Copyright Notice
Copyright © 2020 Imperas Software Limited All rights reserved. This software and
documentation contain information that is the property of Imperas Software Limited. The
software and documentation are furnished under a license agreement and may be used or
copied only in accordance with the terms of the license agreement. No part of the
software and documentation may be reproduced, transmitted, or translated, in any form or
by any means, electronic, mechanical, manual, optical, or otherwise, without prior written
permission of Imperas Software Limited, or as expressly provided by the license
agreement.

Right to Copy Documentation
The license agreement with Imperas permits licensee to make copies of the
documentation for its internal use only. Each copy shall include all copyrights,
trademarks, service marks, and proprietary rights notices, if any.

Destination Control Statement
All technical data contained in this publication is subject to the export control laws of the
United States of America. Disclosure to nationals of other countries contrary to United
States law is prohibited. It is the reader’s responsibility to determine the applicable
regulations and to comply with them.

Disclaimer
IMPERAS SOFTWARE LIMITED., AND ITS LICENSORS MAKE NO WARRANTY
OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
## Table of Contents

1 Preface ...................................................................................................................... 7  
  1.1 Notation ............................................................................................................... 7  
  1.2 Related Documentation ....................................................................................... 7  
  1.3 Glossary / Terminology ...................................................................................... 7  

2 Introduction ................................................................................................................. 9  
  2.1 Prerequisites ........................................................................................................ 9  
  2.2 Obtaining & installing the OP API ..................................................................... 9  
  2.3 Compiling Examples described in this Document .............................................. 9  
  2.4 Shared Objects and executables ........................................................................ 10  
  2.5 Platforms, Modules and Shared Objects ........................................................... 10  

3 Simulating Modules using harness.exe ..................................................................... 11  

4 Imperas Simulation Overview .................................................................................. 12  
  4.1 Simulation Environments .................................................................................. 12  
  4.2 What are OVPsim and CpuManager? ............................................................... 12  
  4.3 Use of OP with Imperas tools ........................................................................... 13  
  4.4 C API and iGen ................................................................................................. 13  

5 Separation of iGen modules and C harness / testbench ............................................14  

6 Documentation on the OP API ................................................................................. 15  

7 Introducing a C harness / testbench .......................................................................... 16  
  7.1 Writing a harness using C++ ............................................................................ 18  

8 The top level or root module ..................................................................................... 21  

9 Configuring the Simulation using parameters .......................................................... 22  
  9.1 Parameter Lists on a Root Module.................................................................... 22  
  9.1.1 The available root module parameters ...................................................... 25  
  9.2 Using Overrides to Set parameter values on module and component instances 25  
  9.3 Configuring Modules and Components from Harness ..................................... 26  
  9.3.1 Module Hardware Parameters ................................................................... 26  
  9.3.2 Component Feature Parameters ................................................................. 27  
  9.3.3 Setting Parameters in the Harness ............................................................... 27  
  9.3.3.1 Parameter List ....................................................................................... 27  
  9.3.3.2 OP_PARAMS Macro ............................................................................ 28  
  9.3.4 Running Example ...................................................................................... 29  
  9.4 Special parameter values ................................................................................... 31  
  9.4.1 MIPS parameter ........................................................................................ 31  
  9.4.2 Endian parameter ...................................................................................... 31  
  9.5 Common error messages related to parameters ................................................ 31  

10 Controlling the simulation using the Command Line............................................... 33  
  10.1 standardCommandLineParser example ............................................................ 33  
  10.2 Verbose Output ................................................................................................. 34  
  10.2.1 Simulation Time Statistics ........................................................................ 34  
  10.3 Passing arguments into application programs ................................................... 35  
  10.4 Use of debugging options ............................................................................... 36  
  10.5 Adding harness specific command line options .............................................. 37
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.5.1</td>
<td>Argument Conflicts</td>
<td>39</td>
</tr>
<tr>
<td>10.6</td>
<td>Other useful command line options</td>
<td>39</td>
</tr>
<tr>
<td>11</td>
<td>Writing your own harness.exe</td>
<td>42</td>
</tr>
<tr>
<td>12</td>
<td>Monitoring / Tracing during simulation</td>
<td>47</td>
</tr>
<tr>
<td>12.1</td>
<td>Processor Model Tracing Operations</td>
<td>47</td>
</tr>
<tr>
<td>12.1.1</td>
<td>Tracing control on the command line</td>
<td>47</td>
</tr>
<tr>
<td>12.1.2</td>
<td>Tracing control with CPU model parameters</td>
<td>48</td>
</tr>
<tr>
<td>12.1.3</td>
<td>Tracing control with OP functions</td>
<td>49</td>
</tr>
<tr>
<td>12.2</td>
<td>Processor Instruction Tracing Example</td>
<td>49</td>
</tr>
<tr>
<td>12.3</td>
<td>Peripheral model parameters</td>
<td>51</td>
</tr>
<tr>
<td>12.3.1</td>
<td>Obtaining parameter lists (using --showoverrides)</td>
<td>51</td>
</tr>
<tr>
<td>12.3.2</td>
<td>Configuration</td>
<td>52</td>
</tr>
<tr>
<td>12.3.3</td>
<td>Diagnostics</td>
<td>54</td>
</tr>
<tr>
<td>12.3.3.1</td>
<td>On a single Instance</td>
<td>54</td>
</tr>
<tr>
<td>12.3.3.2</td>
<td>On all instances in a design</td>
<td>55</td>
</tr>
<tr>
<td>12.4</td>
<td>Adding your own monitors in a harness</td>
<td>55</td>
</tr>
<tr>
<td>12.4.1</td>
<td>Monitoring nets with callbacks</td>
<td>56</td>
</tr>
<tr>
<td>12.4.1.1</td>
<td>Finding nets and declaring net callbacks</td>
<td>56</td>
</tr>
<tr>
<td>12.4.1.2</td>
<td>Simulating with net callbacks</td>
<td>57</td>
</tr>
<tr>
<td>12.4.1.3</td>
<td>monitoringNets/harness/harness.c full listing</td>
<td>58</td>
</tr>
<tr>
<td>12.4.2</td>
<td>Monitoring Bus / Memory / Processor Address accesses using Callbacks</td>
<td>59</td>
</tr>
<tr>
<td>12.4.2.1</td>
<td>Find buses and memories and attach callbacks</td>
<td>59</td>
</tr>
<tr>
<td>12.4.2.2</td>
<td>Running the example</td>
<td>60</td>
</tr>
<tr>
<td>12.4.3</td>
<td>Adding a command to show module contents</td>
<td>61</td>
</tr>
<tr>
<td>12.5</td>
<td>Writing a Monitor module</td>
<td>62</td>
</tr>
<tr>
<td>13</td>
<td>Simulating Processor Exceptions</td>
<td>63</td>
</tr>
<tr>
<td>13.1</td>
<td>Example of Simulating an Unaligned Access Exception</td>
<td>63</td>
</tr>
<tr>
<td>13.2</td>
<td>Enabling simulation of exceptions with overrides on command line</td>
<td>64</td>
</tr>
<tr>
<td>13.3</td>
<td>Enabling simulation of exceptions in modules</td>
<td>65</td>
</tr>
<tr>
<td>13.4</td>
<td>Enabling simulation of exceptions in a C harness</td>
<td>65</td>
</tr>
<tr>
<td>14</td>
<td>Semihosting and Intercept Libraries</td>
<td>66</td>
</tr>
<tr>
<td>14.1</td>
<td>Selecting semihost libraries</td>
<td>66</td>
</tr>
<tr>
<td>14.1.1</td>
<td>Specifying a semihost library in the module.op.tcl</td>
<td>66</td>
</tr>
<tr>
<td>14.1.2</td>
<td>An assembler test program</td>
<td>66</td>
</tr>
<tr>
<td>14.1.3</td>
<td>Adding imperasExit semihost library to harness.c</td>
<td>67</td>
</tr>
<tr>
<td>15</td>
<td>Imperas built-in Application Intercepts</td>
<td>68</td>
</tr>
<tr>
<td>16</td>
<td>Simulator Control Files</td>
<td>70</td>
</tr>
<tr>
<td>16.1</td>
<td>Specify use of Control File on the simulator command line</td>
<td>70</td>
</tr>
<tr>
<td>16.2</td>
<td>Specify use of Control File using IMPERAS_TOOLS</td>
<td>70</td>
</tr>
<tr>
<td>16.3</td>
<td>Specify use of Control File in a C harness</td>
<td>71</td>
</tr>
<tr>
<td>17</td>
<td>Loading an Application Program file</td>
<td>72</td>
</tr>
<tr>
<td>17.1</td>
<td>Load Address Support</td>
<td>72</td>
</tr>
<tr>
<td>17.1.1</td>
<td>LMA vs VMA</td>
<td>73</td>
</tr>
<tr>
<td>17.1.2</td>
<td>Load Physical</td>
<td>73</td>
</tr>
<tr>
<td>17.2</td>
<td>Program Load Options</td>
<td>73</td>
</tr>
<tr>
<td>17.2.1</td>
<td>Set Start Address</td>
<td>73</td>
</tr>
</tbody>
</table>
17.2.2 Zero BSS Section................................................................. 73
17.3 Loading a Program onto Processors using the Command Line......... 74
  17.3.1 Loading onto single or multiple processors............................. 74
  17.3.2 Modification to Load Address.............................................. 74
  17.3.3 Specifying the start Address............................................. 75
  17.3.4 Provide Arguments to Applications....................................... 76
17.4 Common Program Memory.............................................................. 76
17.5 Loading an application from the harness........................................ 77
  17.5.1 Application Loader Controls............................................. 77
  17.5.2 Onto a Processor Instance............................................... 77
  17.5.3 Onto a Bus Instance..................................................... 77
  17.5.4 Onto a Memory Instance................................................ 78
17.6 Getting Information of Loaded Applications.................................... 78
17.7 Example ..................................................................................... 78
17.8 Loading Program Symbolic information.......................................... 81
  17.8.1 Command Line......................................................... 81
  17.8.2 In a Harness.............................................................. 81
17.9 Defining a hardware resident program.......................................... 81
17.10 Custom Application Loaders....................................................... 82
  17.10.1 Writing a custom reader................................................ 82
17.11 Custom Loader using Standard Object Reader................................. 82
18 Attaching a debugger ........................................................................ 84
  18.1 Introduction............................................................................. 84
  18.2 Debug with GDB ..................................................................... 84
  18.2.1 Starting GDB debug session on a single processor................... 85
    18.2.1.1 Command line.......................................................... 85
    18.2.1.2 Root module instance............................................... 85
  18.2.2 Selecting Processor(s) to debug with GDB.............................. 85
    18.2.2.1 Using command line options.................................. 85
    18.2.2.2 In a C harness...................................................... 85
  18.2.3 Configuring GDB............................................................... 86
  18.2.4 Using GDB with Graphical User Interfaces.............................. 87
  18.3 Debug with MPD ..................................................................... 87
    18.3.1 Configure MPD............................................................ 88
      18.3.1.1 Debug Peripheral model constructors......................... 88
      18.3.1.2 Add to source search path.................................. 88
    18.3.2 Using MPD with GUI.................................................. 88
    18.3.3 MPD Batch Mode....................................................... 89
  18.4 Manual Debug connection specifying port..................................... 89
    18.4.1 Do not wait for debug connection.................................. 90
  18.5 Using Control Files.................................................................. 90
  18.6 Example of Debugging Applications.......................................... 90
    18.6.1 Debug all processors.................................................. 92
      18.6.1.1 Using gdbconsole............................................... 92
      18.6.1.2 Using MPD with the Imperas Professional Simulator........ 93
  19 Imperas Graphical Debug Environments......................................... 95
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.1</td>
<td>Imperas iGui</td>
<td>95</td>
</tr>
<tr>
<td>19.1.1</td>
<td>Available Display Windows</td>
<td>95</td>
</tr>
<tr>
<td>19.1.2</td>
<td>Host Code</td>
<td>96</td>
</tr>
<tr>
<td>19.1.3</td>
<td>Initialization files</td>
<td>96</td>
</tr>
<tr>
<td>19.1.4</td>
<td>Limitations</td>
<td>97</td>
</tr>
<tr>
<td>19.2</td>
<td>Imperas eGui (eEclipse based)</td>
<td>97</td>
</tr>
<tr>
<td>20</td>
<td>Controlling Record and Replay of Virtualized Peripheral Input</td>
<td>99</td>
</tr>
<tr>
<td>20.1</td>
<td>Example for Record Replay operation</td>
<td>99</td>
</tr>
<tr>
<td>20.2</td>
<td>Simple CPU Memory and UART module</td>
<td>99</td>
</tr>
<tr>
<td>20.3</td>
<td>Application</td>
<td>100</td>
</tr>
<tr>
<td>20.4</td>
<td>Harness that configures UART and controls operation</td>
<td>100</td>
</tr>
<tr>
<td>20.5</td>
<td>Using instance parameters</td>
<td>102</td>
</tr>
<tr>
<td>20.6</td>
<td>From the command line</td>
<td>103</td>
</tr>
<tr>
<td>21</td>
<td>OP API Compatibility with deprecated ICM API</td>
<td>104</td>
</tr>
<tr>
<td>21.1</td>
<td>Interoperability</td>
<td>104</td>
</tr>
<tr>
<td>21.2</td>
<td>API tracing</td>
<td>104</td>
</tr>
<tr>
<td>22</td>
<td>ICM to OP API Function Conversion</td>
<td>105</td>
</tr>
</tbody>
</table>
1 Preface

The Imperas simulators can use models described in C or C++ and the models can be exported to be used in simulators and platforms using C, C++, SystemC or SystemC TLM2.

This document describes how the simulator is controlled, how virtual platforms are loaded and simulated, how testbenches and harnesses are written.

This document specifically describes how the OVP OP C API is used in C programs for use with Imperas and OVP virtual platform simulators and tools.

1.1 Notation

Code Text representing code, a command or output from iGen.

keyword A word with special meaning.

1.2 Related Documentation

There are several documents available as PDF:

Getting Started
- Imperas Installation and Getting Started Guide

Interface, API, and iGen related
- OVP Peripheral Modeling Guide
- OVPsim Using OVP Models in SystemC TLM2.0 Platforms
- iGen Model Generator Introduction
- iGen Platform and Module Creation User Guide
- Imperas Peripheral Generator Guide (using iGen)

Usage of Modules and Peripherals created using iGen
- Simulation Control of Platforms and Modules User Guide
- Advanced Simulation Control of Platforms and Modules User Guide

1.3 Glossary / Terminology

OP API - OVP Platforms API - C API used for creating and controlling virtual platforms. 2nd generation API, replaces ICM API. iGen creates modules/platforms in C using this API.

iGen - Imperas productivity tool that has a powerful script-based function API that is used to create C/C++/SystemC models and templates. Described in the iGen Model Generator Introduction, and for platforms, in the iGen Platform and Module Generator User Guide.
OVPsim - Simulator for Open Virtual Platforms that executes platforms and models coded in the OVP APIs

CpuManager - Imperas commercial simulator that fully implements the APIs defined by OVP (OVPsim implements a subset)

Platform / Module – a collection of components connected together into a level of hierarchy in a system to be simulated. This is a program in C/C++ making calls into OP API and normally compiled into a shared object/dynamically linked library and loaded by the simulator at run time.

Testbench / Harness – (used interchangeably) – A program in C/C++ making calls into the OP API to connect and control OVP components. It is normally linked to the simulator to provide an .exe binary that can be executed. Used to instance one or more platforms/modules and controls their execution. The main difference, from a platform/module, is that a testbench or harness includes a definition of the function main(), may include a command line parser and is linked to create an executable binary (.exe) file.

Root Module - used to describe the initial platform/module that instances one or more platforms/modules and controls their execution. Used in the testbench / harness.
2 Introduction

Imperas simulation technology enables very high-performance simulation, debug and analysis of platforms containing multiple processors and peripheral models. The technology is designed to be extensible: you can create your own platforms, new models of processors, and other platform components using interfaces and libraries supplied by Imperas. Platform models developed using this technology can be used both with Imperas simulation products and the freely-available OVPsim platform simulator.

Simulations are controlled by using the provided harness.exe program, or for more sophisticated control, and bespoke harness or testbench is written in C/C++ using the OVP OP API.

This document explains the usage of the OP API to write harnesses / testbenches. It explains the structure of modules and the different simulator phases, and how they are controlled.

2.1 Prerequisites

Since harnesses and testbenches for use with Imperas and OVP tools are written in C, an important prerequisite is that you must be proficient in the C language. If you want to use C++ then it is expected that you are proficient in the use of C++ and how it uses a C API.

2.2 Obtaining & installing the OP API

The OP API is part of all Imperas / OVP installations and thus you should already have it installed and be ready for use.

2.3 Compiling Examples described in this Document

The examples use modules, processors, component models and tool chains, available to download from the www.OVPworld.org website or as part of an Imperas installation.

The compilation of the examples utilizes the use of a Makefile, the instructions for which indicate the use of the command make, on Windows systems the MinGW mingw32-make command should be used in its place.

The Makefiles referred to in this document are written for GNU make. Standard Makefiles supplied by Imperas support compilation and linking using GNU tools on both Windows and Linux.

Example scripts will be referred to as (for example) example.sh, this being the extension used on Linux or for Windows MSYS shells. On Windows the script would be called example.bat

SystemC TLM2.0 models can be used on Linux with gcc or on Windows with MinGW/MSys (since SystemC release v2.3.0) or MSVC 8.0. It is assumed that users of
this environment will be familiar with SystemC, TLM2.0 and will have obtained this software from www.accellera.org or similar.

2.4 Shared Objects and executables
The shared objects referred to in this document are either Linux shared objects, with suffix `.so` or Windows dynamic link libraries with suffix `.dll`.

The executables referred to in this document are either Linux or Windows programs and have the suffix `.exe`

2.5 Platforms, Modules and Shared Objects
Modules are created by writing scripts in tcl using iGen API calls and then running iGen to create C code which calls functions from the OP API. A Makefile is provided that will take as input the `module.op.tcl` and run iGen, gcc etc. This will then create the `model.so/.dll` shared objects.

This `model.so/.dll` shared object can then be simulated using the `harness.exe` program provided, or by writing and compiling a bespoke test harness in C.

In this document we will either use binary modules and components from the Imperas provided library, or we will provide them as source in the example directories.
3 Simulating Modules using harness.exe

For many platforms and modules, the harness.exe program will be sufficient to load and execute a simulation and it is not necessary to write your own C testbench / harness.

The harness.exe program is provided in the binary directory which is found, after environment initialization, on the execution PATH. To invoke the program and show the command arguments which are available, type:

```
> harness.exe --help
```

The simplest mode of invocation is to load a module and an application for the processors in the module to execute:

```
> harness.exe --modulefile module/model.so \
   --program application/application.OR1K.elf
```

The argument --modulefile <filename> states which shared object module to load.

The argument --program <filename> states which application program binary to load and run on the simulated processor(s) in the module.

There are a number of examples in Examples/PlatformConstruction that show different modules being created and executed with harness.exe. Some of the examples instead create their own bespoke testbench / harness.
4 Imperas Simulation Overview

Before starting to create models for use with the Imperas simulation environment, you must understand how the components used in that environment interact. This section describes this in detail.

4.1 Simulation Environments

There are currently two simulation environments available that can be used with models and platforms that you create:

- **OVP** allows component models created using OVP modeling technology to be used in C harness, platform and testbench files to create executables, which represent real hardware. These include processor and other behavioral components. The executable can be used to run binary application files compiled and linked for the processors whose models are included. OVP can be used in 3rd party simulation environments (for example, SystemC). It can be used to create a test harness to help validate processor models under construction, or even to create custom simulation environments. OVP has less functionality than the Imperas Professional Simulator Products in some areas and has restricted commercial usage as stipulated in the OVP click-through license agreement.

- **Imperas Professional Simulator Products** enhance the basic capabilities provided by OVPsim, particularly in the areas of debugger integration, tool integration and multiprocessor simulation support (including QuantumLeap parallel simulation). Contact Imperas for more information.

4.2 What are OVPsim and CpuManager?

OVP provides the **OVPsim** simulator and the Imperas Professional product provides the **CpuManager** simulator as dynamic linked libraries (.so suffix on Linux, .dll suffix on Windows) implementing Imperas simulation technology. The shared objects contain implementations of the OP interface functions described later in this document. The OP functions enable instantiation, interconnection and simulation of complex multiprocessor platforms using multicore processors, advanced peripheral devices and complex memory topologies.

Processor models for use with CpuManager and OVPsim are created using the **OVP Virtual Machine Interface (VMI)** API, and are available for download from the www.OVPworld.org website. This API enables the creation of processor models that run at very high simulation speeds (typically hundreds of millions of simulated instructions per second). The use of the API is described in the **OVP Processor Modeling Guide**, also available for download from the www.OVPworld.org website.

The CpuManager simulator is part of the commercial/professional product offering available from Imperas. OVPsim is the freely-available (for Non-Commercial usage) version of the simulator. The simulator can be selected at runtime by the
IMPERAS_RUNTIME environment variable. If it is not set or is set to OVPsim the OVPsim library (which requires an OVP license) will be used. If it is set to CpuManager the CpuManager library (which requires an Imperas license) will be used.

The legacy ICM API is supported by the same products, providing a subset of the functionality offered by OP. In fact, the ICM API is implemented using OP so will be supported for the foreseeable future.

A subset of OP functionality can be used in SystemC TLM2.0. The TLM2.0 C++ interface code is available as source for processor and peripheral models, allowing the use of these models in SystemC TLM2.0 platforms.

4.3 Use of OP with Imperas tools

A program using the OP or ICM APIs must be linked with the Imperas RuntimeLoader library to perform runtime dynamic loading of either the CpuManager or OVPsim dynamic linked libraries, to produce a stand-alone executable. This allows the runtime selection of the CpuManager simulator for any defined platform and so enabling the use of the Imperas tools.

4.4 C API and iGen

An OVP module is written in C code and compiled and linked on the host computer to produce a shared object. An OVP testbench/harness, is written in C code, compiled and linked with the RuntimeLoader (libRuntimeLoader.so) link library on the host computer to produce an executable.

iGen is a program from Imperas that can create either outline or substantially complete C code for a module or testbench/harness using a description written using iGen function calls. Use of iGen is described elsewhere and used in the examples in this document.
5 Separation of iGen modules and C harness / testbench

Even though both the activities of creating a design / module and creating the harness / testbench ultimately make calls into the simulators OP C API there are two distinct usage flows.

A module starts as a script as input to iGen which generates C code that makes calls to the OP API. The iGen function API and use of iGen provide a powerful and easy to use approach to building the structure of your virtual platform to be simulated.

Harnesses and testbenches are written in C and instances modules and tell the simulator how to simulate them. This can include loading programs, loading data files, setting values into registers, scheduling the execution and controlling the interaction with other simulators or with bespoke debug solutions. The OP API has been designed to provide a very efficient and powerful way to specify and control simulations.

For now, assume that:

module creation is done in tcl with iGen
and harnesses / testbenches are written in C using the OP API.

Advanced users can use the OP calls in many different ways and can write code that will not adhere to these assumptions. For more details see the Advanced Simulation Control of Platforms and Modules User Guide.
6 Documentation on the OP API

Provided in the installation is the online OP API Function Reference documentation. This is Doxygen-like API documentation available at:

IMPERAS_HOME/doc/api/op/html/index.html

It includes per function references:
7 Introducing a C harness / testbench

For a first example, we will introduce the simplest module and application we can and then concentrate on the harness we are going to evolve.

Take of a copy of the example, Examples/SimulationControl/minimalHarness:

```bash
> cp -r $IMPERAS_HOME/Examples/SimulationControl/minimalHarness .
> cd minimalHarness
> ls
application    harness    module
```

Examine the application source

```bash
> cat application/application.c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
    printf("Hello world application\n");
    return 0;
}
```

Compile the application

```bash
> make -C application
# Compiling application.c
# Linking application.OR1K.elf
```

Examine the module source

```bash
> cat module/module.op.tcl
ihwnew -name simpleCpuMemory
ihwaddbus -instancename mainBus -addresswidth 32
ihwaddprocessor -instancename cpu1 -vendor ovpworld.org \\
   -library processor -type or1k -version 1.0 \\
   -semihostname or1kNewlib -variant generic
ihwconnect -bus mainBus -instancename cpu1 -busmasterport INSTRUCTION
ihwconnect -bus mainBus -instancename cpu1 -busmasterport DATA
ihwaddmemory -instancename ram1 -type ram
ihwconnect -bus mainBus -instancename ram1 -busslaveport spl \\
   -loaddress 0x0 -hiaddress 0xffffffff
```

Generate the module C code and compile the module

```bash
> make -C module
# iGen Create OP MODULE module
# Host Compiling Module obj/Linux64/module.igen.o
# Host Linking Module object model.dll
```

If you have worked through the iGen Platform and Module Creation User Guide then you should be familiar with the application and simple module, filenames and use of make etc.

The most minimal test harness needs to instance a module and start simulation.
If we examine the harness in the file `harness/harness.c`, we can see it contains a `main` function:

```c
int main(int argc, const char *argv[]) {
    opSessionInit(OP_VERSION);
    opCmdParseStd (argv[0], OP_AC_ALL, argc, argv);
    optModuleP mi = opRootModuleNew(0, 0, 0);
    opModuleNew(mi, "module", "u1", 0, 0);
    opRootModuleSimulate(mi);
    opSessionTerminate();
    return 0;
}
```

It is a normal C `main()` which provides access to the command line arguments passed into the executable using argc, and argv.

The `opSessionInit()` must be at the beginning to initialize and the `opSessionTerminate()` must be at the end to terminate. These are mandatory as the first and last function calls in `main()` and must encompass all other calls to the OP API. `OP_VERSION`, which is passed to the `opSessionInit()` call us defined in op.h and is used to verify that the API version used in this executable matches the version in the dynamically loaded simulator library.

The call to `opCmdParseStd()` tells the simulator to use a standard command line parser

```c
opCmdParseStd (argv[0], OP_AC_ALL, argc, argv);
```

Using the command line parser allows you to put arguments on the command line to control the simulator (for example to load an application program). You may use the `--help` argument to see the supported arguments. All the available arguments are described in the Imperas Simulation Guide. It is also possible to add harness-specific arguments to the standard command line parser, this is described later.

The call to `opRootModuleNew` tells the simulator to create an empty root module and it returns a pointer 'mi' to the created module.

```c
optModuleP mi = opRootModuleNew(0, 0, 0);
```

The call to `opModuleNew` instances a module under this root module, using the 'mi' pointer and finds the module shared object in the directory below us called 'module', and gives it the instance name of 'u1'.

```c
opModuleNew(mi, "module", "u1", 0, 0);
```

Finally, we call the simulator to execute the specified root module:

```c
opRootModuleSimulate(mi);
```
It is compiled and linked, using the provided Makefile, with:

```bash
> make -C harness
# Host Compiling Harness obj/Linux64/platform.o
# Host Linking Harness object model.dll
# Host Linking Harness harness.Linux64.exe
```

Then we can run it passing the program to be loaded for execution by the processor model in the module:

```bash
> harness/harness.$IMPERAS_ARCH.exe --program application/application.OR1K.elf
```

OVPsim started: Mon Jan 04 17:23:43 2016
Hello world application
OVPsim finished: Mon Jan 04 17:23:43 2016

Also, don’t forget, you can pass in standard built-in command line arguments to your harness, for example enable tracing of the instruction execution:

```bash
> harness/harness.$IMPERAS_ARCH.exe --program application/application.OR1K.elf --trace --tracechange
```

OVPsim started: Mon Jan 04 17:23:43 2016
...
We have provided an `example.sh` script (`example.bat` for Windows) which generates the module and compiles the application and harness.

```bash
> ./example.sh
```

7.1 **Writing a harness using C++**

A harness may be written in C++ if desired. This requires configuring the C++ program to call into the C routines of the OP API. An example of this is provided in:

```bash
$IMPERAS_HOME/Examples/SimulationControl/minimalHarnessUsingCPP
```

Take a copy of the example

```bash
> cp -r $IMPERAS_HOME/Examples/SimulationControl/minimalHarnessUsingCPP .
```
> cd minimalHarnessUsingCPP

And examine the C++ harness file `harness/harness.cpp`:

```cpp
#include <iostream>
using namespace std;

extern "C" {
#include "op/op.h"
}

int main(int argc, const char *argv[]) {
    cout << "Running C++ harness...\n";
    opSessionInit(OP_VERSION);
    opCmdParseStd (argv[0], OP_AC_ALL, argc, argv);
    optModuleP mi = opRootModuleNew(0, 0, 0);
    opModuleNew(mi, "module", "u1", 0, 0);
    opRootModuleSimulate(mi);
    opSessionTerminate();
    cout << "Finished running C++ harness.\n";
    return 0;
}
```

Here we see that we have switched to using the C++ `iostream` header file in place of the C `stdio.h`, and added `using namespace std;` to allow implied use of `std::` names.

Next, we see that the `extern "C"` keyword has been used to specify that everything defined in the `op.h` include file uses C linkage and structure conventions.

Finally, we have added use of the C++ `cout` object that is the equivalent of the C `FILE *stdout` stream variable, illustrating that C++ classes and methods can be freely used.

Any of the OP API calls may be used from C++ in this manner.

NOTE: Future releases will include a C++ class library to wrap the OP API C functions so they are simpler to use from C++. Until that becomes available C++ can be used by directly calling the C functions as shown here.

To build and run the example use the command:

```bash
> ./example.sh
# Compiling application.c
# Linking application.OR1K.elf
```
Imperas Tools Overview

```
rm application.o
# iGen Create OP MODULE module
# Copying STUBS module.c.igen.stubs to module.c
# Host Depending obj/Linux32/module.d
# Host Compiling Module obj/Linux32/module.o
# Host Linking Module object model.so
# Host Compiling Platform obj/Linux32/harness.o
# Host Linking Platform harness.Linux32.exe
# Host Linking Platform object model.so

Running C++ harness...
...
OVPsim started: Mon Feb 22 21:58:16 2016

Hello world application from C++ harness

OVPsim finished: Mon Feb 22 21:58:16 2016
...

Finished running C++ harness.
```
8 The top level or root module

A root module must be created (using opRootModuleNew) in the main function. This root module can either:

1. be a self-contained single level combined harness and design by containing all the components in the system (this is how a legacy ICM platform is constructed as it has no hierarchy).
2. be a harness / testbench by creating an instance of the top level of the design (for example, instance a module) and supplying any testbench functionality required to run the design. In fact, it could create instances of several designs and run them independently to perform step and compare testing.
3. include both a modelAttrs definition, so that this module may be instanced within a higher-level design, and contain a main function (with a command line parser, if required) so that it can be used as a top-level design. There will be both a module shared object and a design executable created.

The test harness in the previous example (section 7 “Introducing a C harness / testbench”) is about the simplest that can be created. It is in the form of the 2nd use case above: A main() function makes calls to the different OP simulation control capabilities.

It may be necessary to create a harness / testbench in the style of module creation using OP - i.e. as per the 3rd use case above.

We have provided the example SimulationControl/simpleHarness to demonstrate this approach.

All the other harnesses in this document will use the approach of not using the modelAttrs table, i.e. we are going to focus on the 2nd use case, where we are writing harnesses that instance other modules.
9 Configuring the Simulation using parameters

To increase the flexibility and reusability of a module, it can accept parameters which may be set by the module that creates the instance of this module (its parent) and whose values can be obtained by code in the module, to influence its behavior. Parameter values can be passed to its components (including sub-modules) or can influence the execution of its code. Parameter types include Boolean, Integer, Floating point and String.

9.1 Parameter Lists on a Root Module

There are many parameters that are only for use with the root module and these affect the whole simulation. For example installing a ctrl-C signal handler, or suppressing the simulator's banner, or running in verbose mode. Parameters are described in the HTML documentation; go to:

Imperas/doc/api/op/html/index.html

search for 'OP_FP_'. And look for parameters that must be applied to a rootmodule.

We will look at the example Examples/SimulationControl/configureRootModule (which is a copy of minimalHarness introduced above).

Take a local copy to edit:

```bash
> cp -r $IMPERAS_HOME/Examples/SimulationControl/configureRootModule .
> cd configureRootModule
> ls
application    harness    module
```

and then generate the module C code and compile and run with the provided example script:

```bash
> ./example.sh
```

The harness has a few extra lines of code in the main() to set the parameters on the creation of the root module.

```c
#define MODULE_DIR      "module"
#define MODULE_INSTANCE "u1"
#define CPU_INSTANCE    "cpu1"

int main(int argc, const char *argv[]) {
    opSessionInit(OP_VERSION);
    opCmdParseStd (argv[0], OP_AC_ALL, argc, argv);

    // root module / simulation wide parameters
    // opParamBoolOverride(0, OP_FP_SHOWFORMALS, 1);
    // opParamBoolOverride(0, OP_FP_SHOWBUSES, 1);
    // opParamBoolOverride(0, OP_FP_SHOWDOMAINS, 1);
    // opParamBoolOverride(0, OP_FP_SUPPRESSBANNER, 1);
    // opParamBoolOverride(0, OP_FP_ENABLEIMPERASINTERCEPTS, 1);
```
We create some definitions for the root module name and the module directory name to find the module's shared object, and the instance names for the module and cpu of interest.

We can override the value of a parameter of an object before we instance it by using the correct override function for the type, for example `opParamBoolOverride` function for a Boolean type (there are functions for all the supported types). We make use of the gnu C feature of string concatenation which creates a hierarchical name string to point at the object of interest.

The definitions `OP_FP_` are macros to define the built-in parameters, for example `OP_FP_VERBOSE` is "verbose", so we are passing the string "verbose" as the 2nd argument and ‘1’ (true) as the 3rd to set the simulator in verbose mode.

Also, for the CPU instance "u1/cpu1", we are setting the ‘MIPS’ rating to 200:

but note this is relative to the root module 'mi' (first argument), so we will have set "u1/cpu1/mips" to 200.

If we now build the harness:

```
> make -C harness
```

and run it:

```
> harness/harness.$IMPERAS_ARCH.exe --program application/application.OR1K.elf
```
We will see the simulator output will provide the verbose simulation statistics and the ‘nominal MIPS’ for the processor was set to 200:

```
OVPsim started: Tue Jan 12 10:48:05 2016

Info (OP_AL) Found attribute symbol 'modelAttrs' in file
'/home/graham/itest/Regression_Tests/OVPsim/Examples/OP/single.test.3095/0/module
/model.so'
Info (OP_AL) Found attribute symbol 'modelAttrs' in file
'/home/graham/Imperas/lib/Linux64/ImperasLib/ovpworld.org/processor/or1k/1.0/mod
el.so'
Info (OP_AL) Found attribute symbol 'modelAttrs' in file
'/home/graham/Imperas/lib/Linux64/ImperasLib/ovpworld.org/semihosting/or1kNewlib
/1.0/model.so'
Info (OR_OF) Target u1/cpu1' has object file read from
'application/application.OR1K.elf'
Info (OR_PH) Program Headers:
Info (OR_PD) LOAD           0x00002000 0x00000000 0x000051b8 0x000052d4 RWE
2000
Hello world application
Info
Info ---------------------------------------------------
Info CPU 'u1/cpu1' STATISTICS
Info   Type                  : or1k (generic)
Info   Nominal MIPS          : 200
Info   Final program counter : 0x1774
Info   Simulated instructions: 2,112
Info   Simulated MIPS        : run too short for meaningful result
Info ---------------------------------------------------
Info
Info ---------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 0.00 seconds
Info   User time             : 0.01 seconds
Info   System time           : 0.00 seconds
Info   Elapsed time          : 0.01 seconds
Info ---------------------------------------------------
OVPsim finished: Tue Jan 12 10:48:05 2016
```

Now if we change the harness to remove the previous changes so that we now have the parameter to suppress the banner:

```
opParamBoolOverride(0, OP_FP_SUPPRESSBANNER, 1);
```

after we re-build the harness

```
> make -C harness
```

and then run it, we will see the output without any banner information and just see the semihosted printf from the application:

```
> harness/harness.$IMPERAS_ARCH.exe --program application/application.OR1K.elf
```
**9.1.1 The available root module parameters**

The complete up-to-date list of available parameters is defined in the API documentation at IMPERAS_HOME/doc/api/op/html/index.html. Click on 'globals' in the left hand pane, then 'b' in the right hand pane, to see all the $OP_{FP}*parameters. Click on $OP_{FP\_DEBUGPSECONSTRUCTORS$ to see the first in the list that is applicable to root modules.

Some to try out are:

- $OP_{FP\_FINISHTIME}$
- $OP_{FP\_GDBCONSOLE}$, $OP_{FP\_GDBEGUI}$, $OP_{FP\_GDBIGUI}$
- $OP_{FP\_MPDCONSOLE}$, $OP_{FP\_MPDEGUI}$, $OP_{FP\_MPDIGUI}$
- $OP_{FP\_SHOWBUSES}$, $OP_{FP\_SHOWDOMAINS}$
- $OP_{FP\_WALLCLOCK}$, $OP_{FP\_WALLCLOCKFACTOR}$

Most of these are set with $opParamBoolOverride$, but there are also functions to set parameters that are Enum, Int32, String, Uns32 etc.

Most of the above options are best done from the command line or from control files but can also be added to the harness if they are required permanently.

For example, to set the simulation to always run with a wallclock factor of 5x we can use $OP_{FP\_WALLCLOCKFACTOR}$ on a root module

```c
    opParamDoubleOverride(0, OP_FP_WALLCLOCKFACTOR, 5.0);
```

**9.2 Using Overrides to Set parameter values on module and component instances**

You can set any parameter in your design from the command line or from the C harness. It is often better to set parameters from the command line.

By using the command line `--showoverrides` we can see that there are some parameters on the CPU instance in the module we have instantiated:

```
> harness/harness.$IMPERAS_ARCH.exe --program application/application.OR1K.elf -
-showoverrides
...
--override ul1/cpu1/enableimperasintercepts=0     (Boolean) Enable Imperas
intercepts
--override ul1/cpu1/simulateexceptions=0     (Boolean) Simulate exceptions
--override ul1/cpu1/trace=0     (Boolean) Enable instruction trace
--override ul1/cpu1/tracesshowicount=0     (Boolean) Print count with trace
--override ul1/cpu1/mips=100.000000     (Double) Set the mips rate for this
instance.
...
```

So we can then set some of these:
9.3 Configuring Modules and Components from Harness

In this section we will look at the use of parameters to modify the hardware definition and the component configuration from the harness.

This will use the example

$IMPERAS_HOME/Examples/SimulationControl/moduleParametersWithHarness

9.3.1 Module Hardware Parameters

The module is the hardware definition and typically this is fixed. However, some aspects of the module may be configured using parameters when it is instanced. In this example we will configure the base address at which a peripheral model is instanced.

The parameter to be configured must be defined as a formal parameter of the module on which you wish to apply it. It may also be defined at higher levels and used to configure the lower level modules when they are instanced. The choice of where to apply the parameter depends upon the hierarchy in the design and the level at which the configuration should be made.

Defining the parameters when the module is instanced is illustrated in the topmodule (topmodule/module.op.tcl) for each of the submodules it contains

ihwaddformalparameter -name subUart0Address -type address
iadddocumentatation -name Description -text "Set base address of UART in subsystem 0"
ihwaddformalparameter -name subUart1Address -type address
iadddocumentatation -name Description -text "Set base address of UART in subsystem 1"

which is used to set the configuration parameter on the module when it is instanced.

# configure parameters
ihwsetparameter -handle subUart0 -name baseAddress -value {subUart0Address} -type uns64
ihwsetparameter -handle subUart1 -name baseAddress -value {subUart1Address} -type uns64

In this case we are defining two formal parameters to independently set the base address for the two uarts in the sub modules, we could equally base both of the uart from a single formal parameter.

# configure parameters
We must define the formal parameter in the lowest level at which it is applied. This is shown in the submodule (submodule/module.op.tcl) in which the formal is defined and used to configure the base address at which the UART is instanced in the module.

This allows a generic module to have some configuration aspects that are defined when the hardware platform is created.

9.3.2 Component Feature Parameters
A peripheral or processor component model can include the behavior of different versions or types of hardware, for example a UART model can provide 16450 and 16550 behaviour. The configuration of the model can again be achieved by defining formal parameters on the model itself and then applying these in the module definition, on the command line or in the harness.

Defining the formal parameter for a peripheral model can be found in the documents iGen_Peripheral_Generator_User_Guide and OVP_Peripheral_Modeling_Guide and for a processor model in the OVP_Processor_Modeling_Guide.

9.3.3 Setting Parameters in the Harness
The harness can be used to set the parameters in modules and components using two methods.

The opParamXXX API functions can be used to append to a parameter list of type optParamP that is then passed to the module instance or the parameters can be applied directly using the OP_PARAMS and type macros.

Note that the parameter names are the hierarchical names that are based upon the module on which they are applied.

9.3.3.1 Parameter List
The following shows the parameters being applied by adding the parameters to a list...
The first shows the parameter applied to the formal in the top module.

```c
optParamP paramList = NULL;

// Set formal parameter on the top level module to change uart addresses
paramList = opParamUns64Set(paramList, "subUart0Address", 0x100003e8);
paramList = opParamUns64Set(paramList, "subUart1Address", 0x100003f8);
```

and the alternative is to apply directly to the sub module, rather than through the top module

```c
// Set formal parameter on the top level module to change uart addresses
paramList = opParamUns64Set(paramList, "subUart0/baseAddress", 0x100003e8);
paramList = opParamUns64Set(paramList, "subUart1/baseAddress", 0x100003f8);
```

The following shows the configuration of the `outfile` parameter on the two uarts

```c
// Set parameter on instance of uarts to set logfiles
paramList = opParamStringSet(paramList, "subUart0/uart0/outfile", "subUart0-uart0.txt");
paramList = opParamStringSet(paramList, "subUart1/uart0/outfile", "subUart1-uart0.txt");
```

Finally, the parameter list is passed to the function creating the module.

```c
// instance the topmodule into the root module
opModuleNew(mi, "./topmodule", "top", 0, paramList);
```

### 9.3.3.2 OP_PARAMS Macro

Instead of creating a parameter list the configuration can be applied directly when the module is instanced using macros.

The equivalent of setting the base addresses and the uart outfiles using this method is shown below

```c
opModuleNew(mi, "./topmodule", "top", 0,
    OP_PARAMS (
        OP_PARAM_UNS64_SET ("subUart0/baseAddress", 0x100003e8),
        OP_PARAM_UNS64_SET ("subUart1/baseAddress", 0x100003f8),
        OP_PARAM_STRING_SET ("subUart0/uart0/outfile", "subUart0-uart0.txt"),
        OP_PARAM_STRING_SET ("subUart1/uart0/outfile", "subUart1-uart0.txt")
    )
);
9.3.4 Running Example

The example can be executed from the directory

```
$IMPERAS_HOME/Examples/SimulationControl/moduleParametersWithHarness
```

using the provided scripts, for example on Linux

```
$ ./example.sh
```

Which will generate output similar to the following

```
mingw32-make: Entering directory
'C:/Imperas/Examples/SimulationControl/moduleParametersWithHarness/application'
Compiling application.c
Linking application.RISCV32.elf
rm application.RISCV32.o
mingw32-make: Leaving directory
'C:/Imperas/Examples/SimulationControl/moduleParametersWithHarness/submodule'
# iGen Create OP MODULE  module.c.igen.stubs
# No Update to existing module.c. Compare with module.c.igen.stubs if changes
made to iGen file module.op.tcl
# Host Depending obj/Windows64/module.d
mingw32-make: Leaving directory
'C:/Imperas/Examples/SimulationControl/moduleParametersWithHarness/submodule'
mingw32-make: Entering directory
'C:/Imperas/Examples/SimulationControl/moduleParametersWithHarness/submodule'
# No Update to existing module.c. Compare with module.c.igen.stubs if changes
made to iGen file module.op.tcl
# Host Compiling Module obj/Windows64/module.o
# Host Linking Module object model.dll
mingw32-make: Leaving directory
'C:/Imperas/Examples/SimulationControl/moduleParametersWithHarness/submodule'
mingw32-make: Entering directory
'C:/Imperas/Examples/SimulationControl/moduleParametersWithHarness/topmodule'
# iGen Create OP MODULE  module.c.igen.stubs
# No Update to existing module.c. Compare with module.c.igen.stubs if changes
made to iGen file module.op.tcl
# Host Depending obj/Windows64/module.d
mingw32-make: Leaving directory
'C:/Imperas/Examples/SimulationControl/moduleParametersWithHarness/topmodule'
mingw32-make: Entering directory
'C:/Imperas/Examples/SimulationControl/moduleParametersWithHarness/topmodule'
# No Update to existing module.c. Compare with module.c.igen.stubs if changes
made to iGen file module.op.tcl
# Host Compiling Module obj/Windows64/module.o
# Host Linking Module object model.dll
mingw32-make: Leaving directory
'C:/Imperas/Examples/SimulationControl/moduleParametersWithHarness/topmodule'
mingw32-make: Entering directory
'C:/Imperas/Examples/SimulationControl/moduleParametersWithHarness/harness'
# Host Depending obj/Windows64/harness.d
mingw32-make: Leaving directory
'C:/Imperas/Examples/SimulationControl/moduleParametersWithHarness/harness'
# Host Compiling Harness obj/Windows64/harness.o
```
The way in which the configuration is applied can be modified in the harness and topmodule to show the different configuration options described earlier.

In the top of harness/harness.c there are three defines

```c
// Set the defines to show alternate ways of setting parameters
#define PARAMLIST
#define TOLEVEL
// Define separate formal parameters for each submodule uart (set dualbase in
topmodule/module.op.tcl)
#define DUALBASE
```

PARAMLIST when defined will use the list defined by optParamP to set parameters, otherwise the OP_PARAMS macros will be used.
TOLEVEL can only be defined when PARAMLIST is defined and will set the parameters on the top level module
DUALBASE requires that the variable dualbase in topmodule/module.op.tcl is also set.
This controls whether the uart base addresses are both set or uart1 base address is set relative to uart0.
9.4 Special parameter values

Certain parameters are handled specially by the simulator. These are described in the following sections.

9.4.1 MIPS parameter

All processors support a double parameter called \textit{mips}, used to specify \textit{the nominal processor speed in millions of instructions per second}. This nominal mips rate is used to apportion run time between processors in a multiprocessor simulation. The default nominal mips rate for each processor is 100. The section above shows an example of using a hierarchical parameter setting to configure a processor with a nominal mips rate of 200 MIPS instead.

\begin{verbatim}
  opParamDoubleOverride(mi, MODULE_INSTANCE "" CPU_INSTANCE "/" OP_FP_MIPS, 200);
  opModuleNew(mi, MODULE_DIR, MODULE_INSTANCE, 0, 0);
\end{verbatim}

9.4.2 Endian parameter

The model documentation for processors includes a definition of the endianness supported by that processor. This may be \textit{big}, \textit{little} or \textit{either}.

If the endian is defined as \textit{either} then the model will accept a user string attribute called \textit{endian}, used to specify the endianness of the processor. The \textit{endian} attribute may take the values \textit{big} or \textit{little}.

Some processors allow the endianness to be changed dynamically by software. The \textit{endian} attribute only sets the initial value for the endianness at the start of simulation in this case.

You could set the endianness in the instantiation of the processor, or you could set it in the testbench with a hierarchical parameter setting e.g.:

\begin{verbatim}
  opParamDoubleOverride(mi, MODULE_INSTANCE "" CPU_INSTANCE "/" OP_FP_ENDIAN, OP_ENDIAN_BIG);
  opModuleNew(mi, MODULE_DIR, MODULE_INSTANCE, 0, 0);
\end{verbatim}

Note that the OR1K used in this example does not support the endian parameter.

9.5 Common error messages related to parameters

Some of the common errors are:

\textbf{Error (OP_PNF) Parameter \'<hier param name>\' has no formal parameter defined in the model.}\n
You are trying to call one of the parameter setting functions and the parameter you have specified has not been found - reasons are often spelling mistakes or wrong level of hierarchy. Use the command line argument \texttt{--showoverrides} or the root module parameter \texttt{OP_FP_SHOWFORMALS} to get a list of all the possible parameters in your design that can be set.
Error (OP_DCE) <hier param name> cannot be converted from a double to a boolean
You are setting the value of a parameter to an illegal value - for example
opParamDoubleSet needs the value to be a double etc. - so ensure the correct function is
used, e.g. opParamBoolSet, opParamDoubleSet etc. and that it matches with the value
set.
10 Controlling the simulation using the Command Line

When using a testbench or harness we will want to control the simulation from the command line, pass in arguments to configure the simulation, and set other parameters that will change the simulation run.

There are OP functions that we can use in our harness that will provide different sets of command line arguments. There is the option of basic or full commands, and you can extend the command line parser with your own command line arguments.

It is also possible to pass arguments from the simulator command line into the `main()` of applications running on simulated processors.

10.1 standardCommandLineParser example

Let’s start with a Command Line Parser (CLP), this is the standard command line parser that could be used to add custom arguments. It is configured to include all standard arguments.

If we have a look at the example (which is a copy of `minimalHarness` introduced above) by making a local copy to edit:

```bash
> cp -r $IMPERAS_HOME/Examples/SimulationControl/standardCommandLineParser .
> cd standardCommandLineParser
> ls
application harness module
```

There are two places that are related to the CLP in `harness/harness.c`:

```c
static void cmdParser(optCmdParserP parser) {
}
```

and in `main()`:

```c
optCmdParserP parser = opCmdParserNew(HARNESS_NAME, OP_AC_ALL);
cmdParser(parser);
opCmdParseArgs(parser, argc, argv);
```

and we can compile and run as normal using the provided example script. This time we have put a `--help` in the example script so we can see what command line arguments the simulator is allowing:

```bash
> ./example.sh
```

and we will see the full set of allowed options - this is due to the choice of `OP_AC_ALL` in the `opCmdParserNew` call creating the parser.
If we change this to \texttt{OP\_AC\_NONE}, or \texttt{OP\_AC\_BASIC} and run the example script we will see that we get only very basic arguments allowed. Note you get an error with this basic level of the command line parser because the argument \texttt{--program} is not allowed! This choice would be useful when you want to control the usage of your harness/system such that users could only use the built in programs etc. This would be used when you are defining and using your own arguments.

A more useful set would be: \texttt{OP\_AC\_SIM}.

To see a full list, use \texttt{--help} with either your own harness/platform or the provided simulators: \texttt{harness.exe}, \texttt{iss.exe} etc. To get a more detailed explanation of the different arguments, please read the Control File User Guide.

The above example has shown the creation of a new command line parser using a call to the function \texttt{opCmdParserNew} which returns a parser object \texttt{parser}. This object can be used to add new command line arguments before the command line is processed with calls to \texttt{opCmdParseArgs}.

When you do not wish to add new custom command line arguments, i.e. you only want the standard command line arguments provided by the simulator, the function \texttt{opCmdParseStd} can be used in place of the above two functions as shown below.

```c
int main (int argc, const char *argv[]) {
    ...
    opCmdParseStd(argv[0], OP\_AC\_SIM, argc, argv);
    ...
}
```

The first argument is used in messages and can be the name of the program, \texttt{argv[0]}, as shown or can be any constant string. The second argument controls the arguments that will be available.

10.2 Verbose Output

When verbose output is selected output is generated at the end of simulation giving statistics about the number of instructions executed by each processor, the simulated MIPS rate for each processor, and the total instructions and MIPS rate. This information is present when \texttt{--verbose} is specified on the command line.

The actual performance reported may vary and depends on the performance of the native host. In a typical example the overall simulation speed is apportioned across each processor based upon the nominal MIPS rate configured for each processor.

10.2.1 Simulation Time Statistics

In verbose mode the simulator writes information about simulated and elapsed time. Four time values appear in the \texttt{SIMULATION TIME STATISTICS} paragraph:

```
Info ----------------------------------------------
```
Info SIMULATION TIME STATISTICS
Info   Simulated time : 16.59 seconds
Info   User time : 2.12 seconds
Info   System time : 0.00 seconds
Info   Elapsed time : 2.12 seconds
Info   Real time ratio : 7.81x faster

Simulated time is the duration of the simulation in simulated time. This corresponds exactly to the notion of time in a simulation language such as Verilog and VHDL; it is entirely unrelated to wall-clock time.

User time is the time that the simulation process spent executing instructions on the host machine; system time is the time the host machine spent in the system while executing instructions on behalf of the simulation process. Elapsed time is the overall time taken by the simulation process on the host from start to finish. All three of these times will vary from run to run, depending on the host load average and other factors. Real time ratio shows how much faster than real time this simulation ran.

For each processor, the simulated MIPS line gives the rate at which instructions for that processor were executed in wallclock time. In other words, the simulated MIPS number for a processor is calculated by dividing the number of instructions executed by that processor by the elapsed time for the simulation process. In this example, the reported simulated MIPS for cpu1 is calculated by dividing the simulated instructions (1,658,997,966) by the elapsed time (2.12 seconds) to give 781.2:

Info CPU 'cpu1' STATISTICS
Info   Type : or1k
Info   Nominal MIPS : 100
Info   Final program counter : 0x1dcc
Info   Simulated instructions: 1,658,997,966
Info   Simulated MIPS : 781.2

Provided that a processor does not halt during a simulation, then the simulation ran faster than real time if simulated MIPS exceeds nominal MIPS, and slower than real time if nominal MIPS exceeds simulated MIPS.

When optimizing an application, you should be looking at and minimizing simulated time. When optimizing a model for efficiency, you should be looking at elapsed time.

10.3 Passing arguments into application programs
When running a simulation it is often useful to be able to pass arguments into the main() of the application running on the simulated processor.

Take a copy of the example

> cp -r $IMPERAS_HOME/Examples/SimulationControl/passingArgvIntoApplications .
> cd passingArgvIntoApplications
And examine the application file `application/application.c`:

```c
int main(int argc, char **argv) {
    char *str = (argc >= 2) ? "(Got an Arg)" : "()" ;
    int num = (argc >= 2) ? atoi(argv[1]) : 10; // assume one numerical argument
    // and a default of 10 if not set.
    printf("Passing argument into application. %s num(%d)\n", str, num);
    return 0;
}
```

This shows that the application is expecting and testing for arguments to be passed into it.

If we run without passing in any arguments, as performed in the standard script, we get:

```
> ./example.sh
OVPsim started: Tue Jan  5 13:53:17 2016
Passing argument into application. (no args) num(10)
OVPsim finished: Tue Jan  5 13:53:17 2016
```

We can pass arguments from the simulator command line to the application `argv` with the simulator's `--argv` command line argument.

Any arguments on the `example.sh` command line are passed to the harness executable command line in the script, so we can add additional arguments to the harness as shown:

```
> ./example.sh --argv 21
OVPsim started: Tue Jan  5 13:53:17 2016
Passing argument into application. (Got an Arg) num(21)
OVPsim finished: Tue Jan  5 13:53:17 2016
```

Two important things to note:

- The `--argv` argument **MUST BE THE LAST SIMULATOR ARGUMENT ON THE COMMAND LINE** as all arguments following it are passed down to all the processors applications without further processing
- The semihost libraries for some processors do not support this feature (most do support this feature)

### 10.4 Use of debugging options

There are many different standard debugging options and they are discussed in detail in the various provided debugger documents.

There are options related to *GDB*, for example:

```
> harness.exe --help | grep gdb
--debugprocessor -p processor strings Select processor to attach gdb
--gdbcommandfile des/cpu1=gdb.cmd GDB will run this startup script. e.g..
```
--gdbconsole [platform]   Pop up gdb(s) in console window(s)
--gdbegui [platform]   Start gdb debug in Eclipse (eGui)
--gdbflags strings   Pass additional flags to a gdb e.g. des/cpu1=special_flag
--gdbigui [platform]   Start gdb debug in Imperas gui (iGui)
--gdbinit strings   Pass a file to the gdb to execute before the prompt is displayed
--gdbpath strings   Set the gdb path for a processor. e.g. des/cpu1=/usr/bin/or1k/gdb
and MPD (these are only available in the Imperas simulator, not OVPsim):

> harness.exe --help | grep mpd
--mpdconsole [platform]   Pop up mpd in a console window
--mpdigui [platform]   Start MPD debug in iGui
--mpdegui [platform]   Start MPD debug in eGui (Eclipse)

and eGui:

--equioptions string   Pass these options onto the command line of GUI when started
--equicommants string   Pass these commands to the GUI at startup

Please use the --help option on your harness or one of the simulators or look in the Control File User Guide.

For more information on tracing, please see the section on tracing below.

10.5 Adding harness specific command line options

Often you want to add your own arguments to control a simulation, setting options, providing filenames, etc.

It is easy to use the standard argument parser and to augment it with your own arguments and use them in your harness.

NOTE

It is recommended that all new arguments are added using only lower case characters, for example add an argument as '--myNewArgument' rather than using '--myNewArgument'. Although upper case characters can be used they are converted internally to lower case to prevent problems when moving between different host machine operating systems.

Take a local copy to edit:

> cp -r $IMPERAS_HOME/Examples/SimulationControl/addingNewCommandLineArguments .
> cd addingNewCommandLineArguments

If we look at harness/harness.c we see we added some storage to hold the values of our new arguments:

struct optionsS {

Bool        myarg;
const char*  newString;
}
} options = {
    .myarg = 0,
    .newString = NULL,
};

We add the new arguments to the `cmdParser` function using calls to `opCmdParserAdd` to define them:

```c
static void cmdParser(optCmdParserP parser) {
    opCmdParserAdd(parser, "myarg"   , "m", "bool", "myargs group"  ,
                    OP_FT_BOOLVAL  , &options.myarg,
                    "This is my description [default: 0]",
                    OP_AC_ALL, 0, 0);
    opCmdParserAdd(parser, "newstring", 0 , "string", "myargs group" ,
                    OP_FT_STRINGVAL ,
                    &options.newString,
                    "New String",
                    OP_AC_ALL, 0, 0);
}
```

and we have defined a bool argument `--myarg`, with a short version `--m`, and will put its value into `options.myarg`.

We have also declared a string argument `--newstring` and will hold its value in `options.newString`.

They have both been given the group name 'myargs group' for grouping in the --help listing.

We have added a new function to check the status of the arguments:

```c
static void cmdParserCheck(optCmdParserP parser) {
    if (!opCmdArgUsed (parser,"program")) {
        opMessage("F", HARNESS_NAME, "Argument '--program' must be specified");
    }
    if (options.myarg) {
        opMessage("I", HARNESS_NAME, "Argument 'myarg' selected (%d)",
                  options.myarg);
    }
    if (options.newstring) {
        opMessage("I", HARNESS_NAME, "Argument 'newstring' selected ('%s')",
                  options.newString);
    }
}
```

and to print out their status. Note that we are also now forcing the rule that the `--program` argument must be specified.

And in the main function we now have the following that relates to the command line parser:

```c
optCmdParserP parser = opCmdParserNew(HARNESS_NAME, OP_AC_ALL);
```
If we now run:

./example.sh

We see nothing from the arguments as the script is applying the minimum argument
--program that we require, but if we just invoke the harness executable:

> harness/harness.$IMPERAS_ARCH.exe

we get:

Fatal (harness) Argument '--program' must be specified

and if we run:

> harness/harness.$IMPERAS_ARCH.exe --program application/application.OR1K.elf \   -m -newstring "my new string is here..."

Info (harness) Argument 'myarg' specified
Info (harness) Argument 'newstring' specified ("my new string is here...")

Of course, now that the values are in C in your harness, you can use them as controlling
parameters on modules, processors etc.

10.5.1 Argument Conflicts
The command line parser module checks for duplicate commands at runtime. This
mechanism will detect if the user’s argument names clash with each other or with built in
arguments. If this happens and you get the error message as shown below indicating you
need to use a different name.

Error (CM_DF) This program is trying to install command line flag 'verbose'
which is already in use.
Please study the list below then select another name.
flag short argument description
...

10.6 Other useful command line options
Some of the most useful command line options not already discussed are listed here:

There are a list of 'show' commands used to give visibility of the contents of a design, the
commands available, and the contents of a library:
To show design information of the loaded design

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--showbuses [root]</td>
<td>List all (static) bus connections</td>
</tr>
<tr>
<td>--showdomains [root]</td>
<td>List the (initial) state of each memory domain</td>
</tr>
</tbody>
</table>

To show system, model and other parameters than can be changed in the system, on the command line before the simulation start

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--showmodeloverrides [root]</td>
<td>List overrides requested by models in the platform</td>
</tr>
<tr>
<td>--showoverrides [root]</td>
<td>List all possible platform overrides</td>
</tr>
<tr>
<td>--showsystemoverrides [root]</td>
<td>List overrides in the platform provided by the simulator</td>
</tr>
</tbody>
</table>

To show commands that are available in the design that can be called

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--showcommands [root]</td>
<td>List commands that can be called with --callcommand</td>
</tr>
</tbody>
</table>

To show system environment variables that may be set to control operations of the simulator

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--showenvvars</td>
<td>List all environment variables read by Imperas products</td>
</tr>
</tbody>
</table>

To show a list of items in the VLNV library (set by the IMPERAS_VLNV environment variable):

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--showlibraryextensions</td>
<td>Semihost and other intercept libraries</td>
</tr>
<tr>
<td>--showlibrary</td>
<td>All models and platforms</td>
</tr>
<tr>
<td>--showlibrarymms</td>
<td>Memory model components (caches)</td>
</tr>
<tr>
<td>--showlibrarymodules</td>
<td>Modules</td>
</tr>
<tr>
<td>--showlibraryperipherals</td>
<td>Peripherals</td>
</tr>
<tr>
<td>--showlibraryPlatforms</td>
<td>Platform executables</td>
</tr>
<tr>
<td>--showlibraryprocessors</td>
<td>Processor models</td>
</tr>
</tbody>
</table>

To convert warnings to errors so that simulation cannot continue past a message at a Warning level

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--werror</td>
<td></td>
</tr>
</tbody>
</table>

To modify the interaction between the user and the simulation or between components in the simulation

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--quantum</td>
<td></td>
</tr>
<tr>
<td>--timeprecision</td>
<td></td>
</tr>
<tr>
<td>--wallclock, --wallclockfactor</td>
<td></td>
</tr>
</tbody>
</table>

To control how long the simulation will run

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--finishafter, --finishtime</td>
<td></td>
</tr>
</tbody>
</table>
To automatically install a ctrl-C handler into the simulation platform

--stoponcontrolc

For complete information on these and other arguments please see the explanation in the Control File User Guide.
11 Writing your own harness.exe

As a more complete example, let’s walk through the code that we used to implement harness.exe. We have provided it as an example, so you can understand it and also so you can copy it and use it as the basis for other bespoke harnesses.

> cp -r $IMPERAS_HOME/Examples/SimulationControl/dynamicHarness .
> cd dynamicHarness

The module and application are the same as those used in the simpleHarness example, so we will just look here at the file harness/harness.c.

The example uses the first style of a test harness as described in section 8 in that it does not use an attrsTable and makes OP calls from the main function.

It adds commands to the command line parser to allow user selection of a module shared object either from a VLNV path or from a direct path.

So here is the full file, interspersed with some explanations of some of the more pertinent sections.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "op/op.h"

Declare some type definitions to hold the data:

typedef struct vlnvSpecS {
    const char *vendor;
    const char *library;
    const char *name;
    const char *version;
} vlnvSpec;

typedef struct modelSpecS {
    vlnvSpec          vlnv;
    const char       *file;
    const char       *variant;
    const char       *endian;
    optArgStringListP parameters;
    Uns32             instcount;
} modelSpec;

Provide some extra help information in a string:

const char *helpMessage =
    "To see all command line options:\n"
    "$ harness.exe --help\n"
    "To see the list of models:\n"
    "$ harness.exe --showlibrarymodules\n"
    "To execute a platform from the library:\n"
```
Provide a function used when we have any errors and we want to end the simulation:

```c
static void exitWithErrors(void) {
    opSessionExit(1);
}
```

Provide a function to test for an environment variable being present and if not increment an error variable:

```c
static Uns32 errors = 0;

static void reportEnvironment(const char *program, const char *variable) {
    if(!getenv(variable)) {
        printf("%s: Please set environment variable '%s'\n", program, variable);
        errors++;
    }
}
```

Provide a function that will check for all the required environment variables and exit if any are not found:

```c
static void checkEnvironment(const char *program) {
    reportEnvironment(program, "IMPERAS_HOME");
    reportEnvironment(program, "IMPERAS_VLNV");
    if(errors) {
        exit(1);
    }
}
```

The main function:

```c
int main(int argc, const char **argv) {
    checkEnvironment(argv[0]);
    opSessionInit(OP_VERSION);
}
```
Use the standard command parser with the simulation and library options:

```c
optCmdParserP parser = opCmdParserNew(argv[0], OP_AC_SIM | OP_AC_LIB);
```

provide an extra user message to the command line parser that will be appended at the end of the `–help` output:

```c
opCmdUsageMessage(parser, helpMessage);
```

initialize our data structures to default values:

```c
modelSpec model = { .vlnv={0, "module", 0, "1.0"}, .file=0, .parameters=NULL, .instcount=1};
```

add extra command line arguments, defining their type, values, groups, etc.:

```c
  opCmdParserAdd (parser, "modulevendor", "V", 0, " harness", OP_FT_STRINGVAL, 
                &model.vlnv.vendor, "Module vendor", OP_AC_ALL, 0, 0);
  opCmdParserAdd (parser, "modulelibrary", "L", 0, " harness", OP_FT_STRINGVAL, 
                &model.vlnv.library, "Module library (not usually required)", OP_AC_ALL, 0, 0);
  opCmdParserAdd (parser, "modulename", "N", 0, " harness", OP_FT_STRINGVAL, 
                &model.vlnv.name, "Module name", OP_AC_ALL, 0, 0);
  opCmdParserAdd (parser, "moduleversion", "S", 0, " harness", OP_FT_STRINGVAL, 
                &model.vlnv.version, "Module version (not usually required)", OP_AC_ALL, 0, 0);
  opCmdParserAdd (parser, "modulefile", 0, 0, " harness", OP_FT_STRINGVAL, 
                &model.file, "Path to a a module", OP_AC_ALL, 0, 0);
  opCmdParserAdd (parser, "parameter", 0, 0, " harness", OP_FT_PAIRLIST, 
                &model.parameters, "Module model parameters", OP_AC_ALL, 0, 0);
  opCmdParserAdd (parser, "instancecount", 0, 0, " harness", OP_FT_UNS32VAL, 
                &model.instcount, "Number of instances of the module" , OP_AC_ALL, 0, 0);
```

parse the command line, returning with errors if needed:

```c
Bool ok = opCmdParseArgs(parser, argc, argv);
if(!ok) {
  exitWithErrors();
}

const char *path;
```
generate an error message if we don't have either `--modulename` or `--modulefile` specified on the command line:

```c
if(!model.vlnv.name && !model.file) {
    opMessage("E", "HAR_NPRC", "No module was specified. Suggest --
    modulename <module> "
    "or --modulefile <filename>");
    opPrintf("%s", helpMessage);
    exitWithErrors();
}
```

process the arguments to set the path to the modules directory path using either the `VLANV` or the direct path from `--modulefile`

```c
if(model.vlnv.name) {
    path = opVLNVString(
        0, // use the default VLAN path
        model.vlnv.vendor,
        model.vlnv.library,
        model.vlnv.name,
        model.vlnv.version,
        OP_MODULE,
        1   // report errors
    );
    if(!path) {
        exitWithErrors();
    }
} else {
    path = model.file;
}
```

create a new root module (without an `attrsTable` by specifying the first argument is 0)

```c
optModuleP mi = opRootModuleNew(0, 0, 0);
```

and then iterate to create the number of instances specified on the command line by the `--instancecount` parameter

```c
// Create number of instances of the module specified on the command line
Uns32 i;
for (i = 0; i < model.instcount; i++) {

// Create unique name for this instance
char instName[16];
snprintf(instName, sizeof(instName), "m%d", i);
```

If any `--parameter` options were specified on the command line, create a unique parameter list with a copy of them for this instance
// Create parameter list with any parameters specified on command line
// (Note: unique parameter list required for each opModuleNew call)

    optParamP paramList = NULL;
    optArgStringListP p;
    for (p = model.parameters; p; p= p->next) {
        paramList = opParamStringSet(paramList, p->name, p->value);
    }

and create each instance of the module into the root model \textit{mi}:

\begin{verbatim}
    if (!opModuleNew(mi, path, instName, 0, paramList)) {
        exitWithErrors();
    }
\end{verbatim}

then we simulate and terminate to clean up once complete:

\begin{verbatim}
    opRootModuleSimulate(mi);
    opSessionTerminate();
    exit(0);
\end{verbatim}

Quite simple really... and we can compile and run the examples with the provided script
or pass the \texttt{--help} argument to see the help using:

\begin{verbatim}
> ./example.sh
> ./example.sh --help
\end{verbatim}
12 Monitoring / Tracing during simulation

There are many different ways to monitor the progress of a simulation. You could monitor the instruction execution of the processors, you could monitor writes to peripherals, you could monitor changes to nets, and with the Imperas Verification, Analysis and Profiling (VAP) tools you could monitor higher level items such as function calls/returns, operating system tasks/schedule switches, user application creation and deletion, and more.

In this chapter we will cover the built-in simulator trace capabilities and what can easily be added in a harness / testbench using the OP API. We leave the advanced usage and VAP tools to other manuals.

12.1 Processor Model Tracing Operations

Tracing may be enabled for all or individual processors. By enabling tracing for a processor instance instruction-by-instruction output for that processor instance is generated using the disassembler built in to the processor model.

The trace change mode writes the value of all modified registers when instruction tracing is enabled. Changed values are detected by maintaining a record of all values readable using the register access API at the completion of every instruction.

The register tracing ‘before’ dumps the current processor register state, again using the model-specific register dump format, when tracing is enabled. The order of events for each instruction is:

1. The register state of the processor is dumped;
2. The instruction about to be executed is shown in disassembled form;
3. The instruction is executed.

The register tracing ‘after’ dumps the current processor register state, again using the model-specific register dump format, when tracing is enabled. The order of events for each instruction is:

1. The instruction about to be executed is shown in disassembled form;
2. The instruction is executed;
3. The register state of the processor is dumped.

To keep instruction trace information separate from other simulator output, use -tracefile <filename>.

12.1.1 Tracing control on the command line

The following command line arguments are used to control the tracing that is output when each instruction is executed.

```
--trace -t [processor]    Trace instructions as they are executed
--traceafter <number>    Start tracing instructions after this many have executed
--tracechange [processor] Trace changed registers
```

## --tracecount <number> Trace this number of instructions
## --tracefile <filename> Direct all trace output to this file
## --tracepse [pse] Trace instructions executed by this peripheral model
## --tracereg [processor] Dump registers after each instruction is executed
## --traceregafter [processor] Dump registers after each instruction is executed
## --traceregbefore [processor] Dump registers before each instruction is executed
## --traceshareddata [processor] Trace the use of vmirtWriteListeners
## --traceshowicount [processor] Show instruction count with each instruction

The following turns on the trace buffer which is a special instruction trace mode provided to limit the output to the last N instructions executed before the buffer is flushed or the simulation terminates.

```plaintext
--tracebuffer [processor] Enable the trace buffer
```

It is also possible to enable further tracing of the processor and peripheral models in the simulation by enabling levels of diagnostics on the models.

For processor models:

```plaintext
--cpuflags [processor] CPU processor model specific trace flags
```

For peripheral models:

```plaintext
--modeldiags [pse] Turn on peripheral type specific diagnostics
```

### 12.1.2 Tracing control with CPU model parameters

The following CPU model parameters can be set both on the processor instance and on the command line. It is recommended, for flexibility, that unless specifically required these are used from the command line and not ‘fixed’ in the processor instance in the module.

To control the tracing that is output when each instruction is executed:

```plaintext
OP_FP_TRACE
OP_FP_TRACEREGSAFTER
OP_FP_TRACEREGSBEFORE
OP_FP_TRACESHOWICOUNT
OP_FP_TRACESTART
OP_FP_TRACECHANGE
OP_FP_TRACECOUNT
```

The following illustrates how these model parameters can be applied within a harness to a processor instance named ‘cpu1’ within a module named ‘u1’ to control tracing of the instruction execution.

```plaintext
opParamBoolOverride(mi, "u1/cpu1/" OP_FP_TRACE, 1); // enable instruction tracing
opParamBoolOverride(mi, "u1/cpu1/" OP_FP_TRACESHOWICOUNT, 1); // display instruction count
```
The instructions that are traced can be selected between instruction numbers using the \texttt{opProcessorTraceOnAfter} and \texttt{opProcessorTraceOffAfter} functions.

The trace buffer can be controlled using the \texttt{opProcessorTraceBufferEnable} and \texttt{opProcessorTraceBufferDisable} functions and a trace of the last 512 instructions can be output at any time, after the trace buffer is enabled, using \texttt{opProcessorTraceBufferDump}.

\subsection*{12.2 Processor Instruction Tracing Example}

The control of instruction tracing is illustrated in the example provided as \texttt{Examples/SimulationControl/controllingTracing} which shows both the control by configuring the processor instance and also using the command line.

Take a copy of the example.

\begin{verbatim}
> cp -r $IMPERAS_HOME/Examples/SimulationControl/controllingTracing .
> cd controllingTracing
\end{verbatim}

The harness includes a command line parser that has one additional command \texttt{‘configurecpuinstance’} to allow the control of whether to apply the overrides to control tracing on the processor instance. This allows the same example to be used to show tracing enabled in the harness and using the standard command line arguments.
When this argument is set, the processor instance is configured to enable instruction tracing and register change tracing. The current instruction count is included in the instruction tracing output. The tracing is controlled so that it does not start until instruction 2000 is executed.

This technique allows you to execute a large number of instructions at high speed, then focus on a few instructions of interest which, with tracing turned on will execute less quickly.

```c
// use a custom argument to control if we add to the processor instance configuration
if (options.configurecpuinstance) {
    opMessage("I", HARNESS_NAME, "Adding trace overrides to cpu instance \\
%s", \\
            MODULE_INSTANCE "/" CPU_INSTANCE);
    // enabling instruction tracing
    opParamBoolOverride(mi, MODULE_INSTANCE "/" CPU_INSTANCE "/" \\
            OP_FP_TRACE, 1);
    // display the instruction count
    opParamBoolOverride(mi, MODULE_INSTANCE "/" CPU_INSTANCE "/" \\
            OP_FP_TRACESHOWICOUNT, 1);
    // start tracing after 2000 instructions
    opParamUns64Override(mi, MODULE_INSTANCE "/" CPU_INSTANCE "/" \\
            OP_FP_TRACESTART, 2000);
}
```

Compile the application, module and harness

```bash
> make -C application
> make -C module
> make -C harness
```

The harness can be executed in two ways, first the processor instance is configured to enable tracing after 2000 instructions have been executed

```bash
> harness/harness.$IMPERAS_ARCH.exe --program application/application.OR1K.elf \\
   --configurecpuinstance
```

This generates the following trace output

```
Info (top) Adding trace overrides to cpu instance u1/cpu1
Hello world application
Info 2000: 'u1/cpu1', ADDRESS(_sclose+20): 1.lwz   r3,0x0(r3)
Info 2001: 'u1/cpu1', ADDRESS(__close_r): 1.addi   r1,r1,ADDRESS
Info 2002: 'u1/cpu1', ADDRESS(__close_r+4): 1.sw    0x4(r1),r2

Info 2104: 'u1/cpu1', ADDRESS(_exit+40): 1.ori    r3,r10,0x0
Info 2105: 'u1/cpu1', ADDRESS(_exit): *** INTERCEPT *** (_exit)
```
And secondly, the standard command line arguments are used to perform the same tracing:

```
> harness/harness.$IMPERAS_ARCH.exe --program application/application.OR1K.elf \
  traceshowicount --traceafter 2000
```

This produces the same trace output:

```
Hello world application
Info 2000: 'u1/cpu1', ADDRESS(__sclose+20): 1.lwz r3,0x0(r3)
Info 2001: 'u1/cpu1', ADDRESS(__close_r): 1.addi r1,r1,ADDRESS
Info 2002: 'u1/cpu1', ADDRESS(__close_r+4): 1.sw 0x4(r1),r2
Info 2104: 'u1/cpu1', ADDRESS(_exit+40): 1.ori r3,r10,0x0
Info 2105: 'u1/cpu1', ADDRESS(__exit): *** INTERCEPT *** (__exit)
```

The compilation and execution of the harness in both manners shown above is also provided in the example script, example.sh(.bat)

### 12.3 Peripheral model parameters

A peripheral model can be configured from the harness or on the command line using overrides.

#### 12.3.1 Obtaining parameter lists (using --showoverrides)

The available configuration parameters for a peripheral model can be obtained from:

1. examination of the peripheral model source code or documentation
2. by running the platform and enabling the parameter overrides output

The following shows an example of obtaining the list of available parameters from a module using the `--showoverrides` argument.

We will use the Example we have seen previously described in the document *iGen Platform and Module Creation User Guide, simpleCpuMemoryUart*

```
> cp -r $IMPERAS_HOME/Examples/PlatformConstruction/simpleCpuMemoryUart ./
> cd simpleCpuMemoryUart
> make -C application
> make -C module
> harness.exe --modulefile module/model.so --showoverrides
```

which when executed provides the following output including the list of parameter configuration overrides (we are only showing those associated with the peripheral model `periph0`):
12.3.2 Configuration

A peripheral model in the platform may operate in several modes and you may wish to change the mode of operation in the harness.

In the previous section we obtained the list of all the parameters that can be used to configure and/or modify the behaviour of the peripheral model.

Two of those parameters, we will look at, are `portnum` and `connectnonblocking` which controls the opening of a port for an external connection and how the UART peripheral model starts up respectively. `portnum`, when set, will control the UART peripheral opening of a port and `connectnonblocking` will control if it waits for a connection to be made on the port before allowing the simulation to continue i.e. by default the simulation is blocked until the port connection is made.

First we run the example and add the `portnum` parameter to control the opening of a port for external connection of a terminal, we should see that the simulation waits for a connection to be made.

```
> ./example.sh --override simpleCpuMemoryUart/periph0/portnum=1234
```


Info (PSE_SER) 'simpleCpuMemoryUart/periph0' Waiting for connection on port 1234
The simulation is now waiting for a connection, and will not move forward until a
collection is made.

In another terminal (on Linux) we can make the connection

```
> telnet localhost 1234
Trying 127.0.0.1...
```

In the simulator terminal we see that the connection is made and the simulation continues
to completion

```
Info (PSE_SER) 'simpleCpuMemoryUart/periph0' Connected to 1234
Initializing KinetisUART
Writing to uart - see log file
```

The script also outputs the UART logfile that is generated

```
Hello UART0 world
```

And in the telnet terminal we see that the UART has connected and sent a message prior
to the simulation completing and closing the port.

```
Connected to localhost.
Escape character is '^[]'.
Hello UART0 world
Connection closed by foreign host.
```

Now we also add the `connectnonblocking` parameter

```
> ./example.sh --override simpleCpuMemoryUart/periph0/portnum=1234  
   --override simpleCpuMemoryUart/periph0/connectnonblocking=1
```

the simulation continues (until completion) even if a connection is not made while the
UART model is listening to see if a connection is made in the future.

```
Info (PSE_SER) 'harness/simpleCpuMemoryUart/periph0' Listening for connection on
port 1234
Initializing KinetisUART
Writing to uart - see log file
```

The script also outputs the UART logfile that is generated

```
Hello UART0 world
```
12.3.3 Diagnostics
When there is one or more peripheral model in the platform definition it is often useful to be able to turn on diagnostics to see
1) the internal behavior of the peripheral
2) accesses to the peripheral registers

The actual level of diagnostics that are present in a peripheral model is determined and created by the peripheral model designer.

Typically there are three levels of diagnostics designed into the peripheral (PSE) model that can be set to provide information with further diagnostics enabled for peripheral model native code. Please see the Peripheral Modeling User Guide for further information on the diagnostic levels and their meaning.

12.3.3.1 On a single Instance
The diagnostics can be set for an individual peripheral model using a model parameter, \textit{diagnosticlevel}, as we saw previously when we displayed the available configuration parameters.

\begin{verbatim}
--override simpleCpuMemoryUart/periph0/diagnosticlevel=0 (Uns32) model diagnostics
\end{verbatim}

so if we run the example we see additional diagnostic information from the UART model:

\begin{verbatim}
> ./example.sh --override simpleCpuMemoryUart/periph0/diagnosticlevel=2

Info (UART_UIS) simpleCpuMemoryUart/periph0: Uart initialized in serial channel mode
Initializing KinetisUART
Info (UART_BRC) simpleCpuMemoryUart/periph0: Baud rate changed to 19921
Info (UART_BRC) simpleCpuMemoryUart/periph0: Baud rate changed to 19577
Info (UART_TFT) simpleCpuMemoryUart/periph0: Transmitter fifo threshold set to 1
Info (UART_RFT) simpleCpuMemoryUart/periph0: Receiver fifo threshold set to 1
Info (UART_UW) simpleCpuMemoryUart/periph0: Write to Data register: data=0x0d (')
Writing to uart - see log file
Info (UART_UW) simpleCpuMemoryUart/periph0: Write to Data register: data=0x48 ('H')
Info (UART_UW) simpleCpuMemoryUart/periph0: Write to Data register: data=0x65 ('e')
...
Info (UART_UW) simpleCpuMemoryUart/periph0: Write to Data register: data=0x64 ('d')
...

\end{verbatim}

Overrides can accept wildcards to cause them to be applied to several instances with similar names. For instance thing/uart*/modeldiags.
12.3.3.2 On all instances in a design

The diagnostic level for all models in the simulation can be set using the simulator argument --modeldiags

> ./example.sh --modeldiags 2

```
Info (UART_UIS) simpleCpuMemoryUart/periph0: Uart initialized in serial channel mode
Initializing KinetisUART
Info (UART_BRC) simpleCpuMemoryUart/periph0: Baud rate changed to 19921
Info (UART_BRC) simpleCpuMemoryUart/periph0: Baud rate changed to 19577
Info (UART_TFT) simpleCpuMemoryUart/periph0: Transmitter fifo threshold set to 1
Info (UART_RFT) simpleCpuMemoryUart/periph0: Receiver fifo threshold set to 1
Info (UART_UW) simpleCpuMemoryUart/periph0: Write to Data register: data=0x0d ('d')
Writing to uart - see log file
Info (UART_UW) simpleCpuMemoryUart/periph0: Write to Data register: data=0x48 ('H')
Info (UART_UW) simpleCpuMemoryUart/periph0: Write to Data register: data=0x65 ('e')
Info (UART_UW) simpleCpuMemoryUart/periph0: Write to Data register: data=0x6c ('l')
Info (UART_UW) simpleCpuMemoryUart/periph0: Write to Data register: data=0x64 ('d')
...
```

12.4 Adding your own monitors in a harness

Monitors are different from watchpoints. Monitors are used by tools to monitor a platform without affecting its behavior. They should not be used to stop the simulator.

Watchpoints are used to stop the simulator when particular conditions are met. They are used to implement a debugger.

Monitors are added as callback functions using the following sets of functions. The sets differ in their treatment of the address arguments.

On processor instances

- opProcessorReadMonitorAdd
- opProcessorWriteMonitorAdd
- opProcessorFetchMonitorAdd

The address parameters refer to virtual addresses on the processor’s data domain.

On bus instances

- opBusReadMonitorAdd
- opBusWriteMonitorAdd
- opBusFetchMonitorAdd

The address parameters refer to physical addresses on the bus. Zero is the lowest address on the bus in whichever device on the bus it is decoded at zero (if one exists).
On memory instances

- `opMemoryReadMonitorAdd`
- `opMemoryWriteMonitorAdd`
- `opMemoryFetchMonitorAdd`

The address parameters refer to physical addresses in the memory. Zero is the first address in the memory, no matter what address the memory is decoded at.

These functions all utilize a callback of type `OP_MONITOR_FN` and are used for monitoring accesses in the system. More than one monitor can be applied to fully or partially overlapping regions; they operate independently. This allows tools to be written to monitor and report different aspects of a platform’s operation. The tools can be used simultaneously without interference.

The simulator will call all callbacks but does not guarantee a particular order.

On net instances

- `opNetWriteMonitorAdd`

This function utilizes a callback of type `OP_NET_WRITE_FN` and is used for monitoring nets being written in the system.

The following sections show examples of some of these callbacks being used.

### 12.4.1 Monitoring nets with callbacks

This example is based on the modules and applications described in the example above `PlatformConstruction/simpleHierarchy`. It is a design with two UARTs connected with nets driven by two processors.

We will add harness.c to control it. The harness is based on the harness.c in `SimulationControl/minimalHarness`.

This example will show the use of `opNetWriteMonitorAdd`. It will also show use of some OP introspection functions.

First, take a copy of the example:

```bash
> cp -r $IMPERAS_HOME/Examples/SimulationControl/monitoringNets .
> cd monitoringNets
> ls harness
harness.c Makefile
```

### 12.4.1.1 Finding nets and declaring net callbacks

In the harness we define one function to be called when the net changes:

```c
static OP_NET_WRITE_FN(netCallback) {
```

OP_NET_WRITE \_FN is a macro that declares the callback function we are providing. It gives us the value that has been written to the net, and the userData we will pass in when registering the callback on the net. In our case the userData is a pointer to the net.

We need another function to be called before simulation starts to find the nets and register the callbacks:

```c
static void monitorNets(optModuleP mi) {
    optModuleP mod;
    if (!(mod = opObjectByName (mi, MODULE_INSTANCE, OP_MODULE_EN).Module)) {
        opMessage ("F", "HN_MNE", "Can not find module(%s)",
                                 MODULE_INSTANCE);
    }
    optNetP net = 0;
    while ((net = opNetNext(mod, net))) {
        opPrintf ("monitorNets(%s)\n", opObjectHierName(net));
        opNetWriteMonitorAdd (net, netCallback, net);
    }
}
```

This returns a pointer to the module instance, then scans for all nets within it. For each found net it prints a message, and then registers a callback which will be called whenever the net is written.

Nets passing through module hierarchy are bonded together (flattened) by the simulator to make propagation more efficient at run-time. Net monitoring callbacks have the same effect wherever they are connected to a net.

### 12.4.1.2 Simulating with net callbacks

To test the callbacks:

```bash
> ./example.sh  
...  
monitorNets(simpleHierarchy/write0to1)  
monitorNets(simpleHierarchy/write1to0)  
[application1] Initializing KinetisUART  
[application1]  Reading from uart  
[application0] Initializing KinetisUART  
[application0]  Writing to uart  
netCallback(write0to1) = H  
netCallback(write0to1) = e  
netCallback(write0to1) = l  
netCallback(write0to1) = l  
netCallback(write0to1) = o  
netCallback(write0to1) = 
```
Hello UART world
netCallback(write0to1) = x

[application1] Read termination char(x) from uart


12.4.1.3 monitoringNets/harness/harness.c full listing

```c
#include <string.h>
#include <stdlib.h>
#include "op/op.h"

#define HARNESS_NAME "harness"
#define MODULE_DIR "topmodule"
#define MODULE_INSTANCE "simpleHierarchy"

static OP_NET_WRITE_FN(netCallback) {
    optNetP net = userData;
    opPrintf("netCallback(%s) = %c\n", opObjectName(net), value);
}

static void monitorNets(optModuleP mi) {
    optModuleP mod;
    if (!(mod = opObjectByName (mi, MODULE_INSTANCE, OP_MODULE_EN).Module)) {
        opMessage ("F", "HN_MNE", "Can not find module(%s)", MODULE_INSTANCE);
    }

    optNetP net = 0;
    while ((net = opNetNext(mod, net))) {
        opPrintf ("monitorNets(%s)\n", opObjectHierName(net));
        opNetWriteMonitorAdd(net, netCallback, net);
    }
}

int main(int argc, const char *argv[]) {
    opSessionInit(OP_VERSION);
    opCmdParseStd (HARNESS_NAME, OP_AC_ALL, argc, argv);
    optModuleP mi = opRootModuleNew(0, 0, 0);
    opModuleNew(mi, MODULE_DIR, MODULE_INSTANCE, 0, 0);
    monitorNets(mi);
    opRootModuleSimulate(mi);
    opSessionTerminate();
```
return 0;
}

12.4.2 Monitoring Bus / Memory / Processor Address accesses using Callbacks

This example is based on the modules and applications described in the previous example PlatformConstruction/simpleHierarchy. It is a design with two UARTs connected with nets driven by two processors.

We will add a harness.c to control it. The harness is based on the harness.c in SimulationControl/minimalHarness.

This example will show the use of opBusFetchMonitorAdd, opBusReadMonitorAdd and opBusWriteMonitorAdd to set callbacks across the complete bus address space. The equivalent memory watchpoint functions are also used to watch memory accesses and the equivalent processor callbacks could be used to monitor specific processor accesses.

In this example we have a single processor with a single bus connecting to a memory, so in this case the three different types of monitors will likely show the same information. In a more complex design the different monitor types can be chosen to provide the specific information required.

The example will also make use of some OP introspection functions.

Take a copy of the example:

> cp -r $IMPERAS_HOME/Examples/SimulationControl/monitoringAccesses .
> cd monitoringAccesses
> ls harness
harness.c Makefile

12.4.2.1 Find buses and memories and attach callbacks

We define a function to be called back when an access we are interested in is triggered. The function will print the information:

```c
static OP_MONITOR_FN(monitorCallback) {
  opMessage ("I", PREFIX "_MT",
      "Monitor triggered: "
    "callback '%s': processor '%s': "
    "type '%s': bytes %u : "
    "address Physical 0x" FMT_A0Nx" Virtual 0x" FMT_A0Nx,
    __FUNCTION__,
    processor ? opObjectName(processor) : "artifact", // if no processor
    this is an artifact access
    (const char*)userData,
    bytes,
    addr,
    VA
  );
```
OP_MONITOR_FN is a macro that declares the callback function we are providing. It gives us all the information associated with the access for which we were notified, for example the processor object, the physical and virtual address, the size etc. The userData is set to a string indicating if this callback was registered on a bus or a memory.

The monitorBus function is shown below which iterates over objects of type bus found in the module and installs a callback for any access. The monitorMemory function is similar but iterates over object of type memory found in the module:

```c
static void monitorBus(optModuleP mi) {
    optModuleP mod;
    if (!(mod = opObjectByName (mi, MODULE_INSTANCE, OP_MODULE_EN).Module)) {
        opMessage ("F", PREFIX "_NFW", "Can not find module(%s)",
            MODULE_INSTANCE);
    } // iterate across all buses found in module
    optBusP bus = 0;
    while ((bus = opBusNext(mod, bus))) {
        Addr max = MAX_ADDRESS;
        opMessage("I", PREFIX ", "BM", "Add monitor for '%s' (0x" FMT_A0Nx " to 0x" FMT_A0Nx ")\n",
            opObjectHierName(bus), (Addr)0, max);
        if (options.busshow) {
            opBusShow(bus); // print the bus connections for each bus found
        }
        opBusFetchMonitorAdd(bus, 0, MIN_ADDRESS, max, monitorCallback, "bus-fetch");
        opBusReadMonitorAdd (bus, 0, MIN_ADDRESS, max, monitorCallback, "bus-read");
        opBusWriteMonitorAdd(bus, 0, MIN_ADDRESS, max, monitorCallback, "bus-write");
    }
}
```

12.4.2.2 Running the example

```
> ./example.sh
```

```
Info (BUS_MON_BM) Add monitor for 'simpleHierarchy/mainBus' (0x0000000000000000 to 0x00000000000000000000000000000000)
Info (BUS_MON_BM) Add monitor for 'simpleHierarchy/ram1' (0x0000000000000000 to 0x00000000000000000000000000000000)
Info (BUS_MON_MT) Monitor triggered: callback 'monitorCallback': processor 'cpu1': type 'bus-fetch': bytes 4 : address Physical 0x0000000000000000100 Virtual 0x0000000000000000100
Info (BUS_MON_MT) Monitor triggered: callback 'monitorCallback': processor 'cpu1': type 'memory-fetch': bytes 4 : address Physical 0x0000000000000000100 Virtual 0x0000000000000000100
Info (BUS_MON_MT) Monitor triggered: callback 'monitorCallback': processor 'cpu1': type 'bus-fetch': bytes 4 : address Physical 0x0000000000000000104 Virtual 0x0000000000000000104
```
On occasion you will see an artifact access being reported.

```
Info (BUS_MON_MT) Monitor triggered: callback 'monitorCallback': processor
'artifact' : type 'memory-write' : bytes 52 : address Physical
0x00000000ffffff3c Virtual 0x00000000ffffff3c
```

This is an access that is not a ‘real’ access by the processor but instead is an access being performed by the simulator or a loaded library in the system to access a current value, for example this could be the simulator pre-loading instructions for the code morphing engine.

### 12.4.3 Adding a command to show module contents

In the previous example, to help us see what the design looks like, we added command line options. The first command was `--moduleshow` which uses the `opModuleShow` function to dump what the simulator sees in the design hierarchy. The second command was `--showbuses` which uses `opBusShow` function to provide details of the bus connections. The command `--shownet <net name>` shows where a net is connected. The command `--shownet (without an argument)` shows where each net is connected.

The arguments are added to the standard command line parser, as was previously illustrated in section 10.5 “Adding harness specific command line options”.

When the `--moduleshow` argument is used in the example no simulation is performed and only the module information is generated

```sh
> ./example.sh --moduleshow
...
Module:
   hier : ""
   name : "(null)"
   path : "(null)"
   docNode:
      Formal : [SYS] debugpseconstructors=F (Boolean) (default=F) Start the debugger BEFORE PSE constructors have run
...
Module:
   hier : "simpleHierarchy"
   name : "simpleHierarchy"
   path : "module"
   docNode:
      Bus:
         hier : "simpleHierarchy/mainBus"
         name : "mainBus"
         addrBits : 32
         busPortConns :
            BusPortConn : spl
            BusPortConn : DATA
            BusPortConn : INSTRUCTION
...
```

When the `--busshow` argument is used in the example the bus connection information is generated for each bus when adding the monitor callbacks, simulation is performed.
> ./example.sh --busshow

...  
Info (BUS_MON_BM) Add monitor for 'simpleHierarchy/mainBus' (0x0000000000000000 to 0x00000000ffffffff)
BUS CONNECTIONS: simpleHierarchy/mainBus   (0:ffffffff)
  spl1              on simpleHierarchy/ram1   (0:ffffffff)
  DATA              on simpleHierarchy/cpu1   (0:0)
  INSTRUCTION      on simpleHierarchy/cpu1   (0:0)
...

12.5 Writing a Monitor module

Please see the document Advanced Simulation Control of Platforms and Modules User Guide about writing a module that monitors its ports etc. This can then be instanced in a harness and enabled to monitor buses and nets in the design under test.
13 Simulating Processor Exceptions

By default, a simulation will stop (opRootModuleSimulate will return) if a processor exception occurs.

Some examples of processor exceptions are:
- Executing from memory with no execute permission;
- Read, write or fetch at unaligned address (for processors that require aligned access);
- Attempting to read or write from an address where there is no memory or the memory has insufficient permissions.

In a harness, when opRootModuleSimulate returns for any reason other than end-of-simulation, it returns an optProcessorP object that is the handle of the processor that was executing when the termination condition occurred. To find the exact reason why simulation stopped, use:

```
opStopReason opProcessorStopReason(optProcessorP processor);
```

The opStopReason type returned by this function is an enumeration encoding the possible reasons why simulation stopped.

Instead of stopping simulation on a simulated exception, it is possible to specify that the processor should perform its usual exception actions instead (typically, enter kernel mode and jump to a kernel exception handler). This is usually specified in the iGen input file using the -simulateexceptions option of the ihwaddprocessor command, but may alternatively be done using the OP_FP_SIMULATEEXCEPTIONS processor instance attribute or setting the override parameter ‘simulateexceptions’ on the processor instance using the command line or control file.

13.1 Example of Simulating an Unaligned Access Exception

This example is in the enableProcessorExceptions directory.

Take a local copy to edit:

```
> cp -r $IMPERAS_HOME/Examples/SimulationControl/enableProcessorExceptions .
> cd enableProcessorExceptions
```

A simple module has been provided that instances a processor and memory. By default, the processor is configured so that it does NOT have simulated exceptions enabled.

Compile the test module and application as before using the following commands:

```
make -C module
make -C application
```
To run the simulation, in the `enableProcessorExceptions` directory, run:

```bash
> harness.exe --modulefile module/model.so --program application/asmtest.OR1K.elf --trace
```

You should see the following output:

```
...  
Info 'simpleCpuMemory/cpu1', 0x0000000000010000(_start): l.addi r1,r0,0x0  
Info 'simpleCpuMemory/cpu1', 0x0000000000010004(_start+4): l.addi r2,r0,0x1  
Info 'simpleCpuMemory/cpu1', 0x0000000000010008(_start+8): l.lwz r3,0x0(r2)  
Processor Exception (PC_PRX) Processor 'simpleCpuMemory/cpu1' 0x10008: l.lwz r3,0x0(r2)  
Processor Exception (PC_RAX) Misaligned 4-byte read from 0x1  
...  
```

The two lines:

```
Processor Exception (PC_PRX) Processor ' u1/cpu1' 0x10008: l.lwz r3,0x0(r2)  
Processor Exception (PC_RAX) Misaligned 4-byte read from 0x1  
```

shows that the load from address `0x00000001` has been detected as an unaligned load.

When this happens, `opRootModuleSimulate` returns the processor object handle (to indicate that simulation stopped abnormally).

Calling `opProcessorStopReason` on the processor handle returned by `opRootModuleSimulate` would return `OP_SR_RD_ALIGN` in this case, which can be printed from the value using `opStopReasonString`.

### 13.2 Enabling simulation of exceptions with overrides on command line

Now enable simulation of exceptions by adding the parameter override to the command line as shown:

```bash
> harness.exe --modulefile module/model.so --program application/asmtest.OR1K.elf --trace \  
   --override simpleCpuMemory/cpu1/simulateexceptions=1
```

You should see the following output:

```
...  
Info 'simpleCpuMemory/cpu1', 0x0000000000010000(_start): l.addi r1,r0,0x0  
Info 'simpleCpuMemory/cpu1', 0x0000000000010004(_start+4): l.addi r2,r0,0x1  
Info 'simpleCpuMemory/cpu1', 0x0000000000010008(_start+8): l.lwz r3,0x0(r2)  
Info 'simpleCpuMemory/cpu1', 0x0000000000010100(.text+200): l.j 0x00010024  
Info 'simpleCpuMemory/cpu1', 0x0000000000010204(.text+204): l.nop 0x0  
Info 'simpleCpuMemory/cpu1', 0x0000000000010024(exit): l.addi r1,r2,0x0  
Processor 'simpleCpuMemory/cpu1' terminated at 'exit', address 0x10024  
...  
```
In this example we see the load instruction is executed:

\[
\text{Info 'cpu1', 0x0000000000010000: l.lwz r3,0x0(r2)}
\]

This causes the address of the next instruction executed to be at a processor exception address, 0x00000200, which is the address of the alignment exception handler in the OR1K processor:

\[
\text{Info 'cpu1', 0x0000000000000200: l.j 0x00010024}
\]

In this example, the code at the exception vector simply branches to the exit label, which exits simulation.

### 13.3 Enabling simulation of exceptions in modules

The simulation of exceptions could alternatively be enabled when the module is created or as an override in the harness.

To enable the simulation of exceptions when adding a processor instance in a module definition simply add the `-simulateexceptions` argument to the `ihwaddprocessor` command in the file `module/module.op.tcl`:

```
ihwaddprocessor -instancename cpu1 -vendor ovpworld.org ... -type or1k ... \
    -variant generic \ 
    -simulateexceptions
```

After making this change we see the same behavior as in the previous section when enabling exceptions from the command line:

```
> make -C module
# iGen Create OP MODULE module
# Host Compiling Module obj/Linux32/module.o
# Host Linking Module object model.so
> harness.exe --modulefile module/model.so --program
application/asmtest.OR1K.elf --trace
...
Info 'simpleCpuMemory/cpu1', 0x0000000000010000(_start): l.addi r1,r0,0x0
Info 'simpleCpuMemory/cpu1', 0x0000000000010004(_start+4): l.addi r2,r0,0x1
Info 'simpleCpuMemory/cpu1', 0x0000000000010008(_start+8): l.lwz r3,0x0(r2)
Info 'simpleCpuMemory/cpu1', 0x0000000000000200(.text+200): l.j 0x00010024
Info 'simpleCpuMemory/cpu1', 0x0000000000000204(.text+204): l.nop 0x0
Info 'simpleCpuMemory/cpu1', 0x0000000000010024(exit): l.addi r1,r2,0x0
Processor 'simpleCpuMemory/cpu1' terminated at 'exit', address 0x10024
...
```

### 13.4 Enabling simulation of exceptions in a C harness

If you have created a C harness, you can enable the simulation of exceptions by using an override on the processor instance:

```
opParamBoolOverride(mi, "simpleCpuMemory/cpu1" OP_FP_SIMULATEEXCEPTIONS, 1);
```
14 Semihosting and Intercept Libraries

Imperas *semihosting* allows the default behavior of specified functions or instructions to be modified using a semihosting shared object library that is loaded by the simulator in addition to the processor model.

**14.1 Selecting semihost libraries**

Take a local copy of the *simpleSemihosting* example to edit:

```
> cp -r $IMPERAS_HOME/Examples/SimulationControl/simpleSemihosting .
> cd simpleSemihosting
```

(This a based on the example *SimulationControl/minimalHarness*)

**14.1.1 Specifying a semihost library in the module.op.tcl**

If you look at the *module/module.op.tcl* we can see that for the instance of the processor we already have a semihosting library installed.

```
ihwaddprocessor -instancename cpu1 \
    -vendor ovpworld.org -library processor -type or1k -version 1.0 \ 
    -semihostname or1kNewlib \ 
    -variant generic
```

The *or1kNewlib* semihost library intercepts low-level standard newlib functions such as *read, write, open, close, ...* and makes the application's stdout come to the simulator console for example.

This semihost library will intercept functions such as *open, read, write* etc. so if we have a C application that uses *printf* the output will appear on the host stdout.

Note that *OVPsim* only allows one intercept library, and so in this example to run with *OVPsim* we would need to remove the specification of *or1kNewlib* on the cpu1 instance by editing the *module/module.op.tcl*.

In this example we will add a second new semihost library and assume we are using *CpuManager*, the professional simulator that does not have the intercept library limitation.

Text from all semihost libraries output can be separated from other simulator output by directing it to a file using the command line flag:

```
-semihostlogfile <filename>
```

**14.1.2 An assembler test program**

As a simple first example of semihosting, we have written a simple assembler test, and we have defined a global label, *exit*, on the last instruction of the assembler test:
> cat application/asmtest.S

```
> global _start
_start:
  l.addi r1,r2,0
  ....
  l.muli r1,r2,0
> global exit
exit:
  l.addi r1,r2,0
```

### 14.1.3 Adding imperasExit semihost library to harness.c

This label can be used in conjunction with a standard Imperas semihosting shared object library, whose source is at the following location in the Imperas VLNV library:

```
$IMPERAS_HOME/ImperasLib/source/ovpworld.org/modelSupport/imperasExit/1.0/model
```

This semihosting library terminates simulation immediately after any instruction labeled `exit`. To use the semihosting library we will need to add it to the `harness.c`

```c
optModuleP mi = opRootModuleNew(0, 0, 0);
optModuleP u1 = opModuleNew(mi, "module", "u1", 0, 0);

optProcessorP cpu1 = opObjectByName(u1, "cpu1", OP_PROCESSOR_EN).Processor;
const char *shpath = opVLNVString(0, "ovpworld.org", "modelSupport",
  "imperasExit", 0, OP_EXTENSION, 1);
optProcessorExtensionNew(cpu1, shpath, "sh1", 0);
```

First the root module is created and then a module is instanced in it.

Then we use `opObjectByName` to get a pointer to the processor in the module `u1`. We set up a string for the VLNV path to find the `imperasExit` library. Finally, we add the extension library to `cpu1`, and get the message about `exit` being intercepted to terminate the simulation.

```
> ./example.sh
...
OVPsim started: Thu Jan  7 11:54:24 2016
Processor u1/cpu1' terminated at 'exit', address 0x10000bc
OVPsim finished: Thu Jan  7 11:54:24 2016
...
```
15 Imperas built-in Application Intercepts

Imperas provide a number of functions that can be used in applications that are to run on processor models in the Imperas or OVP simulation environments. These are defined in the header file found at

```
$IMPERAS_HOME/ImpPublic/include/target/application/simulatorIntercepts.h
```

And shown here are the provided functions:

```c
// // Find the processor id from the platform configuration
// int impProcessorId(void);

// // Get the number of simulated instructions
// int impProcessorInstructionCount(void);
```

As can be seen, these are intended to be used to add some reporting into an application that is running on one or more processors and allow the application to identify the processor it is running on and the progress it has made, for example a multicore simulation to test the execution of the algorithm.

To compile the application the linker must include the Imperas library ‘libimperas’ that is provided for a specific toolchain as part of the CrossCompiler installation.

In this example we are utilizing the OR1K processor and the OR1K CrossCompiler, so we will see the link line include ‘-limperas’ to include the library that is found by specifying the library directory ‘-LImperas/lib/Linux32/TargetLibraries/or32-elf’

```
# Linking application.OR1K.elf
Imperas/lib/Linux32/CrossCompiler/or32-elf/bin/or32-elf-gcc  \
  -LImperas/lib/Linux32/TargetLibraries/or32-elf  \
  Imperas/lib/Linux32/TargetLibraries/or32-elf/crt0.o  \
  -o application.OR1K.elf application.o -limperas -lm
```

To enable the use of these function intercepts they must be enabled in the simulator. This can be done

1) by specifying `--imperasintercepts` on the command line, or
2) by setting a parameter on the processor instance in the harness, using
   i. `--enableintercepts` on the `ihnew` command in the module definition
   ii. Setting `OP_FP_ENABLEIMPERASINTERCEPTS` in the harness when instancing the module

1 Other functions are provided but are deprecated and should no longer be used
The example `enablingImperasApplicationIntercepts` shows these options being used.

Take a copy of the example:

```bash
> cp -r $IMPERAS_HOME/Examples/SimulationControl/enablingImperasApplicationIntercepts .
> cd enablingImperasApplicationIntercepts
```

When the example is run using the `example.sh` script the harness is executed three times:

1) without Imperas application intercepts enabled
2) with Imperas application intercepts enabled using parameters in the harness
3) with Imperas application intercepts enabled using the standard simulator command line argument `-imperasintercepts`

# not enabled
```
> harness/harness.$IMPERAS_ARCH.exe --program application/application.OR1K.elf
...
Run: Imperas Intercepts not enabled
** SIMULATOR FAILED TO INTERCEPT 'impProcessorId'
** Check Simulator Initialized with Application Intercepts Enabled
** SIMULATOR FAILED TO INTERCEPT 'impProcessorInstructionCount'
** Check Simulator Initialized with Application Intercepts Enabled
APP: Processor Id 0: instructions executed 0
APP: Processor Id 0: Hello world application
** SIMULATOR FAILED TO INTERCEPT 'impProcessorInstructionCount'
** Check Simulator Initialized with Application Intercepts Enabled
APP: Processor Id 0: instructions executed 0
...
```

# enable using configuration of CPU instance in module
```
> harness/harness.$IMPERAS_ARCH.exe --program application/application.OR1K.elf -
   --configurecpuinstance
...
Run: Imperas Intercepts enabled by configuration of processor instance in module
Info (harness) Adding Imperas intercept overrides to cpu instance u1/cpu1
APP: Processor Id 5: instructions executed 56
APP: Processor Id 5: Hello world application
APP: Processor Id 5: instructions executed 6211
...
```

# enable with standard command line parser
```
> harness/harness.$IMPERAS_ARCH.exe --program application/application.OR1K.elf -
   -imperasintercepts
...
Run: Imperas Intercepts enabled by Standard command line
APP: Processor Id 5: instructions executed 56
APP: Processor Id 5: Hello world application
APP: Processor Id 5: instructions executed 6211
...
```
16 Simulator Control Files

The Imperas simulators can be controlled by providing command line arguments to the command to run the simulation. You can see these with:

```
> harness.exe --help
```

You can alternatively put these commands in a simulator control file.

A simulator control file allows control of extension libraries, overrides, application programs, modules, and model commands in environments that do not have a simulator command line. It also allows the substitution of one VLNV reference with another, provided the function `opVLNVString` is used to obtain the path.

For details of the control file refer to the OVP_Control_File_User_Guide.

These commands can be placed in a simulation control file. An example being:

```
# A simulation control file "control.ic"

-extlib platl/cpul/ext1=vapTools
-override platl/cpul/ext1/functiontrace on

-objfileuseentry platl/cpul2=boot.elf # boot code
-symbolfile platl/cpul=linux.elf # symbols for linux
-callcommand platl/cpul/dumpTLB # show initial contents of TLB

# end of file
```

16.1 Specify use of Control File on the simulator command line

A control file can be specified to the Imperas simulators using the command line argument `-controlfile`. This argument can be repeated to specify multiple control files. e.g.

```
> harness.exe --lvnvname MipsMalta \
   -controlfile control1.ic \
   -controlfile control2.ic
```

16.2 Specify use of Control File using IMPERAS_TOOLS

Control files can be specified using the environment variable `IMPERAS_TOOLS`. Filenames are separated by ': (Linux) or ';' (Windows) This method can be used with CpuManager when the simulator is a shared object that has no command line e.g.

```
> export IMPERAS_TOOLS="control.ic:control2.ic"

> mySystemC.exe # simulator using CpuManager
```
16.3 Specify use of Control File in a C harness

The loading of control files can be controlled from within the C harness. Control files are loaded using the file command parser function `opCmdParseFile` which must be invoked in `main` after calling the new command line parser function `opCmdParserNew`. The command line parser `opCmdParseArgs` should also be called.

```c
int main(int argc, const char *argv[]) {
    opSessionInit(OP_VERSION);

    optCmdParserP cmd = opCmdParserNew(argv[0], OP_AC_ALL);

    opCmdParseArgs (cmd, argc, argv);
    opCmdParseFile (cmd, "control.ic");

    optModuleP mi = opRootModuleNew(0, 0, 0);
    opModuleNew(mi, "module", "u1", 0, 0);

    opRootModuleSimulate(mi);

    opSessionTerminate();
    return 0;
}
```

You can use multiple control files by making multiple calls to `opCmdParseFile`. 
17 Loading an Application Program file

The simulators load application programs into a memory, onto a bus or using a processor virtual memory mapping to the correct virtual address space occupied by a memory or other device.

When a program is loaded onto a processor the program counter of the processor can be set to the start (entry point) address defined in the program or otherwise will default to the processor reset vector address. By setting the processor program counter to the start address of the program the need for correct reset vector code is circumvented allowing just the program (must be baremetal) to be executed, typically from \_start to \_exit. This gives an ideal way of testing how well a C algorithm ports, including Cross Compiler tool chain, between device architectures.

NOTE: As an alternative the function opProcessorPCSet can be used in the harness code to set the PC to the initial execution address.

The simulator loader takes each section address from the application program file and looks for memory which decodes at that address. An error is raised if no memory is mapped at a load address.

The loader uses any address decoding available on the bus, even if the decoded memory is shared with other processors.

There is currently built in support for the ELF, COFF\(^2\) and TI-COFF\(^3\) file formats to load both the program and symbolic information, if present, into the simulator. Alternatively an API is provided allowing a new ‘object loader’ to be created that can be used to load any file type.

The following sections introduce what is available and these are illustrated in the loading sections later.

17.1 Load Address Support

When loading a program onto an instance there are two addressing modes that can be used:

1) Virtual Memory Addressing (VMA)
2) Load Memory Addressing (LMA)

There is also a further modifier ‘Load Physical’ that can be applied to either of these load address modes, when used in conjunction with a processor, to bypass the processor MMU (virtual to physical memory mapping).

\(^2\) The COFF file formats are only supported by the Imperas Professional products, OVP supports only the ELF file format.

\(^3\) TI-COFF is a COFF format extended by TI and used by TI toolchains.
17.1.1 LMA vs VMA

Each section in the loadable file has two addresses.

The first, the VMA, is the address the section will have when it runs.

The second, the LMA, is the address at which the section will be loaded into memory.

In the majority of cases the two addresses, VMA and LMA, will be the same. A typical example of when they are not the same is when a data section is loaded into ROM, and then copied into RAM when the program starts up i.e. the CRT0 executes. In this case the ROM address would be the LMA and the RAM address the VMA.

17.1.2 Load Physical

When load physical is applied the load address bypasses any virtual to physical address mapping the processor creates which would be used and translate from a VMA load address to a physical address.

17.2 Program Load Options

17.2.1 Set Start Address

Setting the start address, initial Program Counter of the processor(s) in the design, to the address of the entry or start symbol in the program allows a simple C program to be executed from the C start-up code rather than from the processor's reset vector. This provides an environment, when used with a semihosting library, which to the application appears to have already been initialized.

It removes the need to execute the reset vector and some of the start-up code used to initialize memory and hardware components and assumes the processor initialization is complete but only works for basic C code ‘bare metal’ application.

It is required when using this method that a semihost library for the Cross Compiler tool chain and associated C library being used is also installed in the simulator.

There is a separate function, opProcessorPCSet(), that allows the initial Program Counter of a processor to be set independently from the loading a program.

NOTE: If multiple programs are loaded onto a single processor and if each has the option set to set the start address, it is the first that is loaded that is used.

17.2.2 Zero BSS Section

In some circumstances, particularly if not starting from the reset vector and running the memory initialization code, it may be required to zero the BSS section.

After the program has been loaded the BSS section will be cleared to zero (‘0’). This emulates the execution of crt0 startup code zeroing the BSS section when the program is being started from the application start and not executing crt0 code.
NOTE: This can mask potential startup issues i.e. if the program is expecting the BSS section to be zero for correct operation but (in error) it is not cleared by the crt0 startup code, the code may appear to work correctly in simulation but fail on real hardware as the actual values in the BSS section would be random.

17.3 Loading a Program onto Processors using the Command Line

The simulator command line allows a program to be loaded a number of ways onto a processor instance in the design; each sets specific criteria for the loading.

The main options to use are --program, --objfile and --objfileuseentry to load the program onto one or more processor instances and set the start program counter for the processor to the entry point (start address) of the application.

Alternatively, use --objfilenoentry to load the program without change to the processor program counter i.e. the typical default is to be set to the reset vector address.

Any of the loading options above can be specified with or without a named processor instance i.e. the program can be loaded onto one or more specified processors (globing is supported) or onto all processors in the platform.

17.3.1 Loading onto single or multiple processors

Example: Loading onto all processors in the design and setting the initial PC to the start address. All processors must be the same type, for example here the OR1K processor type.

> harness.exe --modulefile module/model.so \ 
  --program application/app.OR1K.elf

Example: Loading specific programs onto different processors in the design and setting the initial PC to the start address. The processors in this design are assumed to be, cpu1 an OR1K, cpu2 is an ARM and cpu3 a MIPS32.

> harness.exe --modulefile module/model.so \ 
  --program design/cpu1=application/app1.OR1K.elf \ 
  --program design/cpu2=application/app1.ARM7TDMI.elf \ 
  --program design/cpu3=application/app1.MIPS32.elf

17.3.2 Modification to Load Address

As discussed in the introduction, by default, the program is loaded using the LMA address of the program file. However, the VMA address may be selected from the program file instead by specifying the --elfusevma option. The argument may be applied to all processors or specific processors can be selected.
Example: Loading specific programs onto different processors in the design and setting the initial PC to the start address but loading one of the processors, cpu2, using the VMA.

```bash
> harness.exe --modulefile module/model.so \
  --program design/cpu1=applications/app1.CROSS_A.elf \
  --program design/cpu2=applications/app2.CROSS_B.elf \
  --elfusevma design/cpu2
```

The default operation is to apply the address (LMA or VMA) to be loaded to the processor MMU/TLB, if applicable, which performs, typically, a static translation of the program address to a physical address at which the actual load is applied on the design.

Alternatively the `--loadphysical` argument may be used to cause the load address to bypass the processor MMU and instead be applied directly to the buses connected to the processor instruction or data ports.

Example: Loading specific programs onto different processors in the design and setting the initial PC to the start address but loading one of the processors, cpu1, using physical addressing.

```bash
> harness.exe --modulefile module/model.so \
  --program design/cpu1=applications/app1.CROSS_A.elf \
  --program design/cpu2=applications/app2.CROSS_B.elf \
  --loadphysical design/cpu1
```

17.3.3 Specifying the start Address

There are two equivalent arguments to allow the start address for one or more processors to be specified, either of `--reset` and `--startaddress` can be used to set the initial program counter for one of more processors in a design.

Example: Loading the same program onto all processors in the design and setting the initial PC (the start or reset address) to a specific address on one of the processors, cpu1, using `--startaddress`

```bash
> harness.exe --modulefile module/model.so \
  --program applications/app1.CROSS_A.elf \
  --startaddress design/cpu1=0x00100000
```

and the equivalent using `--reset`

```bash
> harness.exe --modulefile module/model.so \
  --program applications/app1.CROSS_A.elf \
  --reset design/cpu1=0x00100000
```
17.3.4 Provide Arguments to Applications

An application can be provided arguments, as though it was being executed from a command line within an operating system, using \texttt{--argv} which defines ALL following arguments as those to be passed to the application executing on the processor instance(s).

For example, an application (app.c) compiled with GCC could be executed and passed two arguments, that are printed:

Example application source

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    int i;
    for (i=1;i<argc;i++) {
        printf("APP: arg[%u] = %s\n", i, argv[i]);
    }
    return 0;
}
```

Compiled and executed on host machine (for example Linux)

```
> gcc app.c -o app.exe
> app.exe yes no
APP: arg[1] = yes
APP: arg[2] = no
```

The application can be Cross Compiled to any target processor and executed on that processor type in a virtual platform passing the same arguments. Now we have the ability to test, for example, our C algorithm using the correct tool chain and libraries, hence the same binary image as we can execute on the hardware but in the convenience of a virtual platform on the host machine.

```
> make -C application CROSS=OR1K
# Compiling
# Linking

> harness.exe --modulefile module/model.so \ 
    --program applications/app.OR1K.elf \ 
    --argv yes no
APP: arg[1] = yes
APP: arg[2] = no
```

17.4 Common Program Memory

If more than one processor is using the same code memory, the program need be loaded only once; When the simulator starts a processor with no explicitly loaded program, it will look for any other processors of the same architecture with common program memory and, if one is found, use the start address associated with that processor.
17.5 Loading an application from the harness

Application files can be loaded using the C API within a harness. The application program can be loaded onto a processor, a bus or a memory using the three functions opProcessorApplicationLoad, opBusApplicationLoad and opMemoryApplicationLoad respectively.

The load control is defined in each of these functions using the optLoaderControls enumerated type, as defined in the following.

17.5.1 Application Loader Controls

Each of the application load functions is passed optLoaderControls to control how the file is loaded. The values of the enumerated type and their descriptions follow:

- **OP_LDR_DEFAULT**: No special features
- **OP_LDR_ELF_USE_VMA**: Load the application file using the VMA addresses rather than LMA addresses (default) from the file.
- **OP_LDR_PHYSICAL**: (processor only) Do not use the processor MMU to translate the load addresses. Load directly to the addresses specified.
- **OP_LDR_SET_START**: (processor only) Set the PC of the target processor to the start address.
- **OP_LDR_NO_ZERO_BSS**: Do not zero the BSS section after loading.
- **OP_LDR_SYMBOLS_ONLY**: No program will be loaded i.e. no modification to the system, only the symbolic information will be loaded.
- **OP_LDR_FAIL_IS_ERROR**: If an error occurs the simulation will not execute. By default even if only a partial program is loaded correctly the simulation will start execution.
- **OP_LDR_VERBOSE**: Loader will report sections as they are loaded

One or more of the options may be used when a program is loaded.

17.5.2 Onto a Processor Instance

Use opProcessorApplicationLoad to load an application elf file onto a processor.

```
procLocal = opObjectByName(mi, "ul/cpu1", OP_PROCESSOR_EN ).Processor;
const char *myApp = "application/app.OR1K.elf";
opProcessorApplicationLoad (procLocal, myApp, OP_LDR_DEFAULT , 0);
```

17.5.3 Onto a Bus Instance

Use opBusApplicationLoad to load an application elf file onto a bus.
17.5.4 Onto a Memory Instance

Use `opMemoryApplicationLoad` to load an application elf file onto a memory.

```
memoryLocal = opObjectByName(mi, "u1/mem1", OP_MEMORY_EN ).Memory;
const char *myApp = "application/app.OR1K.elf";
opMemoryApplicationLoad (memoryLocal, myApp, OP_LDR_DEFAULT , 0);
```

17.6 Getting Information of Loaded Applications

Each of the application load functions return an application object pointer that can be used to interrogate the loaded application for:

- the attributes with which the application was loaded, `opApplicationControls`
- the elf code for the application pointed to by the object, `opApplicationElfCode`
- the data endian used in the compilation of the application, `opApplicationEndian`
- the entry address for the application, `opApplicationEntry`
- the path from which the file was loaded, `opApplicationPath`

There is a functions to iterate across all the application objects that may be loaded onto each instance type, processor, bus or memory; `opObjectApplicationNext`.

17.7 Example

The example `Examples/SimulationControl/loadingApplicationPrograms` contains the ability to load an application onto a processor, a memory or a bus instance and to find the loaded applications and report their information.

Take a local copy to edit:

```
> cp -r $IMPERAS_HOME/Examples/SimulationControl/loadingApplicationPrograms .
> cd loadingApplicationPrograms
> make -C application
> make -C module
> make -C harness
```

See the file `harness/harness.c`, which can be controlled to perform the loading of the application program onto the different instance types in the module by using the command line.

Load application using the command line argument `--program`:
Load the application using the processor instance in the harness:

```c
if (options.loadprocessor) {
    // load on processor, setting start address
    opProcessorApplicationLoad(
        proc, options.application, OP_LDR_VERBOSE|OP_LDR_SET_START, 0
    );
}
```

which executes the code in `harness.c`:

Load the application using the memory instance in the harness:

```c
if (options.loadmemory) {
    // load on first Memory (assumes a single memory module)
    optMemoryP mem = opMemoryNext(ui, NULL);
    thisApp = opMemoryApplicationLoad(mem, options.application,
        OP_LDR_VERBOSE, 0);
}
```

Which executes code in `harness.c` which also must set the start PC for the processor to the entry point of the application:

```c
if (thisApp) {
    // Loaded on bus or memory so set start address manually
    opProcessorPCSet(proc, opApplicationEntry(thisApp));    // set start address to application entry
}
```

Load the application using the bus instance in the harness:

```c
if (options.loadbus) {
    // Must advance to next phase for the API calls that follow
    opRootModulePreSimulate(mi);
    if (thisApp) {
        // Loaded on bus or memory so set start address manually
        opProcessorPCSet(proc, opApplicationEntry(thisApp));    // set start address to application entry
    }
}
```
// load on first Bus (assumes a single bus module)
optBusP bus = opBusNext(mi, NULL);
thisApp = opBusApplicationLoad(bus, options.application, OP_LDR_VERBOSE, 0);

// Must advance to next phase for the API calls that follow
opRootModulePreSimulate(mi);
if (thisApp) {
    // Loaded on bus or memory so set start address manually
    opProcessorPCSet(proc, opApplicationEntry(thisApp));    // set start address to application entry
}

When the argument --analyzeinstances is used it will report information as shown below about the application loaded onto the instance:

// use a custom argument to control if we analyze the application programs loaded on the instance
if ( options.analyzeinstances) {
    optObjectP object;
    if ( options.loadmemory) {
        object = opObjectByName(
            mi, MODULE_INSTANCE "/" MEM_INSTANCE, OP_MEMORY_EN
        );
    } else if ( options.loadbus) {
        object = opObjectByName(
            mi, MODULE_INSTANCE "/" BUS_INSTANCE, OP_BUS_EN
        );
    } else {
        object = opObjectByName(
            mi, MODULE_INSTANCE "/" CPU_INSTANCE, OP_PROCESSOR_EN
        );
    }

    opMessage("I", MODULE_INSTANCE, "Analyzing Loaded Applications");
    optApplicationP thisApp = NULL;
    while ((thisApp=opObjectApplicationNext(object, thisApp))) {
        optLoaderControls controls = opApplicationControls(thisApp);
        Uns32 elfCode = opApplicationElfCode(thisApp);
        optEndian endian = opApplicationEndian(thisApp);
        Addr entry = opApplicationEntry(thisApp);
        const char *path = opApplicationPath(thisApp);

        opMessage(
            "I", MODULE_INSTANCE,
            "%s : controls %x: elf 0x%x: endian %u: entry 0x" FMT_Ax ": path
            's'\n",
            opObjectHierName(object),
            controls,
            elfCode,
            endian,
            entry,
            path
        );
    }
    }
}
17.8 Loading Program Symbolic information

This is required for debugging and also for the Imperas tools to be able to analyze the executing application.

When the standard command line options or C API functions are used to load a program into simulated memory, the loader also reads the symbol tables included in the file, and records the address-to-symbol mappings. These mapping can then be used:

- When issuing tracing information
- When intercepting a function by name

Sometimes object code may be loaded by another method (e.g. using a boot-loader running on a simulated processor) in which case the simulator has no opportunity to read the symbols. In this situation we can separately associate symbols with a processor without loading the code.

17.8.1 Command Line

Using --symbolfile can be used to select an elf file from which the symbolic information can be read and stored as shown in the following example (that also assumes the use of an alternative loader for an s-record file format without symbolic information)

```bash
> harness.exe --modulefile module/model.so \
   --loadsrecord applications/app.OR1K.srec \
   --symbolfile applications/app.OR1K.elf
```

17.8.2 In a Harness

The function opProcessorApplicationLoad can be used to load the symbolic information from a file. It also allows specific symbol type to be separately specified. When using the program load function the optLoaderControls is set to OP_LDR_SYMBOLS_ONLY so that only the symbolic information is loaded. Alternatively individual symbols can be defined and added to the simulator using the opProcessorApplicationSymbolAdd function.

17.9 Defining a hardware resident program

There may be a program that is always present in the design, for example, a boot loader program, that is always executed (by default) at startup. A program can be defined when a design is created i.e. in the module definition by attaching it to the processor, bus, memory or other instance using ihwaddimage in the iGen script that defines the module.

```
ihwaddimage --handle cpuInstance --filename bootloader.OR1K.elf --name bootloader
```
17.10 Custom Application Loaders

If you wish to load application files of a format not supported by CpuManager, the Imperas Professional Simulator, you can write your own loader and install it in the simulator.

An application file can be tested in a harness with `opApplicationHeaderRead` to see if a specific file format could be read. This function will test the built-in loaders and any installed custom loaders to see if any accept the file type.

A new reader is installed using `opApplicationLoaderInstall`. Take a local copy of the custom application loader example:

```bash
> cp -r $IMPERAS_HOME/Examples/SimulationControl/customApplicationLoader .
> cd customApplicationLoader
> make -C application
> make -C loader
> make -C module
> make -C harness
```

This example creates a Motorola S record file (.s19 suffix) from a compiled ‘Hello World’ application. The custom application loader is created to load a Motorola S record file into memory, including the loading of symbolic information.

The custom loader is found in the loader directory.

In `harness/harness.c` the custom application loader is installed before the processor memory is loaded (otherwise the loader will not be available).

```c
// install the new application loader
const char *loader = "loader/loader." IMPERAS_SHRSUF;
if (!opApplicationLoaderInstall(loader)) {
    opMessage("F", HARNESS_NAME, "Failed to load application loader '%s'",
    loader);
}
```

The simulator tries to read the file "hello.OR1K.s19" using the new loader before trying the built-in loaders.

```bash
> harness/harness.$IMPERAS_ARCH.exe --program application/hello.OR1K.s19
... Info (harness) Loaded application loader 'loader/loader.so'
Hello
...
```

17.10.1 Writing a custom reader

This is covered in the Imperas document Custom_Object_Reader.pdf.

17.11 Custom Loader using Standard Object Reader
opProcessorApplicationRead works in a similar way to the opProcessorApplicationLoad but instead of using one of the installed object readers to load directly it calls a set of callback functions for each section that is identified in the application to load. This allows custom actions for each section of the application file while using the standard readers to read them from the file.

Each of the following callbacks must be provided by the user:

<table>
<thead>
<tr>
<th>Callback</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordSectionFn</td>
<td>listSection</td>
</tr>
<tr>
<td>ordLoadMemoryFn</td>
<td>loadSection</td>
</tr>
<tr>
<td>ordAddSymbolFn</td>
<td>addSymbol</td>
</tr>
<tr>
<td>ordPrintFn</td>
<td>print</td>
</tr>
<tr>
<td>ordMessageFn</td>
<td>message</td>
</tr>
</tbody>
</table>

Please see the document Imperas_Custom_Object_Reader, that is provided with the Imperas professional product packages, for detailed information on how this should be used.

Please contact Imperas if you wish to write your own custom loader using the standard object reader and an example may be provided to you that is not part of the normally published source code.
18 Attaching a debugger

18.1 Introduction

The simulator can be connected to a debugger sockets (socket is a Unix term though sockets are also supported on Linux and Windows).

The debugger could be one of

1) Standard processor GDB
2) Imperas MPD (Multi-Processor Debugger)
3) Imperas iGui or eGui; GUI based debugging interfaces utilizing GDB or MPD
4) Any other debugger that supports the GDB Remote Serial Protocol through a socket.

Depending upon the simulator that is being used, OVPsim or CpuManager, different feature and connection options are supported.

The debug connection can be used:

1) for the debug of application code on processors;
2) the debug and development of peripheral model behavioral code;

If the Imperas Multi-processor debugger\(^4\) is being used it can also:

3) debug of a C platform definition;
4) debug and development of binary interception and tool libraries;
5) debug the native semihost behavior of a peripheral model

When the OVPsim simulator is being used only a single processor may be selected for application debugging; when CpuManager is being used up to 8 processors\(^5\) can be simultaneously debugged, using multiple GDB Remote Serial Protocol

It is required to have a version of GDB specific to the target processor. The OVPworld web site can supply a GDB for most processor models available.

The debug port can be configured so that it waits for a connection (or multiple connections) to be made before starting simulation or to start simulation without waiting and then allow subsequent connections.

18.2 Debug with GDB

A GDB debugger is connected to a single processor instance in the simulation platform. Multiple GDB debuggers can be used to connect to one or more processors. This allows

---

\(^4\) Also requires the availability of a compatible host GDB debugger.

\(^5\) This figure is chosen as a realistic maximum number of processors that it is possible to debug interactively with a console per processor. There is no limit to the number of processor applications that could be debugged simultaneously.
standard processor centric visibility of the software running on each of the processors and
its debugging.
NOTE: When using multiple GDBs the user must ensure that ALL the processors are in
‘run’ mode in order for the simulation to move forward. If one (or more) GDB debuggers
are stopped at an interactive prompt the simulator will remain blocked.

18.2.1 Starting GDB debug session on a single processor
To start a GDB debug sessions on a single processor system is very easy. It can be done
from the command line, when instancing a module, or in the harness using parameters.

18.2.1.1 Command line
A debug session is started from the command line using the --gdbconsole argument. This
can be added to any command line and will start a GDB session, using the processor's
default GDB, for each processor in the design.

```bash
> harness.exe --modulefile module/model.so --program application/application.OR1K.elf --gdbconsole
```

As an argument to --gdbconsole a rootmodule may be provided. This allows the selection
of the processor within one of potentially several rootmodules, that may exist in the
harness, to be debugged.

18.2.1.2 Root module instance
Alternatively, the debug of processor applications can be started using
$OP_FP_GDBCONSOLE$ applied as a parameter to the rootmodule of a design.

```c
//enable debugging
mi = opRootModuleNew(0, HARNESS_NAME,
   OP_PARAMS (OP_PARAM_BOOL_SET($OP_FP_GDBCONSOLE$, 1)
 );
```

18.2.2 Selecting Processor(s) to debug with GDB
When there are a number of processors in the design and only a specific selection of them
should be debugged the --debugprocessor argument is used to specify each instance of
the processor to be debugged.

18.2.2.1 Using command line options
For example, if we have a number of processors in the design, cpu0 to cpu7 and we wish
to debug the applications only on cpu1 and cpu4 we would modify the command line as
shown:

```bash
> harness.exe --modulefile module/model.so --program application/application.OR1K.elf --gdbconsole
   --debugprocessor u1/cpu1
   --debugprocessor u1/cpu4
```

18.2.2.2 In a C harness
In a C harness we can make the same selection using the opProcessorDebug function.
This is applied to each processor instance in the design that we wish to debug. The
following example shows selecting the processor instances which we know by name within the hierarchy

```c
// select the processor to debug
optProcessorP proc;
proc = opObjectByName(mi, "u1/cpu1", OP_PROCESSOR_EN).Processor;
opProcessorDebug(proc);
proc = opObjectByName(mi, " u1/cpu7", OP_PROCESSOR_EN).Processor;
opProcessorDebug(proc);
```

Alternatively, we could search the module we have instanced for a specific matching processor name and when we get a match select the processor for debug

```c
// enable debugging
mi = opRootModuleNew(0, 0, OP_PARAMS(OP_PARAM_BOOL_SET(OP_FP_GDBCONSOLE, 1)));
optModuleP u1 = opModuleNew(mi, MODULE_DIR, MODULE_INSTANCE, 0, 0);

optProcessorP proc = NULL;
while ((proc = opProcessorNext(u1, proc))) {
    opPrintf("found %s\n", opObjectName(proc));
    if (strcmp(CPU_INSTANCE, opObjectName(proc)) == 0) {
        opProcessorDebug(proc);
    }
}
```

18.2.3 Configuring GDB
The GDB that is used and its basic configuration for each specific variant is held within each processor model. In some circumstances it may be required to change the default initialization, add additional configuration or run some common commands at startup.

The following can be specified, on the command line or as part of the processor instance configuration in a harness.

Every processor model supplied by Imperas has been tested with the gdb supplied in the appropriate GNU toolchain. This gdb is usually in your Imperas installation and its path embedded in the processor model. The simulator uses this path to select the correct gdb for the model automatically. However, it is possible to override this mechanism. Select an alternate GDB to be used instead of the default GDB by specifying the full path the GDB executable with `--gdbpath` or `OP_FP_ALTERNATEGDBPATH`.

To run a set of commands on the GDB, after GDB initialization a command file can be specified that provides a list of commands that are executed before the GDB prompt is displayed with `--gdbcommandfile` or `OP_FP_GDBCOMMANDFILE`.

To initialize the GDB before the prompt is displayed a file can be provided with `--gdbinit` or `OP_FP_GDBINIT`.

Special flags may be added onto the command line of the invoked GDB with `--gdbflags` or `OP_FP_ALTERNATEGDBFLAGS`.
For example, to specify an alternative GDB and provide a custom initialization file the following could be used on the command line

```bash
> harness.exe --modulefile module/model.so --program application/application.OR1K.elf --gdbconsole \
  --debugprocessor ul/cpu1 \ 
  --debugprocessor ul/cpu4 \ 
  --gdbpath ul/cpu1=/path/to/my/new/gdb/gdbexe \ 
  --gdbinit ul/cpu1=gdb.init
```

Or applied on the processor in the harness using

```c
// set a new GDB to use
opParamStringOverride(mi, "ul/cpu1/" OP_FP_ALTERNATEGDBPATH, "/path/to/my/new/gdb/gdbexe");
// set new initialization file
opParamStringOverride(mi, "ul/cpu1/" OP_FP_GDBINIT, "gdb.init");
```

### 18.2.4 Using GDB with Graphical User Interfaces

Imperas provide two graphical user interfaces, first is iGui which is based upon TK and the second is eGui, based upon the Eclipse IDE.

One of the use methods, that is described in this section, is for the GUIs to be invoked by the simulator. In GDB mode the GUI will communicate with one or more GDBs which will then communicate to the Imperas simulator. When the OVPsim simulator is used the --debugprocessor argument must be used to specify which processor is to be debugged. When the CpuManager simulator is used, if the number of processor exceeds the number of connections available, the --debugprocessor argument must be used to specify which processors are to be debugged.

To start from the command line, using iGui, add --gdbigui to the command line and for eGui add --gdbegui.

This can also be done in the harness by specifying parameters on the rootmodule, for iGui

```c
mi = opRootModuleNew(0, 0, OP_PARAMS (OP_PARAM_BOOL_SET(OP_FP_GDBIGUI, 1)));
```

And for eGui

```c
mi = opRootModuleNew(0, 0, OP_PARAMS (OP_PARAM_BOOL_SET(OP_FP_GDBEGUI, 1)));
```

### 18.3 Debug with MPD

MPD is the Imperas Multi-Processor Debugger, provided with the M*SDK product package. It allows applications to be debugged on any of the processors within the design and also the debug and development of peripheral model behavioral code within a single
consistent environment. It is described in more detail, in the document “Imperas Debugger User Guide”

The MPD can be started in a separate console using --mpdconsole from the command line or by adding OP_FP_MPDCONSOLE to a root module in the harness.

```
> harness.exe --modulefile module/model.so --program application/application.OR1K.elf --mpdconsole
```

Please refer to the Imperas Multiprocessor Debugger User Guide.

### 18.3.1 Configure MPD

#### 18.3.1.1 Debug Peripheral model constructors

By default, when debugging the Peripheral (PSE), the constructors are executed before getting to a debug prompt. The peripheral constructors generate the peripheral model programmers view information so until these are executed no information regarding the peripheral is available to MPD.

However, you may wish to debug the code from the peripheral model constructors. This is enabled by adding the flag --debugpseconstructors to the command line or by adding OP_FP_DEBUGPSECONSTRUCTORS to a root module.

```
> harness.exe --modulefile module/model.so --program application/application.OR1K.elf --mpdconsole --debugpseconstructors
```

#### 18.3.1.2 Add to source search path

The debug information contained in a file will indicate the source location at which the file was originally located when compiled. If the source has been moved it is possible to indicate alternative locations to search for source using the --searchpath argument.

```
> harness.exe --modulefile module/model.so --program application/application.OR1K.elf \\ --mpdconsole \\ --searchpath /path/to/new/source/directory
```

### 18.3.2 Using MPD with GUI

Imperas provide two graphical user interfaces, one is iGui which is based upon TK and the second is eGui, based upon the Eclipse IDE.

One of the use methods, that is described in this section, is for the GUIs to be invoked by the simulator. In MPD mode the GUI will communicate with MPD which will then communicate to the Imperas simulator.

To start from the command line, using iGui, add --mpdigui to the command line and for eGui add --mpdegui.
This can also be done in the harness by specifying parameters on the rootmodule, for iGui

```c
mi = opRootModuleNew(0, 0, OP_PARAMS (OP_PARAM_BOOL_SET(OP_FP_MPDIGUI, 1)));
```
or for eGui

```c
mi = opRootModuleNew(0, 0, OP_PARAMS (OP_PARAM_BOOL_SET(OP_FP_MPDEGUI, 1)));
```

### 18.3.3 MPD Batch Mode

MPD provides both interactive and batch modes of operation. All commands that can be applied interactively can also be added to a file and provided for batch file operation.

When MPD is operating interactively the commands being applied into the shell are captured into a log file (`idebug.log`), this can be used as a basis for creating a batch file.

Once a batch file has completed executing, the simulation will either terminate or return to an interactive shell.

The file to execute as a batch file is specified using `--batch` on the command line

```c
harness.exe --modulefile module/model.so  --program application/application.OR1K.elf  
   --mpdconsole  
   --batch mpd.batch
```
or by applying the parameter `OP_FP_BATCH` to a root module.

```c
mi = opRootModuleNew(0, 0,  
   OP_PARAMS (OP_PARAM_STRING(OP_FP_BATCH, "mpd.batch")));
```

### 18.4 Manual Debug connection specifying port

So far we have seen that we can start a simulation and connect a debug client automatically using the command line or by configuring the root module. We can also open a specified port and manually connect a debug client.

The port is specified using the `--port` argument on the command line or the parameter `OP_FP_PORT` applied to a root module. If a value of 0 is used the next available port is selected by the OS from its pool, otherwise the nominated port will be opened if available.

If the environment variable IMPERAS_PORT_FILE is set to a filename before starting the simulation then the port number, once opened, will also be written to this file.
18.4.1 Do not wait for debug connection
The default operation is for a listening debug port to be opened and for a connection to be made (blocking mode) before simulation can progress. Alternatively, by applying --nowait to the command line or by adding the parameter OP_FP_RSPNOWAIT to a root module, simulation will progress (non-blocking mode) and a debug connection can be made by GDB or MPD at a later time.

18.5 Using Control Files
As well as specifying arguments on the command line, they may also be included in a control file. This allows, for example, a simulation platform which does not contain the Imperas command line parser to be controlled with command line arguments. The control file is defined using the IMPERAS_TOOLS environment variable and its contents are loaded and parsed directly by the simulator. Control files contain the same syntax as arguments provided on the command line and are described in more detail in the document “OVP Control File User Guide”.

18.6 Example of Debugging Applications
Take a local copy of the example:

```bash
> cp -r $IMPERAS_HOME/Examples/SimulationControl/startingApplicationDebug .
> cd startingApplicationDebug
```

The example consists of a harness, a hardware design (module) and some application software. The design comprises two processors, running application software that can be debugged.

There is a script, example.sh (.bat for Windows) that can be used to build the example and run the same debug session twice, the first time using the command line to invoke and the second time using parameters on the root module to invoke.

```bash
> make -C application
> make -C module
> make -C harness
```

In both of the following cases the gdbconsole is invoked for a single processor instance, cpu1. At the GDB command line we issue the following debugger commands

```bash
break main
break fib
continue
continue
delete 2
continue
quit
```

This results in the following GDB output

```
0x00000100 in start ()
```
Breakpoint 1 at 0xf9c: file fibonacci.c, line 32.
Breakpoint 2 at 0xf48: file fibonacci.c, line 26.

Breakpoint 1, main () at fibonacci.c:32
32     printf("APP: Fibonacci starting...\n");

Breakpoint 2, fib (i=0) at fibonacci.c:26
26     return (i>1) ? fib(i-1) + fib(i-2) : i;

Program exited normally.

When invoked from the command line

```
> harness/harness.$IMPERAS_ARCH.exe \
   --program application/fibonacci.OR1K.elf \
   --gdbconsole --debugprocessor u1/cpu1
```

```
Info (GDBT_PORT) Host: hostname, Port: 33947
Info (GDBT_WAIT) Waiting for remote debugger to connect...
Info (GDBT_CONNECTED) Client connected
APP: Fibonacci starting...
APP: fib(0) = 0
APP: fib(1) = 1
...
APP: fib(29) = 514229
APP: finishing...
APP: fib(29) = 514229
APP: finishing...
...
```

And when invoked from the root module parameter

```
> harness/harness.$IMPERAS_ARCH.exe \
   --program application/fibonacci.OR1K.elf \
   --configurecpuinstance
```

```
Info (top) Adding GDB Console override to root module and selecting u1/cpu1 for debug
found cpu1
found cpu2
Info (GDBT_PORT) Host: hostname, Port: 50795
Info (GDBT_WAIT) Waiting for remote debugger to connect...
Info (GDBT_CONNECTED) Client connected
APP: Fibonacci starting...
APP: fib(0) = 0
APP: fib(1) = 1
...
APP: fib(29) = 514229
APP: finishing...
APP: fib(29) = 514229
APP: finishing...
...
```
18.6.1 Debug all processors

18.6.1.1 Using gdbconsole
If we now invoke the following command line, in which we do not specify the processor to be debugged, we indicate that we wish to debug all the processors in the design using the chosen method, in this case a GDB Console.

```
> harness/harness.$IMPERAS_ARCH.exe \
    --program application/fibonacci.OR1K.elf \
    --gdbconsole
```

Simultaneous debug of multiple processors is only supported in the Imperas professional simulator. When we are configured to use the OVPsim simulator (IMPERAS_RUNTIME=OVPsim) the command above will result in an error indicating that only one processor can be debugged and one processor should be selected using the `-debugprocessor` option:

```
Info (GDBT_PORT) Host: lnx34.impinternal.com, Port: 42623
Error (OP_TMD) There are too many processors (2 when the maximum allowed is 1) to debug.
Suggest using -debugprocessor no more than the maximum.
Candidate processors:
    u1/cpu1
    u1/cpu2
```

When we are configured for the Imperas professional simulator (IMPERAS_RUNTIME=CpuManager), two debug consoles are started, one attached to `cpu1` and the second to `cpu2`, allowing the debug of both (all) applications.

When the same breakpoints are set on the processors, and BOTH processor GDBs are set to run mode, we see:

```
Info (GDBT_PORT) Host: lnx34.impinternal.com, Port: 44008
Info (GDBT_WAIT) Waiting for remote debugger to connect...
Info (GDBT_CONN) Client connected to 'u1/cpu1'
Info (GDBT_CONN) Client connected to 'u1/cpu2'
Info (GDBT_R) Simulator started
Info (GDBT_S) Simulator stopped
Info (GDBT_R) Simulator started
Info (GDBT_S) Simulator stopped
Info (GDBT_R) Simulator started
APP: Fibonacci starting...
APP: fib(0) = 0
APP: fib(1) = 1
... APP: fib(13) = 233
Info (GDBT_S) Simulator stopped
```

```
Info (GDBT_R) Simulator started
Info (GDBT_S) Simulator started
Info (GDBT_R) Simulator started
APP: Fibonacci starting...
APP: fib(0) = 0
APP: fib(1) = 1
... APP: fib(28) = 317811
```
APP: fib(28) = 317811
APP: fib(29) = 514229
APP: finishing...
APP: fib(29) = 514229
APP: finishing...
Info (GDBT_S) Simulator stopped

The logging information is now also showing the points at which the simulator starts and stops. This is useful in this mode because ALL GDBs must be put into run mode before the simulation can continue. In this case the points at which the simulation stops are caused by breakpoints being hit.

18.6.1.2 Using MPD with the Imperas Professional Simulator

```bash
> harness/harness.$IMPERAS_ARCH.exe \
   --program application/fibonacci.OR1K.elf \ 
   --mpdconsole
```

This starts a single debug session with both processors available, in which we can invoke commands on both processors and run the simulation with a single continue command.

Details of the use of the Imperas MPD may be found in the document *Imperas Debugger User Guide.*

---

MPD (32-Bit) 20151203.0 Multiprocessor debugger from www.IMPERAS.com.
Copyright (c) 2005-2015 Imperas Software Ltd. Contains Imperas Proprietary Information.
Licensed Software, All Rights Reserved.
Visit www.IMPERAS.com for multicore debug, verification and analysis solutions.

Info (MPD_SCS) Connecting
Info (MPD_SC) Socket connected
Info (MPD_VC) Server is compatible
idebug (cpu1) > processor
Current processor 'cpu1'
Processors in platform:
   cpu2
   cpu1
idebug (cpu2) > processor cpu1
idebug (cpu1) > break main
Created breakpoint 1 at 0x00000f9c: file fibonacci.c, line 32
idebug (cpu1) > processor cpu2
idebug (cpu2) > break main
Created breakpoint 2 at 0x00000f9c: file fibonacci.c, line 32
idebug (cpu2) > continue
Breakpoint 2 at main triggered for processor cpu2
main () at fibonacci.c:32
32        printf("APP: Fibonacci starting...\n");
idebug (cpu2) > continue

Breakpoint 1 at main triggered for processor cpu1
main () at fibonacci.c:32
32        printf("APP: Fibonacci starting...\n");
idebug (cpu1) > continue
exit (code=0) at ../../../../../../src/gcc-3.4.2/newlib/libc/stdlib/exit.c:64
64      in ../../../../../../src/gcc-3.4.2/newlib/libc/stdlib/exit.c
idebug (cpu1) > ::exit

This sequence of commands in MPD provides the following simulator output

```
... Info (GDBT_PORT) Host: lnx34.impinternal.com, Port: 42885
Info (GDBT_WAIT) Waiting for remote debugger to connect...
Info (GDBT_MPD) Client connected to platform
APP: Fibonacci starting...
APP: fib(0) = 0
APP: fib(1) = 1
APP: fib(2) = 1
APP: fib(3) = 2
APP: fib(4) = 3
APP: fib(5) = 5
... APP: fib(13) = 233
APP: Fibonacci starting...
APP: fib(0) = 0
APP: fib(1) = 1
APP: fib(2) = 1
...
APP: fib(28) = 317811
APP: fib(29) = 514229
APP: finishing...
APP: fib(29) = 514229
APP: finishing...
Info (GDBT_KILL) Client requested shutdown
...```
19 Imperas Graphical Debug Environments

Imperas provide two graphical debug environments, iGui and eGui.

19.1 Imperas iGui

iGui is an easy to use Graphical User Interface written in tcl/TK to provide very quick and easy use of debuggers. It is modeled on the DDD debugger that controls GDB and which is a standard tool within Linux distributions. iGui connects to either a standard GDB or the Imperas Multi-Processor Debugger (MPD).

The Imperas iGui is started using one of the simulator command line arguments --gdbigui or --mpdigui, as discussed briefly in section 18.2.4 “Using GDB with Graphical User Interfaces” and 18.3.2 “Using MPD with GUI”.

When initially started, only the main window will be displayed, all other windows can be displayed as required.

19.1.1 Available Display Windows

The main Window can contain a source code listing pane, an assembler (machine code) pane, a console pane and a command line input. The command line input is always...
present but the other three panes are controlled by tick-box selection under the ‘View’ menu item.

Also selected from the ‘View’ menu item is the ‘Command Tool’ window that provides all the commands to control the simulation execution, e.g. single stepping, continue execution etc.

Two status windows are available, selected under the ‘Status’ menu item.
‘Backtrace’ provides the current frame stack of the currently selected processor.
‘Registers’ provides the standard GDB view of the processor registers.

Two source windows are available, selected under the ‘Source’ menu item.
‘Files’ will provide the list of source files that are associated with the application being debugged. These may be selected to open the source code listing in the source code listing pane from which breakpoints can be inserted or removed. ‘Breakpoints’ will provide a list of the current breakpoints and allow breakpoints to be modified.

Two data windows are available, selected under the ‘Data’ menu item.
‘Display Local Variables’ will provide a list and value of all local variables of the currently selected frame. ‘Display Arguments’ will provide a list and the value of arguments passed into the current function.

Two target windows are available, selected under the ‘Targets’ menu item.
The ‘View’ window provides a list of all the targets, processors and peripherals, within the design, and shows the type and current run state. This is the window that can be used to select the current context for all other Windows i.e. by changing the selection of the component in this window the context of all the other windows changes to reflect this item.

19.1.2 Host Code

The debugger allows simultaneous debug of applications running on processors and behavioral code from peripheral models. Also in the system is the virtual platform source code itself and the native components of peripheral models and semihosting libraries.

It is possible to attach a host GDB debugger to the host code that is executing using the ‘Attach hostcode’ item from the ‘File’ menu item. Once attached the native code can also be debugged.

To switch back from debugging native code, to the application code running on the processors, or behavioral code running on the PSE, the command tool button ‘Nextsimi’ can be used. This will cause an interrupt on the next application instruction to be executed.

19.1.3 Initialization files
Selected under the ‘File’ menu item are ‘Save Ini Global’ and ‘Save Ini Local’ that are used to save your current menu layout configuration into either a global location (specified by the HOME environment variable on Linux and the user home directory on Windows, for example found in C:/Users/) or the local directory as an initialization file (.iguirc).

19.1.4 Limitations

The iGui requires that the GDB that is used supports MI version 2 or higher. Currently all GDB provided by OVP, except for the OR1K processor, are supported.

19.2 Imperas eGui (eclipse based)

The Imperas eGui product is a GUI built on Eclipse that enables you to debug your embedded systems in a standard, easy to use, coherent, use-model built on the Eclipse CDT debug interface.

eGui can be used with GDB for single core software development, or for Imperas professional tools users, eGui can control the Imperas Multi Processor Debugger. The Imperas Multi Processor Debugger enables the simultaneous debugging of host code, cross compiled software running on target CPUs, and the programmers view and source code of peripheral behavioral models.

To use eGui, you will need to have installed the eGui package. Further information can be found in the document eGui Eclipse User Guide.
The Imperas eGui is started using one of the simulator command line arguments -- 
`gdbegui` or `--mpdegui`, as discussed briefly in sections 18.2.4 “Using GDB with 
Graphical User Interfaces” and 18.3.2 “Using MPD with GUI”.

There is a user manual that describes the full capabilities available as *eGui Eclipse User 
Guide*.
20 Controlling Record and Replay of Virtualized Peripheral Input

If a peripheral model communicates with the outside world, e.g. through a real keyboard interface, a simulation may be affected by inputs which cannot be exactly reproduced in subsequent simulation sessions. This makes regression testing or reproduction of particular failures very close to impossible. To overcome this problem, the BHM API presents a simple interface to a record/replay mechanism. It is the responsibility of the peripheral model writer to use this API if replay is required and to ensure that a model using replay does appear to the rest of the system to behave exactly as in the original simulation.

Record/Replay can be controlled from the command line or with instance parameters.

20.1 Example for Record Replay operation

Take a local copy of the recordReplay example:

```bash
> cp -r $IMPERAS_HOME/Examples/SimulationControl/recordReplay .
> cd recordReplay
```

20.2 Simple CPU Memory and UART module

The module for this example is a modified version of

```tcl
ihwnew -name cpuMemUart

ihwaddbus -instancename mainBus -addresswidth 32

ihwaddprocessor -instancename cpu1 -vendor ovpworld.org -library processor -type or1k
  -version 1.0 -semihostname or1kNewlib -variant generic
ihwconnect -bus mainBus -instancename cpu1 -busmasterport INSTRUCTION
ihwconnect -bus mainBus -instancename cpu1 -busmasterport DATA

ihwaddmemory -instancename ram1 -type ram
ihwconnect -bus mainBus -instancename ram1 -busslaveport sp1 -loaddress 0x0 - hiaddress 0xffffffff

ihwaddmemory -instancename ram2 -type ram
ihwconnect -bus mainBus -instancename ram2 -busslaveport sp1 -loaddress 0x0x20000000 -hiaddress 0xffffffff

ihwaddperipheral -instancename uart0 -vendor freescale.ovpworld.org
  -library peripheral -type KinetisUART -version 1.0
ihwconnect -bus mainBus -instancename uart0 -busslaveport bport1
  -loaddress 0x100000000 -hiaddress 0x100013f7
```

PlatformConstruction/simpleCpuMemoryUart

with the removal of the nets and log file. It uses the Freescale KinetisUart (which can communicate using either nets, or be virtualized and connect to a console via a socket).
No parameters are set when instancing the peripheral (we will do this in the harness). Note that the UART sits between two RAMs in the address space.

## 20.3 Application

The application initializes the UART and then writes a banner to the UART, then loops reading and echoing input until an 'x' is read, whereupon it exits.

```c
> cat application/application.c
...
int main(int argc, char **argv) {
    initFreeScaleKinetisUart(UART0_BASE);
    if ((argc == 2) && !strcmp(argv[1],"replay")) { // if 'replay' is arg to application
        printf ("[application] Reading from uart (replay) [type 'x' to exit']\n\n");
        writeMessFreescaleKinetisUart (UART0_BASE, "in REPLAY MODE.\n\n");}
    else {
        printf ("[application] Reading from uart (record) [type 'x' to exit']\n\n");
        writeMessFreescaleKinetisUart (UART0_BASE, "Reading from UART.\nType chars to be recorded. [type 'x' to exit']\n\n");
    }
    char c;
    while ((c = uart_getchar ()) != 0) {
        if (c == 'x') {
            printf ("[application1] Read termination char(x) from uart\n\n");
            break;
        }
        printf ("uartGetChar(%c)\n", c);
        char s[] = {c, '\0'};
        writeMessFreescaleKinetisUart (UART0_BASE, s);
    }
    return 0;
}
```

## 20.4 Harness that configures UART and controls operation

File `harness/harness.c` shows how record and replay may be controlled from a simulation harness. Firstly, we need to declare we are going to have command line arguments for control:

```c
struct optionsS {
    Bool record;
    Bool replay;
    Bool useport;
} options = {
    .record = 0,
    .replay = 0,
    .useport = 0,
};
```

```c
static void cmdParser(optCmdParserP parser) {
```
Then we set some parameters, first for the simulation on the root module:

```c
opParamBoolOverride(0, OP_FP_STOPONCONTROLC, 1);
```

And then set some parameters on the peripheral instance putting it into record or replay depending on command line arguments. We are also defining the directory/file name where the recordings will be stored:

```c
if (options.record) {
    opParamStringOverride(0,
        MODULE_INSTANCE " /" PERIPHERAL_INSTANCE " /" "record",
        "recdir/uart0.record");
} else if (options.replay) {
    opParamStringOverride(0,
        MODULE_INSTANCE " /" PERIPHERAL_INSTANCE " /" "replay",
        "recdir/uart0.record");
}
```

Depending upon the command line argument given when running the simulation, we will connect it to the UART with port numbers or the built-in console:

```c
if (options.useport) {
    opParamUns32Override(0,
        MODULE_INSTANCE " /" PERIPHERAL_INSTANCE " /" "portnum", 0);
} else {
    opParamBoolOverride(0,
        MODULE_INSTANCE " /" PERIPHERAL_INSTANCE " /" "console", 1);
}
```

The 'console' parameter configures the peripheral instance to pop up a console window for us to type into.
If instead we use the --myuseport command line argument, then we would be telling the UART to use a TCP port. In this mode, the UART would open a port and wait for us to connect something to it, such as the provided terminal.exe. The use of 'console' simply makes this easier and uses our provided console window automatically, resolving port numbers, etc. [Note that if you start the simulation in this mode, and the UART is awaiting a connection to a port - you either need to connect to the port and exit, or kill the simulation process to exit.]

For more information on the parameters that this UART can use, please see the documentation on it which may be found in:

$IMPERAS_HOME/ImperasLib/source/freescale.ovpworld.org/peripheral/KinetisUART/1.0/doc/OVP_Peripheral_Specific_Information_KinetisUART.pdf

Then we instance the root module and design module as normal:

```c
optModuleP mi = opRootModuleNew(0, 0, 0);
opModuleNew(mi, MODULE_DIR, MODULE_INSTANCE, 0, 0);
opRootModuleSimulate(mi);
opSessionTerminate();
return 0;
```  

### 20.5 Using instance parameters

We saw in the harness above, that we configured the specific UART instance to be set to record or replay mode.
if (options.record) {
    opParamStringOverride(0,
        MODULE_INSTANCE ""/"" PERIPHERAL_INSTANCE ""/"" "record",
            "recdir/uart0.record");
} else if (options.replay) {
    opParamStringOverride(0,
        MODULE_INSTANCE ""/"" PERIPHERAL_INSTANCE ""/"" "replay",
            "recdir/uart0.record");
}

Record or replay can be turned on / off for each peripheral instance that requires this behavior. It is usual to set record or replay for all peripherals or no peripherals so that the whole simulation behaves consistently.

### 20.6 From the command line

If you are using an Imperas simulator, record / replay can be turned on from the command line:

```bash
> harness/harness.$IMPERAS_ARCH.exe \
    --program u1/cpu1=application/application.OR1K.elf \
    --modelrecorddir recdir
```

Replay is similar; a directory is specified which contains pre-recorded events:

```bash
> harness/harness.$IMPERAS_ARCH.exe \
    --program u1/cpu1=application/application.OR1K.elf \
    --modelreplaydir recdir \
    --argv replay
```
21 OP API Compatibility with deprecated ICM API

The OP API replaces the ICM API used in earlier Imperas products. A version of the ICM provides ongoing support for existing designs, implemented internally using OP functions. Therefore all necessary ICM functions will be supported for as long as they are required.

There is no need to convert an ICM platform to an OP module unless:
- The platform must be modified to use sub-modules
- The platform must be modified to be included in an OP design.

21.1 Interoperability

A design must use either OP or ICM. The two styles must not be mixed. OP modules can load other OP modules, but not ICM. ICM cannot load OP modules.

The processor, peripheral and intercept APIs have not changed. All leaf level models and plug-ins can be used with OP or ICM.

The GDB RSP interface, the multiprocessor debugger and other Imperas tools support platforms that use ICM or OP.

OP introduces a new way to construct a platform. Once constructed, the execution and scheduling of processor, peripheral and other modules is unchanged from previous versions of the simulator.

21.2 API tracing

Set the environment variables `IMPERAS_OP_TRACE=1` to turn on tracing of entry to, and exit from OP functions. Output is to the standard output of the console or shell that invokes the simulator.
# 22 ICM to OP API Function Conversion

The ICM API is being replaced by the OP API.

To help with converting between the two APIs a conversion table has been provided, ICM_to_OP_ConversionTable.html, please see below.

This table is correct as of the date of this document.

<table>
<thead>
<tr>
<th>ICM</th>
<th>OP</th>
</tr>
</thead>
<tbody>
<tr>
<td>icmAbortRead</td>
<td>opProcessorReadAbort</td>
</tr>
<tr>
<td>icmAbortWrite</td>
<td>opProcessorWriteAbort</td>
</tr>
<tr>
<td>icmAddBoolAttr</td>
<td>opParamBoolSet</td>
</tr>
<tr>
<td>icmAddBusFetchCallback</td>
<td>opBusFetchMonitorAdd</td>
</tr>
<tr>
<td>icmAddBusReadCallback</td>
<td>opBusReadMonitorAdd</td>
</tr>
<tr>
<td>icmAddBusWriteCallback</td>
<td>opBusWriteMonitorAdd</td>
</tr>
<tr>
<td>icmAddControlFile</td>
<td>(not required)</td>
</tr>
<tr>
<td>icmAddDoubleAttr</td>
<td>opParamDoubleSet</td>
</tr>
<tr>
<td>icmAddFetchCallback</td>
<td>opProcessorFetchMonitorAdd</td>
</tr>
<tr>
<td>icmAddInt32Attr</td>
<td>opParamInt32Set</td>
</tr>
<tr>
<td>icmAddInt64Attr</td>
<td>opParamInt64Set</td>
</tr>
<tr>
<td>icmAddInterceptObject</td>
<td>opProcessorExtensionNew</td>
</tr>
<tr>
<td>icmAddNetCallback</td>
<td>opNetWriteMonitorAdd</td>
</tr>
<tr>
<td>icmAddPacketnetCallback</td>
<td>opPacketnetWriteMonitorAdd</td>
</tr>
<tr>
<td>icmAddPortMapCB</td>
<td>opBusPortConnMapNotify</td>
</tr>
<tr>
<td>icmAddProcessorDelay</td>
<td>opProcessorDelayAdd</td>
</tr>
<tr>
<td>icmAddPseInterceptObject</td>
<td>opPeripheralExtensionNew</td>
</tr>
<tr>
<td>icmAddPtrAttr</td>
<td>opParamPtrSet</td>
</tr>
<tr>
<td>icmAddReadCallback</td>
<td>opProcessorReadMonitorAdd</td>
</tr>
<tr>
<td>icmAddStringAttr</td>
<td>opParamStringSet</td>
</tr>
<tr>
<td>icmAddSymbol</td>
<td>opProcessorApplicationSymbolAdd</td>
</tr>
<tr>
<td>icmAddUns32Attr</td>
<td>opParamUns32Set</td>
</tr>
<tr>
<td>icmAddUns64Attr</td>
<td>opParamUns64Set</td>
</tr>
<tr>
<td>icmAddWriteCallback</td>
<td>opProcessorWriteMonitorAdd</td>
</tr>
<tr>
<td>icmAdvanceTime</td>
<td>opRootModuleTimeAdvance</td>
</tr>
<tr>
<td>icmAdvanceTimeDouble</td>
<td>opRootModuleTimeAdvance</td>
</tr>
<tr>
<td>icmAllVlnvFiles</td>
<td>opVLNVIter</td>
</tr>
<tr>
<td>ICM</td>
<td>OP</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>icmAtExit</td>
<td>opSessionAtExit</td>
</tr>
<tr>
<td>icmBanner</td>
<td>opBanner</td>
</tr>
<tr>
<td>icmBridgeBuses</td>
<td>opDynamicBridge</td>
</tr>
<tr>
<td>icmCLPDefaultObjectFile</td>
<td>opCmdDefaultApplication</td>
</tr>
<tr>
<td>icmCLParseArgUsed</td>
<td>opCmdArgUsed</td>
</tr>
<tr>
<td>icmCLParseArgs</td>
<td>opCmdParseArgs</td>
</tr>
<tr>
<td>icmCLParseFile</td>
<td>opCmdParseFile</td>
</tr>
<tr>
<td>icmCLParseStd</td>
<td>opCmdParseStd</td>
</tr>
<tr>
<td>icmCLParser</td>
<td>opCmdParserNew</td>
</tr>
<tr>
<td>icmCLParserAdd</td>
<td>opCmdParserAdd</td>
</tr>
<tr>
<td>icmCLParserOld</td>
<td>(not required)</td>
</tr>
<tr>
<td>icmCLParserUsageMessage</td>
<td>opCmdUsageMessage</td>
</tr>
<tr>
<td>icmCallCommand</td>
<td>opCommandStringCall</td>
</tr>
<tr>
<td>icmCancelTrigger</td>
<td>opModuleTriggerDelete</td>
</tr>
<tr>
<td>icmClearAddressBreakpoint</td>
<td>opProcessorBreakpointAddrClear</td>
</tr>
<tr>
<td>icmClearICountBreakpoint</td>
<td>opProcessorBreakpointICountClear</td>
</tr>
<tr>
<td>icmConnRestoreState</td>
<td>opFIFOStateRestore</td>
</tr>
<tr>
<td>icmConnRestoreStateFile</td>
<td>opFIFOStateRestoreFile</td>
</tr>
<tr>
<td>icmConnSaveState</td>
<td>opFIFOStateSave</td>
</tr>
<tr>
<td>icmConnSaveStateFile</td>
<td>opFIFOStateSaveFile</td>
</tr>
<tr>
<td>icmConnectMMCBus</td>
<td>opMMCBusConnect</td>
</tr>
<tr>
<td>icmConnectMemoryToBus</td>
<td>opMemoryBusConnect</td>
</tr>
<tr>
<td>icmConnectPSEBus</td>
<td>opPeripheralBusConnectSlave</td>
</tr>
<tr>
<td>icmConnectPSEBusDynamic</td>
<td>opPeripheralBusConnectSlaveDynamic</td>
</tr>
<tr>
<td>icmConnectPSENet</td>
<td>opObjectNetConnect</td>
</tr>
<tr>
<td>icmConnectPSEPacketnet</td>
<td>opPeripheralPacketnetConnect</td>
</tr>
<tr>
<td>icmConnectProcessorBusByName</td>
<td>opProcessorBusConnect</td>
</tr>
<tr>
<td>icmConnectProcessorBusses</td>
<td>opProcessorBusConnect</td>
</tr>
<tr>
<td>icmConnectProcessorConn</td>
<td>opProcessorFIFOConnect</td>
</tr>
<tr>
<td>icmConnectProcessorNet</td>
<td>opObjectNetConnect</td>
</tr>
<tr>
<td>icmDebugReadProcessorMemory</td>
<td>opProcessorRead</td>
</tr>
<tr>
<td>icmDebugThisProcessor</td>
<td>opProcessorDebug</td>
</tr>
<tr>
<td>icmDebugWriteProcessorMemory</td>
<td>opProcessorWrite</td>
</tr>
<tr>
<td>icmDeleteWatchPoint</td>
<td>opWatchpointDelete</td>
</tr>
<tr>
<td>ICM</td>
<td>OP</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>icmDerateProcessor</td>
<td>opProcessorDerate</td>
</tr>
<tr>
<td>icmDisableTraceBuffer</td>
<td>opProcessorTraceBufferDisable</td>
</tr>
<tr>
<td>icmDisassemble</td>
<td>opProcessorDisassemble</td>
</tr>
<tr>
<td>icmDocChildNode</td>
<td>opDocChildNext</td>
</tr>
<tr>
<td>icmDocIsText</td>
<td>opDocIsTitle</td>
</tr>
<tr>
<td>icmDocNextNode</td>
<td>opDocChildNext</td>
</tr>
<tr>
<td>icmDocSectionAdd</td>
<td>opDocSectionAdd</td>
</tr>
<tr>
<td>icmDocText</td>
<td>opDocText</td>
</tr>
<tr>
<td>icmDocTextAdd</td>
<td>opDocTextAdd</td>
</tr>
<tr>
<td>icmDumpRegisters</td>
<td>opProcessorRegDump</td>
</tr>
<tr>
<td>icmDumpTraceBuffer</td>
<td>opProcessorTraceBufferDump</td>
</tr>
<tr>
<td>icmEnableTraceBuffer</td>
<td>opProcessorTraceOnAfter</td>
</tr>
<tr>
<td>icmErrors</td>
<td>opErrors</td>
</tr>
<tr>
<td>icmExit</td>
<td>opProcessorExit</td>
</tr>
<tr>
<td>icmExitSimulation</td>
<td>opSessionExit</td>
</tr>
<tr>
<td>icmFindInterceptObject</td>
<td>opObjectByName</td>
</tr>
<tr>
<td>icmFindMMCByName</td>
<td>opObjectByName</td>
</tr>
<tr>
<td>icmFindPSEInterceptObject</td>
<td>opObjectByName</td>
</tr>
<tr>
<td>icmFindProcessorByName</td>
<td>opObjectByName</td>
</tr>
<tr>
<td>icmFindProcessorDoubleAttribute</td>
<td>opObjectParamNext</td>
</tr>
<tr>
<td>icmFindProcessorNetPort</td>
<td>opObjectByName</td>
</tr>
<tr>
<td>icmFindProcessorStringAttribute</td>
<td>opObjectByName</td>
</tr>
<tr>
<td>icmFindPseByName</td>
<td>opObjectByName</td>
</tr>
<tr>
<td>icmFindPseNetPort</td>
<td>opObjectNetPortConnNext</td>
</tr>
<tr>
<td>icmFinish</td>
<td>opProcessorFinish</td>
</tr>
<tr>
<td>icmFlushMemory</td>
<td>opMemoryFlush</td>
</tr>
<tr>
<td>icmFlushProcessorMemory</td>
<td>opProcessorFlush</td>
</tr>
<tr>
<td>icmFreeAttrList</td>
<td>(not required)</td>
</tr>
<tr>
<td>icmFreeBus</td>
<td>(not required)</td>
</tr>
<tr>
<td>icmFreeBusBridge</td>
<td>(not required)</td>
</tr>
<tr>
<td>icmFreeFifo</td>
<td>(not required)</td>
</tr>
<tr>
<td>icmFreeMMC</td>
<td>(not required)</td>
</tr>
<tr>
<td>icmFreeMemory</td>
<td>(not required)</td>
</tr>
<tr>
<td>icmFreeNet</td>
<td>(not required)</td>
</tr>
<tr>
<td>ICM</td>
<td>OP</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>icmFreePSE</td>
<td>(not required)</td>
</tr>
<tr>
<td>icmFreeProcessor</td>
<td>(not required)</td>
</tr>
<tr>
<td>icmFreeze</td>
<td>opProcessorFreeze</td>
</tr>
<tr>
<td>icmGetAllPlatformCommands</td>
<td>opObjectCommandNext</td>
</tr>
<tr>
<td>icmGetAllProcessorCommands</td>
<td>opProcessorCommandIterAll</td>
</tr>
<tr>
<td>icmGetBusHandle</td>
<td>opParamPtrValue</td>
</tr>
<tr>
<td>icmGetBusPortAddrBits</td>
<td>opBusPortAddrBitsDefault</td>
</tr>
<tr>
<td>icmGetBusPortAddrBitsMax</td>
<td>opBusPortAddrBitsMax</td>
</tr>
<tr>
<td>icmGetBusPortAddrBitsMin</td>
<td>opBusPortAddrBitsMin</td>
</tr>
<tr>
<td>icmGetBusPortBytes</td>
<td>opBusPortAddrHi</td>
</tr>
<tr>
<td>icmGetBusPortDesc</td>
<td>opBusPortDescription</td>
</tr>
<tr>
<td>icmGetBusPortDomainType</td>
<td>opBusPortDomainType</td>
</tr>
<tr>
<td>icmGetBusPortDomainTypeString</td>
<td>opBusPortDomainTypeString</td>
</tr>
<tr>
<td>icmGetBusPortMustBeConnected</td>
<td>opBusPortMustConnect</td>
</tr>
<tr>
<td>icmGetBusPortName</td>
<td>opObjectName</td>
</tr>
<tr>
<td>icmGetBusPortType</td>
<td>opBusPortType</td>
</tr>
<tr>
<td>icmGetBusPortTypeString</td>
<td>opBusPortTypeString</td>
</tr>
<tr>
<td>icmGetCurrentProcessor</td>
<td>opProcessorCurrent</td>
</tr>
<tr>
<td>icmGetCurrentTime</td>
<td>opModuleCurrentTime</td>
</tr>
<tr>
<td>icmGetException</td>
<td>opProcessorExceptionCurrent</td>
</tr>
<tr>
<td>icmGetExceptionInfoCode</td>
<td>opExceptionCode</td>
</tr>
<tr>
<td>icmGetExceptionInfoDescription</td>
<td>opExceptionDescription</td>
</tr>
<tr>
<td>icmGetExceptionInfoName</td>
<td>opExceptionName</td>
</tr>
<tr>
<td>icmGetFaultAddress</td>
<td>opProcessorFaultAddress</td>
</tr>
<tr>
<td>icmGetFifoHandle</td>
<td>opParamPtrValue</td>
</tr>
<tr>
<td>icmGetFifoPortDesc</td>
<td>opFIFOPortDescription</td>
</tr>
<tr>
<td>icmGetFifoPortName</td>
<td>opObjectName</td>
</tr>
<tr>
<td>icmGetFifoPortType</td>
<td>opFIFOPortType</td>
</tr>
<tr>
<td>icmGetFifoPortTypeString</td>
<td>opFIFOPortTypeString</td>
</tr>
<tr>
<td>icmGetFifoPortWidth</td>
<td>opFIFOPortWidth</td>
</tr>
<tr>
<td>icmGetImagefileElfcode</td>
<td>opApplicationElfCode</td>
</tr>
<tr>
<td>icmGetImagefileEndian</td>
<td>opApplicationEndian</td>
</tr>
<tr>
<td>icmGetImagefileEntry</td>
<td>opApplicationEntry</td>
</tr>
<tr>
<td>icmGetInterceptObjectName</td>
<td>opObjectName</td>
</tr>
<tr>
<td>ICM</td>
<td>OP</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>icmGetInterceptObjectReleaseStatus</td>
<td>opObjectReleaseStatus</td>
</tr>
<tr>
<td>icmGetInterceptObjectVisibility</td>
<td>opObjectVisibility</td>
</tr>
<tr>
<td>icmGetInterceptVLNV</td>
<td>opObjectVLNV</td>
</tr>
<tr>
<td>icmGetMMCName</td>
<td>opObjectName</td>
</tr>
<tr>
<td>icmGetMMCPortName</td>
<td>opObjectName</td>
</tr>
<tr>
<td>icmGetMMCRleaseStatus</td>
<td>opObjectReleaseStatus</td>
</tr>
<tr>
<td>icmGetMMCVisibility</td>
<td>opObjectVisibility</td>
</tr>
<tr>
<td>icmGetMMCVLNV</td>
<td>opObjectVLNV</td>
</tr>
<tr>
<td>icmGetMemoryHandle</td>
<td>opParamPtrValue</td>
</tr>
<tr>
<td>icmGetMode</td>
<td>opProcessorModeCurrent</td>
</tr>
<tr>
<td>icmGetModeInfoCode</td>
<td>opModeCode</td>
</tr>
<tr>
<td>icmGetModeInfoDescription</td>
<td>opModeDescription</td>
</tr>
<tr>
<td>icmGetModeInfoName</td>
<td>opModeName</td>
</tr>
<tr>
<td>icmGetNetHandle</td>
<td>opParamPtrValue</td>
</tr>
<tr>
<td>icmGetNetPortDesc</td>
<td>opNetPortDescription</td>
</tr>
<tr>
<td>icmGetNetPortMustBeConnected</td>
<td>opNetPortMustConnect</td>
</tr>
<tr>
<td>icmGetNetPortName</td>
<td>opObjectName</td>
</tr>
<tr>
<td>icmGetNetPortType</td>
<td>opNetPortType</td>
</tr>
<tr>
<td>icmGetNetPortTypeString</td>
<td>opNetPortTypeString</td>
</tr>
<tr>
<td>icmGetNextBusPortInfo</td>
<td>opObjectBusPortNext</td>
</tr>
<tr>
<td>icmGetNextEventTime</td>
<td>opEventTimeNext</td>
</tr>
<tr>
<td>icmGetNextException</td>
<td>opProcessorExceptionNext</td>
</tr>
<tr>
<td>icmGetNextFifoPortInfo</td>
<td>opObjectFIFOPortNext</td>
</tr>
<tr>
<td>icmGetNextInstructionAddress</td>
<td>opProcessorInstructionAttributes</td>
</tr>
<tr>
<td>icmGetNextInterceptParamInfo</td>
<td>opObjectFormalNext</td>
</tr>
<tr>
<td>icmGetNextMCMBusPortInfo</td>
<td>opObjectBusPortNext</td>
</tr>
<tr>
<td>icmGetNextMCMParamInfo</td>
<td>opObjectFormalNext</td>
</tr>
<tr>
<td>icmGetNextMode</td>
<td>opProcessorModeNext</td>
</tr>
<tr>
<td>icmGetNextNetPortInfo</td>
<td>opObjectNetPortNext</td>
</tr>
<tr>
<td>icmGetNextPSEBusPortInfo</td>
<td>opObjectBusPortNext</td>
</tr>
<tr>
<td>icmGetNextPSENNetPortInfo</td>
<td>opObjectNetPortNext</td>
</tr>
<tr>
<td>icmGetNextPSEPacketnetPort</td>
<td>opObjectPacketnetPortNext</td>
</tr>
<tr>
<td>icmGetNextPSEPParamInfo</td>
<td>opObjectFormalNext</td>
</tr>
<tr>
<td>icmGetNextParamEnum</td>
<td>opFormalEnumNext</td>
</tr>
<tr>
<td>ICM</td>
<td>OP</td>
</tr>
<tr>
<td>---------------------------------------------------------</td>
<td>---------------------------------------------------------</td>
</tr>
<tr>
<td>icmGetNextProcessorBusPortInfo</td>
<td>opObjectBusPortNext</td>
</tr>
<tr>
<td>icmGetNextProcessorFifoPortInfo</td>
<td>opObjectFIFOPortNext</td>
</tr>
<tr>
<td>icmGetNextProcessorNetPortInfo</td>
<td>opObjectNetPortNext</td>
</tr>
<tr>
<td>icmGetNextProcessorParamInfo</td>
<td>opObjectFormalNext</td>
</tr>
<tr>
<td>icmGetNextReg</td>
<td>opProcessorRegNext</td>
</tr>
<tr>
<td>icmGetNextRegGroup</td>
<td>opProcessorRegGroupNext</td>
</tr>
<tr>
<td>icmGetNextRegInGroup</td>
<td>opRegGroupRegNext</td>
</tr>
<tr>
<td>icmGetNextTriggeredWatchPoint</td>
<td>opRootModuleWatchpointNext</td>
</tr>
<tr>
<td>icmGetPC</td>
<td>opProcessorPC</td>
</tr>
<tr>
<td>icmGetPCDS</td>
<td>opProcessorPCDS</td>
</tr>
<tr>
<td>icmGetPSEDoc</td>
<td>opObjectDocNodeNext</td>
</tr>
<tr>
<td>icmGetPSEHandle</td>
<td>opParamPtrValue</td>
</tr>
<tr>
<td>icmGetPSEName</td>
<td>opObjectName</td>
</tr>
<tr>
<td>icmGetPSEReleaseStatus</td>
<td>opObjectReleaseStatus</td>
</tr>
<tr>
<td>icmGetPSESaveRestoreSupported</td>
<td>opObjectSaveRestoreSupported</td>
</tr>
<tr>
<td>icmGetPSEVisibility</td>
<td>opObjectVisibility</td>
</tr>
<tr>
<td>icmGetPSEVlnv</td>
<td>opObjectVLAN</td>
</tr>
<tr>
<td>icmGetPacketnetMaxBytes</td>
<td>opPacketnetMaxBytes</td>
</tr>
<tr>
<td>icmGetPacketnetName</td>
<td>opObjectName</td>
</tr>
<tr>
<td>icmGetPacketnetPortDesc</td>
<td>opPacketnetPortDescription</td>
</tr>
<tr>
<td>icmGetPacketnetPortMustBeConnected</td>
<td>opPacketnetPortMustConnect</td>
</tr>
<tr>
<td>icmGetPacketnetPortName</td>
<td>opObjectName</td>
</tr>
<tr>
<td>icmGetPacketnetPortNet</td>
<td>opPacketnetPortConnPacketnet</td>
</tr>
<tr>
<td>icmGetParamDesc</td>
<td>opFormalDescription</td>
</tr>
<tr>
<td>icmGetParamEnumDesc</td>
<td>opFormalEnumDescription</td>
</tr>
<tr>
<td>icmGetParamEnumName</td>
<td>opObjectName</td>
</tr>
<tr>
<td>icmGetParamEnumValue</td>
<td>opFormalEnumValue</td>
</tr>
<tr>
<td>icmGetParamName</td>
<td>opObjectName</td>
</tr>
<tr>
<td>icmGetParamType</td>
<td>opFormalType</td>
</tr>
<tr>
<td>icmGetParamTypeString</td>
<td>opFormalTypeString</td>
</tr>
<tr>
<td>icmGetPlatformName</td>
<td>opObjectName</td>
</tr>
<tr>
<td>icmGetPlatformPurpose</td>
<td>opModulePurpose</td>
</tr>
<tr>
<td>icmGetPlatformReleaseStatus</td>
<td>opObjectReleaseStatus</td>
</tr>
<tr>
<td>icmGetPlatformStopReason</td>
<td>opRootModuleStopReason</td>
</tr>
<tr>
<td>ICM</td>
<td>OP</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>(icmGetProcessorAMP)</td>
<td>opProcessorAMP</td>
</tr>
<tr>
<td>(icmGetProcessorClocks)</td>
<td>opProcessorClocks</td>
</tr>
<tr>
<td>(icmGetProcessorClocksUntilTime)</td>
<td>opProcessorClocksUntilTime</td>
</tr>
<tr>
<td>(icmGetProcessorClocksUntilTimeDouble)</td>
<td>opProcessorClocksUntilTimeDouble</td>
</tr>
<tr>
<td>(icmGetProcessorDataBus)</td>
<td>opObjectBusPortConnNext</td>
</tr>
<tr>
<td>(icmGetProcessorDefaultSemihost)</td>
<td>opProcessorDefaultSemihost</td>
</tr>
<tr>
<td>(icmGetProcessorDelay)</td>
<td>opProcessorDelay</td>
</tr>
<tr>
<td>(icmGetProcessorDesc)</td>
<td>opProcessorDescription</td>
</tr>
<tr>
<td>(icmGetProcessorDoc)</td>
<td>opObjectDocNodeNext</td>
</tr>
<tr>
<td>(icmGetProcessorElfcode)</td>
<td>opProcessorElfCodes</td>
</tr>
<tr>
<td>(icmGetProcessorEndian)</td>
<td>opProcessorEndian</td>
</tr>
<tr>
<td>(icmGetProcessorFamily)</td>
<td>opProcessorFamily</td>
</tr>
<tr>
<td>(icmGetProcessorGdbFlags)</td>
<td>opProcessorGdbFlags</td>
</tr>
<tr>
<td>(icmGetProcessorGdbPath)</td>
<td>opProcessorGdbPath</td>
</tr>
<tr>
<td>(icmGetProcessorGroupH)</td>
<td>opProcessorGroupH</td>
</tr>
<tr>
<td>(icmGetProcessorGroupL)</td>
<td>opProcessorGroupL</td>
</tr>
<tr>
<td>(icmGetProcessorHandle)</td>
<td>opParamPtrValue</td>
</tr>
<tr>
<td>(icmGetProcessorHelper)</td>
<td>opProcessorHelper</td>
</tr>
<tr>
<td>(icmGetProcessorICount)</td>
<td>opProcessorICount</td>
</tr>
<tr>
<td>(icmGetProcessorInstructionBus)</td>
<td>opObjectBusPortConnNext</td>
</tr>
<tr>
<td>(icmGetProcessorLoadPhysical)</td>
<td>opProcessorLoadPhysical</td>
</tr>
<tr>
<td>(icmGetProcessorName)</td>
<td>opObjectName</td>
</tr>
<tr>
<td>(icmGetProcessorQLQualified)</td>
<td>opProcessorQLQualified</td>
</tr>
<tr>
<td>(icmGetProcessorReleaseStatus)</td>
<td>opObjectReleaseStatus</td>
</tr>
<tr>
<td>(icmGetProcessorTime)</td>
<td>opProcessorTime</td>
</tr>
<tr>
<td>(icmGetProcessorVariant)</td>
<td>opProcessorVariant</td>
</tr>
<tr>
<td>(icmGetProcessorVisibility)</td>
<td>opObjectVisibility</td>
</tr>
<tr>
<td>(icmGetProcessorVlnv)</td>
<td>opObjectVlnv</td>
</tr>
<tr>
<td>(icmGetRegByIndex)</td>
<td>opProcessorRegByIndex</td>
</tr>
<tr>
<td>(icmGetRegByName)</td>
<td>opProcessorRegByName</td>
</tr>
<tr>
<td>(icmGetRegByUsage)</td>
<td>opProcessorRegByUsage</td>
</tr>
<tr>
<td>(icmGetRegGroupByName)</td>
<td>opProcessorRegGroupByName</td>
</tr>
<tr>
<td>(icmGetRegGroupName)</td>
<td>opRegGroupName</td>
</tr>
<tr>
<td>(icmGetRegInfoAccess)</td>
<td>opRegReadOnly</td>
</tr>
<tr>
<td>ICM</td>
<td>OP</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>icmGetRegInfoAccessString</td>
<td>opRegAccessString</td>
</tr>
<tr>
<td>icmGetRegInfoBits</td>
<td>opRegBits</td>
</tr>
<tr>
<td>icmGetRegInfoDesc</td>
<td>opRegDescription</td>
</tr>
<tr>
<td>icmGetRegInfoGdbIndex</td>
<td>opRegIndex</td>
</tr>
<tr>
<td>icmGetRegInfoGroup</td>
<td>opRegGroup</td>
</tr>
<tr>
<td>icmGetRegInfoName</td>
<td>opRegName</td>
</tr>
<tr>
<td>icmGetRegInfoReadOnly</td>
<td>opRegReadOnly</td>
</tr>
<tr>
<td>icmGetRegInfoUsage</td>
<td>opRegUsageEnum</td>
</tr>
<tr>
<td>icmGetRegInfoUsageString</td>
<td>opRegUsageString</td>
</tr>
<tr>
<td>icmGetSMPChild</td>
<td>opProcessorChild</td>
</tr>
<tr>
<td>icmGetSMPData</td>
<td>opProcessorData</td>
</tr>
<tr>
<td>icmGetSMPIndex</td>
<td>opProcessorIndex</td>
</tr>
<tr>
<td>icmGetSMPNextSibling</td>
<td>opProcessorSiblingNext</td>
</tr>
<tr>
<td>icmGetSMPParent</td>
<td>opProcessorParent</td>
</tr>
<tr>
<td>icmGetSMPPrevSibling</td>
<td>opProcessorSiblingPrevious</td>
</tr>
<tr>
<td>icmGetStatus</td>
<td>opModuleFinishStatus</td>
</tr>
<tr>
<td>icmGetStopReason</td>
<td>opProcessorStopReason</td>
</tr>
<tr>
<td>icmGetVlnvString</td>
<td>opVlnvString</td>
</tr>
<tr>
<td>icmGetWatchPointCurrentValue</td>
<td>opWatchpointRegCurrentValue</td>
</tr>
<tr>
<td>icmGetWatchPointHighAddress</td>
<td>opWatchpointAddressHi</td>
</tr>
<tr>
<td>icmGetWatchPointLowAddress</td>
<td>opWatchpointAddressLo</td>
</tr>
<tr>
<td>icmGetWatchPointPreviousValue</td>
<td>opWatchpointRegPreviousValue</td>
</tr>
<tr>
<td>icmGetWatchPointRegister</td>
<td>opWatchpointReg</td>
</tr>
<tr>
<td>icmGetWatchPointTriggeredBy</td>
<td>opWatchpointTriggeredBy</td>
</tr>
<tr>
<td>icmGetWatchPointType</td>
<td>opWatchpointType</td>
</tr>
<tr>
<td>icmGetWatchPointUserData</td>
<td>opWatchpointUserData</td>
</tr>
<tr>
<td>icmHalt</td>
<td>opProcessorHalt</td>
</tr>
<tr>
<td>icmIgnoreMessage</td>
<td>opMessageDisable</td>
</tr>
<tr>
<td>icmInitInternal</td>
<td>opModuleNewFromAttrs</td>
</tr>
<tr>
<td>icmInitPlatform</td>
<td>opModuleNew</td>
</tr>
<tr>
<td>icmInstallObjectReader</td>
<td>opApplicationLoaderInstall</td>
</tr>
<tr>
<td>icmInterrupt</td>
<td>opInterrupt</td>
</tr>
<tr>
<td>icmInterruptRSP</td>
<td>opInterruptRSP</td>
</tr>
<tr>
<td>icmIsFrozen</td>
<td>opProcessorFrozen</td>
</tr>
<tr>
<td>ICM</td>
<td>OP</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>icmIterAllChildren</td>
<td>opProcessorIterChildren</td>
</tr>
<tr>
<td>icmIterAllDescendants</td>
<td>opProcessorIterDescendants</td>
</tr>
<tr>
<td>icmIterAllModelParameters</td>
<td>opModuleFormalsShow</td>
</tr>
<tr>
<td>icmIterAllProcessors</td>
<td>opProcessorIterAll</td>
</tr>
<tr>
<td>icmIterAllUserAttributes</td>
<td>opModuleFormalsShow</td>
</tr>
<tr>
<td>icmLastMessage</td>
<td>opLastMessage</td>
</tr>
<tr>
<td>icmLegalUsageEnable</td>
<td>(not required)</td>
</tr>
<tr>
<td>icmLoadBus</td>
<td>opBusApplicationLoad</td>
</tr>
<tr>
<td>icmLoadModelHook</td>
<td>opSessionInit</td>
</tr>
<tr>
<td>icmLoadProcessorMemory</td>
<td>opProcessorApplicationLoad</td>
</tr>
<tr>
<td>icmLoadProcessorMemoryOffset</td>
<td>opProcessorApplicationLoad</td>
</tr>
<tr>
<td>icmLoadSymbols</td>
<td>opMemoryApplicationLoad</td>
</tr>
<tr>
<td>icmMMCRestoreState</td>
<td>opMMCStateRestore</td>
</tr>
<tr>
<td>icmMMCRestoreStateFile</td>
<td>opMMCStateRestoreFile</td>
</tr>
<tr>
<td>icmMMCSaveState</td>
<td>opMMCStateSave</td>
</tr>
<tr>
<td>icmMMCSaveStateFile</td>
<td>opMMCStateSaveFile</td>
</tr>
<tr>
<td>icmMMRegBits</td>
<td>opMMRegisterBits</td>
</tr>
<tr>
<td>icmMMRegDescription</td>
<td>opMMRegisterDescription</td>
</tr>
<tr>
<td>icmMMRegName</td>
<td>opMMRegisterName</td>
</tr>
<tr>
<td>icmMMRegOffset</td>
<td>opMMRegisterOffset</td>
</tr>
<tr>
<td>icmMapExternalMemory</td>
<td>opBusSlaveNew</td>
</tr>
<tr>
<td>icmMapExternalNativeMemory</td>
<td>opBusSlaveNew</td>
</tr>
<tr>
<td>icmMapLocalMemory</td>
<td>opMemoryNew</td>
</tr>
<tr>
<td>icmMapNativeMemory</td>
<td>opMemoryNativeDynamic</td>
</tr>
<tr>
<td>icmMemoryRestoreState</td>
<td>opMemoryStateRestore</td>
</tr>
<tr>
<td>icmMemoryRestoreStateFile</td>
<td>opMemoryStateRestoreFile</td>
</tr>
<tr>
<td>icmMemorySaveState</td>
<td>opMemoryStateSave</td>
</tr>
<tr>
<td>icmMemorySaveStateFile</td>
<td>opMemoryStateSaveFile</td>
</tr>
<tr>
<td>icmMessage</td>
<td>opMessage</td>
</tr>
<tr>
<td>icmMessageQuiet</td>
<td>opMessageQuiet</td>
</tr>
<tr>
<td>icmMessageSetNoWarn</td>
<td>opMessageSetNoWarn</td>
</tr>
<tr>
<td>icmMessageSetQuiet</td>
<td>opMessageSetQuiet</td>
</tr>
<tr>
<td>icmMessageVerbose</td>
<td>opMessageVerbose</td>
</tr>
<tr>
<td>icmNetPortDirection</td>
<td>opNetPortType</td>
</tr>
<tr>
<td>ICM</td>
<td>OP</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>icmNetPortName</td>
<td>opObjectName</td>
</tr>
<tr>
<td>icmNetRestoreState</td>
<td>opNetStateRestore</td>
</tr>
<tr>
<td>icmNetRestoreStateFile</td>
<td>opNetStateRestoreFile</td>
</tr>
<tr>
<td>icmNetSaveState</td>
<td>opNetStateSave</td>
</tr>
<tr>
<td>icmNetSaveStateFile</td>
<td>opNetStateSaveFile</td>
</tr>
<tr>
<td>icmNewBus</td>
<td>opBusNew</td>
</tr>
<tr>
<td>icmNewBusBridge</td>
<td>opBridgeNew</td>
</tr>
<tr>
<td>icmNewBusWithHandle</td>
<td>opParamPtrSet</td>
</tr>
<tr>
<td>icmNewFifo</td>
<td>opFIFONew</td>
</tr>
<tr>
<td>icmNewFifoWithHandle</td>
<td>opParamPtrSet</td>
</tr>
<tr>
<td>icmNewMMC</td>
<td>opMMCNew</td>
</tr>
<tr>
<td>icmNewMemory</td>
<td>opMemoryNew</td>
</tr>
<tr>
<td>icmNewMemoryWithHandle</td>
<td>opParamPtrSet</td>
</tr>
<tr>
<td>icmNewNet</td>
<td>opNetNew</td>
</tr>
<tr>
<td>icmNewNetWithHandle</td>
<td>opParamPtrSet</td>
</tr>
<tr>
<td>icmNewPSE</td>
<td>opPeripheralNew</td>
</tr>
<tr>
<td>icmNewPSEWithHandle</td>
<td>opParamPtrSet</td>
</tr>
<tr>
<td>icmNewPacketnet</td>
<td>opPacketnetNew</td>
</tr>
<tr>
<td>icmNewProcessor</td>
<td>opProcessorNewWithSemihost</td>
</tr>
<tr>
<td>icmNewProcessorIASAttrs</td>
<td>opProcessorNewFromAttrs</td>
</tr>
<tr>
<td>icmNewProcessorWithHandle</td>
<td>opParamPtrSet</td>
</tr>
<tr>
<td>icmNextBusPortMMRegInfo</td>
<td>opBusPortMMRegisterNext</td>
</tr>
<tr>
<td>icmNextInterceptObject</td>
<td>opObjectExtensionNext</td>
</tr>
<tr>
<td>icmNextMmc</td>
<td>opMMCNext</td>
</tr>
<tr>
<td>icmNextProcessor</td>
<td>opProcessorNext</td>
</tr>
<tr>
<td>icmNextProcessorNetPort</td>
<td>opObjectNetPortConnNext</td>
</tr>
<tr>
<td>icmNextPse</td>
<td>opPeripheralNext</td>
</tr>
<tr>
<td>icmNextPseNetPort</td>
<td>opObjectNetPortConnNext</td>
</tr>
<tr>
<td>icmNoBanner</td>
<td>opNoBanner</td>
</tr>
<tr>
<td>icmOverride</td>
<td>opParamUns64Override</td>
</tr>
<tr>
<td>icmParamBoolDefaultValue</td>
<td>opFormalBoolDefaultValue</td>
</tr>
<tr>
<td>icmParamDoubleLimits</td>
<td>opFormaldoubleLimits</td>
</tr>
<tr>
<td>icmParamInt32Limits</td>
<td>opFormalInt32Limits</td>
</tr>
<tr>
<td>icmParamInt64Limits</td>
<td>opFormalInt64Limits</td>
</tr>
<tr>
<td>ICM</td>
<td>OP</td>
</tr>
<tr>
<td>---------------------------------------------------------------------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>icmParamStringDefaultValue</td>
<td>opFormalStringDefaultValue</td>
</tr>
<tr>
<td>icmParamUns32Limits</td>
<td>opFormalUns32Limits</td>
</tr>
<tr>
<td>icmParamUns64Limits</td>
<td>opFormalUns64Limits</td>
</tr>
<tr>
<td>icmPrintAllBusConnections</td>
<td>opModuleShow</td>
</tr>
<tr>
<td>icmPrintAllPacketnetConnections</td>
<td>(not required)</td>
</tr>
<tr>
<td>icmPrintBusConnections</td>
<td>opModuleBusShow</td>
</tr>
<tr>
<td>icmPrintNetConnections</td>
<td>opNetShow</td>
</tr>
<tr>
<td>icmPrintf</td>
<td>opPrintf</td>
</tr>
<tr>
<td>icmProcessorIsVisible</td>
<td>(not required)</td>
</tr>
<tr>
<td>icmProcessorRestoreState</td>
<td>opProcessorStateRestore</td>
</tr>
<tr>
<td>icmProcessorRestoreStateFile</td>
<td>opProcessorStateRestoreFile</td>
</tr>
<tr>
<td>icmProcessorSaveState</td>
<td>opProcessorStateSave</td>
</tr>
<tr>
<td>icmProcessorSaveStateFile</td>
<td>opProcessorStateSaveFile</td>
</tr>
<tr>
<td>icmReadBus</td>
<td>opBusRead</td>
</tr>
<tr>
<td>icmReadMemory</td>
<td>opMemoryRead</td>
</tr>
<tr>
<td>icmReadObject</td>
<td>opProcessorApplicationRead</td>
</tr>
<tr>
<td>icmReadObjectFileHeader</td>
<td>opApplicationHeaderRead</td>
</tr>
<tr>
<td>icmReadObjectFileHeaderInfo</td>
<td>opApplicationHeaderRead</td>
</tr>
<tr>
<td>icmReadObjectFileInfo</td>
<td>opProcessorApplicationRead</td>
</tr>
<tr>
<td>icmReadProcessorMemory</td>
<td>opProcessorRead</td>
</tr>
<tr>
<td>icmReadReg</td>
<td>opProcessorRegReadByName</td>
</tr>
<tr>
<td>icmReadRegInfoValue</td>
<td>opProcessorRegRead</td>
</tr>
<tr>
<td>icmResetErrors</td>
<td>opResetErrors</td>
</tr>
<tr>
<td>icmResetWatchPoint</td>
<td>opWatchpointReset</td>
</tr>
<tr>
<td>icmSetAddressBreakpoint</td>
<td>opProcessorBreakpointAddrSet</td>
</tr>
<tr>
<td>icmSetBusAccessWatchPoint</td>
<td>opBusAccessWatchpointNew</td>
</tr>
<tr>
<td>icmSetBusReadWatchPoint</td>
<td>opBusReadWatchpointNew</td>
</tr>
<tr>
<td>icmSetBusWriteWatchPoint</td>
<td>opBusWriteWatchpointNew</td>
</tr>
<tr>
<td>icmSetContextString</td>
<td>(not required)</td>
</tr>
<tr>
<td>icmSetDebugMode</td>
<td>(not required)</td>
</tr>
<tr>
<td>icmSetDebugNotifiers</td>
<td>opSessionDebuggerNotifiersAdd</td>
</tr>
<tr>
<td>icmSetDebugStopTime</td>
<td>opRootModuleSetDebugStopTime</td>
</tr>
<tr>
<td>icmSetExceptionWatchPoint</td>
<td>opProcessorExceptionWatchpointNew</td>
</tr>
<tr>
<td>ICM</td>
<td>OP</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>icmSetICountBreakpoint</td>
<td>opProcessorBreakpointICountSet</td>
</tr>
<tr>
<td>icmSetMemoryAccessWatchPoint</td>
<td>opMemoryAccessWatchpointNew</td>
</tr>
<tr>
<td>icmSetMemoryReadWatchPoint</td>
<td>opMemoryReadWatchpointNew</td>
</tr>
<tr>
<td>icmSetMemoryWriteWatchPoint</td>
<td>opMemoryWriteWatchpointNew</td>
</tr>
<tr>
<td>icmSetModeWatchPoint</td>
<td>opProcessorModeWatchpointNew</td>
</tr>
<tr>
<td>icmSetPC</td>
<td>opProcessorPCSet</td>
</tr>
<tr>
<td>icmSetPSEGdbgPath</td>
<td>(not required)</td>
</tr>
<tr>
<td>icmSetPSEdiagnosticLevel</td>
<td>opPeripheralDiagnosticLevelSet</td>
</tr>
<tr>
<td>icmSetPersonality</td>
<td>opLicPersonalitySet</td>
</tr>
<tr>
<td>icmSetPlatformStatus</td>
<td>(not required)</td>
</tr>
<tr>
<td>icmSetProcessorAccessWatchPoint</td>
<td>opProcessorAccessWatchpointNew</td>
</tr>
<tr>
<td>icmSetProcessorGdbBasic</td>
<td>(not required)</td>
</tr>
<tr>
<td>icmSetProcessorGdbPath</td>
<td>(not required)</td>
</tr>
<tr>
<td>icmSetProcessorReadWatchPoint</td>
<td>opProcessorReadWatchpointNew</td>
</tr>
<tr>
<td>icmSetProcessorWriteWatchPoint</td>
<td>opProcessorWriteWatchpointNew</td>
</tr>
<tr>
<td>icmSetProduct</td>
<td>opProductSet</td>
</tr>
<tr>
<td>icmSetRegisterWatchPoint</td>
<td>opProcessorRegWatchpointNew</td>
</tr>
<tr>
<td>icmSetSimulationRandomSeed</td>
<td>opRootModuleSetSimulationRandomSeed</td>
</tr>
<tr>
<td>icmSetSimulationStopTime</td>
<td>opRootModuleSetSimulationStopTime</td>
</tr>
<tr>
<td>icmSetSimulationStopTimeDouble</td>
<td>opRootModuleSetSimulationStopTime</td>
</tr>
<tr>
<td>icmSetSimulationTimePrecision</td>
<td>opRootModuleSetSimulationTimePrecision</td>
</tr>
<tr>
<td>icmSetSimulationTimePrecisionDouble</td>
<td>opRootModuleSetSimulationTimePrecision</td>
</tr>
<tr>
<td>icmSetSimulationTimeSlice</td>
<td>opRootModuleSetSimulationTimeSlice</td>
</tr>
<tr>
<td>icmSetSimulationTimeSliceDouble</td>
<td>opRootModuleSetSimulationTimeSlice</td>
</tr>
<tr>
<td>icmSetTextOutputFn</td>
<td>opSessionTextRedirect</td>
</tr>
<tr>
<td>icmSetWallClockFactor</td>
<td>opRootModuleSetWallClockFactor</td>
</tr>
<tr>
<td>icmSimulate</td>
<td>opProcessorSimulate</td>
</tr>
<tr>
<td>icmSimulatePlatform</td>
<td>opRootModuleSimulate</td>
</tr>
<tr>
<td>icmSimulationEnding</td>
<td>opRootModulePostSimulate</td>
</tr>
<tr>
<td>icmSimulationStarting</td>
<td>opRootModulePreSimulate</td>
</tr>
<tr>
<td>icmTerminate</td>
<td>opSessionTerminate</td>
</tr>
<tr>
<td>icmTraceOffAfter</td>
<td>opProcessorTraceOffAfter</td>
</tr>
<tr>
<td>icmTraceOnAfter</td>
<td>opProcessorTraceOnAfter</td>
</tr>
<tr>
<td>icmTriggerAfter</td>
<td>opModuleTriggerAdd</td>
</tr>
<tr>
<td>ICM</td>
<td>OP</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>icmTryVInvString</td>
<td>opVLNVString</td>
</tr>
<tr>
<td>icmUnbridgeBuses</td>
<td>opDynamicUnbridge</td>
</tr>
<tr>
<td>icmUnfreeze</td>
<td>opProcessorUnfreeze</td>
</tr>
<tr>
<td>icmVAAbort</td>
<td>opVAAbort</td>
</tr>
<tr>
<td>icmVMessage</td>
<td>opVMessage</td>
</tr>
<tr>
<td>icmVPrintf</td>
<td>opVPrintf</td>
</tr>
<tr>
<td>icmWriteBus</td>
<td>opBusWrite</td>
</tr>
<tr>
<td>icmWriteMemory</td>
<td>opMemoryWrite</td>
</tr>
<tr>
<td>icmWriteNet</td>
<td>opNetWrite</td>
</tr>
<tr>
<td>icmWriteNetPort</td>
<td>opNetWrite</td>
</tr>
<tr>
<td>icmWritePacketnet</td>
<td>opPacketnetWrite</td>
</tr>
<tr>
<td>icmWriteProcessorMemory</td>
<td>opProcessorWrite</td>
</tr>
<tr>
<td>icmWriteReg</td>
<td>opProcessorRegWriteByName</td>
</tr>
<tr>
<td>icmWriteRegInfoValue</td>
<td>opProcessorRegWrite</td>
</tr>
<tr>
<td>icmYield</td>
<td>opProcessorYield</td>
</tr>
</tbody>
</table>