arm

Fast Models

Version 1.0

Tutorials

Fast Models

Tutorials

Release information

Document history

Issue	Date	Confidentiality	Change
0100-00	22 March 2023	Non-Confidential	Initial release

Proprietary Notice

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 the Agreement shall prevail.

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. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm's trademark usage guidelines at https://www.arm.com/company/policies/trademarks.

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

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

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.

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.

Contents

1. Introduction

1.1 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` `underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<and>`	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.

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

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

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

	This is important information and needs your attention.
Note	

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

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

1.2 Useful resources

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

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.

- Confidential documents are available to licensees only through the product package.

Arm product resources	Document ID	Confidentiality
Arm® Compiler for Embedded User Guide	100748	Non-Confidential
Arm® Development Studio	–	Non-Confidential
Arm® Development Studio Debugger Command Reference	101471	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 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

1.3 Other information

See the Arm website for other relevant information.

- Arm® Developer.
- Arm® Documentation.
- Technical Support.
- Arm® Glossary.

2. Constructing a simple Fast Model

In this tutorial, we will build a simple model of an ARMCortexR52x1CT. Then we will write hello world and run it on the model.

2.1 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 SGCanvas.

- 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 The instructions here are compact instructions for constructing a model using System Canvas. Please refer to the Fast Models 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 2-1: Create new Project
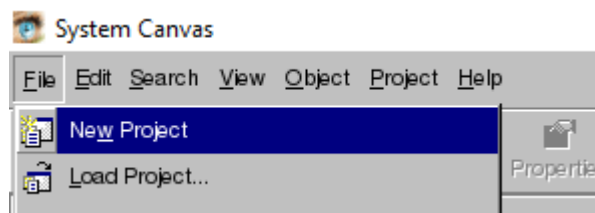
3. Click on the **Block Diagram** tab.

Figure 2-2: Click Block Diagram tab
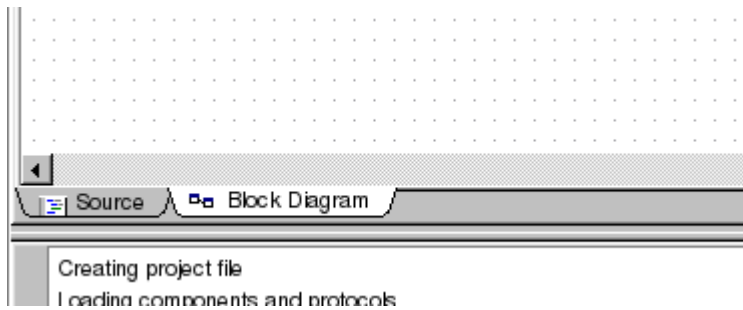
4. Add the following components to the canvas by clicking and dragging them from the Component list on the right onto the canvas:

- MasterClock

- ARMCortexR52x1CT

- PVBusDecoder

- RAMDevice

Figure 2-3: Drag components
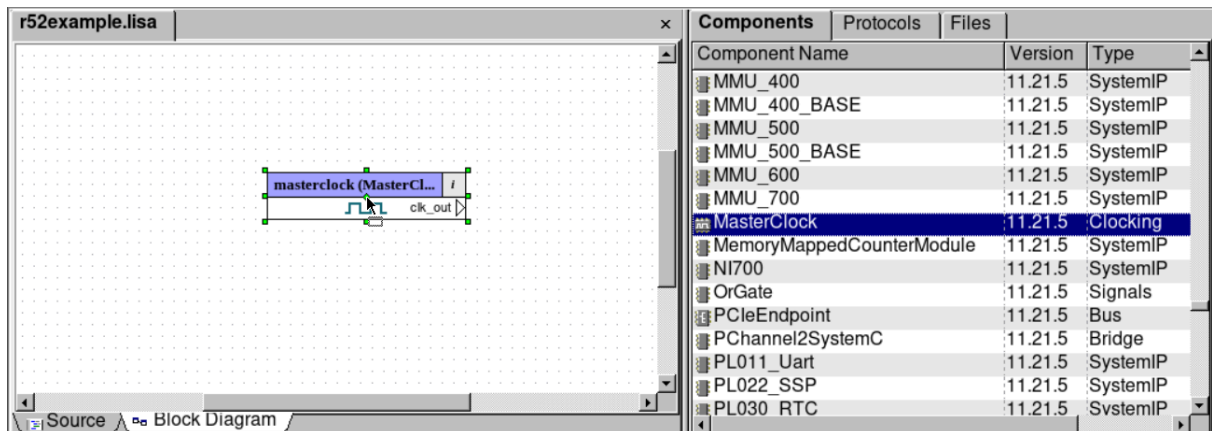
The **Block Diagram** view shows the components:

Figure 2-4: All components in Canvas
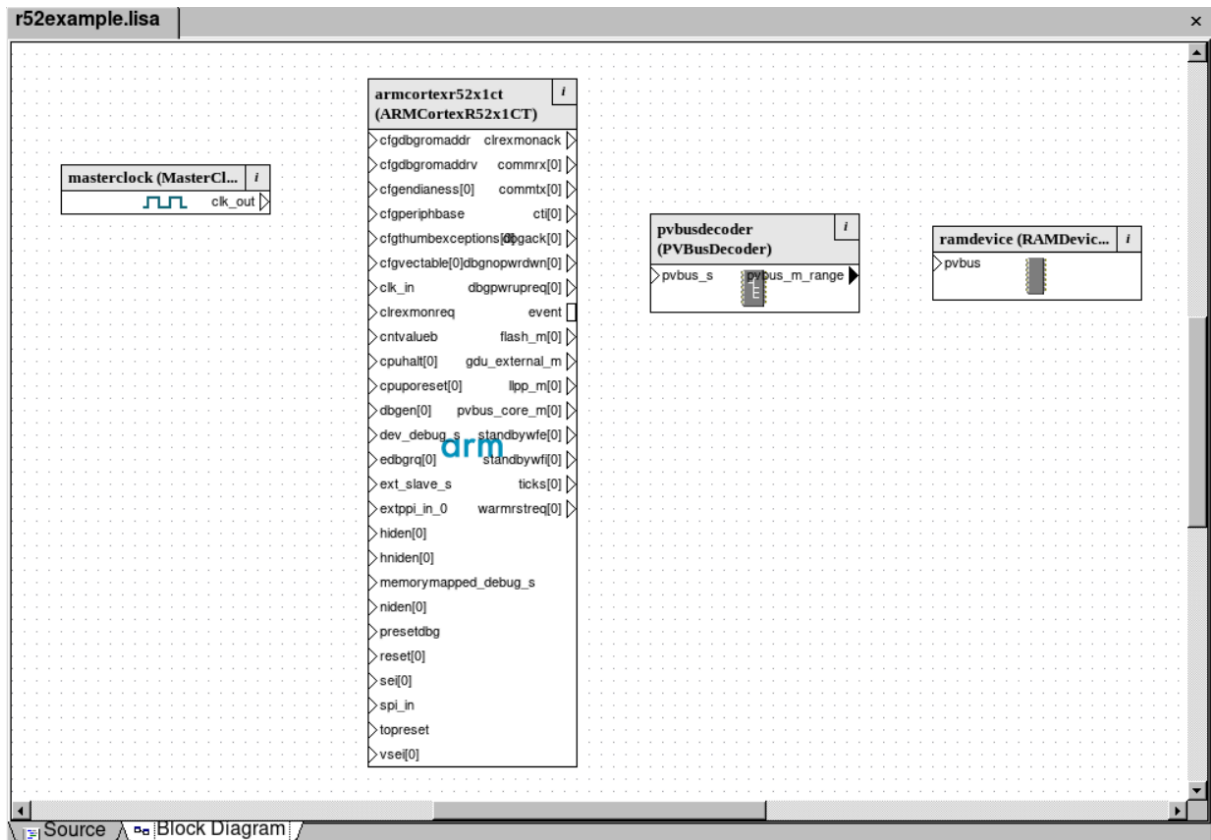
5. Connect the components using the **Connect** tool on the tool bar:

   - Connect MasterClock `clk_out` to ARMCortexR52x1CT `clk_in`

   - Connect ARMCortexR52x1CT `pvbus_core_m[0]` to PVBusDecoder `pvbus_s`

   - Connect PVBusDecoder `pvbus_m_range` to RAMDevice `pvbus`

Figure 2-5: Connected system
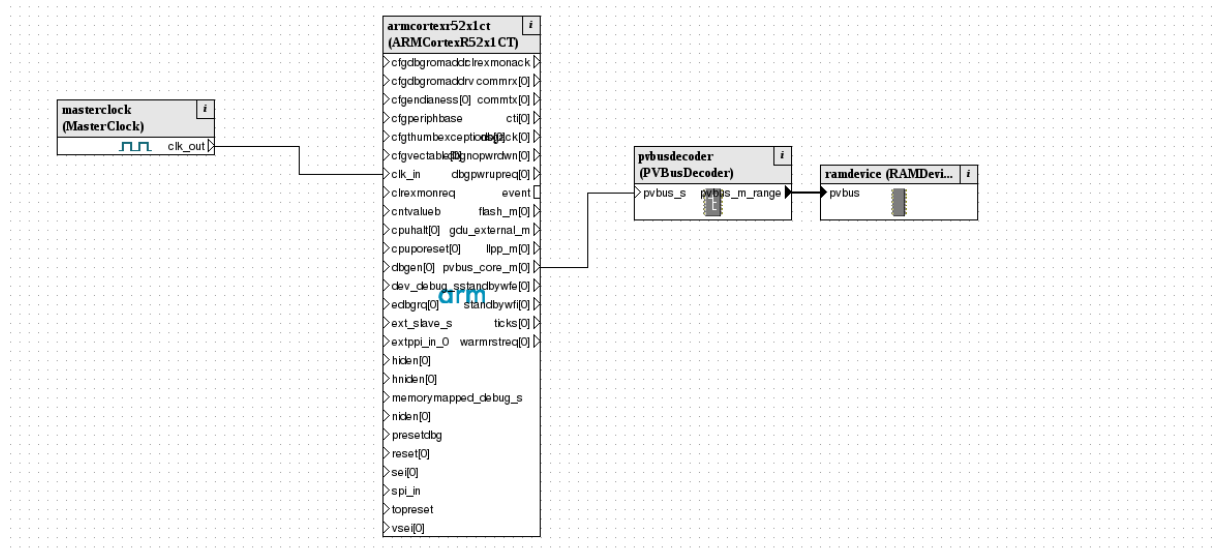
6. Map the PVBusDecoder to your RAMDevice:

   - Right click on the `pvbus_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 `0x0FFFFF` (1MB mapped to RAM)

Figure 2-6: Mapping the PVBusDecoder
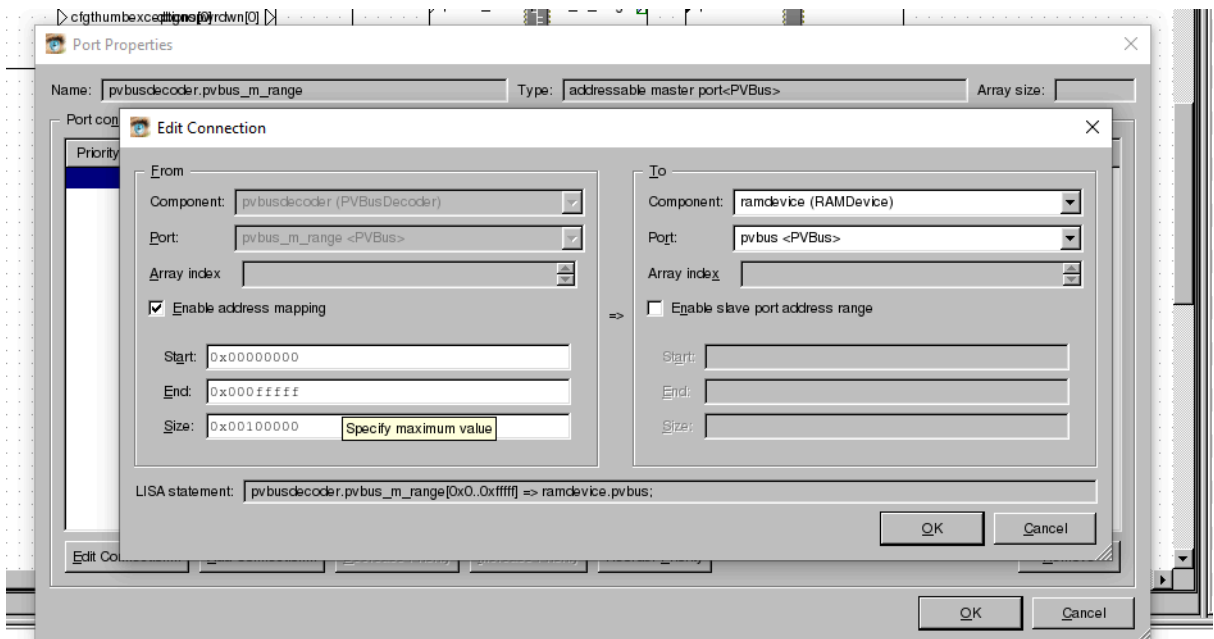
At this point if you click on the **Source** tab you will see:

```
// This file was generated by System Generator Canvas
// -------------------------------------------------
component r52example
{
    composition
    {
        pvbusdecoder : PVBusDecoder();
        armcortexr52x1ct : ARMCortexR52x1CT();
        ramdevice : RAMDevice();
        masterclock : MasterClock();
    }
    connection
    {
    masterclock.clk_out => armcortexr52x1ct.clk_in;
    rmcortexr52x1ct.pvbus_core_m[0] => pvbusdecoder.pvbus_s;
    pvbusdecoder.pvbus_m_range[0x0..0xfffff] => ramdevice.pvbus;
    }
}
```

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 2-7: Set your target
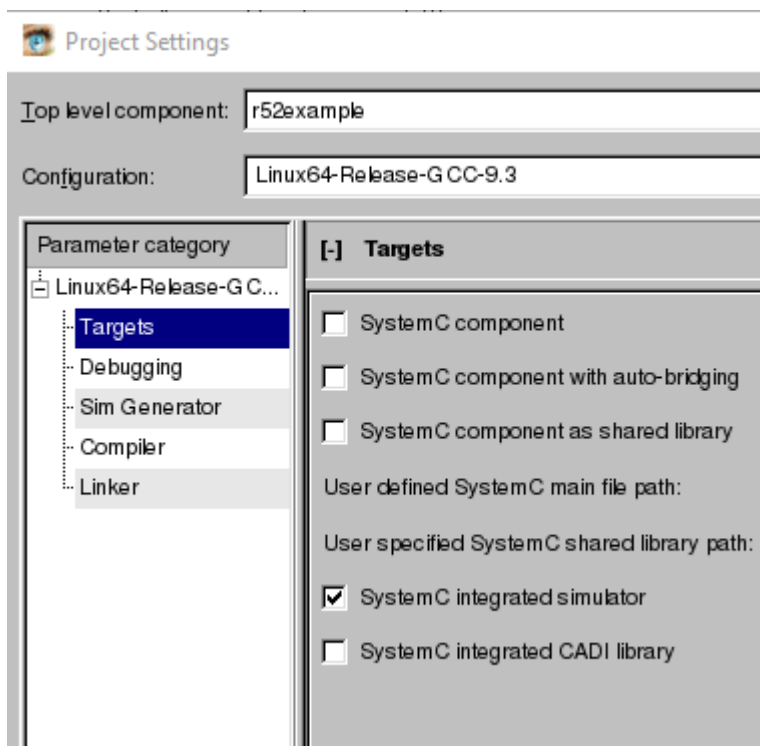
8. Build the model by clicking the **Build** button. If the build is successful, the log view should display the text Model Build process completed successfully. The model is called isim_system and is created in a subdirectory named after the build configuration (for example `<build_dir>/Linux64-Release-GCC-9.3/isim_system`).

Figure 2-8: Build the model
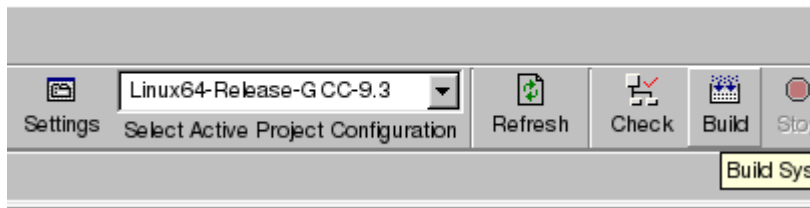
Figure 2-9: Successful build
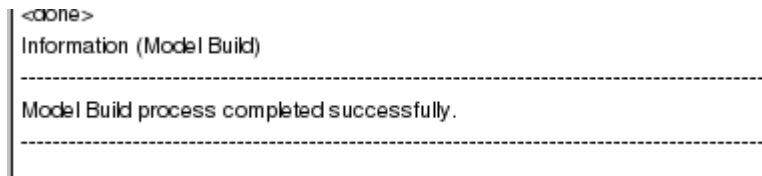
2.2 Code to run on the simple model

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 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`.

2.3 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
```

2.4 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

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

```
./Linux64-Release-GCC-9.3/isim_system -a image.axf
```

*   In Microsoft Windows, if you are using Visual Studio 2019, 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.
```

3. Constructing a dual core Fast Model

In this tutorial, we will build a dual core model containing an ARMCortexA57x2 and write and then run software on it.

3.1 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 SGCanvas.

- 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 | The instructions here are compact instructions for constructing a model using System Canvas. Please refer to the Fast Models 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 on 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

   - ARMCortexA57x2

   - RAMDevice

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

   - Connect MasterClock `clk_out` to ARMCortexA57x2 `clk_in`

   - Connect ARMCortexA57x2 `pvbus_m0` to RAMDevice `pvbus`

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();
        armcortexa57x2ct : ARMCortexA57x2CT();
    }
    connection
    {
    masterclock.clk_out => armcortexa57x2ct.clk_in;
    armcortexa57x2ct.pvbus_m0 => ramdevice.pvbus;
    }
}
```

6. Click the **Settings** button and select the configuration and set the Target to be a `SystemC Integrated Simulator`.

7. 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`.

3.2 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_print

    // 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`.

3.3 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
```

3.4 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

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

```
./Linux64-Release-GCC-9.3/isim_system -a image.axf
```

- In Microsoft Windows, if you are using Visual Studio 2019, 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.

4. 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.

4.1 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 either the Component Architecture Debug Interface (CADI) or 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.

4.2 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 4-1: New Debug Connection button
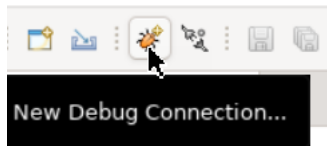
3. In the **New Debug Connection** dialog box, select **Model Connection**, then click **Next**:

   Figure 4-2: New Debug Connection dialog
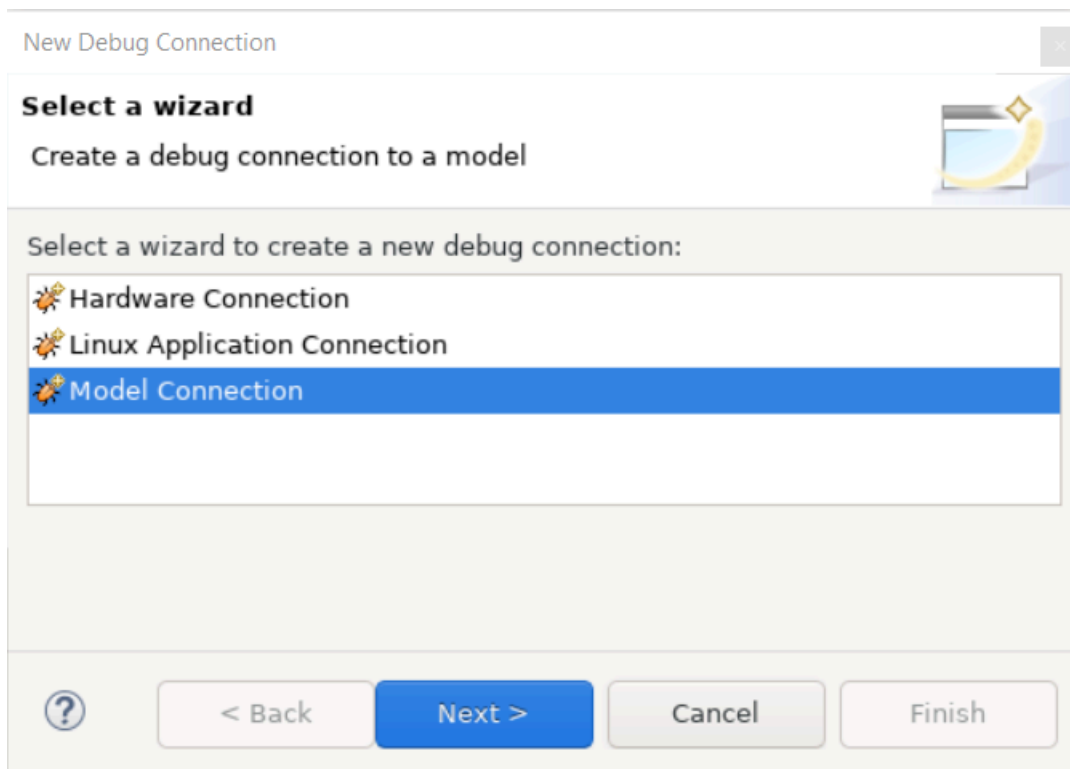
4. In the **Debug Connection** dialog box, give the debug connection a name, for example **a57example**, then click **Next**.

5. In the **Target Selection** dialog box, click **Add a new model...**:

Figure 4-3: Model Connection dialog
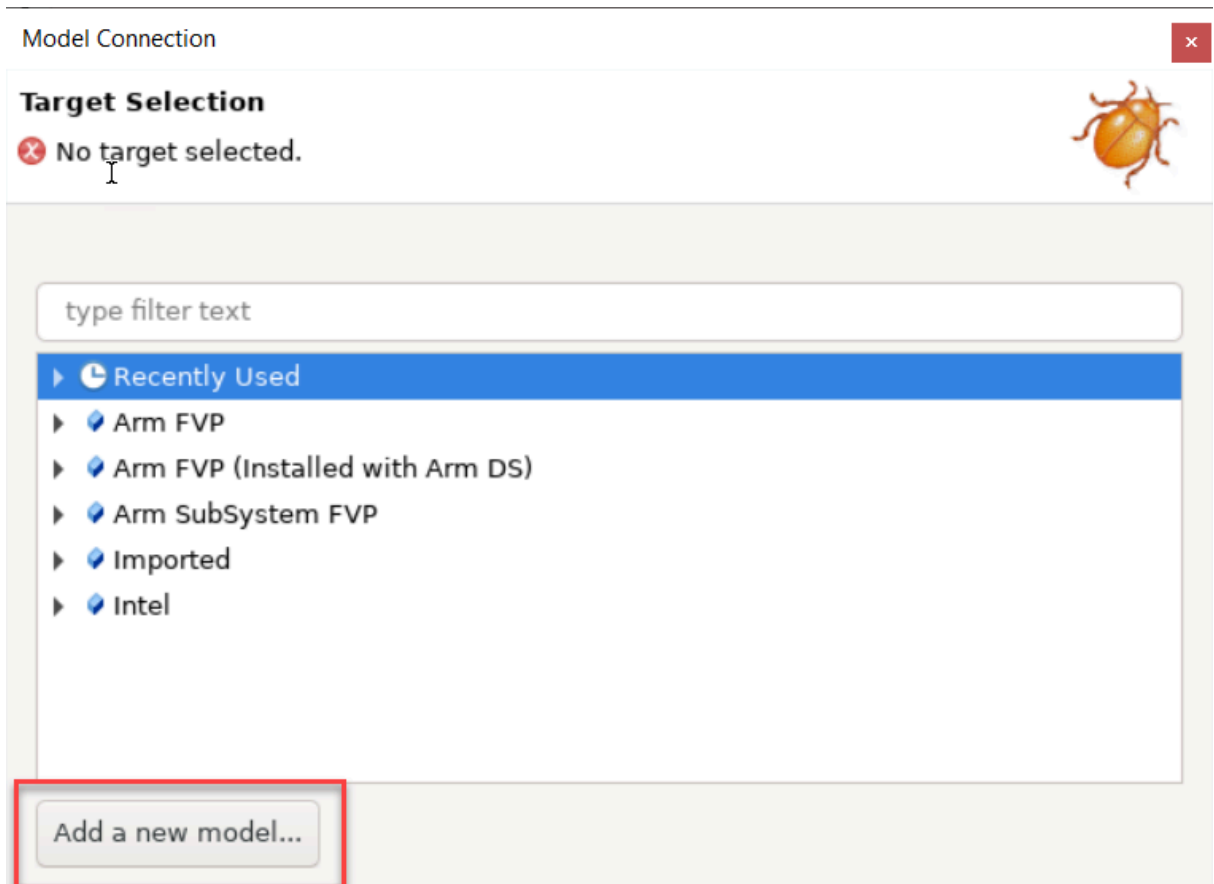
6. In the **Select Model Interface** dialog box, select the debug interface to use to connect to the model. You can select either CADI or Iris for this tutorial because all Fast Models support both interfaces, but we recommend Iris. It provides the debug functionality that is available in CADI but also provides event tracing and better support for remote connections. Also, CADI is deprecated and will be removed in a future release. Click **Next**.

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

Figure 4-4: New Model dialog
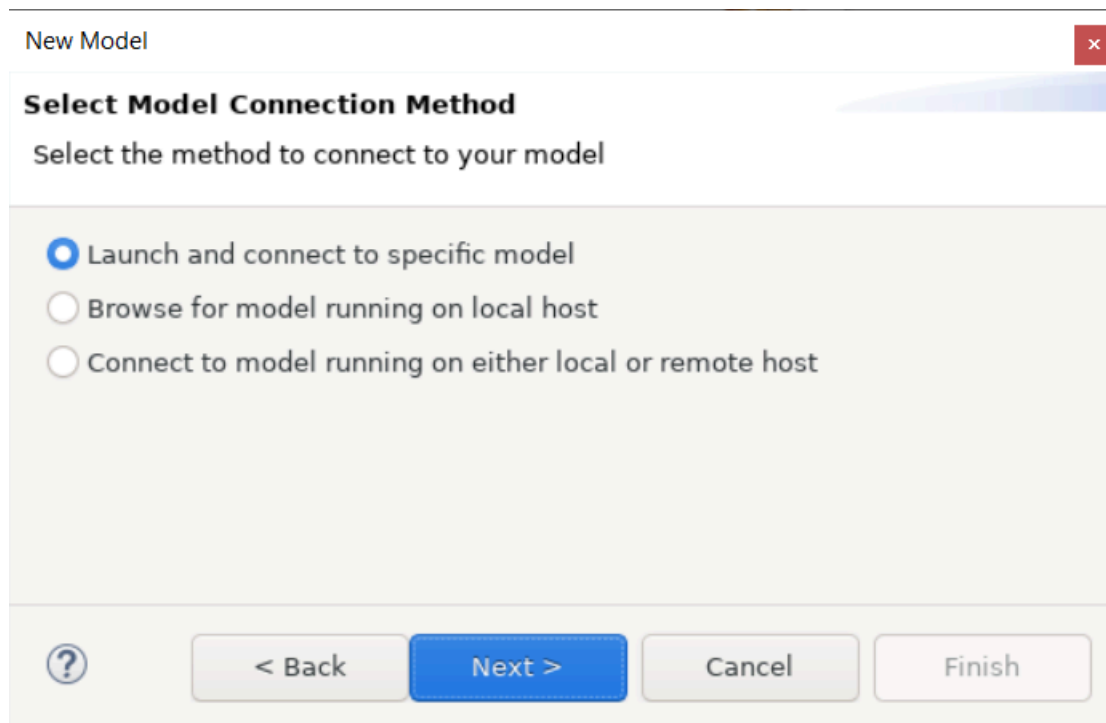
Note that you can only select the other options in this dialog box if you had previously launched the FVP on a local or a remote host using the `--iris-connect` option.

Click **Next**.

8. 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 4-5: Model Selection from File System
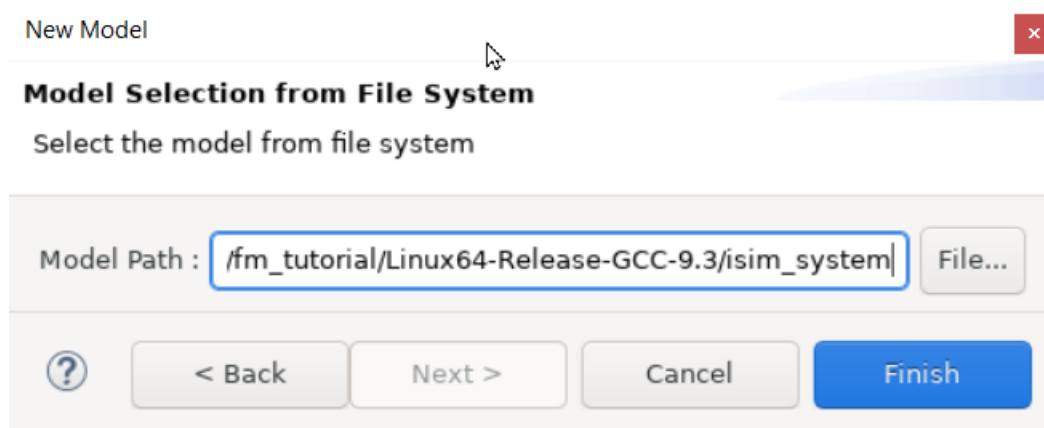
Arm Development Studio imports the FVP so that it appears in the list of available model connections.

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

4.3 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

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

    Figure 4-6: Edit configuration and launch
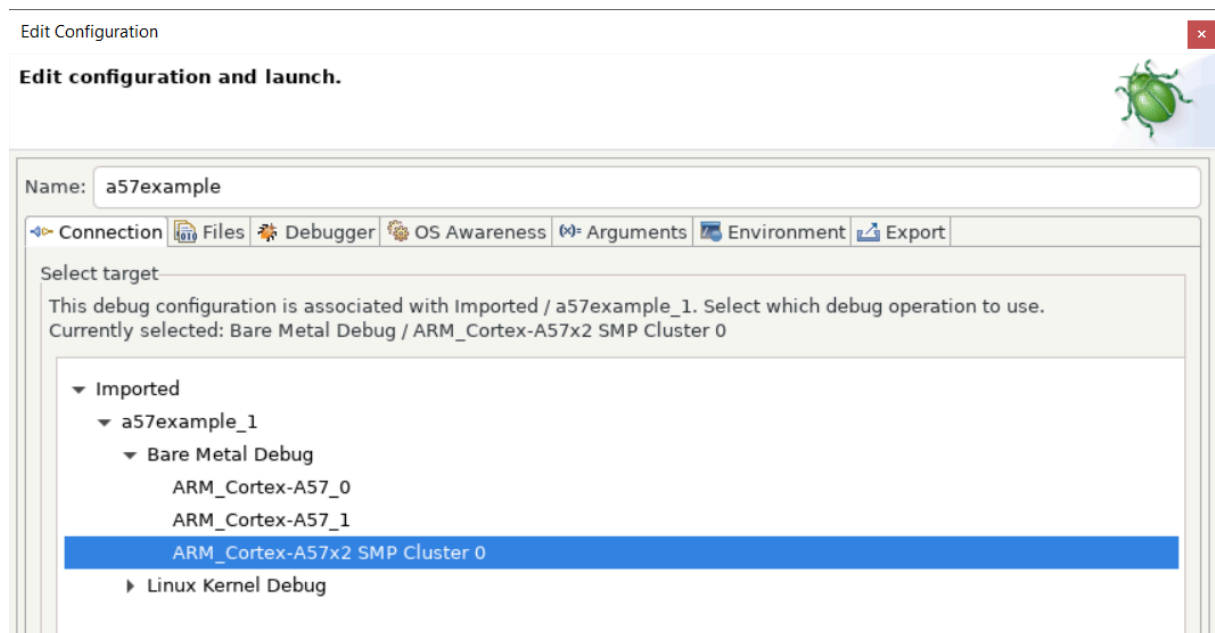
2. In the **Files** tab, navigate to the application image file from the Dual Core tutorial, `image.axf`. Arm Development Studio will download this file from the host and run it on the FVP.
3. In the Debugger tab, select the **Debug from entry point** radio button, then click **Debug**. The Iris server starts running, the FVP is launched, and the debugger connects to the FVP. Execution stops at the image entry point.
4. Click **Continue** in the **Debug Control**:

    Figure 4-7: Continue button
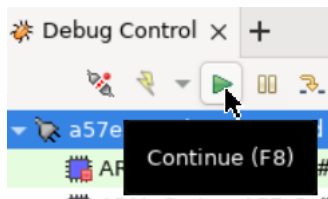
Results

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

Figure 4-8: Target Console output
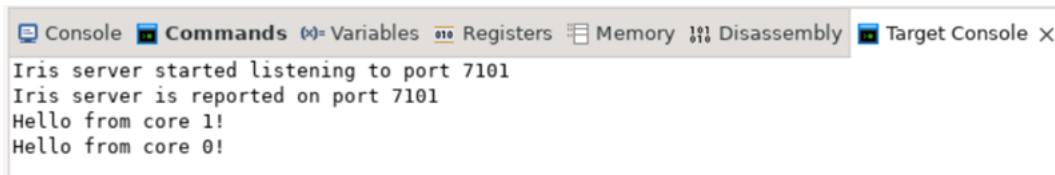
4.4 Step through the application

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 4-9: Connect to Target button
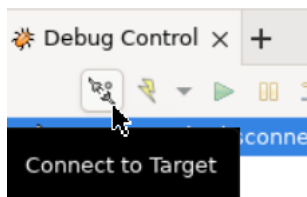
3. Arm Development Studio now displays the source file `startup.s` in the Editor view, highlighting in green the line at which execution stopped:

Figure 4-10: Editor view

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

Figure 4-11: Step instruction button
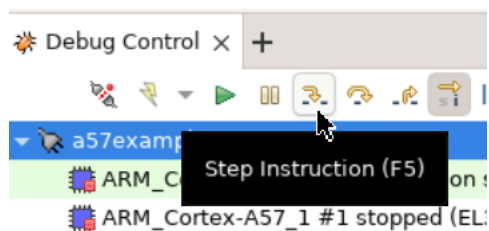
4.5 View the contents of memory

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.

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

Figure 4-12: Adding the Characters column
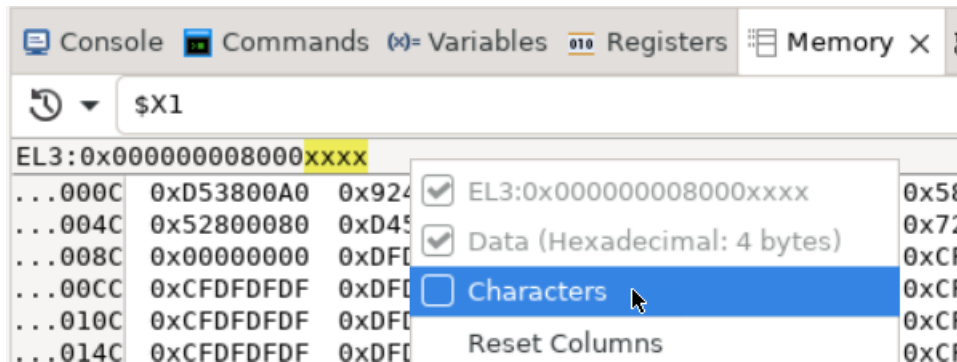
4. Add a breakpoint to the line following the label `secondary` in the source code by double clicking in the column containing the line number:

Figure 4-13: Add a breakpoint
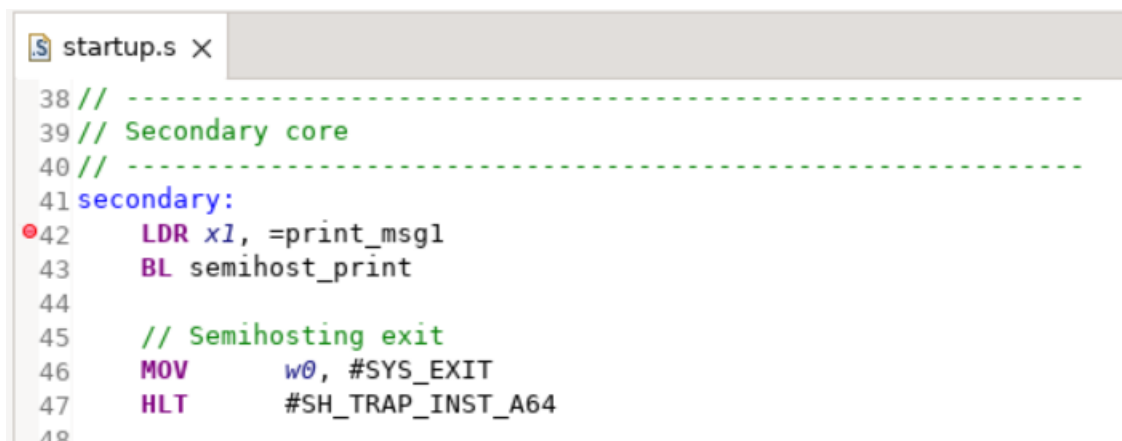
5. Press **F5** to step a single instruction. The **Characters** column now shows the contents of the address in `x1`.

6. Highlight the string `Hello from core 1!` and the memory view highlights the hexadecimal values of the ASCII characters stored in memory:

Figure 4-14: Memory view
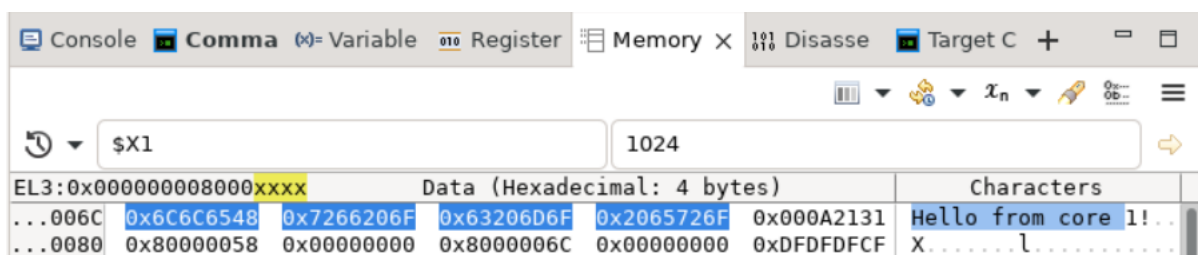
4.6 Useful information

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

- Using FVPs with Arm Development Studio.

- Tutorial: Using FVPs.

- Connect to new or custom models.

5. 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.

5.1 Introduction to ToggleMTIPlugin

Tracing an entire simulation 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 that start and stop trace at a specific instruction count, or that 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 version 11.20 or later. ToggleMTIPlugin is not included in earlier versions.

- A compiler supported by Fast Models. See Requirements for Fast Models in *Fast Models User Guide* for a list of supported operating systems and compilers.

- An Arm account that enables you to log into Product Download Hub to download Fast Models products.

5.2 Installation and setup

To demonstrate ToggleMTIPlugin we first need a Fixed Virtual Platform (FVP) and an image to run on it.

Procedure

1. Download and install the Third Party Add-ons for Fast Models package for your Fast Models version and host platform, from Product Download Hub. This step creates the directory `$PVLIB_HOME/images/`, containing various sample images that you can run on an FVP. We will use one of these images to demonstrate ToggleMTIPlugin.

2. Build one of the Fast Models example FVPs to run the image on. The following command builds the FVP_Base_Cortex-A57x1 example, creating an FVP called isim_system:

```
cd $PVLIB_HOME/examples/LISA/FVP_Base/Build_Cortex-A57x1
simgen -b -p FVP_Base_Cortex-A57x1.sgproj --configuration Linux64-Release-GCC-9.3
```

3. From the generated `Linux64-Release-GCC-9.3` directory, run one of the images installed by the Third Party Add-ons package on `isim_system`, and also load the TarmacTrace plug-in:

```
./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.

4. 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 toggle TarmacTrace at specific points in the code.

5.3 How to use ToggleMTIPlugin

The 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 runtime parameter `TRACE.ToggleMTIPlugin.disable_mti_runtime` that you set using a CADI or Iris-enabled 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.

5.4 Toggle trace using a debugger

To demonstrate this method, we will use Model Debugger, which is a CADI-enabled debugger that is supplied in the Fast Models package and also in some FVP packages.

Procedure

1. Start Model Debugger, for example by typing `modeldebugger` in the terminal on Linux, assuming the executable is on your path, or using the Start Menu on Windows.

2. Run the model 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. ToggleMTIPlugin.so 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.

-S

     Starts the CADI debug server, which enables the debugger to connect to the model.

The full command is:

```
./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 \
--plugin $PVLIB_HOME/plugins/Linux64_GCC-9.3/ToggleMTIPlugin.so \
-C TRACE.ToggleMTIPlugin.use_hlt=0 \
-C TRACE.ToggleMTIPlugin.disable_mti_from_start=1 \
-S
```

3. In Model Debugger, select **File > Connect to model…**, then select FVP_Base_Cortex-A57x1.

4. In the **Select Targets** dialog, the `cluster0.cpu0` checkbox is selected by default. Also select ToggleMTIPlugin, then click **OK**.
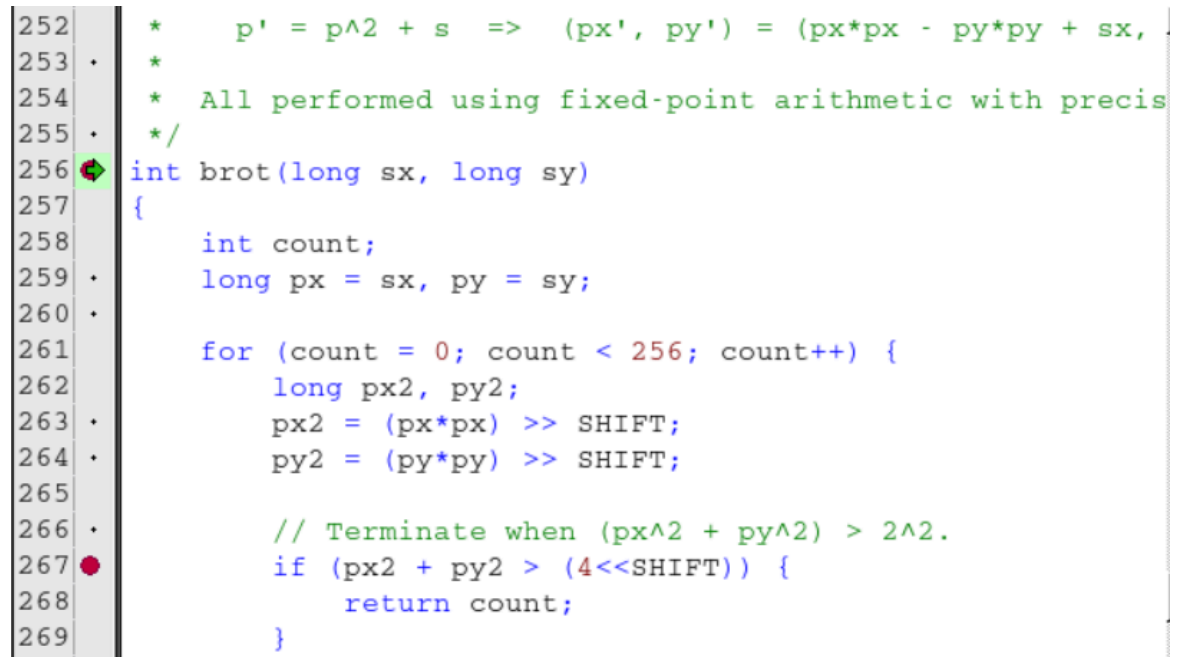
Figure 5-1: Select targets
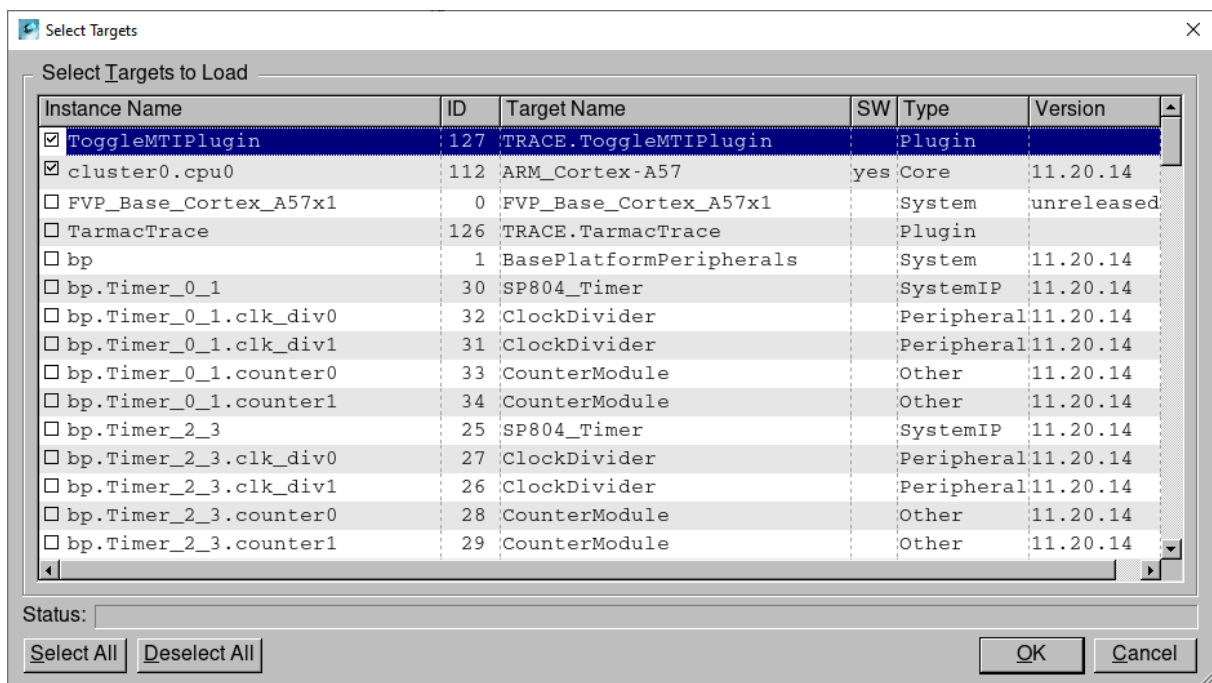
5. In the Model Debugger window for the `cluster0.cpu0` target, set breakpoints on the lines in the source code where you want tracing to start and stop:

   a. Select **File > Open Source…**.

   b. Select `$PVLIB_HOME/images/brot.c`.

c. Set two breakpoints in `brot.c` by double clicking in the left hand margin, for example, in the `brot()` function:

Figure 5-2: Set a breakpoint

```
252      *     p' = p^2 + s  =>  (px', py') = (px*px - py*py + sx,
253  ·   *
254      *  All performed using fixed-point arithmetic with precis
255  ·   */
256 ◈ int brot(long sx, long sy)
257     {
258         int count;
259  ·      long px = sx, py = sy;
260  ·
261         for (count = 0; count < 256; count++) {
262             long px2, py2;
263  ·          px2 = (px*px) >> SHIFT;
264  ·          py2 = (py*py) >> SHIFT;
265
266  ·          // Terminate when (px^2 + py^2) > 2^2.
267 ●          if (px2 + py2 > (4<<SHIFT)) {
268                 return count;
269             }
```

6. Click **Run**. The simulation starts running, and stops at the first breakpoint. Because we disabled trace from the start of the simulation, the terminal from which we ran the command to launch Model Debugger does not yet show any trace output.

7. Switch focus to the Model Debugger window for the ToggleMTIPlugin target, then select **Debug > Set Parameters**.

8. Click the **List View** tab, then clear the `disable_mti_runtime` checkbox, then click **OK**:

Figure 5-3: Clear disable_mti_runtime parameter

Parameter name	Value	Type	Default	Mode	Description
☐ diagnostics	0	bool	0		Print diagnostics
☑ disable_mti_from_start	1	bool	0		Enable/Disable MTI callbacks from start of simulation
hlt_imm16	0xf000	int	0xf000		16-bit integer used in HLT instruction meant to be used b...
☐ use_hlt	0	bool	1		If true, use HLT #imm16 instruction to toggle MTI behaviour
☐ disable_mti_runtime	0	bool	0	runtime	Enable/Disable MTI callbacks at run-time

Category View \ List View \ Tree View

Status:

☑ Hexadecimal display for numeric values   OK   Cancel

> **Note** The other parameters in this dialog are fixed as they can only be set at initialization time.

9. Switch focus back to the Model Debugger window for `cluster0.cpu0` and click **Run**.

10. Execution stops at the second breakpoint. Now the terminal shows some TarmacTrace output:

Figure 5-4: Trace output

```
In process: FVP_Base_Cortex_A57x1.thread_p_6 @ 36900020 ns
36900020000 ps R cpsr 600003cd
36904270000 ps IT (3690427) 80000270 9b027c44 O EL3h_s : MUL      x4,x2,x2
36904270000 ps R X4 0000000000000000
36904280000 ps IT (3690428) 80000274 934dfc84 O EL3h_s : ASR      x4,x4,#13
36904280000 ps R X4 0000000000000000
36904290000 ps IT (3690429) 80000278 8b040067 O EL3h_s : ADD      x7,x3,x4
36904290000 ps R X7 0000000000012000
36904300000 ps IT (3690430) 8000027c f14020ff O EL3h_s : CMP      x7,#8,LSL #12
36904300000 ps R cpsr 200003cd
36904310000 ps IS (3690431) 80000280 5400006d O EL3h_s : B.LE     {pc}+0xc ; 0x8000028c
36904320000 ps IT (3690432) 80000284 2a0503e0 O EL3h_s : MOV      w0,w5
36904320000 ps R X0 0000000000000000
5 clk CADI E simulation_stopped
```

11. To turn trace off again, select the `disable_mti_runtime` checkbox in the Model Debugger window for ToggleMTIPlugin.

12. Click **Run** again to resume the simulation.

5.5 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 armcortexa57x2ct.cpu0.enable_trace_special_hlt_imm16=1

    Enables the use of parameter trace_special_hlt_imm16.

-C armcortexa57x2ct.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 armcortexa57x2ct.cpu0.enable_trace_special_hlt_imm16=1 \
-C armcortexa57x2ct.cpu0.trace_special_hlt_imm16=5 \
-C armcortexa57x2ct.cpu1.enable_trace_special_hlt_imm16=1 \
-C armcortexa57x2ct.cpu1.trace_special_hlt_imm16=5
```

Results

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

Figure 5-5: Trace output

```
5000000000000 ps cpu1 IT (5) 80000010 d53800a0 O EL3h_s : MRS      x0,MPIDR_EL1
5000000000000 ps cpu1 R X0 0000000080000001
6000000000000 ps cpu1 IT (6) 80000014 92401c00 O EL3h_s : AND      x0,x0,#0xff
6000000000000 ps cpu1 R X0 0000000000000001
7000000000000 ps cpu1 IT (7) 80000018 d44000a0 O EL3h_s : HLT      #5
Hello from core 1!
7000000000000 ps cpu0 R cpsr 000003cd
5000000000000 ps cpu0 IT (5) 80000010 d53800a0 O EL3h_s : MRS      x0,MPIDR_EL1
5000000000000 ps cpu0 R X0 0000000080000000
6000000000000 ps cpu0 IT (6) 80000014 92401c00 O EL3h_s : AND      x0,x0,#0xff
6000000000000 ps cpu0 R X0 0000000000000000
7000000000000 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.
```

Related information

Constructing a Dual Core Fast Model on page 18

6. 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, and introduce an example component and platform which demonstrate this behavior.

6.1 What is MTI?

MTI stands for Model Trace Interface and 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 tend to be extern components and the mechanism of exposing these traces is not visible.

Exposing and generating trace information by a component is just half of the MTI story, this trace data needs to be consumed, which typically is done by an MTI plugin.

6.2 Setting up the trace source in the LISA component

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

Procedure

1. We need to define a pointer of type `sg::EventSource` in the resources section and create the actual 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 three fields, two integers and a string.

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

   behaviour init()
   {
       composition.init();

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

2. Now the `sg::EventSource` object is created, 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 information about each field. This includes the field's name, a description, type, and size. You must define them in the order of the template parameters of the trace object. In our example, int int string.

```
/* provide field information

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

6.3 Generating trace data in the LISA component

With the trace source configured and registering done 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);
```

6.4 The example, what is in it, and what does it do?

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

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

Figure 6-1: Platform seen in System Canvas
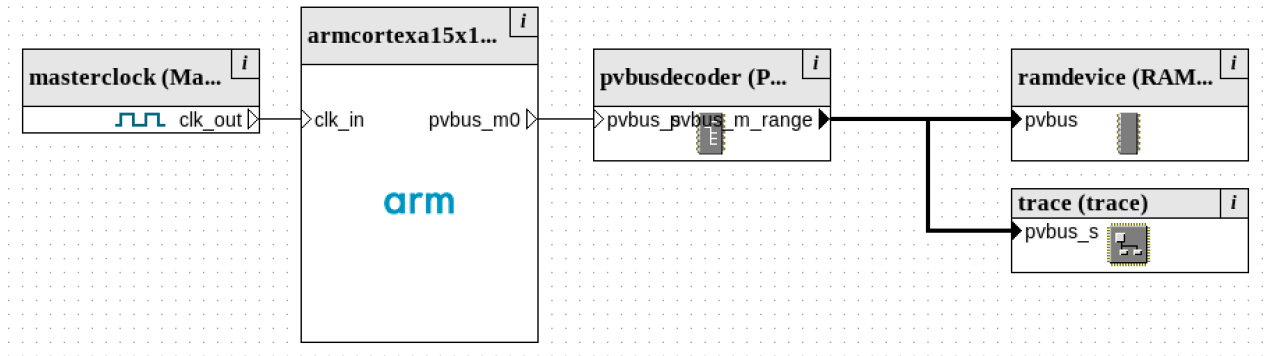
In this platform we have:

- MasterClock

- Core

- PVBusDecoder

- RAMDevice

- trace component

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

- The address and data to be written are extracted from the `pv::WriteTransaction` object.

- A string is created containing the extracted data.

- All three variables are used when triggering the trace source using the `fire()` function.

```
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();
}
```

6.5 Using the 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-A15x1
simgen -b -p GeneratingTraceFromLISA.sgproj --configuration Linux64-Release-GCC-9.3
```

The elf image to load onto the core is already built, 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 be able to capture the generated trace, we are going to 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 0xcafebeef to trace peripheral (0x100000000)
 LISA_TRACE_WRITE: Address=0x0000000010000000 Value=0xcafebeef String="Addres=
 0x0000000010000000    data=0xcafebeef"

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.

7. 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.

7.1 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.

7.2 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 behaviour, `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 occured, therefore calling `signal_out.setValue()` is gated by the `init_complete` flag. This flag is set in the `reset()` behaviour 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: 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);
```

7.3 Using the parameter to change the signal

`SignalDriver` exposes a parameter to change the signals.

The parameter has the following properties:

`type(bool)`

Setting it to `True` or `1` drives the signal `Set`, setting it to `False` or `0` drives 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()` behaviour 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()` behaviour 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;
}
```

7.4 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;
```

> **Note**
>
> If the debugger shows 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()` behaviour 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()` behaviour 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;
}
```

7.5 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 currently `set`, and 0 returned if the signal is `clear`

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

Note

The `write()` behaviour 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()` behaviour 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();
  }
}
```
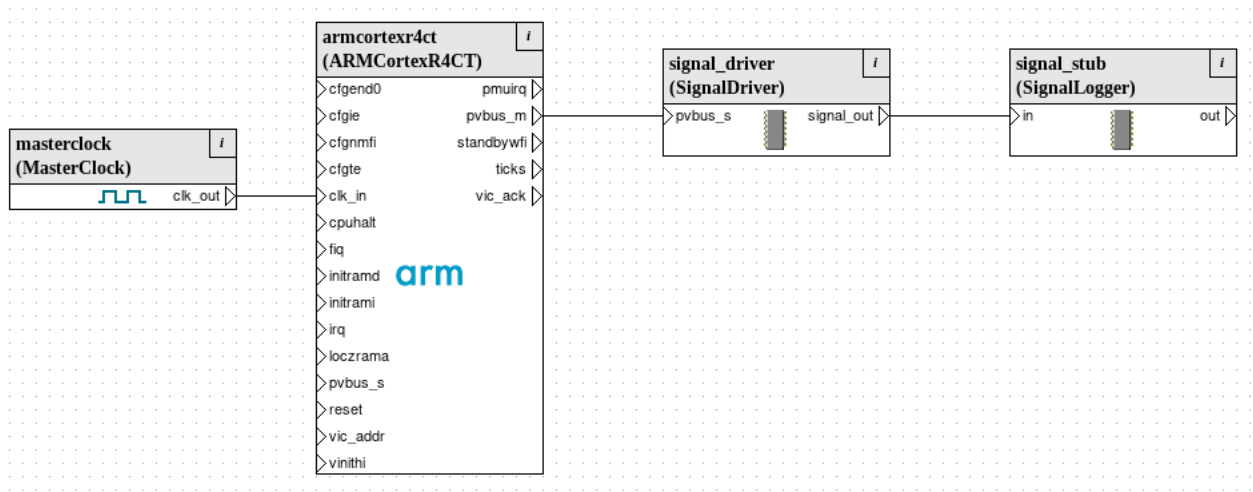
7.6 SignalDriver example

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

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

Figure 7-1: Platform seen in sgcanvas

In this platform we have:

- MasterClock

- Core

- SignalDriver

- SignalLogger

In the example the `SignalLogger` component is being 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 the System Canvas or run `simgen` directly:

```
cd $PVLIB_HOME/examples/LISAPlus/SignalDriver
simgen -b -p SignalDriver.sgproj --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 Model Debugger.

2. The next step is to run the example. To be able to capture the generated trace, we are going to use the `GenericTrace` MTI plugin:

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

3. With the model started and the CADI server enabled, we need to connect Model Debugger to the model. Connect to both the core and the `SignalDriver` component:

Figure 7-2: Model Debugger connect dialog
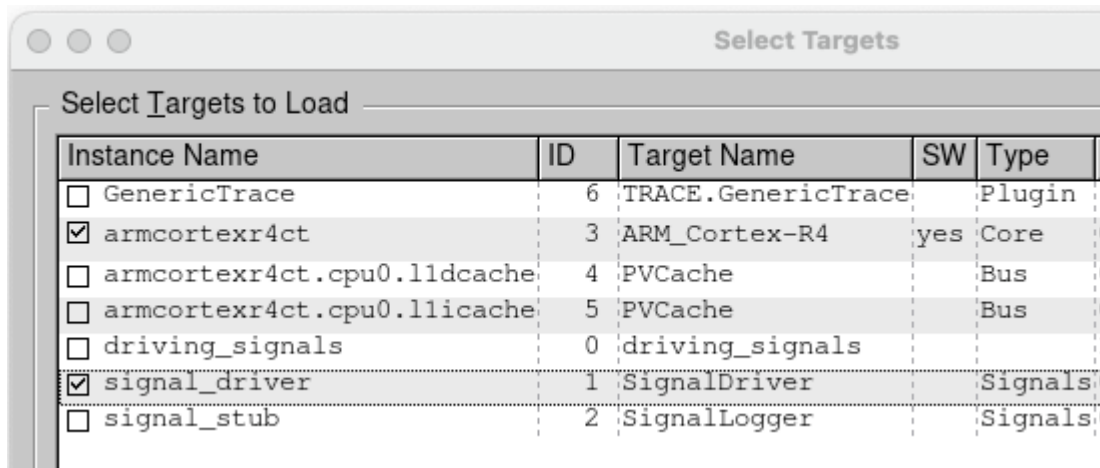
4. Viewing the `signalDriver` component, the current state of the signal can be seen in the register view. The default setting is for the signal to be driven `sg::Signal::Clear` and therefore the register shows this as `false`:

Figure 7-3: Model Debugger register view
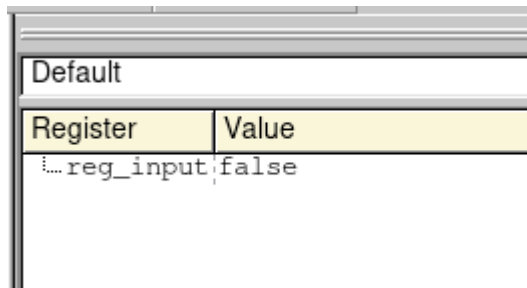
5. Now we will change the register value and observe the signal changes. Set the register value to `true` or `1`:

Figure 7-4: Model Debugger register view
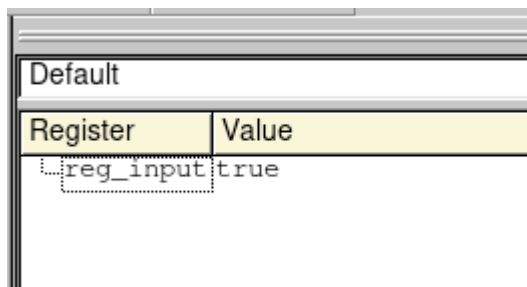

On the terminal we can see that:

a. `SignalDriver` is driving the signal set and this change occurred due to changing the register.

b. The `signalLogger` stub component confirmed the new value of the signal:

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

6. Similar behaviour can be seen when using the parameter. Viewing the `signalDriver`'s parameter, we can see that it is set, matching the current state of the signal:

Figure 7-5: Model Debugger parameter view
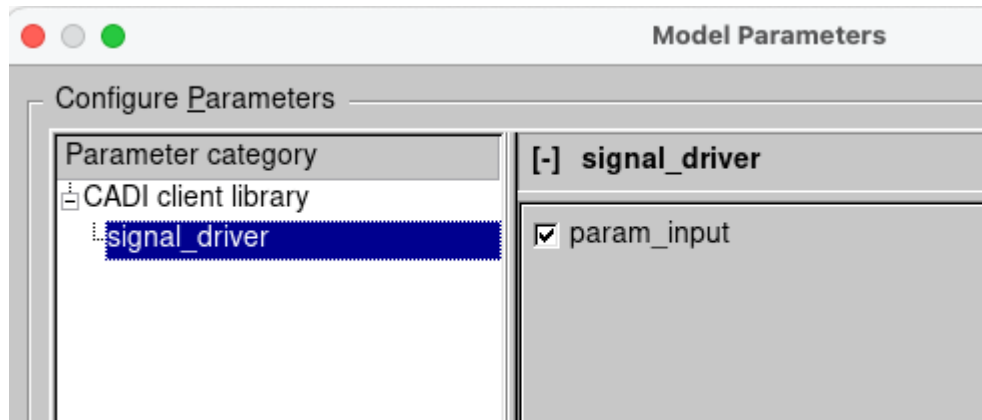
Clear the checkbox to see this reflected in the traces:

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

7. Lastly, we can test the bus behaviour by using the Model Debugger window connected to the core. From here we can write `1` to any address:

Figure 7-6: Model Debugger memory view

Writing to any 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
```

> **Note**
>
> `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 will return the same value.