text{core_clock_period} * \text{fixed_cpi_value}$ is rounded to the nearest picosecond.



You can combine the CPI value with other timing annotation features. Therefore, the average CPI value that you observe can be different from the fixed CPI value that you specify.

Assuming we want an average CPI value of 0.75, the fraction can be applied to the simulation by using the `cpi_mul` and `cpi_div` model parameters as follows:

```
echo 10000 | ./Linux64-Release-GCC-9.3/isim_system -C bp.secure_memory=0 /
-a $PVLIB_HOME/images/dhrystone_v8.axf -C cluster0.cpi_mul=3 -C cluster0.cpi_div=4
--plugin=$PVLIB_HOME/plugins/Linux64_GCC-9.3/GenericTrace.so /
-C TRACE.GenericTrace.trace-sources=INST /
-C TRACE.GenericTrace.trace-file=trace.log --stat
```

For each instruction, a simulated time of 7500 ps or 0.7500 ticks can be observed using the GenericTrace plug-in. The `--stat` output shows an average CPI value of approximately 0.75 $((0.034300 * 10^{12}) / (10000 * 4579328))$:

```
--- FVP_Base_Cortex_A57 statistics: -----
Simulated time           : 0.034300s
User time at initialisation : 0.210786s
User time                 : 12.758075s
System time at initialisation : 0.255376s
System time               : 6.596258s
Wall time at initialisation : 0.541965s
Wall time                  : 19.388600s
Performance index         : 0.00
cluster0.cpu0              : 0.24 MIPS ( 4579304 Inst)
cluster0.cpu1              : 0.00 MIPS ( 8 Inst)
cluster0.cpu2              : 0.00 MIPS ( 8 Inst)
cluster0.cpu3              : 0.00 MIPS ( 8 Inst)
Total                      : 0.24 MIPS ( 4579328 Inst)
Memory highwater mark at initialisation : 0x19a2f000 bytes ( 0.401 GB )
Memory highwater mark      : 0x1c0e7000 bytes ( 0.438 GB )
-----
```

8.7 Additional timing annotation features

In addition to the CPI parameters, there are other parameters that can be used to change the observed CPI of the core.

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`.



- 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.

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

export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. Please follow Arm’s trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners.

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 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
0100-04	19 November 2025	Non-Confidential	Update for v11.30
0100-03	16 May 2025	Non-Confidential	Update for v11.29
0100-02	19 February 2025	Non-Confidential	Update for v11.28
0100-01	12 June 2024	Non-Confidential	Update for v11.26
0100-00	22 March 2023	Non-Confidential	Initial release

Change history

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.
monospace <u>underline</u>	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: <div>MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2></div>
SMALL CAPITALS	Terms that have specific technical meanings as defined in the <i>Arm® Glossary</i> . For example, IMPLEMENTATION DEFINED , IMPLEMENTATION SPECIFIC , UNKNOWN , and UNPREDICTABLE .



We recommend the following. If you do not follow these recommendations your system might not work.



Your system requires the following. If you do not follow these requirements your system will not work.



You are at risk of causing permanent damage to your system or your equipment, or harming yourself.



This information is important and needs your attention.



A useful tip that might make it easier, better or faster to perform a task.



A reminder of something important that relates to the information you are reading.

Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Arm documents are available on developer.arm.com/documentation.

Confidential documents are only available to licensees, when logged in. Each document link in the tables below provides direct access to the online version of the document.

Arm product resources	Document ID	Confidentiality
Arm® Compiler for Embedded User Guide	100748	Non-Confidential
Arm® Debugger Command Reference	101471	Non-Confidential
Arm® Development Studio	–	Non-Confidential
Arm® Development Studio Getting Started Guide	101469	Non-Confidential
Arm® Development Studio User Guide	101470	Non-Confidential
Fast Models Reference Guide	100964	Non-Confidential
Fast Models Tools User Guide	109415	Non-Confidential
Fast Models User Guide	100965	Non-Confidential
Product Download Hub	–	Non-Confidential

Non-Arm resources	Document ID	Organization
ELF object file format, Section Attribute Flags	–	https://www.sco.com