Advanced Simulation Control of Platforms and Modules User Guide

Imperas Software Limited
Imperas Buildings, North Weston,
Thame, Oxfordshire, OX9 2HA, UK
docs@imperas.com

Author:	Imperas Software Limited
Version:	2.1.1
Filename:	Advanced_Simulation_Control_of_Platforms_and_Modules_User_Guide.doc
Project:	Advanced Simulation Control of Platforms and Modules User Guide
Last Saved:	Friday, 22 January 2021

Copyright Notice

Right to Copy Documentation

Destination Control Statement

Disclaimer

Table of Contents

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 detailed information of how the simulator is controlled, how virtual platforms are loaded and simulated, how test benches and harnesses are written.

This is an advanced user guide.

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.
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

Also, in your installation there is also the online OP API Function Reference documentation. This is correct-by-construction Doxygen-like API documentation available at:

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

1.3 Glossary / Terminology

OP API - OVP Platforms API - C API used for creating and controlling virtual platforms. OP is a 2nd generation API and 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

Platform / Module (used interchangeably) - 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 - program in C/C++ making calls into OP API to connect and control OVP components. Normally linked to the simulator to provide an .exe binary that can be executed. Used to instantiate one or more platforms/modules and controls their execution. The main difference, from a platform/module, is that a testbench or harness includes a call to 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 instantiates 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 test bench is written in C/C++ using the OVP OP API.

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

2.1 Prerequisites

Since harnesses and test benches 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.

This document includes usage of iGen which uses the TCL scripting language, so it is beneficial to have some basic understanding of TCL.

This is the ***advanced*** usage guide, so please ensure you have read and worked through *Simulation Control of Platforms and Modules User Guide* before reading this advanced guide.

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 processor and component models and toolchains, available to download from the www.OVPworld.org website or as part of an Imperas installation.

The compilation of the examples makes use of Makefiles and GNU make. The instructions indicate the use of the command *make* on Linux systems and MinGW *mingw32-make* command on Windows systems.

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, for example, as *example.sh.* The shell (extension *sh)* script files may be used on Linux and in Windows MSYS shells. The batch (extension bat) files may be used in Windows explorer or in a Windows command shell.

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 (Imperas/OVP has been used with version 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.systemc.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 both have the suffix *.exe*

2.5 Platforms, Modules and Shared Objects

Modules are created by writing scripts using iGen API calls and then using iGen to generate C code that calls functions from the OP API. A Makefile is provided that will take as input a file *module.op.tcl* and execute iGen and the host compiler and linker to create the *model.so/.dll* shared objects.

The *model.so/.dll* shared object can then be loaded and simulated using the *harness.exe* program (provided in the installation binary directory), or by writing a bespoke test harness in C using the OP API.

In this document we will either use binary modules and components form the Imperas provided library, or we will provide them as source in the example directories.

3 Tracing the API calls and seeing what is happening

Sometime it is not clear what is happening and what order things are happening.

Set the environment variable *IMPERAS_OP_TRACE=1* to turn on tracing of entry to and exit from OP API functions. Output is to the standard output of the console or shell that invokes the simulator.

In the process of converting some platforms from use of the deprecated ICM API to the OP API we found it very useful to be able to trace the API calls used in platform/module creation.

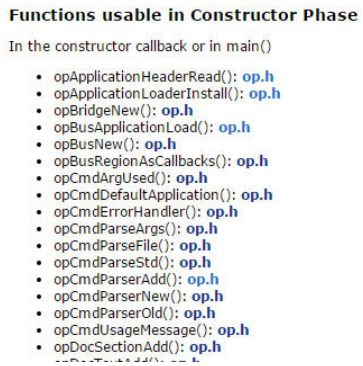
4 Detailed OP API Documentation

For a definition of each function call, macro, and structure that make up the OP API, please refer to the OP function definition documentation that is part of the doxygen generated documentation located as part of an installation at:

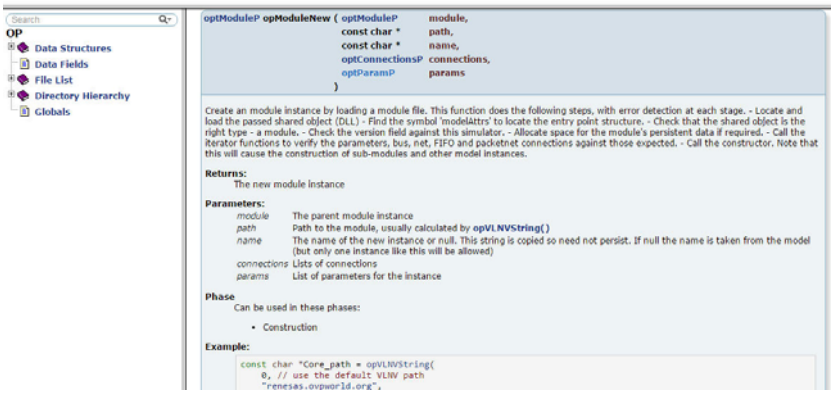
IMPERAS_HOME/doc/api/op/html/index.html

For example:

A good place to start is with the 'Phases and the functions that can be called from them' link. This provides a list of the phases, with links to those API functions that can be used in that phase:



And then clicking on the *'op.h'* link shows you detailed documentation on that function:



5 Structure of Harnesses/Modules & Simulation Phases

The document Simulation_Control_of_Platforms_and_Modules_User_Guide explained a simple test harness. It includes a *main()* function with several simple sections: command line parser, module instantiation, simulation, and was bounded with a couple of session control calls.

```
> cat Examples/SimulationControl/minimalHarness/harness/harness.c

#include <string.h>
#include <stdlib.h>

#include "op/op.h"

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

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

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

    opRootModuleSimulate(mi);

    opSessionTerminate();
    return 0;
}
```

It has a *main()* that makes the calls to the API calls of interest. This was a very simplistic introduction to the power and possibilities that are possible with the OP API related to test benches and harnesses to control simulation.

5.1 A little more complete test harness

In the *Example/SimulationControl/minimalHarness*, we saw a really minimal harness, now we will look at one that has placeholders for all the full capabilities that are available. Subsequent examples will add code into these placeholders.

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

Again we see the application and module directories. We discussed those previously. They have not changed in this example. We will focus on the harness code.

Looking at *harness/harness.c* we see there are several callbacks with some being empty. Being empty at this stage they are just 'placeholders' for us to put code in later.

These callbacks are defined using macros, are also defined in the *modelAttrs* table, and are called at different stages of the simulation process.

For example:

```
    OP_CONSTRUCT_FN(static OP_CONSTRUCT_FN(moduleConstruct) {
```

declares a function *moduleConstruct* that will be called by the simulator in the phase of constructing all the modules.

The simulator has five main phases:

- Construction
- Pre Simulation
- Simulation
- Post Simulation
- Destruction

5.1.1 Constructor - moduleConstruct

In the *OP_CONSTRUCT_FN(moduleConstruct*) callback function declaration we instance the design to be simulated. In this example it is a single module instance:

```
static OP_CONSTRUCT_FN(moduleConstruct) {
    const char *u1_path = "module";
    opModuleNew(
        mi,       // parent module
        u1_path,      // modelfile
        "u1",   // name
        0,
        0
    );
}
```

The constructor call back function is the only *mandatory* callback function and it constructs the structural contents of this harness including creation of component instances, buses, nets etc. and connection to components

5.1.2 Command Line Parsing - cmdParser

There is then the declaration of a function we can edit that will process the command line arguments. It is here will put any processing that is specific for our design / test bench.

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

```
}
```

The OP standard parser gives a consistent method of parsing standard data types, but also allows the user to specify any of the rich set of options accepted by the simulator. Please refer to the OVP Control File User Guide to see the standard options.

5.1.3 Local module persistent data - optModuleObject

Often we will need to hold local data for our testbench that can be called at different times in the simulation. This is in the optModuleObject structure.

```
typedef struct optModuleObjectS {
    // insert module persistent data here
} optModuleObject;
```

5.1.4 Initialization - modulePreSimulate

This is followed by several more declarations of callback functions. You can put bespoke code into these callbacks knowing that they will be called at the appropriate times or phases of the simulation.

```
static OP_PRE_SIMULATE_FN(modulePreSimulate) {
// insert modulePreSimulate code here
}
```

The *OP_PRE_SIMULATE_FN(modulePreSimulate)* callback function is called when simulation is about to begin.

5.1.5 Simulation - moduleSimulate

```
static OP_SIMULATE_STARTING_FN(moduleSimulate) {
// insert moduleSimulate code here
}
```

Gets called every time *opRootModuleSimulate ()* is called. Called every time the simulator starts simulating.

5.1.6 Reporting - modulePostSimulate

```
static OP_POST_SIMULATE_FN(modulePostSimulate) {
// insert modulePostSimulate code here
}
```

Called when simulation has finished, but before any destructors are called.

5.1.7 Destructor - moduleDestruct

```
static OP_DESTRUCT_FN(moduleDestruct) {
// insert moduleDestruct code here
}
```

If required, code to close files, free memory etc.

5.1.8 modelAttrs table

A complete *modelAttrs* table is:

```
optModuleAttr modelAttrs = {
    .versionString      = OP_VERSION,
    .type               = OP_MODULE,
    .name               = MODULE_NAME,
    .objectSize         = sizeof(optModuleObject),
    .releaseStatus      = OP_UNSET,
    .purpose            = OP_PP_BAREMETAL,
    .visibility         = OP_VISIBLE,
    .constructCB        = moduleConstruct,
    .preSimulateCB      = modulePreSimulate,
    .simulateCB         = moduleSimulate,
    .postSimulateCB     = modulePostSimulate,
    .destructCB         = moduleDestruct,
};
```

It defines this module and provides the relevant data and pointers that are needed. Some items like *versionString*, *type*, *releaseStatus*, *purpose*, *visibility* should be accepted and left alone.

The others are only required if they are being used in your module. There is no performance issue in just leaving them as above. The simulator looks up the modelAttrs table to find which callbacks are defined and then calls them as it needs to.

5.1.9 The main() program

Lastly, by convention at the bottom of the file, the main() program entry point:

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

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

    optModuleP mi = opRootModuleNew(&modelAttrs, MODULE_NAME, 0);

    opRootModuleSimulate(mi);

    opSessionTerminate();
    return 0;
}
```

5.1.10 Complete harness code

The complete harness is:

```
#include <string.h>
#include <stdlib.h>

#include "op/op.h"

#define MODULE_NAME "top"

struct optionsS {
} options = {
};

static OP_CONSTRUCT_FN(moduleConstruct) {
    const char *u1_path = "module";
    opModuleNew(
        mi,             // parent module
        u1_path,        // modelfile
        "u1",           // name
        0,
        0
    );
}

static void cmdParser(optCmdParserP parser) {
}

typedef struct optModuleObjectS {
    // insert module persistent data here
} optModuleObject;
```

```
static OP_PRE_SIMULATE_FN(modulePreSimulate) {
// insert modulePreSimulate code here
}

static OP_SIMULATE_STARTING_FN(moduleSimulate) {
// insert moduleSimulate code here
}

static OP_POST_SIMULATE_FN(modulePostSimulate) {
// insert modulePostSimulate code here
}

static OP_DESTRUCT_FN(moduleDestruct) {
// insert moduleDestruct code here
}

optModuleAttr modelAttrs = {
    .versionString      = OP_VERSION,
    .type               = OP_MODULE,
    .name               = MODULE_NAME,
    .objectSize         = sizeof(optModuleObject),
    .releaseStatus      = OP_UNSET,
    .purpose            = OP_PP_BAREMETAL,
    .visibility         = OP_VISIBLE,
    .constructCB        = moduleConstruct,
    .preSimulateCB      = modulePreSimulate,
    .simulateCB         = moduleSimulate,
    .postSimulateCB     = modulePostSimulate,
    .destructCB         = moduleDestruct,
};

int main(int argc, const char *argv[]) {
    opSessionInit(OP_VERSION);
    optCmdParserP parser = opCmdParserNew(MODULE_NAME, OP_AC_ALL);
    cmdParser(parser);
    opCmdParseArgs(parser, argc, argv);
    optModuleP mi = opRootModuleNew(&modelAttrs, MODULE_NAME, 0);
    opRootModuleSimulate(mi);
    opSessionTerminate();
    return 0;
}
```

5.2 Order of Platform construction

In the above section we saw that a harness can have several callbacks defined in the modelAttrs table. These callbacks are called at different phases of the simulation.

For a test bench this might seem like overkill to specify different phases and callbacks.

This approach comes into its own when you realize that in fact all modules can have callbacks and that all callbacks of one type are called in one phase. Thus all construction callbacks from all modules are called before all pre-simulate callbacks etc. This allows extremely sophisticated and complex platforms to be model relatively easily

This section summarizes the operation of a hierarchical platform.

- Host computer calls the program entry point : main
  - start opSessionInit
  - construct a command line parser opCmdParserNew, opCmdParserAdd (with added arguments) or use the standard command line parser opCmdParseArgs
  - create instance of root module opRootModuleNew
    - call the constructor moduleConstruct
      - create instance of the design opModuleNew
        - call parameter iterator
        - call interface iterators
        - call module constructor moduleConstruct
          - create model instances opProcessorNew
          - create module instances opModuleNew
  - run the simulator opRootModuleSimulate (maybe more than once)
    - call pre-simulate functions in all modules (first time only)
    - run the simulator
  - finish opSessionTerminate
    - call post-simulation functions for all modules
    - call destructors

Higher-level modules are constructed before lower modules.

Leaf components (processors, memories etc) can be created at any level.

A module can instance itself (so long as there is code to prevent infinite recursion).

The simulator can determine the interface to a module without constructing it.

6 Modules revisited (using the OP API and C)

In the document *iGen_Platform_and_Module_Creation_User_Guide* we saw the creation of modules using iGen from a tcl script input. When iGen is run on a *module.op.tcl* for the first time it creates two main files:

module.c file - this is the code you can edit

module.igen.h - this is code you should not edit - it is the structure of the module and will be generated each time from the *module.op.tcl* iGen run

there is also a file

module.c.igen.stubs - this is a template of *module.c* - this is written each time, and is used for comparison purposes - you can ignore it for now

6.1 A module in C with all user callbacks

Let's look at the process of running iGen on a module and look at the generated C:

```
> cd Examples/SimulationControl/simpleHarness/module
> cat 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 sp1 \
        -loaddress 0x0 -hiaddress 0xffffffff
```

```
> make
# iGen Create OP MODULE module
# Host Compiling Module obj/Linux32/module.o
# Host Linking Module object model.so
```

and let's look at the generated C files, first the one defining the structure that we don't edit:

```
> cat module.igen.h

#define MODULE_NAME "simpleCpuMemory"

static OP_CONSTRUCT_FN(moduleConstruct) {

    // Bus mainBus
```

```
optBusP mainBus_b = opBusNew(mi, "mainBus", 32, 0, 0);


// Processor cpu1

const char *cpu1_path = opVLNVString(
    0, // use the default VLNV path
    "ovpworld.org",
    "processor",
    "or1k",
    "1.0",
    OP_PROCESSOR,
    1   // report errors
);

optProcessorP cpu1_c = opProcessorNew(
    mi,
    cpu1_path,
    "cpu1",
    OP_CONNECTIONS(
        OP_BUS_CONNECTIONS(
            OP_BUS_CONNECT(mainBus_b, "INSTRUCTION"),
            OP_BUS_CONNECT(mainBus_b, "DATA")
        )
    ),
    OP_PARAMS(
         OP_PARAM_STRING_SET("variant", "generic")
    )
);

const char *or1kNewlib_0_expath = opVLNVString(
    0, // use the default VLNV path
    0,
    0,
    "or1kNewlib",
    0,
    OP_EXTENSION,
    1   // report errors
);

opProcessorExtensionNew(
    cpu1_c,
    or1kNewlib_0_expath,
    "or1kNewlib_0",
```

```
        0
    );

    // Memory ram1

    opMemoryNew(
        mi,
        "ram1",
        OP_PRIV_RWX,
        (0xffffffff) - (0x0),
        OP_CONNECTIONS(
            OP_BUS_CONNECTIONS(
                OP_BUS_CONNECT(mainBus_b, "sp1", .slave=1,
                    .addrLo=0x0, .addrHi=0xffffffff)
            )
        ),
        0
    );

}

optModuleAttr modelAttrs = {
    .versionString      = OP_VERSION,
    .type               = OP_MODULE,
    .name               = MODULE_NAME,
    .objectSize         = sizeof(optModuleObject),
    .releaseStatus      = OP_UNSET,
    .purpose            = OP_PP_BAREMETAL,
    .visibility         = OP_VISIBLE,
    .constructCB        = moduleConstruct,
    .preSimulateCB      = modulePreSimulate,
    .simulateCB         = moduleSimulate,
    .postSimulateCB     = modulePostSimulate,
    .destructCB         = moduleDestruct,
};
```

and then look at the *module.c* file we can edit:

```
> cat module.c

#include <string.h>
#include <stdlib.h>

#include "op/op.h"
```

```
typedef struct optModuleObjectS {
    // insert module persistent data here
} optModuleObject;

////////////////////////////////////////////////////////////////////////////////
//                      U S E R   F U N C T I O N S
////////////////////////////////////////////////////////////////////////////////

static OP_PRE_SIMULATE_FN(modulePreSimulate) {
// insert modulePreSimulate code here
}

static OP_SIMULATE_STARTING_FN(moduleSimulate) {
// insert moduleSimulate code here
}

static OP_POST_SIMULATE_FN(modulePostSimulate) {
// insert modulePostSimulate code here
}

static OP_DESTRUCT_FN(moduleDestruct) {
// insert moduleDestruct code here
}

#include "module.igen.h"
```

We can see it is just the placeholders of the callbacks and then it includes the iGen generated constructor code we don't edit.

6.2 Simple C module

Above we saw a module with all its callbacks, *modelAttrs* etc.

It is possible to write more simplistic modules with just a *main()*, in the same way that the *Examples/SimulationControl/minimalHarness* was structured, just creating instances of buses, memories, processors etc.

Unless there is a specific reason, our suggestion is

a) use iGen from *module.op.tcl*, and then

b) if you need to, edit only the generated *module.c* file.

7 Adding C code to a module

In this chapter we will create a module using a tcl iGen script, and then edit the C to add some specific C code to add some behaviors that are not possible to add using iGen from tcl.

There will be examples of adding a monitor to detect changes on nets, adding watchpoints to see memory accesses, and adding behavior using C.

7.1 Monitoring nets

Monitoring nets from C code in the modules will be shown using the following example:

```
Examples\SimulationControl\monitoringNetsInModule
```

It is a copy of:

```
PlatformConstruction/simpleCpuMemoryUart
```

Which creates a module using iGen from tcl, and has a simple application that writes to a UART and logs the output to a file. It also has two nets that the UART can configure to use. There is no need for our own C harness, so we use the provided *harness.exe*.

So let's follow the process we used:

```
> cp Examples\ PlatformConstruction/simpleCpuMemoryUart .
```

and run:

```
./example.sh

OVPsim started: Wed Jan 13 15:29:54 2016
Initializing KinetisUART
Writing to uart.
OVPsim finished: Wed Jan 13 15:29:54 2016
```

and then see the output:

```
> cat uartTTY0.log
Hello UART0 world
```

So now let's make some very simple changes in *module.c*. First add the call back declaration:

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

```
}
```

and in the pre-simulate callback add our callback on all nets in the module:

```
static OP_PRE_SIMULATE_FN(modulePreSimulate) {
        // set up net monitoring
         optNetP net = 0;
         while ((net = opNetNext(mi, net))) {
             opPrintf ("monitorNets(%s)\n", opObjectHierName(net));
              opNetWriteMonitorAdd(net, netCallback, net);
         }

      // and put UART into 'using nets' mode, using local hierarchical path
      opParamBoolOverride(mi,  "periph0/directReadWrite", 1);
}
```

NOTE that we have to set the periph0/directReadWrite parameter of the UART, or we get the callbacks registered, but it just writes to the logfile and not the nets.

and then we get:

```
./example.sh
OVPsim started: Wed Jan 13 15:35:10 2016

monitorNets(harness/simpleCpuMemoryUart/directWrite)
monitorNets(harness/simpleCpuMemoryUart/directRead)
Initializing KinetisUART
Writing to uart.

netCallback(directWrite) =
netCallback(directWrite) = H
netCallback(directWrite) = e
netCallback(directWrite) = l
netCallback(directWrite) = l
netCallback(directWrite) = o
netCallback(directWrite) =
netCallback(directWrite) = U
netCallback(directWrite) = A
netCallback(directWrite) = R
netCallback(directWrite) = T
netCallback(directWrite) = 0
netCallback(directWrite) =
netCallback(directWrite) = w
netCallback(directWrite) = o
netCallback(directWrite) = r
netCallback(directWrite) = l
```

```
netCallback(directWrite) = d
netCallback(directWrite) =


OVPsim finished: Wed Jan 13 15:35:10 2016
```

7.2 Listing of C code for module monitoring its nets

```
> cat Examples/SimulationControl/monitoringNetsInModule/module/module.c
#include <string.h>
#include <stdlib.h>
#include "op/op.h"

typedef struct optModuleObjectS {
    // insert module persistent data here
} optModuleObject;

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

//                      U S E R   F U N C T I O N S
static OP_PRE_SIMULATE_FN(modulePreSimulate) {
    // set up net monitoring
            optNetP net = 0;
            while ((net = opNetNext(mi, net))) {
                    opPrintf ("monitorNets(%s)\n", opObjectHierName(net));
                opNetWriteMonitorAdd(net, netCallback, net);
            }

    // and put UART into 'using nets' mode, using local hierarchical path
    opParamBoolOverride(mi,  "periph0/directReadWrite", 1);
}

static OP_SIMULATE_STARTING_FN(moduleSimulate) {
// insert moduleSimulate code here
}
static OP_POST_SIMULATE_FN(modulePostSimulate) {
// insert modulePostSimulate code here
}
static OP_DESTRUCT_FN(moduleDestruct) {
// insert moduleDestruct code here
}

#include "module.igen.h"
```

7.3 Monitoring Memory locations and Buses using callbacks

A previous example Examples/SimulationControl/monitoringMemory/harness added monitoring for memory and buses into the harness.

To add this code to a module is very simple. We have created an example in Examples/SimulationControl/monitoringMemoryInModule and then copied in the relevant code from monitoringMemory/harness/harness.c into module/module.c with very few changes.

The code in SimulationControl/monitoringMemoryInModule/module/module.c now looks like:

```
> cat SimulationControl/monitoringMemoryInModule/module/module.c
```

```
#include <string.h>
#include <stdlib.h>

#include "op/op.h"


typedef struct optModuleObjectS {
    // insert module persistent data here
    Uns32 count;
} optModuleObject;

#define PREFIX          "BUS_MON"
#define MIN_ADDRESS 0x0
#define MAX_ADDRESS 0xffffffff

//
// triggered when registered access happens and prints information of access
//
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);
}

//
// iterate across the buses found in the module and register callbacks
// for read, write and fetch
//
static void monitorBus(optModuleP mi) {
    // iterate across all busses found in module
    optBusP bus = 0;
    while ((bus = opBusNext(mi, bus))) {
        Addr max = MAX_ADDRESS; // TODO: should be call to opBusMaxAddress(bus);
        opMessage("I", PREFIX "_BM", "Add monitor for '%s' (0x" FMT_A0Nx " to 0x" FMT_A0Nx ")\n",
```

```
                                opObjectHierName(bus), (Addr)0, max);
        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");
    }
}

//
// iterate across the memories found in the module and register callbacks for read, write and fetch
//
static void monitorMemory(optModuleP mi) {
    // iterate across all memories found in module
    optMemoryP memory = 0;
    while ((memory = opMemoryNext(mi, memory))) {
        Addr max = opMemoryMaxAddress(memory);
        opMessage("I", PREFIX "_BM", "Add monitor for '%s' (0x" FMT_A0Nx " to 0x" FMT_A0Nx ")\n",
                                opObjectHierName(memory), (Addr)0, max);
        opMemoryFetchMonitorAdd(memory, 0, MIN_ADDRESS, max, monitorCallback, "memory-fetch");
        opMemoryReadMonitorAdd (memory, 0, MIN_ADDRESS, max, monitorCallback, "memory-read");
        opMemoryWriteMonitorAdd(memory, 0, MIN_ADDRESS, max, monitorCallback, "memory-write");
    }
}

static OP_PRE_SIMULATE_FN(modulePreSimulate) {
// insert modulePreSimulate code here
        // Setup bus monitors
        monitorBus(mi);
        // Setup memory monitors
        monitorMemory(mi);
}

static OP_SIMULATE_STARTING_FN(moduleSimulateStart) {
// insert moduleSimulate code here
}

static OP_POST_SIMULATE_FN(modulePostSimulate) {
// insert modulePostSimulate code here
}

static OP_DESTRUCT_FN(moduleDestruct) {
// insert moduleDestruct code here
}

#include "module.igen.h"
```

Note the use of the module instance 'mi' in the two iterators:

```
    while ((bus = opBusNext(mi, bus))) {
```

```
    while ((memory = opMemoryNext(mi, memory))) {
```

And when it is run we get:

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

```
OVPsim started: Wed Jan 13 16:26:23 2016

Info (BUS_MON_BM) Add monitor for 'harness/simpleCpuMemory/mainBus' (0x0000000000000000
            to 0x00000000ffffffff)
```

```
Info (BUS_MON_BM) Add monitor for 'harness/simpleCpuMemory/ram1' (0x0000000000000000
        to 0x00000000ffffffff)

Info (BUS_MON_MT) Monitor triggered: callback 'monitorCallback': processor 'cpu1' : type 'bus-fetch' : bytes 4 :
        address Physical 0x0000000000000100 Virtual  0x0000000000000100
Info (BUS_MON_MT) Monitor triggered: callback 'monitorCallback': processor 'cpu1' : type 'bus-fetch' : bytes 4 :
        address Physical 0x0000000000000104 Virtual  0x0000000000000104
...
Info (BUS_MON_MT) Monitor triggered: callback 'monitorCallback': processor 'cpu1' : type 'bus-fetch' : bytes 4 :
        address Physical 0x0000000000000f38 Virtual  0x0000000000000f38
Info (BUS_MON_MT) Monitor triggered: callback 'monitorCallback': processor 'cpu1' : type 'bus-write' : bytes 4 :
        address Physical 0x00000000ffffffe4 Virtual  0x00000000ffffffe4
Info (BUS_MON_MT) Monitor triggered: callback 'monitorCallback': processor 'cpu1' : type 'bus-fetch' : bytes 4 :
        address Physical 0x0000000000000f3c Virtual  0x0000000000000f3c
...
Info (BUS_MON_MT) Monitor triggered: callback 'monitorCallback': processor 'cpu1' : type 'bus-fetch' : bytes 4 :
        address Physical 0x000000000000171c Virtual  0x000000000000171c
Info (BUS_MON_MT) Monitor triggered: callback 'monitorCallback': processor 'cpu1' : type 'bus-read' : bytes 4 :
        address Physical 0x0000000000004d8c Virtual  0x0000000000004d8c
Info (BUS_MON_MT) Monitor triggered: callback 'monitorCallback': processor 'cpu1' : type 'bus-fetch' : bytes 4 :
        address Physical 0x000000000000166c Virtual  0x000000000000166c
...

Info (BUS_MON_MT) Monitor triggered: callback 'monitorCallback': processor 'cpu1' : type 'bus-fetch' : bytes 4 :
         address Physical 0x0000000000001780 Virtual  0x0000000000001780
Info (BUS_MON_MT) Monitor triggered: callback 'monitorCallback': processor 'cpu1' : type 'bus-fetch' : bytes 4 :
        address Physical 0x000000000000176c Virtual  0x000000000000176c
Info (BUS_MON_MT) Monitor triggered: callback 'monitorCallback': processor 'cpu1' : type 'bus-fetch' : bytes 4 :
        address Physical 0x0000000000001770 Virtual  0x0000000000001770
Info (BUS_MON_MT) Monitor triggered: callback 'monitorCallback': processor 'cpu1' : type 'bus-fetch' : bytes 4 :
        address Physical 0x0000000000004674 Virtual  0x0000000000004674

OVPsim finished: Wed Jan 13 16:26:23 2016
```

7.4 Adding behavioral C code in modules

The examples above showed the use of C code to monitor objects in a module.

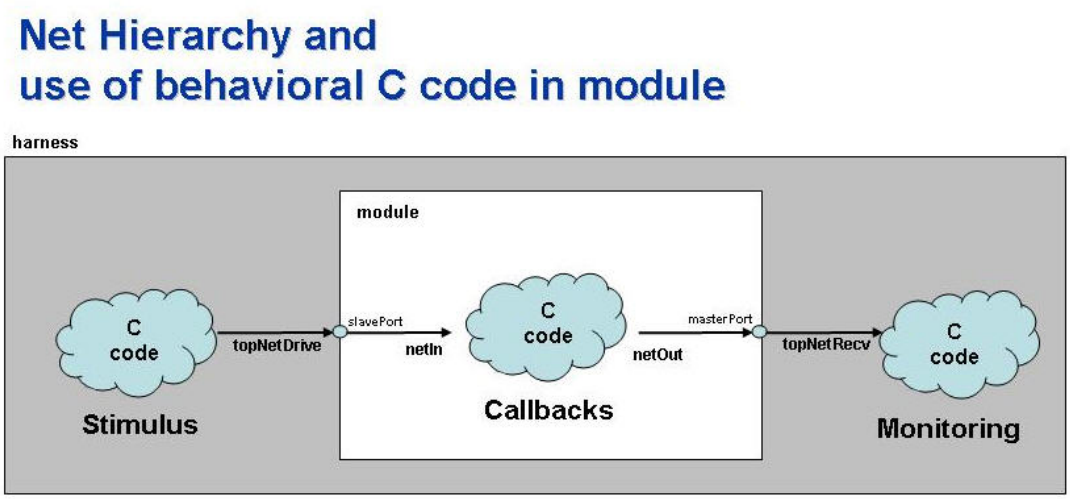
In this very small example there is a small amount of behavioral C code in the middle of the module. It reads a net value and writes it to another net inverted. Yes a behavioral not gate...

```
> cp -r Examples/SimulationControl/behavioralCcodeInModule .
```

There is no harness for this example, we will use the provided *harness.exe*.

7.4.1 A mixed TCL and C module

There is a simple module with net in, net out and some C code, and harness that instantiates it.



The module is created from two files:

module/module.op.tcl	the structural definition of the module
module/module.c	the C code that is called by the structural code

The iGen tcl input script should need no discussion:

```
> cat module/module.op.tcl
ihwnew -name module

ihwaddnet -instancename netIn
ihwaddnet -instancename netOut

ihwaddnetport -instancename slavePort
ihwaddnetport -instancename masterPort

ihwconnect -netport slavePort -net netIn
ihwconnect -netport masterPort -net netOut
```

We initially wrote the *module.op.tcl*, and then ran *make -C module* and this generated the initial copy of module.c, which now it exists, the make system will not overwrite so it is safe for us to edit and change.

The C code we write in *module/module.c* needs a little explanation. Let's first look at it in its entirety, and then explain each section:

```
> cat module/module.c
```

```
#include <assert.h>
#include <string.h>
#include <stdlib.h>

#include "op/op.h"
```

```
typedef struct optModuleObjectS {
    // insert module persistent data here

    optNetP netIn;
    optNetP netOut;

} optModuleObject;

//
// This function is called when the input net is written
// It writes the same value to the output net
//
static OP_NET_WRITE_FN(netCallback) {
    optModuleObjectP object = userData;

    Uns32 newValue = (value) ? 0 : 1;
    opPrintf(FMT_TIME ": sub_netCallback: %s = %u\n",
            opModuleCurrentTime(opObjectRootModule(object->netOut)),
            opObjectHierName(object->netOut), newValue);

    opNetWrite(object->netOut, newValue);
}

static OP_PRE_SIMULATE_FN(modulePreSimulate) {
//
// Get handles to the two nets
//
    opPrintf("modulePreSimulate: %s setting up callback on netIn\n",
            opObjectHierName(mi));

    object->netIn  = opObjectByName(mi, "netIn", OP_NET_EN).Net;
    object->netOut = opObjectByName(mi, "netOut").Net;
    assert(object->netIn != NULL && object->netOut != NULL);

    opNetWriteMonitorAdd(object->netIn, netCallback, object);

    opPrintf("modulePreSimulate: %s done\n", opObjectHierName(mi));
}

static OP_SIMULATE_STARTING_FN(moduleSimulate) {
// insert moduleSimulate code here
}

static OP_POST_SIMULATE_FN(modulePostSimulate) {
// insert modulePostSimulate code here
```

```
}

static OP_DESTRUCT_FN(moduleDestruct) {
// insert moduleDestruct code here
}

#include "module.igen.h"
```

The declaration of:

```
typedef struct optModuleObjectS {
    optNetP netIn;
    optNetP netOut;
} optModuleObject;
```

is declaring module persistent variables to hold references to the nets in the harness.

The declaration of the pre-simulate callback:

```
static OP_PRE_SIMULATE_FN(modulePreSimulate) {
```

obtains handles to the nets and saves them in the module persistent data

```
    object->netIn  = opObjectByName(mi, "netIn", OP_NET_EN).Net;
    object->netOut = opObjectByName(mi, "netOut", OP_NET_EN).Net;
    assert(object->netIn != NULL && object->netOut != NULL);
```

Note the use of *assert*. This is to check that we have got the correct names of our nets. We don't want to change them in the *module.op.tcl* and forget to update the C. The good thing about *assert* is it tells us file and line numbers so it is easy to see what has gone wrong;

And then we need to register the *netCallback* function to be called back when the *object->netIn* is written to. This is a net callback and will be triggered when the net is written to:

```
    opNetWriteMonitorAdd(object->netIn, netCallback, object);
```

The 3rd argument is passing the *object* in so that when the callback is called it has access to the module persistent data, in this case the handles to the nets.

The definition of the *netCallback* is where we write some behavior:

```
static OP_NET_WRITE_FN(netCallback) {
    optModuleObjectP object = userData;

    Uns32 newValue = (value) ? 0 : 1;
```

```
    opPrintf(FMT_TIME ": sub_netCallback: %s = %u\n",
            opModuleCurrentTime(opObjectRootModule(object->netOut)),
            opObjectHierName(object->netOut), newValue);

    opNetWrite(object->netOut, newValue);
}
```

The callback is defined using a macro that provides several variables for our use.
userData is the 3rd argument of the call that registers the callback
value is the value being written to the net
With:

```
    Uns32 newValue = (value) ? 0 : 1;
```

we are creating a simple inverter or not gate function.

And:

```
    opNetWrite(object->netOut, newValue);
```

writes the new value to the output net.

Note that the call back was registered on the input net.

```
    opNetWriteMonitorAdd(object->netIn, netCallback, object);
```

Then there are the declarations of the other user callback functions - and these are left as doing nothing.

This is compiled with the normal make command:

```
> make -C module
# Igen Create OP MODULE module
# Host Depending obj/Linux32/module.d
# Host Compiling Module obj/Linux32/module.o
# Host Linking Module object model.so
```

7.4.2 Advancing time and writing nets in a harness

The harness has in it a main() which creates the top module and an instance of our submodule. It then calls a function we have written to monitor the nets, and a function to control the simulation:

```
int main(int argc, const char *argv[]) {
...
    optModuleP mi = opRootModuleNew(0, "top", 0);
    constructModule(mi);
```

```
    monitorNets(mi);

    opRootModulePreSimulate(mi);
    simulate(mi);
...
}
```

The *constructModule* creates the sub module instance:

```
optNetP topNetDrive;
optNetP topNetRecv;
...
static void constructModule(optModuleP mi) {
    topNetDrive = opNetNew(mi, "topNetDrive", 0, 0);
    topNetRecv = opNetNew(mi, "topNetRecv", 0, 0);

    opModuleNew(
        mi,       // parent module
        "module/model",       // modelfile
        "u1",   // name
        OP_CONNECTIONS(
            OP_NET_CONNECTIONS(
                OP_NET_CONNECT(topNetDrive, "slavePort"),
                OP_NET_CONNECT(topNetRecv, "masterPort")
            )
        ),
        0
    );
}
```

Noting that it creates two global nets (topNetDrive, topNetRecv) , and then connects them up to the slave and master ports of the module. The sub module file will be located at: *module/model.*

As in other examples, in this harness we are going to register a callback on the output net to see any writes to it:

```
static void monitorNets(optModuleP mi) {
    opPrintf("monitorNets: %s setting up callback on nets\n", opObjectHierName(mi));

    assert(topNetDrive != NULL && topNetRecv != NULL);

    opNetWriteMonitorAdd(topNetRecv, netCallback, topNetRecv);

    opPrintf("monitorNets: %s : done\n", opObjectHierName(mi));
```

```
}
```

Note the use of *assert*:

```
    assert(topNetDrive != NULL && topNetRecv != NULL);
```

This is to check that we have in fact located the nets in our global persistent data.

We are registering a call back on the net *topNetRecv* being written to and when it is, we want our function, *netCallback* to be called. We will pass in the handle of the net that has changed (*topNetRecv*).

And when the net is written to, our function:

```
static OP_NET_WRITE_FN(netCallback) {
    optNetP net = userData;
    opPrintf(FMT_TIME ": top_netCallback: %s = %d\n",
             opModuleCurrentTime(opObjectRootModule(net)),
            opObjectHierName(net), value);
}
```

will be called. It just writes out that the net was written to, and its new value, and the time of the write.

All that remains is to have some stimulus in our harness. This is in our simulate function:

```
static void simulate(optModuleP mi) {
    opPrintf("\n");
    opPrintf(FMT_TIME ": simulate: %s starting\n",
            opModuleCurrentTime(mi), opObjectHierName(mi));

    opRootModuleSetSimulationStopTime(mi, 0.001) ;
    opRootModuleSimulate(mi);
    opPrintf("\n");
    opPrintf(FMT_TIME ": simulate: %s writing 0\n",
            opModuleCurrentTime(mi), opObjectHierName(topNetDrive));
    opNetWrite(topNetDrive, 0);

    opRootModuleSetSimulationStopTime(mi, 0.002);
    opRootModuleSimulate(mi);
    opPrintf("\n");
    opPrintf(FMT_TIME ": simulate: %s writing 1\n",
            opModuleCurrentTime(mi), opObjectHierName(topNetDrive));
    opNetWrite(topNetDrive, 1);

    opRootModuleSetSimulationStopTime(mi, 0.003);
```

```
    opRootModuleSimulate(mi);

    opPrintf("\n");
    opPrintf(FMT_TIME ": simulate: %s done\n\n",
            opModuleCurrentTime(mi), opObjectHierName(mi));
}
```

the key bit is:

```
    opRootModuleSetSimulationStopTime(mi, 0.001) ;
    opRootModuleSimulate(mi);
...
    opNetWrite(topNetDrive, 0);
```

which sets a time in the future at which the simulator should stop and then, when stopped (return from opRootModuleSimulate), write to the net.

In other words:

- run
- set value
- run
- set value
- ...

i.e. very simple stimulus.

7.4.2.1 The full harness listing

```
> cat harness/harness.c

#include <assert.h>
#include <string.h>
#include <stdlib.h>

#include "op/op.h"

optNetP topNetDrive;
optNetP topNetRecv;

static OP_NET_WRITE_FN(netCallback) {
    optNetP net = userData;
    opPrintf(FMT_TIME ": top_netCallback: %s = %d\n",
            opModuleCurrentTime(opObjectRootModule(net)),
            opObjectHierName(net), value);
}

static void monitorNets(optModuleP mi) {
```

```
    opPrintf("monitorNets: %s setting up callback on nets\n",
            opObjectHierName(mi));

    assert(topNetDrive != NULL && topNetRecv != NULL);

    opNetWriteMonitorAdd(topNetRecv, netCallback, topNetRecv);

    opPrintf("monitorNets: %s : done\n", opObjectHierName(mi));
}

static void constructModule(optModuleP mi) {
    topNetDrive = opNetNew(mi, "topNetDrive", 0, 0);
    topNetRecv = opNetNew(mi, "topNetRecv", 0, 0);

    opModuleNew(
        mi,       // parent module
        "module/model",       // modelfile
        "u1",   // name
        OP_CONNECTIONS(
            OP_NET_CONNECTIONS(
                OP_NET_CONNECT(topNetDrive, "slavePort"),
                OP_NET_CONNECT(topNetRecv, "masterPort")
            )
        ),
        0
    );
}

static void simulate(optModuleP mi) {
    opPrintf("\n");
    opPrintf(FMT_TIME ": simulate: %s starting\n",
            opModuleCurrentTime(mi), opObjectHierName(mi));

    opRootModuleSetSimulationStopTime(mi, 0.001) ;
    opRootModuleSimulate(mi);
    opPrintf("\n");
    opPrintf(FMT_TIME ": simulate: %s writing 0\n",
            opModuleCurrentTime(mi), opObjectHierName(topNetDrive));
    opNetWrite(topNetDrive, 0);

    opRootModuleSetSimulationStopTime(mi, 0.002);
    opRootModuleSimulate(mi);
    opPrintf("\n");
    opPrintf(FMT_TIME ": simulate: %s writing 1\n",
            opModuleCurrentTime(mi), opObjectHierName(topNetDrive));
    opNetWrite(topNetDrive, 1);
```

```
    opRootModuleSetSimulationStopTime(mi, 0.003);
    opRootModuleSimulate(mi);

    opPrintf("\n");
    opPrintf(FMT_TIME ": simulate: %s done\n\n",
            opModuleCurrentTime(mi), opObjectHierName(mi));
}

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

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

    optModuleP mi = opRootModuleNew(0, "top", 0);
    constructModule(mi);

    monitorNets(mi);

    opRootModulePreSimulate(mi);
    simulate(mi);

    opSessionTerminate();
    return 0;
}
```

7.4.3 Running the simulation

```
> ./example.sh
...
OVPsim started: Thu Jan 14 15:01:23 2016

monitorNets: top setting up callback on nets
monitorNets: top : done
modulePreSimulate: top/u1 setting up callback on netIn
modulePreSimulate: top/u1 done

0: simulate: top starting

0.001: simulate: top/topNetDrive writing 0
0.001: sub_netCallback: top/u1/netOut = 1
0.001: top_netCallback: top/topNetRecv = 1

0.002: simulate: top/topNetDrive writing 1
0.002: sub_netCallback: top/u1/netOut = 0
0.002: top_netCallback: top/topNetRecv = 0
```

```
0.003: simulate: top done

OVPsim finished: Thu Jan 14 15:01:23 2016
```

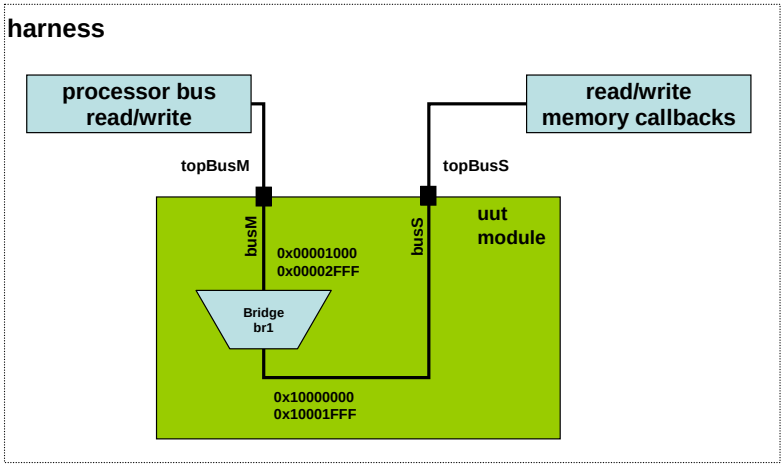
8 Using Hierarchical Connections

When hierarchy is used there are specific connection types that can be made between the different levels. These can be a bus, a FIFO, a net or a packetnet. The following examples show the use of each.

8.1 Bus

There is an example provided as Examples/PlatformConstruction/busHierarchy that shows the connection of busses between levels of hierarchy.

This example contains a simple module that has a two bus ports connected by a bridge. The bridge maps an address range on the input (slave) port from 0x10000000 to 0x10001fff to an address range on the output (master) port from 0x00000000 to 0x00001fff. So, for example, an access at 0x10000004 will be translated to an access at 0x00000004.



The module is instanced in a test harness and the bus connections from the module are connected to buses instanced in the harness. Monitors are added to the bus to detect read or write accesses on the buses.

```
static OP_BUS_SLAVE_READ_FN(readCallback) {
    const char *bus = userData;
    opPrintf("readCallback: %s, " FMT_Ax "\n", bus, addr);
}

static OP_BUS_SLAVE_WRITE_FN(writeCallback) {
    const char *bus = userData;
    opPrintf("writeCallback: %s, " FMT_Ax "\n", bus, addr);
}
```

Using

```
char *busName = "topBusS";
optBusP bus = opObjectByName(mi, busName, OP_BUS_EN).Bus;
opBusSlaveNew(bus, "all", 0, OP_PRIV_RWX, 0, 0xffffffff,
                                    readCallback, writeCallback, 0, busName);
```

The harness performs bus read/writes accesses on the bus to test the connections. These will be debug artifact accesses, as there is no processor specified in the read/write calls. [Note that the last argument to the calls is the *debugAccess* bit, and when set to 1 means that the read/write should not perturb the platform state - and in this case as there is no processor that is the case anyway.]

```
    Bool okRead  = opBusRead  (bus, 0, base, &b, sizeof(b), 1);
    Bool okWrite = opBusWrite (bus, 0, base, &b, sizeof(b), 1);
```

When the example script is executed the example is built and then executed. The following shows the expected output:

```
...
modulePreSimulate: top setting up callbacks on slave bus
modulePreSimulate: top done
moduleSimulate: top starting
moduleSimulate: top reading and writing to 0
moduleSimulate: top         read: 0, write: 0
moduleSimulate: top reading and writing to 1000
readCallback: topBusS, 10000000
writeCallback: topBusS, 10000000
moduleSimulate: top         read: 1, write: 1
moduleSimulate: top reading and writing to 2000
readCallback: topBusS, 10001000
writeCallback: topBusS, 10001000
moduleSimulate: top         read: 1, write: 1
moduleSimulate: top reading and writing to 3000
moduleSimulate: top         read: 0, write: 0
moduleSimulate: top done
...
```

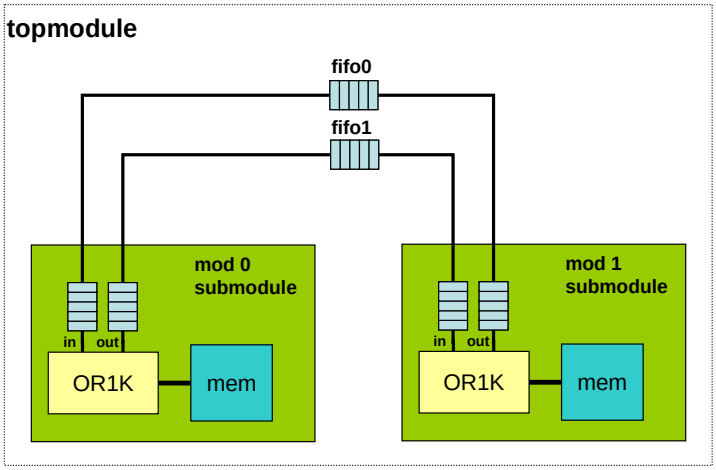
8.2 FIFO

There is an example provided as Examples/PlatformConstruction/FIFOHierarchy that shows the connection of buses between levels of hierarchy.

This example contains two modules.

The first module ‘submodule’ instances an OR1K processor and two FIFOs instances connected between the OR1K processor FIFO ports and ports on the module.

The second module ‘topmodule’ is used to instance ‘submodule’ twice and to connect the two modules FIFO out ports to the other modules FIFO in ports, as shown in the diagram



And as shown in the following extract from topmodule/module.op.tcl

```
# Create two modules that will communicate via the FIFO
ihwaddmodule  -instancename mod0 -modelfile submodule/model
ihwaddmodule  -instancename mod1 -modelfile submodule/model

ihwaddfifo -instancename fifo0 -width 32
ihwaddfifo -instancename fifo1 -width 32
```

When the example is run the processors within each ‘submodule’, that is ‘mod0’ and ‘mod1’, execute the application messageIn and a messageOut respectively to transfer a message using the FIFO connections between them.

The application makes use of inline assembler in order to generate the correct instruction/operation. There is a special purpose register added to the OR1K processor that allows access to the read and write FIFOs.

Message out application messageout.c

```
#include <stdio.h>

const char *message = "This is the Imperas simulator communicating with FIFOs\n";

static void writeToFifo(unsigned int value) {
   asm volatile("l.mtspr r0,%0,0x123" :: "r"(value));
}

int main() {
```

```
    unsigned int i = 0;
    unsigned int c;

    while((c=message[i++])) {
        writeToFifo(c);
    }

    writeToFifo(0);

    return 0;
}
```

Message in application messagein.c

```
#include <stdio.h>

static unsigned int readFromFifo(void) {
    unsigned int result;
    asm volatile("l.mfspr %0,r0,0x123" : "=r"(result));
    return result;
}

int main() {

    unsigned int c;

    while((c=readFromFifo())) {
        printf("%c", c);
    }

    return 0;
}
```

When the applications are executed in the virtual platform the output message is displayed when reception is complete as shown.

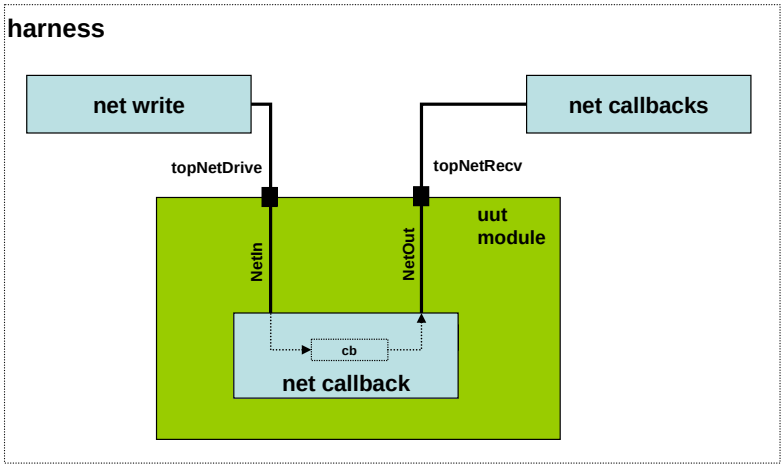
```
This is the Imperas simulator communicating with FIFOs
```

8.3 Net

There is an example provided as Examples/PlatformConstruction/netHierarchy that shows the connection of buses between levels of hierarchy.

This example contains a module and a test harness.

The module has hardware generated that contains two net ports and connected nets, ‘netIn’ and ‘netOut’.



The module C file installs a net callback function on the input net to transfer the value to the output net

Net callback function, writes value from netIn onto netOut

```
static OP_NET_WRITE_FN(netCallback) {

    optModuleObjectP object = userData;

    opPrintf("netRepeater: netCallback: set '%s' to %u\n",
                        opObjectName(object->netOut),
                        value);

    opNetWrite(object->netOut, value);
}
```

Install callback onto the net netIn

```
    opNetWriteMonitorAdd(object->netIn, netCallback, object);
```

the harness instantiates the module and adds a callback onto netOut to monitor any changes to its value.

Defining callback

```
static OP_NET_WRITE_FN(netCallback) {

    opPrintf("uut: netCallback: topNetRecv = %u\n", value);
}
```

Adding callback

```
    opNetWriteMonitorAdd(topNetRecv, netCallback, 0);
```

running the example script builds and then executes the simulation which should provide the following output

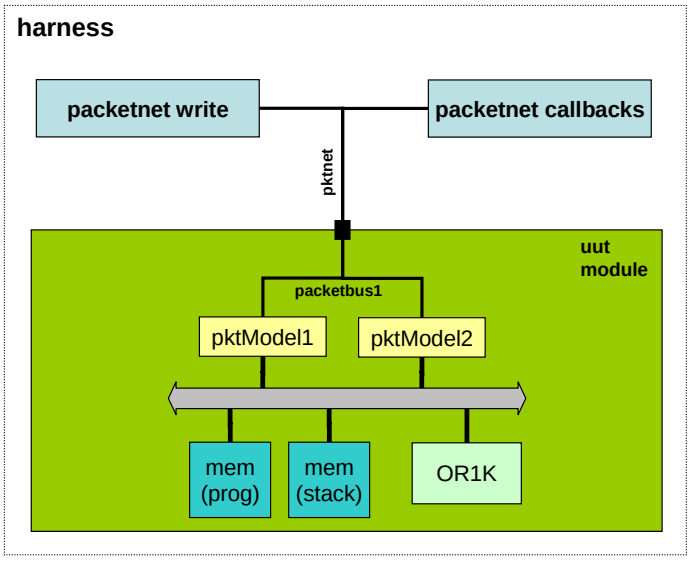
```
uut: setting up callback on net 'topNetRecv'
netRepeater: setting up net repeater 'uut/netIn' to 'uut/netOut'
uut: simulate: writing net 'topNetRecv' to 0
netRepeater: netCallback: set 'netOut' to 0
uut: netCallback: topNetRecv = 0
uut: simulate: writing net 'topNetRecv' to 1
netRepeater: netCallback: set 'netOut' to 1
uut: netCallback: topNetRecv = 1
uut: simulate: done
```

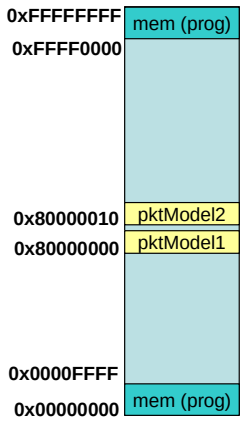
8.4 Packetnet

There is an example provided as Examples/PlatformConstruction/packetnetHierarchy that shows the connection of buses between levels of hierarchy.

This example contains a module, a test harness and a peripheral using a packetnet.

The module instances an OR1K processor and two packetnet peripherals whose registers are based at locations 0x80000000 and 0x80000010 in the processor memory space.





Platform Memory Map

The application running on the OR1K processor starts a transfer by the first packetnet peripheral and then on the second.

```
int main(int argc, char **argv)
{

   LOG("PACKETNET TEST Application\n\n");
   delay(1000);

   // Write to the transmit register of the first instance, causing the model
   // to transmit. The value 0x77 goes in the packet header, so we can see which
   // instance was triggered.
   writeReg8(0x80000000, 0, 0x77);

   delay(1000);
   // Write to the transmit register of the second instance, causing the model
   // to transmit. The value 0x88 goes in the packet header, so we can see which
   // instance was triggered.
   writeReg8(0x80000010, 0, 0x88);

   delay(1000);
   LOG("PACKETNET TEST Application DONE\n\n");
   return 1;
}
```

The packetnet ports are instanced in the harness which allows them to be monitored and written to.

The harness installs a callback on the packetnet input to monitor packets being received

```
static OP_PACKETNET_WRITE_FN(packetnetFunction) {
   netPacketP p = data;
   if(p && bytes) {
      opMessage("I", HARNESS_NAME "_PKT_RXD",
         "Platform PKT testbench Trigger bytes=%u {%02x}, {%s}, {%02x}
UD=0x%x\n",
         bytes,
         p->head,
         p->data,
         p->tail,
         (Uns32)(UnsPS)userData
      );

      // increment this tail value. In our simple protocol this tells anyone
      // watching the packet that we received it.
      p->tail++;
   }
}
```

Using

```
    //
    // Add callback onto the packetnet connection
    //
    opPacketnetWriteMonitorAdd(pktnet_pkn, packetnetFunction, 0);
```

and writes a frame of data to the packetnet interface

```
//
// Send a packetnet and reports the fact
//
static void packetWrite(optModuleP mi) {
    optPacketnetP pkn = opObjectByName(mi, "pktnet",
                                                OP_PACKETNET_EN).Packetnet;

    netPacket p = { 1, {'I', 'C', 'M', 0, 0, 0 }, 0 };
    opMessage("I", HARNESS_NAME "_PKT_TXS",
        "%s: writePkt START {%02x}, {%s}, {%02x}",
        opObjectHierName(pkn),
        p.head,
        p.data,
        p.tail
    );

    // This is the function that triggers the packetnet.
    // All models with a callback installed on this packetnet
    // will be notified while this function is active.

    opPacketnetWrite(pkn, &p, sizeof(netPacket));

    // Now that the function has returned, we are certain that all
    // models have been notified. Their notifier callbacks have finished,
    // and if they chose to modify the data in the packet, we can see the
    // modification here.

    opMessage("I", HARNESS_NAME "_PKT_TXE",
        "%s: writePkt DONE {%02x}, {%s}, {%02x}",
        opObjectHierName(pkn),
        p.head,
        p.data,
        p.tail
    );
    opPrintf("\n");
}
```

The example is built and executed with the example script, the execution should provide output similar to the following:

```
...
Info PACKETNET TEST Application


Info PACKETNET TEST Application

Info (PKT_PSETXS) testpacketnet/pktModel1: PSE to packetnet START {77} {PSETX} {00}
Info (PKT_PSERXD) testpacketnet/pktModel2: Peripheral PKT model Trigger bytes=8 {77}, {PSETX}, {00} UD=33
Info (testpacketnet_PKT_RXD) Platform PKT testbench Trigger bytes=8 {77}, {PSETX}, {01} UD=0x0
Info (PKT_PSETXE) testpacketnet/pktModel1: PSE to packetnet DONE  {77} {PSETX} {02}

Info (PKT_PSETXS) testpacketnet/pktModel2: PSE to packetnet START {88} {PSETX} {00}
Info (PKT_PSERXD) testpacketnet/pktModel1: Peripheral PKT model Trigger bytes=8 {88}, {PSETX}, {00} UD=33
Info (testpacketnet_PKT_RXD) Platform PKT testbench Trigger bytes=8 {88}, {PSETX}, {01} UD=0x0
Info (PKT_PSETXE) testpacketnet/pktModel2: PSE to packetnet DONE  {88} {PSETX} {02}

Info PACKETNET TEST Application DONE

Info (testpacketnet_PKT_TXS) testpacketnet/pktNet: writePkt START {01}, {APITX}, {00}
Info (PKT_PSERXD) testpacketnet/pktModel1: Peripheral PKT model Trigger bytes=8 {01}, {APITX}, {00} UD=33
Info (PKT_PSERXD) testpacketnet/pktModel2: Peripheral PKT model Trigger bytes=8 {01}, {APITX}, {01} UD=33
Info (testpacketnet_PKT_TXE) testpacketnet/pktNet: writePkt DONE {01}, {APITX}, {02}

...
```

9 Introducing Basic Processor Introspection

Previously, we saw an example which used standard instantiation attributes to enable tracing. The trace file output was generated in a fixed order. Occasionally, it might be necessary to generate trace information in a different format: for example, if a platform is being used to generate trace output to compare with the output from another tool, comparison is much easier if the format from the platform can be made to exactly match the other tool.

The OP API contains a number of functions allowing processor instance registers to be read and written. These can be used to construct test platforms that generate trace output from a simulation run in whatever format required. These access functions allow:

- Access to instance program counter;
- Access to any processor register by name;
- Dump of processor registers;
- Disassembly of the current instruction;
- Access to the count of instructions executed by the processor.

There is also a function available that allows a processor model to be stepped by a single instruction, opProcessorSimulate, which will be used in this example.

9.1 Reading and Writing Registers

There are a set of functions that allow access to the current program counter and to access the program counter while determining if the current instruction is being executed in the delay slot (for processors that support delay slot instructions) of a processor.

The processor can be found using the *opProcessorNext* function applied to a module, for example to get the first processor in a module (mi), use the following

```
optProcessorP processor = opProcessorNext(mi, NULL);
```

To access the current program counter, use opProcessorPC:

```
Addr currentpc = opProcessorPC(processor);   // get current PC
```

The return value from *opProcessorPC* is of type *Addr*, which is a 64-bit unsigned integer. For processors with address widths less than 64 bits, this value should be cast to an appropriate sized value if it is to be used subsequently in an arithmetic expression; for example:

```
Uns32 currentpc = (Uns32)opProcessorPC(processor);   // get current PC as 32-bit value
```

For processors with delay slot instructions, it is sometimes useful to know whether the current instruction is a delay slot instruction. To do this, use *opProcessorPCDS*:

```
Uns8  delaySlotOffset;
```

```
Uns32 branchPC = opProcessorPCDS(processor, &delaySlotOffset);
```

opProcessorPCDS behaves as follows:

1. If the current instruction is not a delay slot instruction, it returns the current program counter and sets the *byref* value *delaySlotOffset* to 0;
2. If the current instruction is a delay slot instruction, it returns the address of the preceding branch instruction and sets the byref value *delaySlotOffset* to the current instruction byte offset from the branch instruction. For example, if there is a branch instruction at 0x1000 with a delay slot instruction at 0x1004, then if *opProcessorPCDS* is called when the processor is executing the delay slot instruction at 0x1004, it will return 0x1000 and set *delaySlotOffset* to 4.

The current value of any processor register can be found using *opProcessorRegRead*, which fills a *byref* argument buffer with the current value of a named register:

```
Bool opProcessorRegRead (optProcessorP processor, optRegP reg, void *buffer);
```

To write a processor register, there is a similar function *opProcessorRegWrite*:

```
Bool opProcessorRegWrite(optProcessorP processor, optRegP reg, void *buffer);
```

The following code snippet shows how a processor register called *R1* can be masked with a bitmask *REG_FLAG_MASK* in a platform:

```
#define REG_FLAG_MASK 0x00f0000f
...
Uns32 regR1;
optRegP reg = opProcessorRegByName(processor, "R1");
if(reg) {
   if (opProcessorRegRead(processor, reg, regR1)) {
      regR1 = regR1 & REG_FLAG_MASK;
      opProcessorRegWrite(processor, reg, regR1);
   }
}
```

It is the responsibility of the calling function to ensure that the buffer value is the correct size to hold the register data. For example, the above example implicitly requires that register *R1* is a 32-bit register which will fit in a value of type *Uns32*. The *opRegBits* function may be used to verify this.

The function *opProcessorPCSet* can be used to set the processor's start-address without knowing the name of the PC in the particular model being used (not everyone calls it 'PC').

It is also possible within a platform to iterate over all the registers in a processor instance to determine their names and sizes (in bits) using three functions: *opProcessorRegNext*, *opRegName* and *opRegBits*.

opProcessorRegNext returns an opaque pointer of type *optRegP*, which describes a single processor register. It takes as an argument the previously-returned *optRegP* value; when passed a *NULL* pointer, it returns the first *optRegP* pointer for a processor mode. It can therefore be used to iterate over all register descriptions for a processor in a simple loop:

```
optRegP info = 0;    // initiate loop with NULL pointer

while((info=opProcessorRegNext(processor, info))) {
   . . .
}
```

Given an *optRegP* pointer, the name of the register it corresponds to can be found using *opRegName* and the register size in bits can be found using *opRegBits*:

```
optRegP info = 0;    // initiate loop with NULL pointer

while((info=opProcessorRegNext(processor, info))) {

   const char *name = opRegName(info);
   Uns32          bits = opRegBits(info);

   opPrintf("Found %u-bit register %s\n", name, bits);
}
```

The *optRegP* returned by *opProcessorRegNext* can be used, if required, to identify the register to read or write using *opProcessorRegRead* or *opProcessorRegWrite*. Alternatively the register can be read or written directly using its name using *opProcessorRegReadByName* and *opProcessorRegWriteByName* respectively.

opRegUsageEnum returns an enumeration and *opRegUsageString* returns a string describing if the register has special use. Extending the above example to show this information

```
optRegP info = 0;    // initiate loop with NULL pointer

while((info=opProcessorRegNext(processor, info))) {

   const char *name = opRegName(info);
   Uns32          bits = opRegBits(info);

   opPrintf("Found %u-bit register %s: usage (%d) %s\n",
                                 name,
```

```
                    bits,
                    opRegUsageEnum(info),
                    opRegUsageString(info));
```

9.2 Generating Disassembly Output

Processor models contain instruction disassembly functionality that can be accessed from a platform using *opProcessorDisassemble*, which returns a string disassembly of an instruction at a passed address. For example, to print the disassembled instruction at the current program counter:

```
opPrintf("%s", opProcessorDisassemble(processor, opProcessorPC(processor)));
```

9.3 Dumping Registers

Processor models also contain functionality to dump all processor register values in a standard format. This can done using *opProcessorRegDump*:

```
opProcessorRegDump(processor);
```

9.4 Instruction Counts

Every processor also maintains a count of the number of instructions that it has executed (as a 64-bit unsigned integer). This can be accessed using the OP function *opProcessorICount*; for example, to print the number of instructions executed at the end of simulation:

```
opPrintf(
   "Simulation finished, "FMT_64u" instructions executed...\n",
   opProcessorICount(processor)
);
```

NOTE: The macro *FMT_64u* defines a format string that will correctly print a 64-bit unsigned integer on both Linux and Windows hosts. It is defined with other similar macros in *ImpPublic/include/host/impTypes.h*.

9.5 Example Processor Introspection

An example is provided in *Examples/SimulationControl/basicProcessorIntrospection* that utilizes the functions described above.

Take a copy of the example and run the script which will build both the test application, the hardware definition in the module and the test harness.

```
> cp -r Examples/SimulationControl/basicProcessorIntrospection .
```

```
> cd basicProcessorIntrospection
> example.sh
```

```
# Compiling application.c
# Linking application.OR1K.elf
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 Depending obj/Linux32/harness.d
# Host Compiling Platform obj/Linux32/harness.o
# Host Linking Platform harness.Linux32.exe
# Host Linking Platform object model.so
...
Info (MODULE) Introspect processor ‘cpu1’
Found 32-bit register R0: usage (0) general
Found 32-bit register R1: usage (2) stack pointer
Found 32-bit register R2: usage (3) frame pointer
Found 32-bit register R3: usage (0) general
...
Found 32-bit register TTCR: usage (0) general
Found 32-bit register TTMR: usage (0) general
Found 32-bit register EXCPT: usage (0) general
```

9.6 Example Custom Processor Trace

An example is provided in *Examples/SimulationControl/customProcessorTrace* that utilizes the functions described above

Take a copy of the example and run the script which will build both the test application, the hardware definition in the module and the test harness.

```
> cp -r Examples/SimulationControl/customProcessorTrace .
> cd customProcessorTrace
> example.sh
```

When the simulation executes, the following should be observed:

```
Info (MODULE) Trace processor 'cpu1'
1 : 0x104 : l.addi   r3,r0,0x0
2 : 0x108 : l.addi   r4,r0,0x0
3 : 0x10c : l.addi   r5,r0,0x0
4 : 0x110 : l.addi   r6,r0,0x0
5 : 0x114 : l.addi   r7,r0,0x0
```

```
6 : 0x118 : l.addi   r8,r0,0x0
…
2100 : 0x177c : l.j      0x0000176c
2101 : 0x1780 : l.nop    0x0
2102 : 0x176c : l.jal    0x00004674
2103 : 0x1770 : l.ori    r3,r10,0x0
2104 : 0x4674 : l.addi   r1,r1,0xfffffffc
2105 : 0x1774 : l.jalr   r4
Simulation finished, stop reason 'CPU has exited' after 2105 instructions executed
```

10 Simulating a Design

10.1 Simulator Scheduler

The scheduler controls the execution of instructions on each of the processor models that may be in the platform and also when peripheral models timed events occur.
It is the recommended approach to use the simulator internal built-in scheduler to execute simulation. However, in circumstances when you wish to control how the processor models instruction execution are scheduled and how time is moved forward OP API functions can be used in its place.

How processors execute instructions and how time is moved forward may be controlled when using the standard scheduler by setting the processor MIPS rate and the platform quantum.
When using a custom scheduler it is the combination of the arguments passed to the functions that determine how instructions executed and time are related.

It is possible to create the same execution with both the standard and custom schedulers as shown in the example in section 10.1.3 Example below.

10.1.1 The standard built in (default) simulator scheduler algorithm

The platforms/modules created using iGen and the OP API would normally be simulated using the default scheduling algorithm. By using the default algorithm the harness/modules can be directly imported into the Imperas professional tools without any modification.

The default scheduling is performed by a call to the `opRootModuleSimulate` function. This runs all processor and peripheral instances in the platform. There is one argument, a root module.

```
opRootModuleSimulate( myRootModule )
```

The default scheduling algorithm described in section 10.6 is used in *harness.exe*, and all the examples in the documents up to this point.

10.1.2 Writing a custom scheduler

Examples of when a custom scheduler would be useful are:

1. in co-simulation to allow explicit control of components
2. when fine grain control of the interaction between components in a system is required

A custom scheduling algorithm is created using the *opProcessorSimulate* and *opRootModuleTimeAdvance* functions in place of the `opRootModuleSimulate` function.

The function *opProcessorSimulate* is applied to only one processor instance. To simulate the platform all processors and peripherals in the platform must be scheduled. The function *opProcessorSimulate* is used for each processor instance in turn to make them execute a fixed number of instructions. The number of instructions a processor can execute in a given slice of time is a product of the performance of the processor and the length of time the time slice occupies.

The sequence to simulate a platform is to schedule each processor to execute the number of instructions it can nominally achieve in the time slice. Once all processors have executed any instructions they should perform in a time slice the platform time is moved forward in time by the appropriate amount using the *opRootModuleTimeAdvance* function. In moving time forward any peripheral functionality that is waiting for an amount of time to expire within this time slice will execute its behavior. *opRootModuleTimeAdvance* returns *False* if the new time is at or beyond a requested stop time (see *opRootModuleSetSimulationStopTime*).

10.1.3 Example

This example is found in the customScheduler directory.

```
$IMPERAS_HOME/Examples/SimulationControl/customScheduler
```

The example shows how a custom scheduler is used to control the execution of an application on a processor while moving time forward so that events within peripheral models occur at the correct rate.

```
#define INSTRUCTIONS_PER_SECOND         10000000
#define QUANTUM_TIME_SLICE              0.00001
#define INSTRUCTIONS_PER_TIME_SLICE     (INSTRUCTIONS_PER_SECOND *
                                         QUANTUM_TIME_SLICE)
```

```
    // must advance to next phase for the API calls that follow
    opRootModulePreSimulate(mi);

    if (options.custom) {

        // run simulation with custom scheduling
        optTime             myTime  = QUANTUM_TIME_SLICE;
        optStopReason stopReason = OP_SR_SCHED;
        do {

            // move time forward by time slice on root module
            // NOTE: This matches the standard scheduler which moves time forward in
            //       the system and then executes instructions on all processors
            opRootModuleTimeAdvance(mi, myTime);
```

```
            opMessage(
                "I", HARNESS_NAME,
                "Advance Time to %g seconds",
                (double)myTime
            );

            // run processor for number of instructions calculated for time slice
            stopReason = opProcessorSimulate(proc,
                                        INSTRUCTIONS_PER_TIME_SLICE);
            if ((stopReason!=OP_SR_SCHED) && (stopReason!=OP_SR_HALT)) {
                opMessage(
                    "I", HARNESS_NAME,
                    "Simulation Complete (%s)",
                    opStopReasonString(stopReason)
                );

                break;  // finish simulation loop

            }

            myTime += QUANTUM_TIME_SLICE;

        } while (1);

    } else {

        // run with built in standard scheduler
        opRootModuleSimulate(mi);

    }
```

Compile the test harness, example module and application as before using the following commands in the memory directory:

```
> make –C harness
> make –C module
> make –C application CROSS=OR1K
```

To run the simulation, in the customScheduler directory, run :

```
# run with custom scheduler
> harness/harness.${IMPERAS_ARCH}.exe \
        --program application/application.${CROSS}.elf \
        --output custom.log \
        --custom \
```

```
        $@
```

You should see the following output:

```
Info (harness) Running with custom scheduler
Info (16550_BRS) u1/uartTTY0: baud rate=1152000  parity=N  data bits=5  total bits=7
character delay=6usec
Info (harness) Advance Time to 1e-05 seconds
Info (harness) Advance Time to 2e-05 seconds
Info (16550_UWR) u1/uartTTY0: Write to Data register: data=0x48 ('H')
Info (harness) Advance Time to 3e-05 seconds
Info (16550_UWR) u1/uartTTY0: Write to Data register: data=0x65 ('e')
Info (harness) Advance Time to 4e-05 seconds
…
Info (16550_UWR) u1/uartTTY0: Write to Data register: data=0x6c ('l')
Info (harness) Advance Time to 0.00085 seconds
Info (harness) Advance Time to 0.00086 seconds
Info (16550_UWR) u1/uartTTY0: Write to Data register: data=0x6f ('o')
Info (harness) Advance Time to 0.00087 seconds
Info (16550_UWR) u1/uartTTY0: Write to Data register: data=0x20 (' ')
Info (harness) Advance Time to 0.00088 seconds
Info (16550_UWR) u1/uartTTY0: Write to Data register: data=0x38 ('8')
Info (harness) Advance Time to 0.00089 seconds
Info (16550_UWR) u1/uartTTY0: Write to Data register: data=0x0a (' ')
Info (harness) Advance Time to 0.0009 seconds
Info (harness) Simulation Complete (CPU has exited)
Info (harness) Time at Completion 0.0009 seconds
```

To run the simulation, with the standard scheduler, in the customScheduler directory, run :

```
# run with standard scheduler
> harness/harness.${IMPERAS_ARCH}.exe \
        --program application/application.${CROSS}.elf \
        --output standard.log \
        $@
```

You should see the similar output i.e. the same but without the time advance information

```
Info (harness) Running with standard scheduler
Info (16550_BRS) u1/uartTTY0: baud rate=1152000  parity=N  data bits=5  total bits=7
character delay=6usec
Info (16550_UWR) u1/uartTTY0: Write to Data register: data=0x48 ('H')
Info (16550_UWR) u1/uartTTY0: Write to Data register: data=0x65 ('e')
```

```
…
Info (16550_UWR) u1/uartTTY0: Write to Data register: data=0x6c ('l')
Info (16550_UWR) u1/uartTTY0: Write to Data register: data=0x6c ('l')
Info (16550_UWR) u1/uartTTY0: Write to Data register: data=0x6f ('o')
Info (16550_UWR) u1/uartTTY0: Write to Data register: data=0x20 (' ')
Info (16550_UWR) u1/uartTTY0: Write to Data register: data=0x38 ('8')
Info (16550_UWR) u1/uartTTY0: Write to Data register: data=0x0a (' ')
Info (harness) Time at Completion 0.0009 seconds
```

Note the time at completion is the same in both cases.

10.2 Custom Simulation Tracing with Harness

10.2.1 Controlling Instructions Executed on a Processor

Previous examples have used the function opRootModuleSimulate, which simulates a design using a built-in scheduling algorithm that simulates each processor for many instructions, or until the simulation is finished, before returning to the harness. For this example, we instead want to simulate a processor one instruction at a time, performing custom instruction tracing after each one completes. To do this, use opProcessorSimulate:

```
optStopReason opProcessorSimulate (optProcessorP  processor, Uns64  instructions);
```

opProcessorSimulate runs the passed processor for up to *instructions* more instructions and then returns. The precise reason why simulation stopped is indicated by the return code provided by the enumerated type optStopReasonE:

Stop Reason	Description
OP_SR_SCHED	Scheduler expired.
OP_SR_YIELD	Yield encountered.
OP_SR_HALT	CPU is halted.
OP_SR_EXIT	CPU has exited.
OP_SR_FINISH	Simulation finish.
OP_SR_RD_PRIV	Read privilege exception.
OP_SR_WR_PRIV	Write privilege exception.
OP_SR_RD_ALIGN	Read align exception.
OP_SR_WR_ALIGN	Write align exception.
OP_SR_FE_PRIV	Fetch privilege exception.
OP_SR_ARITH	Arithmetic exception.
OP_SR_INTERRUPT	Interrupt simulation.
OP_SR_FREEZE	Frozen (by opProcessorFreeze).
OP_SR_WATCHPOINT	Memory watchpoint is pending.
OP_SR_BP_ICOUNT	Instruction count breakpoint is pending.
OP_SR_BP_ADDRESS	Address breakpoint is pending.
OP_SR_RD_ABORT	Read abort exception.
OP_SR_WR_ABORT	Write abort exception.

OP_SR_FE_ABORT	Fetch abort exception.

The three most common return codes are:
OP_SR_SCHED: processor successfully simulated the required number of instructions and returned
OP_SR_EXIT: processor has exited (but in a multiprocessor platform, other processors may still be running)
OP_SR_FINISH: simulation has finished

In practice, it is usually sufficient to continue simulation while the return code from opProcessorSimulate is OP_SR_SCHED, for example:

```
while(opProcessorSimulate(processor, 1)==OP_SR_SCHED) {
   . . .
}
opPrintf(
   "Simulation finished, “FMT_64u” instructions executed\n",
   opProcessorICount(processor)
);
```

Note that simulating one instruction at a time is much less efficient than simulating for a larger number of instructions, so this should only be used when needed, such as when tracing each instruction as we are doing here.

10.2.2 Generating Disassembly Information

The disassembly information is normally generated when the built-in tracing is enabled, using --trace. However, API functions are also available to allow some custom tracing to be generated within the harness.

The disassembly output can be obtained as a string using the opProcessorDisassemble function and providing the processor and the address of interest

```
opPrintf(“0x%08x : %s\n”, thisPC, opProcessorDisassemble (processor, thisPC);
```

10.2.3 Accessing Registers

The registers of a processor can each be individually accessed using the either an iterator function opProcessorRegNext or by name using opProcessorRegByName. These can then be queried and values or other information provided.

All of the standard core registers may also be displayed directly from the model as a listing using the opProcessorRegDump function. This will format the registers as dictated by the model or if there is no model function as 4 registers per line.

10.2.4 Example

The following example uses the functions described above to control the order of instruction disassembly, register dumping and instruction execution.

This example is found in the processorRegisterAccess directory.

```
$IMPERAS_HOME/Examples/ControlSimulation/processorRegisterAccess
```

The module is defined in module/module.op.tcl as a platform with a single processor and memory. The harness that instantiate the module and controls the simulation is in harness/harness.c, is as follows:

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

    opCmdParseStd(MODULE_NAME, OP_AC_ALL, argc, argv);

    optModuleP mr = opRootModuleNew(0, MODULE_NAME, 0);

    const char *u1_path = "module";
    optModuleP mi = opModuleNew(
        mr,                 // parent module
        u1_path,            // modelfile
        MODULE_INSTANCE,    // name
        0,
        0
    );

    // get the handle for the processor in the module
    optProcessorP processor = opObjectByName(mi, "cpu1",
OP_PROCESSOR_EN).Processor;

    // construction complete
    opRootModulePreSimulate(mr);

    Bool done = False;
    while(!done) {

        Uns32 currentPC = (Uns32)opProcessorPC(processor);

        // disassemble instruction at current PC
        opPrintf("** Instruction Disassemble\n");
        opPrintf(
            "0x%08x : %s\n",
                currentPC,
                opProcessorDisassemble(processor, currentPC)
```

```
        );

        // execute one instruction and check for none scheduler return
        opPrintf("** Instruction Execution\n");
        done = (opProcessorSimulate(processor, 1) != OP_SR_SCHED);

        // dump registers
        opPrintf("** Register Dump\n");
        opProcessorRegDump(processor);
    }

    // print number of instructions executed at end of simulation and reason simulation
stopped
    opPrintf(
        "Simulation finished, "FMT_64u" instructions executed, StopReason is '%s'\n",
              opProcessorICount(processor),
              opStopReasonString(opProcessorStopReason(processor))
    );

    opSessionTerminate();
    return 0;
}
```

Compile the module, test harness and application using the following commands in the processorRegisterAccess directory:

```
make -C module
make -C harness
make -C application
```

To run the simulation, in the processorRegisterAccess directory, run :

```
./harness/harness.Linux32.exe --program application/asmtest.OR1K.elf
```

You should see the following output:

```
** Instruction Disassemble
0x01000074 : l.addi   r1,r0,0x0
** Instruction Execution
** Register Dump
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000000   R2 : deadbeef   R3 : deadbeef
 R4 : deadbeef   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : deadbeef   R10: deadbeef   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
```

```
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: deadbeef   R31: deadbeef
 PC : 01000078   SR : 00008001   ESR: deadbeef   EPC: deadbeef
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:0 CF:0 OF:0
--------------- --------------- --------------- ---------------
… etc …

** Instruction Disassemble
0x01000120 : l.addi   r1,r0,0x0
** Instruction Execution
** Register Dump
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000000   R2 : 00000001   R3 : ffffffff
 R4 : 80000000   R5 : 7fffffff   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : deadbeef   R10: deadbeef   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: 80000000   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: deadbeef   R31: deadbeef
 PC : deadbeef   SR : 00008801   ESR: deadbeef   EPC: deadbeef
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:0 CF:0 OF:1
--------------- --------------- --------------- ---------------

Simulation finished, 44 instructions executed, StopReason is 'CPU has exited'
```

10.3 Interrupt a Running Simulation

Normally, opProcessorSimulate and opRootModuleSimulate will run until they have completed the requested number of simulated instructions (for opProcessorSimulate) or time has advanced until the time specified by opRootModuleSetSimulationStopTime (for opRootModuleSimulate), or until a processor model has performed some explicit action that terminates the simulation loop early (for example, halting or exiting) or the simulation completes and all processors have finished execution.

10.3.1 Interrupt Simulation (from a Cntrl-C Handler)

Occasionally, it may be required that the opProcessorSimulate or opRootModuleSimulate call be terminated early by some external event. For example, the harness may implement an interrupt handler so that when a user presses Ctrl-C the simulation loop should

immediately terminate. This can be done using the opInterrupt[1] API call from within a signal handler, shown in the following code snippet (Linux only):

```
#include <signal.h>

//
// LINUX signal handler to interrupt the running simulation
//
static void ctrlCHandler(Int32 nativeSigNum, siginfo_t *sigInfo, void *context) {
    opInterrupt();
}

//
// Install a LINUX signal handler to trap any CtrlC
//
static void installCtrlCHandler(void) {

    struct sigaction sa = {{0}};
    sa.sa_sigaction = ctrlCHandler;
    sa.sa_flags     = SA_SIGINFO;
    sigfillset(&sa.sa_mask);
    sigaction(SIGINT, &sa, NULL);
}
```

Within the main function, the Ctrl-C handler is installed:

```
int main(int argc, char ** argv) {

    . . .

    // install a signal handler to trap any CtrlC
    installCtrlCHandler();

    . . .
}
```

When the user presses Ctrl-C, while this example is running, a call to the ctrlCHandler occurs which calls opInterrupt. This will cause any active opProcessorSimulate or opRootModuleSimulate call to return, and the stopReason for the processor that stops will be set to OP_SR_INTERRUPT.

This needs to be handled in the main routine, for example:

[1] The API function opInterrupt is the only API call that can be made when a simulation is running. This causes the simulator to return when it reaches a stable state from which it can do so.

```
optProcessorP stoppedProcessor;

// simulate until done or ctrl-C
while((stoppedProcessor=opRootModuleSimulate(mr))) {
    if(opProcessorStopReason(stoppedProcessor)==OP_SR_INTERRUPT) {
        opPrintf(
            "%s: interrupt after " FMT_64u " instructions...\n",
            opObjectName ((optObjectP)stoppedProcessor),
            opProcessorICount(stoppedProcessor)
        );
    } else {
        break;
    }
}
```

In this example, when an interrupt occurs, the platform prints a message and continues simulation by calling opRootModuleSimulate again (which will continue from where it was interrupted). In real cases, applications will typically enter a command interpreter instead at this point.

In the common case that simulation needs to be interrupted on a Ctrl-C event, The API provides a method that does not require OS-specific signal handler code: simply specify OP_FP_STOPONCONTROLC, either applied to the root module as an override

```
opParamBoolOverride(0, OP_FP_STOPONCONTROLC, 1);
optModuleP mr = opRootModuleNew(0, 0, 0);
```

or applied as parameters to the root module

```
optModuleP mr = opRootModuleNew(
                0,
                0,
                OP_PARAMS(
                    OP_PARAM_BOOL_SET(OP_FP_STOPONCONTROLC, 1)
                )
            );
```

This will have exactly the same effect as an OS-specific interrupt handler calling opInterrupt.

10.3.2 Interrupt a Specific Processor

It may be required that a specific processor is interrupted when it accesses an area of memory. This can be achieved using the opProcessorYield API call. This will cause the simulator to return from the opProcessorSimulate or opRootModuleSimulate function after the instruction in which the opProcessorYield was called has completed.

In this example opProcessorYield is called when one of the processors makes a write to a specific address range that triggers the memory watchpoint callback.

```
OP_BUS_SLAVE_WRITE_FN(extMemoryWrite) {
    opMessage("I", "EXTERNAL_MEMORY_WRITE",
                    "%s access at address 0x" FMT_Ax " data 0x%08x",
              processor ? opObjectHierName(processor) : "artifact", addr, *(Uns32 *)data);

   // Calling this API function will interrupt the simulator
   opProcessorYield(processor);
 }
```

10.3.3 Example

This example is found in the interruptSimulation directory.

```
$IMPERAS_HOME/Examples/SimulationControl/interruptSimulation
```

Compile the test harness, module and application using the following commands in the interruptSimulation directory:

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

To run the simulation, in the interruptSimulation directory, run:

```
> ./harness/harness.Linux32.exe --program application/application.OR1K.elf
```

You should see output as in example multiprocessor as the two processors execute the application.

After a number of iterations the application will make a write that will cause a call to opInterrupt, when data written is 610. This will cause the simulator to return from the opRootModuleSimulate and a message will be generated of the form:

```
CPU (writer): fib(15) = 610
Info (MEMORY_ACCESS) top/u1/cpu1 Write at address 0xe0001004 (0xe0001004)
data 0x00000262 (610)
Info (HARNESS) cpu1: interrupt after 1504523 instructions 'Interrupt simulation'
Info (MEMORY_ACCESS) top/u1/cpu2 Read at address 0xe0001004 (0xe0001004) data
0x00000262 (610)
Info (HARNESS) cpu2: interrupt after 1600036 instructions 'Interrupt simulation'
CPU (reader): munge(610) = 185745
```

The application will make a write that will cause a call to opProcessorYield, when the data written is 6765. This will cause the simulator to return from opRootModuleSimulate and a message will be generated of the form:

```
CPU (writer): fib(20) = 6765
Info (MEMORY_ACCESS) top/u1/cpu1 Write at address 0xe0001004 (0xe0001004)
data 0x00001a6d (6765)
Info (HARNESS) cpu1: interrupt after 3147106 instructions 'Yield was encountered'
Info (MEMORY_ACCESS) top/u1/cpu2 Read at address 0xe0001004 (0xe0001004) data
0x00001a6d (6765)
Info (HARNESS) cpu2: interrupt after 3200018 instructions 'Yield was encountered'
CPU (reader): munge(6765) = 22879230
```

You may also press Ctrl-C repeatedly while the application runs; each time, a line will be generated of the form:

```
Info (HARNESS) cpu1: interrupt after <ins count> instructions 'Interrupt simulation'
```

or:

```
Info (HARNESS) cpu2: interrupt after <ins count> instructions 'Interrupt simulation'
```

depending which processor is running when the Ctrl-C is hit.

10.3.4 Important Notes

10.3.4.1 API Usage in Ctrl-C Handler

When in a Ctrl-C or other similar handler and you want to cause the simulation to be interrupted i.e. return from the opProcessorSimulate or opRootModuleSimulate functions the opInterrupt may be used. However, it is important that no other API calls should be made from within a handler of this type. To do so may result in unexpected behavior.

10.3.4.2 opInterrupt Usage

One important point about opInterrupt is that it is not intended to be asynchronously thread-safe. In other words, it is not appropriate to asynchronously call opInterrupt when the simulation thread is not suspended. In virtual platform simulations with multiple asynchronous threads, the interrupting thread should be designed to work as follows:

1. It should suspend the simulating thread using any appropriate means;
2. It should call opInterrupt to notify the suspended thread that an interrupt has been requested;
3. It should restart the simulating thread so that the interrupt request can be acted on.

If this sequence is not followed, simulator data structures may become corrupted.

10.4 Generating External Events to a Processor

Processor models written using the VMI interface can be made to react to external interrupt events on named ports. For example, a processor model can be made to perform

a hard reset on an event on its ‘reset’ port or take an exception and execute from an exception vector on a change to an interrupt or exception port input.

These types of ‘interrupt’ and ‘exception’ events are signaled to processor models using nets.
A net is a hardware connection that can signal state; typically asserted and de-asserted are signaled but a net can be used to pass an unsigned 32bit value.

10.4.1 Processor Reset

The processor ‘reset’ port can be connected to a net that can be driven from the test harness or from another component (typically a peripheral) in the design.

When the ‘reset’ is asserted[2] by writing a ‘1’ to the net the processor is held in a reset state until the ‘reset’ is de-asserted by writing a ‘0’ to the net.

The following illustrates controlling the ‘reset’ connection from a test harness.

The net object ‘reset’ must be found in the module. This can be done by searching for the object by name, as shown, or the net could be exported from the module using net ports and connected in the harness

```
optNetP reset = opObjectByName(mi, "reset", OP_NET_EN).Net;
```

and then may be used to assert the reset

```
opNetWrite(reset, 1);
```

or de-assert the reset

```
opNetWrite(reset, 0);
```

the reset is connected to the processor in the module (module.op.tcl snippet):

```
# connect reset to cpu1
ihwconnect -instancename cpu1 -netport reset -net reset
```

and may be driven from a peripheral in the module, as well as from the harness

```
ihwaddperipheral -instancename resetControlPeripheral  \
                 -type resetControl \
                 -modelfile peripheral/pse.pse
```

[2] All OVP/Imperas processor models use a high ‘1’ written to the ‘reset’ net to cause the assertion of the reset; this may not be representative of the true hardware signal polarity.

```
ihwconnect -instancename resetControlPeripheral -netport resetOut  -net reset
```

10.4.2 Processor Startup Reset

The processors in a design can be brought out of reset under control of other components, for example an application on one processor can write a peripheral register which releases the reset on a second processor which allows it to start execution.

The mechanism used can be modeled to match the same mechanism in the hardware definition.

The startup reset can also be controlled from the harness. For example the following code extract shows a two processor system in which the second processor, cpu2, is brought out of reset 0.1 seconds after the first processor, cpu1, using the cpu2 reset connection, reset2.

Obtain the reset connections in the design

```
    // get reset nets
    optNetP reset2 = opObjectByName(mi, "reset2", OP_NET_EN).Net;
```

assert the reset on processor cpu2

```
    // Hold processor cpu2 in reset
    opNetWrite(reset2, 1);
```

simulate for the required amount of time

```
    opRootModuleSetSimulationStopTime(mr, 0.1);
    opRootModuleSimulate(mr);
```

release processor cpu2 reset

```
    // Release processor2
    opNetWrite(reset2, 0);
```

simulate until both processors complete

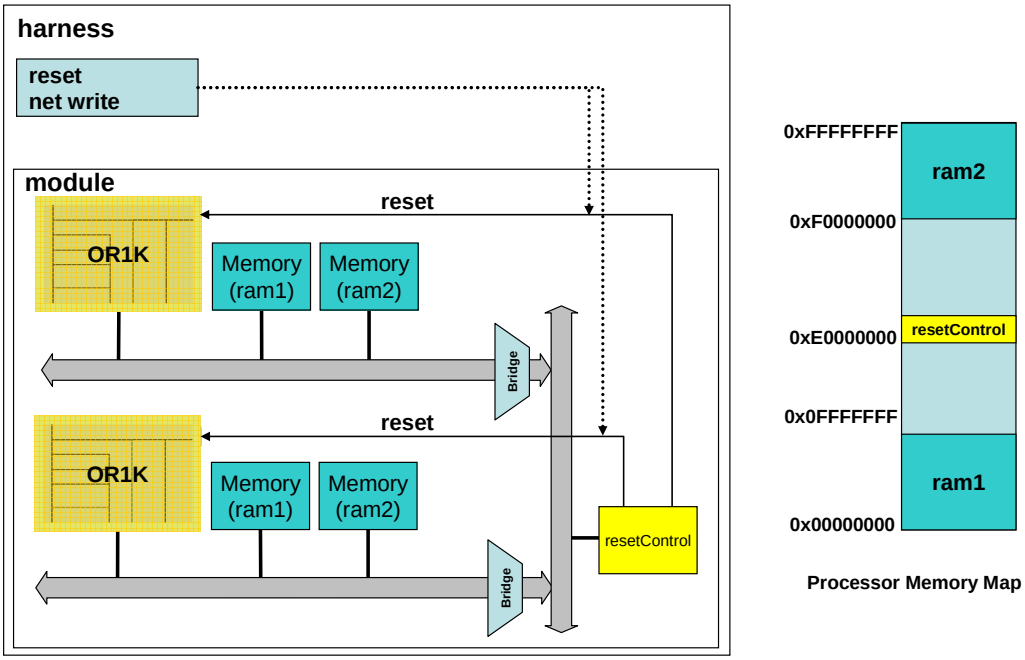
```
    opRootModuleSimulate(mr);
```

10.4.3 Processor Reset Example

This example is found in the processorResetControl directory.

$IMPERAS_HOME/Examples/SimulationControl/processorResetControl

This example includes a design that instantiates two processors and one peripheral model. Two reset lines connect from the peripheral one to each of the processors and is driven by the peripheral, as shown in the following diagram.



The application on processor cpu1 is controlling the reset of processor cpu2 the application of which is running a fibonacci application which includes the generation of code at the reset vector so that on a reset it jumps back to the initial entry point at ‘start’.

The peripheral model has a control register that can be accessed by either processor that is used to toggle the value of one or both of the reset connections. It also opens a socket, using the support included in the BHM API calls, and monitors characters received over the socket. If an ‘r’ character is received both reset lines are driven high then low to generate a reset, if a ‘q’ character is received the simulation is terminated. The simulation will also terminate if the socket is disconnected (terminal window closed).

The control register behavioral code of the peripheral, peripheral/user.c, is as follows:

```
// Control Register Write
PPM_REG_WRITE_CB(controlRegisterWrite) {

   // if bit one set toggle reset1
   if (SWAP4(data) & 1) {
      if(ppmReadNet(handles.resetOut1)) {
         bhmMessage("I", PSENAME "_CRWC1",
                  " Control Register Write: Release reset1");
```

```
            ppmWriteNet(handles.resetOut1, 0);
        } else {
            bhmMessage("I", PSENAME "_CRWS1",
                    " Control Register Write: Reset reset1");
            ppmWriteNet(handles.resetOut1, 1);
        }
    }
    // if bit two set toggle reset2
    if (SWAP4(data) & 2) {
        if(ppmReadNet(handles.resetOut2)) {
            bhmMessage("I", PSENAME "_CRWC2",
                    " Control Register Write: Release reset2");
            ppmWriteNet(handles.resetOut2, 0);
        } else {
            bhmMessage("I", PSENAME "_CRWS2",
                     " Control Register Write: Reset reset2");
            ppmWriteNet(handles.resetOut2, 1);
        }
    }
    // store current write value
    *(Uns32*)user = data;
}
```

Note that the data value written is byte swapped. This is because the OR1K processor being used uses big endian data ordering, the PSE uses host endian data ordering and on the x86 this is little endian.

The main routine of the peripheral, is as follows:

```
    if (channel) {

                bhmSerWriteN(channel, startMessage, sizeof(startMessage));

                while (1) {
                        //
                        // Non Blocking read
                        //
                        Uns8 buffer[MAXREAD];
                        Uns8 bytes = bhmSerReadN(channel, buffer, 1);

                        if (bytes) {
                                // echo value back to terminal
                                bhmSerWriteN(channel, buffer, 1);

                                if (buffer[0] == 'r') {
```

```
                bhmMessage("I", PSENAME "_MT", "generate reset on reset1");
                        ppmWriteNet(handles.resetOut1, 1);
            ppmWriteNet(handles.resetOut1, 0);
            bhmMessage("I", PSENAME "_MT", "generate reset on reset2");
            ppmWriteNet(handles.resetOut2, 1);
            ppmWriteNet(handles.resetOut2, 0);
                    }
                    if (buffer[0] == 'q') {
            bhmMessage("I", PSENAME "_MT", "Force Finish Simulation");
                        bhmFinish();
                    }
                }
                // polling delay
                bhmWaitDelay(100);

        }
```

Open a socket, waiting for a connection to be made

```
    bhmSerOpenAuto();
```

Attempt to read a single character from the socket. This is a non-blocking read (there is an equivalent blocking read, bhmSerReadN) so if nothing is available it returns with result 0.

```
    bhmSerReadN(channel, buffer, 1);
```

If a character is available and it is ‘r’ the peripheral toggles the ‘reset’ net (handles.resetOut is the handle to the opened port on the peripheral to which the reset line is connected) to generate a reset signal causing the processor to reset.

```
    ppmWriteNet(handles.resetOut1, 1);
    ppmWriteNet(handles.resetOut,1 0);
```

Compile the test harness, module, peripheral and application using the following commands in the processorResetControl directory:

```
> make –C harness
> make –C module
> make –C peripheral
> make –C application CROSS=OR1K
```

To run the simulation, in the processorResetControl directory, run:

```
> ./harness/harness.Linux32.exe \
        --program top/u1/cpu1=application/application1.OR1K.elf \
```

```
 --program top/u1/cpu2=application/application2.OR1K.elf \
--override top/u1/resetControlPeripheral/outfile=uart.log \
--output imperas.log
```

You should see output as follows:

Simulator log, showing the following.

Initially both processors are held in reset (reset lines active)

```
Info (HARNESS) Reset processor cpu1 and cpu2
Info (HARNESS) Simulate to 0.001
Info (HARNESS) Release processor cpu1
Info (HARNESS) Simulate to End
```

After 0.001 seconds of simulation time processor cpu1 reset line, reset1, is released and the processor starts executing application1. This application executes a delay before writing to the resetControl peripheral to release the reset to processor cpu2 which then starts executing application2 which executes the fibonacci sequence several times.

```
CPU1: starting...
CPU1: delay (1000000)
CPU1: release cpu2 reset
Info (RESET_CONTROL_CRWC2) top/u1/resetControlPeripheral:  Control Register
Write: Release reset2
CPU2: starting...
CPU2: fib(0) = 0
CPU2: fib(1) = 1
CPU2: fib(2) = 1
CPU2: fib(3) = 2
CPU1: delay (1000000)
CPU2: fib(4) = 3
…
CPU2: fib(28) = 317811
CPU2: fib(29) = 514229
CPU2: fib(30) = 832040
CPU2: fib(31) = 1346269
CPU2: fib(32) = 2178309
CPU1: reset cpu2
```

After a delay, coded within application 1, processor cpu1 performs a reset on processor cpu2, which we see starts re-executing from the reset vector

```
Info (RESET_CONTROL_CRWS2) top/u1/resetControlPeripheral:  Control Register
Write: Reset reset2
```

```
Info (RESET_CONTROL_CRWC2) top/u1/resetControlPeripheral:  Control Register
Write: Release reset2
CPU2: starting...
CPU2: fib(0) = 0
CPU2: fib(1) = 1
CPU2: fib(2) = 1
CPU2: fib(3) = 2
CPU1: delay (1000000)
CPU2: fib(4) = 3
…
CPU2: fib(30) = 832040
CPU2: fib(31) = 1346269
CPU2: fib(32) = 2178309
CPU2: fib(33) = 3524578
CPU2: finishing...
Info (HARNESS) Simulation finished
Info (HARNESS) cpu1 executed 4500024183 instructions, StopReason 'CPU has exited'
Info (HARNESS) cpu2 executed 5203371011 instructions, StopReason 'CPU has exited'
```

10.5 Processor External Interrupt

In the same way that a ‘reset’ can be signaled to a processor so too can interrupts, that cause the processor to take an exception i.e. to change execution to an exception vector.

Typically a processor interrupt port will be driven by an interrupt controller and the interrupt will remain active until acknowledged by interrupt handler software included in the application that is executing on the processor.

The processor interrupt port, e.g.: ‘intr0’ can be connected to a net that can be driven from the test harness or from another component (typically a peripheral) in the design.

When the net connected to the ‘intr0’ port is written the processor model will be invoked to process the interrupt. The processor model may react to an edge i.e. a change in the level on the port or it may react to a level i.e. it will continue to process the interrupt while the level is in an active state.

The following illustrates controlling the interrupt pin ‘intr0’ on the processor from a test harness which can be found in the processorExternalInterruptControl directory.

```
$IMPERAS_HOME/Examples/SimulationControl/processorExternalInterruptControl
```

First we need to find the name of the net, if any, connected to the net port on the processor in the module (mi)

```
    // get the handle for the processor in the module (only one)
```

```
optProcessorP processor = opProcessorNext(mi, 0);

// get interrupt port connection on processor (if made)
optNetPortConnP intNC = opObjectByName(processor,
                                       INTERRUPT_PORT_NAME,
                                      OP_NETPORTCONN_EN).NetPortConn;
```

NOTE: When we know that an object returned by opObjectByName can only be a certain type we can extract the correct union member. To determine the object, if it is not a certain type, the function opObjectType can be used.

If there is no net connection we can create our own new net in the harness and make the connection otherwise we can obtain the current net already connected:

```
optNetP intr0 = 0;
if(!intNC) {
    intr0 = opNetNew(mi, INTERRUPT_PORT_NAME "_net", 0, 0);
    opObjectNetConnect( processor, intr0, INTERRUPT_PORT_NAME);
} else {
    intr0 = opNetPortConnNet(intNC);
}
```

and then we can use the net to write and assert the interrupt (we show writing ‘1’ to assert but this could equally well be any other value)

```
opNetWrite(intr0, 1);
```

and similarly the interrupt can be de-asserted

```
opNetWrite(intr0, 0);
```

The above example can be compiled and run.

Initially instructions are executed to enable interrupts on the processor

```
Info (HARNESS) Creating net connection to 'intr0' on 'cpu'
Info (HARNESS) Simulate 20 instructions
Info 'top/u1/cpu', 0x0000000000001000: l.ori    r30,r0,0x0
Info 'top/u1/cpu', 0x0000000000001004: l.ori    r1,r0,0x7
…
Info 'top/u1/cpu', 0x0000000000001020: l.nop    0x0
Info 'top/u1/cpu', 0x0000000000001018: l.sfeq   r1,r30
Info 'top/u1/cpu', 0x000000000000101c: l.bnf    0x00001018
```

The harness generates the interrupt and runs one instruction to ensure that the processor will see the interrupt line active.

NOTE: If the interrupt is level sensitive and the simulation is not executed the processor may not see the interrupt ever being active.

The processor is enabled to simulate exceptions so when the interrupt line becomes active the interrupt exception, at address 0x00000800, is executed.

```
Info (HARNESS) Generate Interrupt on 'intr0'
Info (HARNESS) Simulate 1 instruction
Info 'top/u1/cpu', 0x0000000000001020: *** FETCH EXCEPTION ***
Info (HARNESS) Release Interrupt on 'intr0'
Info (HARNESS) Simulate 20 instructions
Info 'top/u1/cpu', 0x0000000000000800: l.addi   r30,r30,0x1
Info 'top/u1/cpu', 0x0000000000000804: l.addi   r2,r0,0xffffe000
Info 'top/u1/cpu', 0x0000000000000808: l.slli   r2,r2,0x10
Info 'top/u1/cpu', 0x000000000000080c: l.sw     0x0(r2),r30
Info 'top/u1/cpu', 0x0000000000000810: l.rfe
```

One the return from exception is executed the processor continues from the instruction that was executed.

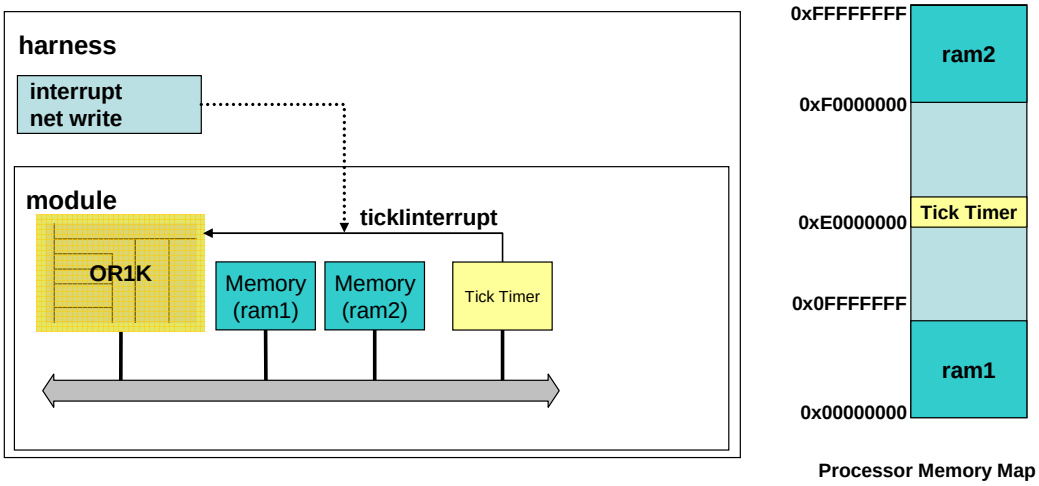
```
Info 'top/u1/cpu', 0x000000000000101c: l.bnf    0x00001018
Info 'top/u1/cpu', 0x0000000000001020: l.nop    0x0
Info 'top/u1/cpu', 0x0000000000001018: l.sfeq   r1,r30
…
Info 'top/u1/cpu', 0x0000000000001020: l.nop    0x0
Info 'top/u1/cpu', 0x0000000000001018: l.sfeq   r1,r30
Info 'top/u1/cpu', 0x000000000000101c: l.bnf    0x00001018
Info 'top/u1/cpu', 0x0000000000001020: l.nop    0x0
Info 'top/u1/cpu', 0x0000000000001018: l.sfeq   r1,r30
Info (HARNESS) Simulation finished
Info (HARNESS) cpu executed 41 instructions, StopReason 'Scheduler has expired'
```

10.5.1 Processor External Interrupt Timer Tick Example

This example is found in the processorExternalInterrupt directory.

```
$IMPERAS_HOME/Examples/PlatformConstruction/processorExternalInterrupt
```

This example includes a design that instantiates a processor and a peripheral model (a simple timer generating a regular interrupt tick), as shown in the following diagram.



The interrupt line, *intr0*, is connected to the processor in the module definition

```
# interrupt connection
ihwaddnet -instancename tickInterrupt
# connect tickInterrupt to cpu1 intr0
ihwconnect -instancename cpu1 -netport intr0 -net tickInterrupt
```

and may be driven from a peripheral in the module (as well as from the harness)

```
ihwaddperipheral -instancename timerPeripheral  \
                 -type simpleTimer \
                 -modelfile peripheral/pse.pse

# timer tick interrupt
ihwconnect -instancename timerPeripheral -netport tickOut  -net tickInterrupt
```

The application on the processor is a simple assembler routine that enables interrupts on the processor and installs reset and interrupt handler software at the vector addresses 0x0000 and 0x0800 respectively. The assembler routine is compiled and loaded without setting the initial PC, so the processor will start execution from its reset vector.

The peripheral model can be configured for a tick rate using the parameter ‘rate’, in the example the default is used to generate a tick every 0.01 seconds. A register, when

written, is used by the processor application to acknowledge and clear the generated tick interrupt.

The regular tick is generated within a loop delayed using bhmWaitDelay in the behavioral code of the peripheral, peripheral/user.c, is as follows:

```
   while (1) {
      // delay for tick rate
      bhmWaitDelay(delayRate);

      if(PSE_DIAG_LOW)
         bhmMessage("I", "TIMER_PSE", "Generating tick at %g seconds",
                  (double) (bhmGetCurrentTime()/1000000));
      ppmWriteNet(handles.tickOut, 1);
   }
```

And the behavior of the interrupt acknowledge register, in the register write callback of the peripheral, is as follows:

```
// interrupt acknowledge (used to clear interrupt)
PPM_REG_WRITE_CB(intAck) {

   *(Uns32*)user = data;

   if(ppmReadNet(handles.tickOut)) {
      if(PSE_DIAG_HIGH)
         bhmMessage("I", "TIMER_PSE", "Interrupt Acknowledge, clear interrupt");
      ppmWriteNet(handles.tickOut, 0);
   }

}
```

The application enables the interrupts on the OR1K processor and then waits in a loop for 8 interrupts to be generated. The main loop is

```
      l.ori       r1,r0,8        // r1 = 8 (loop count)
loop1:
      l.sfeq      r1,r30          // r1==r30?
      l.bnf       loop1           // go to loop if not true
      l.nop                       // (delay slot)
```

which tests the value in register r30 and r1 (r1 is initialized to 8) and r30 is incremented in the interrupt handler code

```
.global interruptVector
```

```
.org 0x800
        //////////////////////////////////////////////////////////////////
        // EXTERNAL INTERRUPT HANDLER (AT 0x800)
        //////////////////////////////////////////////////////////////////
interruptVector:
        l.addi r30,r30,1            // increment count of external exceptions
        l.addi r2,r0,0xe000         // set r2
        l.slli r2,r2,16             // shift to 0xe0000000
        l.sw   0(r2),r30            // write interrupt Acknowledge register
        l.rfe                       // return from exception
```

which also write to the interrupt acknowledge register of the peripheral to clear down the interrupt.

NOTE: The OR1K processor interrupt line is level sensitive so if the interrupt source is not cleared immediately on the OR1K executing the return from exception routine the interrupt will become active once again and the exception routine executed again.

Compile the module, peripheral and application using the following commands in the processorExternalInterrupt directory:

```
> make –C module
> make –C peripheral
> make –C application CROSS=OR1K
```

To run the simulation, in the processorExternalInterrupt directory, run:

```
> harness.exe --modulefile module \
        --objfilenoentry application/asmtest.OR1K.elf \
        --override processorExternalInterrupt /timerPeripheral/diagnosticlevel=3 \
        --output imperas.log
```

1) We are using –objfilenoentry to load the program without changing the initial PC of the processor so that we start execution from the reset vector (0x0000)
2) We are turning on diagnostics on the peripheral so that we can see messages generated when the interrupt is generated and when it is acknowledged by the processor application

You should see output as follows:

```
Info (TIMER_PSE) ...Interrupt/timerPeripheral: Generating tick at 0.01 seconds
Info (TIMER_PSE) ...Interrupt/timerPeripheral: Interrupt Acknowledge, clear interrupt
Info (TIMER_PSE) ...Interrupt/timerPeripheral: Generating tick at 0.02 seconds
Info (TIMER_PSE) ...Interrupt/timerPeripheral: Interrupt Acknowledge, clear interrupt
Info (TIMER_PSE) ...Interrupt/timerPeripheral: Generating tick at 0.03 seconds
Info (TIMER_PSE) ...Interrupt/timerPeripheral: Interrupt Acknowledge, clear interrupt
```

```
Info (TIMER_PSE) ...Interrupt/timerPeripheral: Generating tick at 0.04 seconds
Info (TIMER_PSE) ...Interrupt/timerPeripheral: Interrupt Acknowledge, clear interrupt
Info (TIMER_PSE) ...Interrupt/timerPeripheral: Generating tick at 0.05 seconds
Info (TIMER_PSE) ...Interrupt/timerPeripheral: Interrupt Acknowledge, clear interrupt
Info (TIMER_PSE) ...Interrupt/timerPeripheral: Generating tick at 0.06 seconds
Info (TIMER_PSE) ...Interrupt/timerPeripheral: Interrupt Acknowledge, clear interrupt
Info (TIMER_PSE) ...Interrupt/timerPeripheral: Generating tick at 0.07 seconds
Info (TIMER_PSE) ...Interrupt/timerPeripheral: Interrupt Acknowledge, clear interrupt
Info (TIMER_PSE) ...Interrupt/timerPeripheral: Generating tick at 0.08 seconds
Info (TIMER_PSE) ...Interrupt/timerPeripheral: Interrupt Acknowledge, clear interrupt
```

10.6 Standard Multiprocessor Scheduling Algorithm

This example uses the standard multiprocessor scheduling algorithm built-in to the simulator that is used when the `opRootModuleSimulate` function is called. The standard scheduling algorithm works as follows:

1. Simulation time is broken into *time slices*. By default, each time slice is 0.001 seconds (one millisecond).
2. The simulator selects the first processor and simulates it for one time slice. It in fact does this by calculating the *number of instructions* that should be executed by that processor in a time slice, and then simulating for that number of instructions. The number of instructions in a time slice is:
   (processor nominal MIPS rate) x 1e6 x (time slice duration)
   In this example, each processor has the default nominal MIPS rate of 100 MIPS. This means that each processor will execute 100 x 1e6 x 0.001 = 100,000 instructions per time slice
3. When the first processor has simulated for 100,000 instructions, it is suspended and the next processor is simulated for the time slice.
4. When all processors have simulated the time slice, simulated time is moved on and the next slice is started.

This algorithm is an approximation designed to give realistic simulation results with very high simulator performance: the simulator is not designed to be cycle accurate.

The simulation algorithm is configurable in several ways:

10.6.1 Changing the Time Slice Size

The size of the time slice (in seconds) can be set with:

```
Bool opRootModuleSetSimulationTimeSlice(optTime newSliceSize);
```

where type `optTime` is a `long double`. Shorter time slices may approximate real system behavior more closely, but degrade simulator performance.

10.6.2 Changing Processor Nominal MIPS Rate

The nominal MIPS rate for each processor can be set with a parameter on the processor instance. See the section titled *MIPS Parameter* in the *Simulation Control of Platforms and Modules User Guide* for an example of this.

10.6.3 Writing Custom Scheduling Algorithms

If the standard multiprocessor scheduling algorithm does not do what is required, a custom algorithm can be built around calls to `opProcessorSimulate` for each processor. This function will simulate a specified processor for an exact number of instructions.

Please note:

1) The user must ensure that the appropriate number of instructions are executed on each processor, in a multicore platform.
2) When a platform also includes peripheral models time must be updated at an appropriate rate.

11 Parallel Simulation: QuantumLeap™

As of VMI version 6.0.0, Imperas Professional Simulation products implement a parallel simulation algorithm called *QuantumLeap*™, which enables multicore platform simulation to be distributed over separate threads on multiple cores of the host machine for improved performance.

QuantumLeap allows for the parallel execution of both processor and peripheral models on host processors. This section describes its use with processors; refer to the OVP Peripheral Modeling Guide for information about parallelization of peripherals.

11.1 License and Runtime for QuantumLeap

Note that QuantumLeap is only supported as a licensed feature of the Imperas Professional Tools. The Imperas professional simulator, CpuManager, is selected using the IMPERAS_RUNTIME environment variable:

```
export IMPERAS_RUNTIME=CpuManager
```

You will need a license for QuantumLeap™. This will be one of IMP_SIMPARALLEL or IMP_SIMPARALLEL_MAX (see below for difference). Contact Imperas for more information.

11.2 Example

A number of examples are found in the Demo/Processors tree in directories ‘many_core’ and ‘multi_core’ that instance a number of processors and run the same standard application on each of them. These examples utilize the Imperas ISS product which allows the selection of the processor type, the number of processors and what memory is included.

In this section we will use the example found at

```
$IMPERAS_HOME/Demo/Processors/OPENCORES/or1k/generic/many_core
```

The QuantumLeap algorithm can be enabled by appending the commands (--*parallel* or --*parallelmax*) to the command line, when the command line parser has been included. If no command line parser is available a control file can be used (control files are described in the *document doc/ovp/OVP_Control_File_User_Guide*).

To run the simulation, with QuantumLeap enabled, in the many_core directory, run:

```
> iss.exe --verbose \
   --program ../../../Applications/dhrystone/dhrystone.OR1K-O3-g.elf \
   --processorvendor ovpworld.org --processorname or1k --variant generic \
    --numprocessors 4   –output ql.log --parallel -argv 400000
```

Which is based upon the Run_QL_AMP4_Dhrystone.sh script. Note The argument --argv must be the last argument on the command line. All arguments following this argument are passed to the applications argc argv.

You should see the following output as the four processors execute the Dhrystones application on parallel host processors:

```
...
CpuManagerMulti Parallel started: Tue Feb 23 09:18:42 2016
…
 Dhrystone Benchmark, Version 2.1 (Language: C)
 …
Info TOTAL
Info   Simulated instructions: 16,496,458,632
Info   Simulated MIPS        : 7199.9
Info ---------------------------------------------------
Info
Info ---------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 41.24 seconds
Info   User time             : 8.76 seconds
Info   System time           : 0.28 seconds
Info   Elapsed time          : 2.29 seconds
Info   Real time ratio       : 18.00x faster
Info ---------------------------------------------------

CpuManagerMulti Parallel finished: Tue Feb 23 09:18:45 2016
...
```

Note that the banners emitted at the start and end of simulation include an indication that QuantumLeap parallel simulation is now enabled.

These results show a marked improvement in performance over the non QuantumLeap run that can be performed by removing the *--parallel* argument

```
> iss.exe --verbose \
   --program ../../../Applications/dhrystone/dhrystone.OR1K-O3-g.elf \
   --processorvendor ovpworld.org --processorname or1k --variant generic \
    --numprocessors 4   --output non-ql.log  --argv 400000
```

```
…
CpuManagerMulti started: Tue Feb 23 09:18:22 2016
…
Info TOTAL
Info   Simulated instructions: 16,496,458,632
Info   Simulated MIPS        : 2029.1
Info ---------------------------------------------------
Info
Info ---------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 41.24 seconds
Info   User time            : 8.13 seconds
Info   System time          : 0.00 seconds
Info   Elapsed time         : 8.14 seconds
Info   Real time ratio      : 5.07x faster
Info ---------------------------------------------------

CpuManagerMulti finished: Tue Feb 23 09:18:31 2016
…
```

11.3 QuantumLeap Results

The actual performance reported may vary and depends on the performance of the native host. In this example (run on a 3.4Ghz Dell Quad Core i7-3770 desktop machine) the overall simulation speed is about 7200 simulated OR1K MIPS, approximately a quarter for each processor. This is almost four times as fast as the same application run without QuantumLeap.

11.4 QuantumLeap Scheduling Algorithm

The QuantumLeap scheduling algorithm is similar in many respects to the standard multiprocessor scheduling algorithm. The exact details of the algorithm are proprietary, but some general characteristics are given here.

Time moves forward in quanta which are calculated in exactly the same way as for the standard algorithm. During each quantum, processors may run in parallel in independent native threads, but they are all synchronized at the quantum end before the next quantum is started. Any processor may also cause the simulation to revert to synchronous mode during a quantum if the simulator detects that synchronous operation is required (for example, execution of a test-and-set instruction). In such a case, all other processors are safely stopped while the atomic action is carried out on the processor requiring synchronization.

Provided that synchronizing instructions and accesses to shared registers are correctly described, the simulation is deterministic in the absence of unguarded spin locks (demonstrate this by running this example simulation several times: instruction counts for

each processor will remain the same from run to run). See the OVP Processor Modeling Guide for a detailed description of how to make processor models compatible with QuantumLeap.

The actual simulation results can differ between the normal multiprocessor algorithm and the QuantumLeap algorithm, because of detailed scheduling differences. In the normal multiprocessor algorithm, preceding processors in the schedule list for this quantum will all have finished the quantum before an intermediate processor runs, and subsequent processors will not have run any instructions at all. In the QuantumLeap algorithm, all other processors can be in some deterministic intermediate state between the start and end of the quantum when an intermediate processor interacts with them. This usually affects instruction counts and sometimes program results, but in a correctly-designed program the standard and QuantumLeap results represent alternative legal paths through the parallel program. If you examine instruction counts for this example program running with and without QuantumLeap, you will see that they differ slightly, but the results are the same.

Any instruction that is intercepted is guaranteed to be run in synchronous mode with all other processors stopped. This means that legacy intercept libraries can be used with QuantumLeap without modification.

Sometimes QuantumLeap results are non-deterministic. This can either be due to legal constructs such as unguarded spin locks (often used to defer expensive synchronization instructions) or by real program synchronization bugs. QuantumLeap determinism can be a useful tool for validating parallel algorithm correctness.

11.5 QuantumLeap Options

Command line arguments *–-parallelopt*, *–-parallelthreads* and *–-parallelmax* can be used to control details of the simulation, as described below.

11.5.1 Option -parallelopt

QuantumLeap algorithm behavior can be modified using command line option *--parallelopt*. This option is a bitfield, which currently defines the following bits:

Bit 0: enable *nice* scheduling behavior
When this bit is 0, QuantumLeap operates in a *greedy* mode, in which the algorithm assumes that it can freely use all resources of the host to achieve the fastest possible simulation. Setting this bit enables *nice* mode, which suspends native threads more frequently so that more resources are available to other processes on the host machine.

The effect of nice mode depends on the operating system type and version. Often, QuantumLeap simulation runs little or no slower; on some operating system versions, the effect may be to slow simulation more significantly. Validate performance on your operating system before deciding whether it is appropriate to use this option.

Bit 1: don't fix affinity
When this bit is 0, QuantumLeap attempts to fix the affinity of a simulated core to a particular host core to avoid costs involved in synchronizing host caches that can occur when host processes are moved from one host core to another. Setting this bit disables affinity fixing so that simulated core processes can migrate between host cores.

Bit 2: use legacy thread allocation algorithm
When the number of host threads available for parallel simulation is less than the number of simulated cores, QuantumLeap has to share the limited host resources amongst the simulated cores. There are two algorithms available: the default algorithm statically allocates simulated cores to fixed host cpus (or sets of host cpus) and relies upon the host operating system to schedule those threads fairly; the alternative algorithm schedules host threads *explicitly* so that no more than a specified limit execute concurrently (see the *–parallelthreads* option below). Setting this bit enables the alternative algorithm.

The default value of *–parallelopt* is 0, specifying *greedy* mode simulation, *fixed affinities* and *static thread allocation without explicit scheduling*.

Example
To rerun the previous simulation with greedy scheduling behavior and no fixed affinities:

```
> iss.exe --verbose \
   --program ../../../Applications/dhrystone/dhrystone.OR1K-O3-g.elf \
   --processorvendor ovpworld.org --processorname or1k --variant generic \
    --numprocessors 4  –output ql.log --parallel --parallelopt 2 -argv 400000
```

11.5.2 Option -parallelthreads

QuantumLeap option *--parallelthreads* can be used to specify the maximum number of parallel threads that should execute at once. This option can be useful in (for example) regression test runs, to restrict a particular simulation to use of a smaller-than-normal set of the available processor resources, to ensure that some resources are available for other runs that might be occurring in parallel on the same machine.

11.5.2.1 Example

To run a simulation in which no more than three parallel threads execute at once:

```
> iss.exe --verbose \
   --program ../../../Applications/dhrystone/dhrystone.OR1K-O3-g.elf \
   --processorvendor ovpworld.org --processorname or1k --variant generic \
    --numprocessors 4   –output ql.log --parallel -- parallelthreads 3 -argv 400000
```

Note that standard QuantumLeap supports up to 4 parallel threads. To specify more than this, a separate license is required; contact Imperas for details.

11.5.3 Option -parallelmax

QuantumLeap option --*parallelmax* can be used in a control file to specify that a simulation should run as many threads as possible in parallel for maximum performance. This option requires a separate license; contact Imperas for details.

11.5.3.1 Example

To run a simulation using maximum parallelization:

```
> iss.exe --verbose \
   --program ../../../Applications/dhrystone/dhrystone.OR1K-O3-g.elf \
   --processorvendor ovpworld.org --processorname or1k --variant generic \
    --numprocessors 4  –output ql.log  --parallel --parallelmax -argv 400000
```

12 Memory Operations

12.1 Implicit Processor Model Memory

When a new processor instance is created, and no specific memories are created, then by default an implicit RAM memory that covers the entire addressable memory by the processor type is also created.

This memory can be directly accessed within the harness using the following functions: *opProcessorRead* and *opProcessorWrite*. In addition, theses functions can also be used to read and write memory without causing side effects in the processor model, or in any TLM2.0 models connected to the processor. The function *opProcessorApplicationLoad* can be used to load an object file.

12.1.1 Loading object files

An object file can be loaded into processor memory using the *opProcessorApplicationLoad* function:

```
optApplicationP opProcessorApplicationLoad(
   optProcessorP processor,
   const char   *objectFile,
   optLoaderControls  controls,
   Addr         loadOffset
);
```

The `optLoaderControls` are used to control how the file is loaded into memory and is defined as:

```
OP_LDR_DEFAULT:        No special features
OP_LDR_PHYSICAL:       Use object file physical addresses if available
OP_LDR_VERBOSE:        Report each section as it is loaded
OP_LDR_NO_ZERO_BSS:    Do not zero the extent of the BSS section if present
OP_LDR_SET_START:      Set the PC to the code start address
OP_LDR_SYMBOLS_ONLY    Read the symbols but do not load the code or data
OP_LDR_ELF_USE_VMA     Load ELF files using VMA addresses instead of LMA.
OP_LDR_FAIL_IS_ERROR   Failure to load will prevent the simulation from running
```

For ELF files, the Load Memory Address (LMA) is used as the load address by default. Setting *OP_LDR_ELF_USE_VMA* in the *attrs* argument will cause the Virtual Memory Address (VMA) to be used instead.

In the following example the memory is loaded from file *hello.OR1K.elf* using physical address information, and the PC will be set to the entry address defined in the object file:

```
opProcessorApplicationLoad(processor, “hello.OR1K.elf”, OP_LDR_PHYSICAL, 0);
```

The algorithm used is as follows:
1. Find the processor passed in argument #1.

2. Find the bus connected to the instruction or data port[3] on that processor.
3. Load the specified object file into memory on that bus.

The loader takes each section address from the object 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.

If more than one processor is using the same code memory, the program need be loaded only once; When OVPsim 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.

An object file which is not directly related to a processor (e.g. a data file) can be loaded into memory on a bus using *opBusApplicationLoad*.

The functions *opProcessorApplicationLoad* and *opBusApplicationLoad* return an *optApplicationP* which can be interrogated using *opApplicationControls*, *opApplicationElfCode*, *opApplicationEndian* and *opApplicationEntry* to find respectively the control features used when the application was loaded, the 16-bit processor architecture code, endianness and the executable start address.

12.1.1.1 Supported object formats

The simulator currently supports:
ELF Used by all GNU tool chains
TI COFF An extended version of the COFF format, used by compilers supplied by Texas Instruments

12.1.1.2 Loading Symbols in object files

When *opProcessorApplicationLoad* loads an object file into simulated memory it also reads the symbol tables included in the object file, and records the address-to-symbol mappings. These mapping can then be used:

- When issuing tracing information
- When intercepting a function by name (see *opProcessorExtensionNew*)

Sometimes object code might be loaded by another route (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 the function *opProcessorApplicationLoad* can be used with the controls set for *OP_LDR_SYMBOLS_ONLY* to associate symbols addresses within a processor without loading the code. In this example, instruction tracing will include code labels found in *program.elf*, though the code came from another source:

In *module.op.tcl* the *bootloader.elf* file is specified as part of the hardware i.e. it is always present and loaded into the processor memory. This may then execute and re-locate the

[3] The port used is dependent upon the processor model and the attributes of the section being loaded.

code to a different memory space, for example to copy from a ROM area to a RAM area for execution.

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

In the module C code callback the program symbol file can be loaded separately that contains the symbol information at the execution addresses.

```
static OP_PRE_SIMULATE_FN(modulePreSimulate) {
   opProcessorApplicationLoad(processor, “program.elf”, OP_LDR_SYMBOLS_ONLY, 0);
}
```

12.2 Loading by Directly Reading and Writing Data

The memory space can also be read and written directly using the *opProcessorRead* and *opProcessorWrite* functions. These functions transfer N bytes of data between a local buffer and the simulated memory space using the simulated memory address.

12.2.1 Example Loading Program from Hex format file

This example is found in the loadingApplicationProgramHexFormatFile directory.

```
$IMPERAS_HOME/Examples/SimulationControl/loadingApplicationProgramHexFormatFile
```

The example shows the use of the write memory and read memory functions to perform the loading of a program. The program is provided in the form of a hex file with address and data pairs.
The file loader is written in standard C code as part of the harness. In the same way any file format can be supported by either incorporating available C code of a reader or creating a new one.

The hex file format used in this example is based upon a simple sequence of address and data, with comments marked using ‘#’ and symbols marked with ‘:’ (note symbols will only be loaded if the Imperas professional simulator is being used), for example:

```
:01000074 _start
01000074 0000209c ;
01000078 0100409c ;
```

In this example the loader is implemented so that multiple consecutive addresses can be written with the same value.

```
#load memory addresses with 0x00000000
01000078-01000088 00000000 ;
```

Following is the loader, found in the file *harness/harness.c*, which takes the name of the file containing the data to be loaded and a switch to control the byte swapping.
The initial code is responsible for opening the file, reading it a line at a time and parsing the line

```
static int loadHexFile(optProcessorP processor, const char *fileName) {

    FILE *fp;
    char inBuf[MAX_LINE_LENGTH + 1];
    int address, endAddress, data, dataCheck;

    fp = fopen(fileName, "r");

    if (!fp) {
        opMessage("E", HARNESS_NAME "_HEX_LDR",
                               "Failed to open Memory Initialization File '%s'",
                               fileName);
        return -1;
    }

    opMessage("I", HARNESS_NAME "_HEX_LDR",
                           "Loading Hex file '%s'",
                           fileName);

    while ( fgets( inBuf,MAX_LINE_LENGTH, fp) != 0 ) {

        if ( inBuf[0] == '#') {
            // ignore header
        } else if ( inBuf[0] == ':') {
```

If the initial character is a ‘:’ this indicates that the line is defining a symbol to be loaded.

```
            // symbol
            Uns32 symbolAddress;
            char  symbol[16];
            if ( sscanf(inBuf, ":%08x %s", &symbolAddress, symbol) == 2 ) {
                opMessage("I", HARNESS_NAME "_HEX_LDR_SYM",
                                        "Found symbol '%s' at address 0x%08x",
                                        symbol,
                                        symbolAddress);
```

The following section of code checks the IMPERAS_RUNTIME environment variable to check for the Imperas Professional simulator. If it is being used the symbols of the application can be loaded. If the symbol is the '_start' symbol this is also used to set the PC.

```
            const char *runTime=getenv("IMPERAS_RUNTIME");
            if(strcmp(runTime, "CpuManager") == 0) {
               opProcessorApplicationSymbolAdd(processor,
                                    symbol,
                                    symbolAddress,
                                    4,
                                    ORD_SYMBOL_TYPE_FUNC,
                                    ORD_SYMBOL_BIND_GLOBAL);
            } else {
               opMessage("W", HARNESS_NAME "_HEX_LDR_SF",
                                    "Symbols cannot be loaded with this product runtime (%s),
"
                                    "supported in Imperas Professional simulator",
                                    runTime);
            }
            if(strcmp(symbol, "_start") == 0) {
               opMessage("I", HARNESS_NAME "_HEX_LDR_START",
                                    "Set start address to 0x%08x",
                                    symbolAddress);
               opProcessorPCSet(processor, symbolAddress);
            }

         } else {
            opMessage("W", HARNESS_NAME "_HEX_LDR_SND",
                                   "Found symbol line '%s' but not decoded",
                                   inBuf);
         }

      } else {
```

The address and data information is used to write the data to the correct address in the processor memory space. A read back of the same address is carried out to see that it was correctly written. The data word will be swapped if the Host endian does not match the endian of the simulated processor because of the use of OP_HOSTENDIAN_TARGET in the opProcessorWrite and opProcessorRead functions.

```
         if ( sscanf(inBuf, "%08x-%08x %08x ;", &address, &endAddress, &data) != 3 ) {
            sscanf(inBuf, "%08x %08x ;", &address, &data);
            endAddress = address;
         }
```

```
        do {
            //
            // Access the memory through the processor memory space
            //
            if(!opProcessorWrite(
                processor,      // processor
                address,        // memory address
                &data,          // data buffer of data to write
                4,              // number of bytes in one object
                1,              // number of objects
                True,           // debug access (not a true processor access)
                OP_HOSTENDIAN_TARGET
            )) {
                opMessage("E", HARNESS_NAME "_HEX_LDR_FW",
                                "Failed Data Write at 0x%08x",
                                (Uns32)address);
                return -1;
            }

            opProcessorRead(
                processor,
                address,
                &dataCheck,
                4,
                1,
                True,
                OP_HOSTENDIAN_TARGET
            );

            if(data != dataCheck) {
                opMessage("E", HARNESS_NAME "_HEX_LDR_FRB",
                                "Failed Data Read Back at 0x%08x (0x%08x 0x%08x)",
                                (Uns32)address, data, dataCheck);
                return -1;
            }

            opMessage("I", HARNESS_NAME "_HEX_LDR_LD",
                                "Load address 0x%08x : 0x%08x",
                                address, data);

        } while ( address++ < endAddress);
    }
}

opMessage("I", HARNESS_NAME "_HEX_LDR",
                    "Load Complete");
```

```
    if (fclose(fp)!=0) {
        opMessage("E", HARNESS_NAME "_HEX_LDR_CF",
                                "Failed to close Memory Initialization File");
        return -1;
    }

    return 0;
}
```

> ⇒ Note if the processor uses virtual addressing the address of the **opProcessorWrite** and **opProcessorRead** functions will be translated to a physical memory address using the current virtual address mapping.

The main() routine in the platform file, creates a platform with a single OR1K processor and a region of memory from 0x00100000 to 0xffffffff. The memory is loaded by a call to the hexLoader routine that has been described above. If the load fails an error message is generated.

```
// Loading application hex file
// Load Hex file into Processor Memory
if (loadHexFile(proc, options.hexfile, False)) {
    opMessage("E", HARNESS_NAME, "Load of Hex File '%s' Failed", options.hexfile);
}
```

To run the example, take a copy of the example and compile the test harness and the module hardware definition using the following command in the copied `loadingApplicationProgramHexFormatFile` directory:

```
make -C harness
make -C module
```

In the application directory you will find an assembler file, asmtest.S, and the same file as hex, asmtest.hex. To run the simulation, type :

```
./harness/harness.Linux32.exe --hexfile application/asmtest.hex
```

If you are using the Imperas professional simulator, you should see output similar to the following:

```
Info (harness_HEX_LDR) Loading Hex file 'application/asmtest.hex'
Info (harness_HEX_LDR_SYM) Found symbol '_start' at address 0x01000074
Info (harness_HEX_LDR_START) Set start address to 0x01000074
Info (harness_HEX_LDR_LD) Load address 0x01000074 : 0x0000209c
Info (harness_HEX_LDR_LD) Load address 0x01000078 : 0x0100409c
Info (harness_HEX_LDR_LD) Load address 0x0100007c : 0xffff609c
Info (harness_HEX_LDR_LD) Load address 0x01000080 : 0x0008809c
```

```
Info (harness_HEX_LDR_LD) Load address 0x01000084 : 0x000884b0
Info (harness_HEX_LDR_LD) Load address 0x01000088 : 0x0000a484
Info (harness_HEX_LDR_LD) Load address 0x0100008c : 0xffffa3a0
Info (harness_HEX_LDR_LD) Load address 0x01000090 : 0x0000a4a0
Info (harness_HEX_LDR_LD) Load address 0x01000094 : 0x0100a5a0
Info (harness_HEX_LDR_LD) Load address 0x01000098 : 0x002804d4
Info (harness_HEX_LDR_SYM) Found symbol '__exit' at address 0x0100009c
Info (harness_HEX_LDR_LD) Load address 0x0100009c : 0x0000c084
Info (harness_HEX_LDR_LD) Load address 0x010000a0 : 0x0000209c
Info (harness_HEX_LDR_LD) Load address 0x010000a1 : 0x0000209c
Info (harness_HEX_LDR) Load Complete
Info 'u1/cpu1', 0x0000000001000074(_start): l.addi   r1,r0,0x0
Info 'u1/cpu1', 0x0000000001000078(_start+4): l.addi   r2,r0,0x1
Info 'u1/cpu1', 0x000000000100007c(_start+8): l.addi   r3,r0,0xffffffff
Info 'u1/cpu1', 0x0000000001000080(_start+c): l.addi   r4,r0,0x800
Info 'u1/cpu1', 0x0000000001000084(_start+10): l.muli   r4,r4,0x800
Info 'u1/cpu1', 0x0000000001000088(_start+14): l.lwz    r5,0x0(r4)
Info 'u1/cpu1', 0x000000000100008c(_start+18): l.addic  r5,r3,0xffffffff
Info 'u1/cpu1', 0x0000000001000090(_start+1c): l.addic  r5,r4,0x0
Info 'u1/cpu1', 0x0000000001000094(_start+20): l.addic  r5,r5,0x1
Info 'u1/cpu1', 0x0000000001000098(_start+24): l.sw     0x0(r4),r5
Info 'u1/cpu1', 0x000000000100009c(__exit): *** INTERCEPT *** (__exit)
```

if you are using the OVPsim simulator, you should see output similar to the following:

```
Info (harness_HEX_LDR) Loading Hex file 'application/asmtest.hex'
Info (harness_HEX_LDR_SYM) Found symbol '_start' at address 0x01000074
Warning (harness_HEX_LDR_SF) Symbols cannot be loaded with this product runtime
(OVPsim), supported in Imperas Professional simulator
Info (harness_HEX_LDR_START) Set start address to 0x01000074
Info (harness_HEX_LDR_LD) Load address 0x01000074 : 0x0000209c
Info (harness_HEX_LDR_LD) Load address 0x01000078 : 0x0100409c
Info (harness_HEX_LDR_LD) Load address 0x0100007c : 0xffff609c
Info (harness_HEX_LDR_LD) Load address 0x01000080 : 0x0008809c
Info (harness_HEX_LDR_LD) Load address 0x01000084 : 0x000884b0
Info (harness_HEX_LDR_LD) Load address 0x01000088 : 0x0000a484
Info (harness_HEX_LDR_LD) Load address 0x0100008c : 0xffffa3a0
Info (harness_HEX_LDR_LD) Load address 0x01000090 : 0x0000a4a0
Info (harness_HEX_LDR_LD) Load address 0x01000094 : 0x0100a5a0
Info (harness_HEX_LDR_LD) Load address 0x01000098 : 0x002804d4
Info (harness_HEX_LDR_SYM) Found symbol '__exit' at address 0x0100009c
Warning (harness_HEX_LDR_SF) Symbols cannot be loaded with this product runtime
(OVPsim), supported in Imperas Professional simulator
Info (harness_HEX_LDR_LD) Load address 0x0100009c : 0x0000c084
Info (harness_HEX_LDR_LD) Load address 0x010000a0 : 0x0000209c
Info (harness_HEX_LDR_LD) Load address 0x010000a1 : 0x0000209c
```

```
Info (harness_HEX_LDR) Load Complete
Info 'u1/cpu1', 0x0000000001000074: l.addi   r1,r0,0x0
Info 'u1/cpu1', 0x0000000001000078: l.addi   r2,r0,0x1
Info 'u1/cpu1', 0x000000000100007c: l.addi   r3,r0,0xffffffff
Info 'u1/cpu1', 0x0000000001000080: l.addi   r4,r0,0x800
Info 'u1/cpu1', 0x0000000001000084: l.muli   r4,r4,0x800
Info 'u1/cpu1', 0x0000000001000088: l.lwz    r5,0x0(r4)
Info 'u1/cpu1', 0x000000000100008c: l.addic  r5,r3,0xffffffff
Info 'u1/cpu1', 0x0000000001000090: l.addic  r5,r4,0x0
Info 'u1/cpu1', 0x0000000001000094: l.addic  r5,r5,0x1
Info 'u1/cpu1', 0x0000000001000098: l.sw     0x0(r4),r5
Info 'u1/cpu1', 0x000000000100009c: l.lwz    r6,0x0(r0)
Processor Exception (PC_PRX) Processor 'u1/cpu1' 0x100009c: l.lwz    r6,0x0(r0)
Processor Exception (PC_RPX) No read access at 0x0
```

Note that:

1. When using the Imperas professional simulator the symbols are loaded and the semihost library (or1kNewlib, loaded in the module) intercepts '__exit' and terminates the simulation.
2. When using the OVPsim simulator, no symbols are loaded, the code in '__exit', loads from address 0x00000000 which causes an exception – this is because there is no memory mapped at this address.
3. The simulator reports the exception and returns from opRootModuleSimulate (using opRootModuleStopReason or opProcessorStopReason would show the status was `OP_SR_RD_PRIV` although this isn't shown in this example). The `OP_FP_SIMULATEEXCEPTIONS` instance parameter could be used to cause the exception to be simulated by the processor instead.

12.2.2 Reading and writing memory without side-effects

The functions `opProcessorRead and opProcessorWrite` can be used to access processor memory without making any change to the processor state, for example if used with a debugger, rather than as part of a platform model. This table compares their behavior with the debugAccess argument, True and False:

Function	debugAccess	TLM2.0	Effect on tlb	Bad access
`opProcessorRead`	True	transport_dbg	None	returns 'False'
`opProcessorWrite`	True	transport_dbg	None	returns 'False'
`opProcessorRead`	False	b_transport	might update	bus err if supported
`opProcessorWrite`	False	b_transport	might update	bus err if supported

12.2.3 Swapping data to host endian

The functions `opProcessorRead and opProcessorWrite` can be used to convert the data endianness between that of the simulated processor and the host.

The `endian` argument controls the treatment of byte order in the host memory pointed to by `buffer`:

Endian	Effect
OP_HOSTENDIAN_HOST	Byte swapped, if necessary, to be host endian
OP_HOSTENDIAN_TARGET	No swapping; result will be same as target processor
OP_HOSTENDIAN_BIG	Byte swapped, if necessary, to be big endian
OP_HOSTENDIAN_LITTLE	Byte swapped, if necessary, to be little endian

If required, the bytes in each group of `objectSize` bytes, will be reversed, throughout the whole buffer (if `objectSize = 1 byte`, there can be no swapping).

A request to read or write can cross boundaries between different types of memory, or regions where no device exists. The functions return `True` if the entire buffer was read or written successfully, `False` if any part failed.

The `processor` argument refers to the target processor for the read or write. The required endianness is that of the **data** endian of the processor, which might differ from the **code** endian.

12.3 Explicit Local and External Memory

The previous example in this section "Memory Operations" has used an implicit RAM memory that covers the entire address space that can be accessed by the processor type. Instead of doing this, processor address spaces can be explicitly specified to contain separate RAMs and ROMs, with some perhaps shared between processors in a multiprocessor system. This is typically created in the module iGen definition, see the document *iGen_Platform_and_Module_Creation_User_Guide*. It is also possible to specify that certain address ranges will be modeled by callback functions, which is useful for modeling simple memory-mapped devices such as uarts[4].

12.3.1 Local Memory

The following would normally be generated as part of the module definition using iGen but is shown here to provide the complete methodology for hardware definition.

In order to use an explicit address space mapping, it is first necessary to create a bus to which all address-mapped components will be connected. A bus is defined using the function `opBusNew`, which takes a bus name and bit width as arguments, for example:

[4] But note that in general, it is much better to use Imperas *PSE* objects to model peripherals, instead of coding them directly in OP, for many reasons:

1. PSE models run in a protected address space and cannot crash the simulator;
2. PSE models allow concepts such as simulation time and threading to be handled elegantly;
3. A platform consisting of processor models and PSEs is ideally suited to debug with the Imperas debugger;
4. PSEs can be analyzed using tools built with Imperas intercept technology without having to modify and recompile the platform.

See the *OVP Peripheral Modeling Guide* for detailed information on PSEs.

```
optBusP bus = opBusNew(“bus”, 32);
```

This example defines a new bus called ‘bus’ which is 32 bits wide.

The bus must be connected to any processor that uses it by defining the optConnectionsP in the opProcessorNew function using the OP_BUS_CONNECTIONS and OP_BUS_CONNECT macros. The processor connects by two busses, the instruction bus and the data bus (the simulator permits processors to have distinct data and instruction busses). Most processors use the same address space for both data and instruction accesses, so often the bus arguments have the same value:

```
optProcessorP cpu1_c = opProcessorNew(
                mi,
                cpu1_path,
                "cpu1",
                OP_CONNECTIONS(
                   OP_BUS_CONNECTIONS(
                      OP_BUS_CONNECT(mainBus_b, "INSTRUCTION"),
                      OP_BUS_CONNECT(mainBus_b, "DATA")
                   )
                ),
                0
            );
```

Any number of memory objects can then be defined and connected to the bus. A memory is defined using `opMemoryNew`, which takes a memory name, access privileges, address range and connections as arguments, for example:

```
    opMemoryNew(
       mi,
       "mem1",
       OP_PRIV_RWX,
       0x003fffff,
       OP_CONNECTIONS(
          OP_BUS_CONNECTIONS(
             OP_BUS_CONNECT(mainBus_b, "sp1", .slave=1,
                                    .addrLo=0x01000000, .addrHi=0x013fffff)
          )
       ),
       0
    );
```

This example defines a new memory called *mem1* which has an address range `0:0x3fffff` (i.e. it is of size `0x400000`). The access privileges for the memory are defined by the enumeration type `optPrivE` in `op.h`:

OP_PRIV_NONE	No access permitted.
OP_PRIV_R	Read permitted.
OP_PRIV_W	Write permitted.
OP_PRIV_RW	Read and write permitted.
OP_PRIV_X	Execute permitted.
OP_PRIV_RX	Read and execute permitted.
OP_PRIV_WX	Write and execute permitted.
OP_PRIV_RWX	Read, write and execute permitted.
OP_PRIV_ALIGN	Force accesses to be aligned.

⇒ Note that the address argument to `opMemoryNew` is the *memory upper bound*, not the *memory size*. This is so that it is possible to define a memory of size 2^64 bytes, i.e. to cover the full range of a 64-bit address space.
⇒ The *highAddr* is the high address within the memory, it is NOT the address at which the memory is decoded when connected onto a bus. The decoded address range for the memory is *bus base address* to *bus base address + highAddr*.

The memory is connected to a bus in this example using port *sp1* of the memory (memories may be multiport, and can be connected to several busses using different port names, however, it is usual to use one or more bridges to connect memory regions from multiple busses onto a single bus on which the memory resides). The memory is connected with memory address `0` mapped to bus address `0x01000000`.

Memories defined with `opMemoryNew` use the simulator's internal memory modeling capabilities. It is possible as an alternative to specify that a memory range should be modeled using a callback function instead. This is done using `opBusSlaveNew`:

12.3.2 External Memory: Mapping an address region to a callback

In this example, an address region is mapped to read and write callbacks supplied by a user's functions.

```
typedef struct extMemDescS {
   void *localSource;
   void *localSink;
} extMemDesc;

// called when a read occurs in the range 0x00400000, 0x00400fff,
// copies data from localSource;

OP_BUS_SLAVE_READ_FN(extMemReadCB) {
   extMemDesc *p = userData;
   memcpy(data, p->localSource, bytes)
}
```

```
// called when a write occurs in the range 0x00400000, 0x00400fff,
// copies data to localSink;

OP_BUS_SLAVE_WRITE_FN(extMemWriteCB) {
   extMemDesc *p = userData;
   memcpy(p->localSink, data, bytes)
}

static extMemDesc extMem;

opBusSlaveNew(
   bus, "external", 0, OP_PRIV_RWX, 0x00400000, 0x00400fff,
   extMemReadCB, extMemWriteCB, 0, &extMem
);
```

This example specifies that the range `0x400000:0x400fff` on the bus should not be modeled using simulated memory, but should instead be implemented using two callback functions, `extMemReadCB` and `extMemWriteCB`. These callback functions are specified using the `OP_BUS_SLAVE_READ_FN` and `OP_BUS_SLAVE_WRITE_FN` macros. Any time a simulated processor or device performs a memory read or write in this address range, the appropriate callback function will be called. The write callback will be passed the value being written in the `data` argument. The read callback should fill the `data` buffer with `bytes` bytes of data (the required contents for a read at the passed address).

12.3.2.1 Generating an Invalid access

During the read or write callback functions, defined by OP_BUS_SLAVE_READ_FN and OP_BUS_SLAVE_WRITE_FN only, the client might decide that the access cannot be completed. To signal this, either function `opProcessorReadAbort` or `opProcessorWriteAbort` should be called by the client in the read or write callback respectively.

Use `opProcessorReadAbort` when the callback was initiated by either of these functions:

```
opProcessorRead
opBusRead
```

Use `opProcessorWriteAbort` when the callback was initiated by either of these functions:

```
opProcessorWrite
opBusWrite
```

In no other context should `opProcessorReadAbort` or `opProcessorWriteAbort` be called.

If the initiating processor model implements `rdAbortExceptCB` or `wrAbortExceptCB` callback functions in its `vmiAttrs` structure, then the appropriate callback will be invoked to allow the processor model to handle the abort. Otherwise, simulation will be terminated with a memory abort error message.

12.3.2.2 Processor Instruction Execution

When the memory represented by or accessed through an external memory callback is used to store the executable binary to be executed by the processor the external memory callback will be called for the processor instruction fetch access but also as an artifact of simulation.

In order to distinguish between a real instruction fetch and a simulation artifact the optProcessorP processor argument should be used within the callback. If the read is a processor instruction fetch the *processor* argument will be a pointer to the processor making the access. If the read is a simulation artifact then the *processor* argument will be NULL, indicating that it is not a processor making this access.

12.3.2.3 Example

This example is found in the `usingExternalMemory` directory.

```
$IMPERAS_HOME/Examples/PlatformConstruction/usingExternalMemory
```

The example shows how a region of memory could be mapped externally in the module to simulate an area memory with read only privileges.

Module iGen defines the fixed memory

And the module C file is used to define the dynamic memory added using callbacks

```
   // map the address range 0x00400000:0x004000ff externally to the processor,
   opBusSlaveNew (
                bus, "externalRW", OP_PRIV_R, 0x00400000, 0x004000ff,
                extMemReadCB, extMemWriteCB, 0
   );
   // map the address range 0x00400110:0x0040010f externally to the processor,
// generate abort
   opBusSlaveNew (
                bus, "externalRW", OP_PRIV_R, 0x00400100, 0x0040010f,
                extMemReadCB, extMemWriteCB, 0
   );
   // map the address range 0x00400110:0x004001ff externally to the processor,
// read only
   opBusSlaveNew (
```

```
            bus, "externalRW", OP_PRIV_R, 0x00400110, 0x0040011f,
            extMemReadCB, extMemWriteCB, 0
    );
```

The callbacks are defined using the macros as:

```
static OP_BUS_SLAVE_READ_FN(extMemReadCB) {

    *(Uns32 *)data = (Uns32)0xcefaedfe;

    opMessage( “I”, "EXTERNAL MEMORY”, “Reading  0x%08x from 0x%08x\n",
        data, (Uns32)address
    );
}

static OP_BUS_SLAVE_WRITE_FN(extMemWriteCB) {

    opMessage(“I”, "EXTERNAL MEMORY”, “Writing 0x%08x to 0x%08x\n",
        (Int32)value, (Int32)address
    );
}
```

This very simple ROM implementation returns the fixed pattern `0xcefaedfe` for any read from the ROM area and ignores any write (obviously a real example can do something much more sophisticated than this if required).

Compile the test platform and application as before using the following commands in the `memory` directory:

```
make –C module
make –C application
```

To run the simulation, in the `memory` directory, run :

```
harness.exe –modulefile module --program application/asmtest.OR1K.elf –trace –
tracechange --traceshowicount
```

You should see the following output:

```
. . . lines deleted . . .

Processor Exception (PC_PRX) Processor 'platform/cpu1' 0x1000098: l.sw
0x0(r4),r5
Processor Exception (PC_WPX) No write access at 0x400000
```

Note that:

1. The load and store to the external memory region ‘externalRW’ are performed correctly
2. The load to the external memory region ‘externalR’ is performed correctly but the store causes an exception – this is because the external region was specified to have read access permission only.
3. Although the read memory callback returns the value `0xcefaedfe`, the value that gets loaded into register `R5` of the OR1K processor is `0xfeedface`. This is because the native host (x86) is little-endian, whereas the OR1K processor is big-endian. Depending on the processor being used, memory callbacks may be required to perform endian swapping to get the desired results.
4. The simulator reports the exception (and returns `OP_SR_WR_PRIV` from `opRootModuleSimulate`, although this isn’t explicitly shown in this example). The `OP_FP_SIMULATEEXCEPTIONS` instance attribute could be used to cause the exception to be simulated instead.

12.3.3 External Memory: Using Native Memory

Some usages of the OP API require that simulated memory be allocated by the program rather than using *opMemoryNew* and allowing the simulator to allocate and maintain the memory. The function *opMemoryNativeDynamic* enables such use of native memory in a platform.

Note that since each call to *opMemoryNativeDynamic* requires a contiguous block of memory, this method is not suitable for modeling memory whereby the size is similar to, or larger than the memory of the host machine. Conversely, memories created using *opMemoryNew* can be specified to be as large as desired, and backing the store for such memories is allocated on demand, using a sparse implementation.

12.3.3.1 Example

This example is found in the `nativeMemory` directory.

```
$IMPERAS_HOME/Examples/PlatformConstruction/nativeMemory
```

iGen definition of static module components

```
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 mem1 -type ram
ihwconnect -bus mainBus -instancename mem1 \
                -busslaveport sp1 -loaddress 0xf0000000 -hiaddress 0xffffffff
```

this creates the following fixed definition

```
   optBusP bus_b = opBusNew(mi, "bus", 32, 0, 0);

   opMemoryNew(
      mi,
      "mem1",
      OP_PRIV_RWX,
      (0xffffffff) - (0xf0000000),
      OP_CONNECTIONS(
         OP_BUS_CONNECTIONS(
            OP_BUS_CONNECT(bus_b, "sp1", .slave=1,
                                    .addrLo=0xf0000000, .addrHi=0xffffffff)
         )
      ),
      0
   );
```

In the generated module C file we can then add the dynamic memory

```
typedef struct optModuleObjectS {
   Uns32 msize;    // size of allocated memory
   void *mem;      // allocated memory
} optModuleObject;

// forward declaration of component constructor
static OP_CONSTRUCT_FN(instantiateComponents);

static OP_CONSTRUCT_FN(moduleConstructor) {

   // instantiate module components
   instantiateComponents(mi, object);

   // create native memory (64K)
   object->msize = 0x10000;
   object->mem   = STYPE_ALLOC_N(Uns8, object->msize);

   // terminate if memory allocation failed
   if (!object->mem)
```

```
        opMessage("F", "MODULE", "Failed to allocate native memory");

    // get first bus in module
    optBusP bus = opBusNext(mi, NULL);
    if (!bus)
        opMessage("F", "MODULE", "Did not find a bus in module '%s'",
opObjectName(mi));

    // connect memory to bus at address range 0x0000:0xffff
    opMemoryNativeNew(
        mi, "native", OP_PRIV_RWX, object->msize-1, object->mem,
        OP_CONNECTIONS(
            OP_BUS_CONNECTIONS(
                OP_BUS_CONNECT(bus, "sp1", .addrLo=0x0, .addrHi=object->msize-1)
            )
        ),
        0
    );
}
```

> ⇒ NOTE the use of STYPE_ALLOC_N which is an Imperas provided wrapped function (see ImpPublic/include/hostapi/impAlloc.h) with the same result as malloc.

A processor connected to 'bus' will see memory mapped at 0x0 to 0xFFFF and 0xF0000000 to 0xFFFFFFFF, however the command line argument *--showbuses w*ill only show the statically defined areas

```
BUS CONNECTIONS: nativeMemory/mainBus   (0:ffffffff)
   sp1            on nativeMemory/mem1   (f0000000:ffffffff)
   DATA           on nativeMemory/cpu1   (0:0)
   INSTRUCTION    on nativeMemory/cpu1   (0:0)
```

To see all the regions the command line argument *--showdomains* can be used

```
BUS simpleCpuMemory/mainBus: DOMAIN simpleCpuMemory/mainBus REGIONS:
    1:    0x00000000:0x00000fff type:RAM  priv:rwx raw:rwx list:0 opt:RW flags:MD..... \
                  blocks:0    device:.. (master -,native,0x08e76230:0x08e7722f)
    2:    0xf0000000:0xffffffff type:RAM  priv:rwx raw:rwx list:0 opt:.. flags:..... blocks:0    device:..
```

> ⇒ Making the native memory smaller than the boundaries described by `opMemoryNativeDynamic` can cause memory corruptions, because the simulator may attempt to write beyond the bounds of the allocated space.

The allocated region of memory should be freed in the post simulation callback

```
static OP_POST_SIMULATE_FN(modulePostSimulate) {
   STYPE_FREE(mem);    // free 64K area
}
```

12.3.4 External Memory: Combining Callbacks and Native Memory

Occasionally, it can be useful to specify memory regions that combine aspects of mapping using external callbacks and mapping using native memory pointers (in other words, a combination of the effects of *opBusSlaveNew* and *opMemoryNativeDynamic*). For example, it might be the case that a memory region should be mapped natively for read accesses, but use a callback for write accesses. In such cases, function *opBusSlaveNew* can be used:

```
void opBusSlaveNew (
   optBusP       bus,
   const char*   portName,
   optProcessorP processor,
   optPriv       priv,
   Addr          lowAddr,
   Addr          highAddr,
   optBusSlaveReadFn  readCB,
   optBusSlaveWriteFn writeCB,
   void*         nativeMemory,
   void*         userData
);
```

To specify how the memory is used, three arguments may be used in various combinations:

1. *readCB*: if non-*NULL*, this indicates that read accesses to the memory should use this callback. If *NULL*, then read accesses should be performed directly using pointer *memory*.
2. *writeCB*: if non-*NULL*, this indicates that write accesses to the memory should use this callback. If *NULL*, then write accesses should be performed directly using pointer *memory*.
3. memory: this is a native pointer to be used for read or write accesses when either the read or write callback function is *NULL*.

12.3.5 Debugging Bus Connections

When there are many connections to a bus, visualizing the connections can be difficult, to help, there are functions to help show what is connected on a bus or all busses in a module:

```
void opBusShow(optBusP bus);
void opModuleBusShow(optModuleP module);
```

Given a bus, the opBusShow function prints details of all the master (e.g. processors) and slaves (e.g. memories) currently connected to that bus. As an example, the output might look like this:

```
BUS MASTERS: 2
  PORT 'DATA' of 'cpu1'
  PORT 'INSTRUCTION' of 'cpu1'
BUS SLAVES: 2
  0x00000000:0x003fffff: PORT 'mp1' of 'mem1'
  0x00400000:0x00400fff: MAPPED r-- RCB:0x8048808 WCB:0x8048857
  0x00401000:0xffffffff: PORT 'mp2' of 'mem2'
```

Given a module, the opModuleBusShow function prints details of all the master and slaves currently connected to each of the busses in the module. As an example, the output might look like this:

```
BUS MASTERS: 2
```

The same output can also be obtained using the command line argument *--showbusses*.

The opBusShow, opModuleBusShow and –showbuses provide a static view of the platform bus connections, they do not show any dynamic bus connections.
The command line arguments are processed as the initial platform construction has been completed and so they also cannot show any dynamic changes during execution.

The memories have a dynamic view using the memory domains. These can be displayed using the opModuleDomainDebug function or the command line argument --showdomains. The command line argument provides the initial view after construction, the opModuleDomainDebug could be called at any time to provide the current dynamic memory mappings.

12.4 Adding Memory Access Callbacks

Adding monitor or watchpoint callbacks across address regions allows address accesses to be observed. A monitor or watchpoint callback can be executed whenever there is either a read, a write or a fetch access to a specified range of memory addresses.

The monitoring of address accesses can show the behavior of a processor as it runs an application.

The watchpoint callback can be used to stop the simulation i.e. return from the opRootModuleSimulate or opProcessorSimulate functions.

The monitors or watchpoints can be placed on a processor access, a bus access or a memory access.

12.4.1.1 Adding a Memory Monitor

The monitor callbacks can be created for address ranges accessed by a processor, on a bus or to a memory.

12.4.1.2 Monitoring Processor Accesses

The access, read, write or instruction fetch, from a specific processor can be monitored using `opProcessorReadMonitorAdd` , `opProcessorWriteMonitorAdd` and `opProcessorFetchMonitorAdd` functions respectively.

```
// watch read accesses to the address range 0x01000000:0x01000fff
opProcessorReadMonitorAdd(processor, 0x01000000, 0x01000fff, readCallBack, 0);

// watch write accesses to the address range 0x01000000:0x01000fff
opProcessorWriteMonitorAdd(processor, 0x01000000, 0x01000fff, writeCallBack, 0);

// watch instruction fetch accesses to the address range 0x01000000:0x01000fff
opProcessorFetchMonitorAdd(processor, 0x01000000, 0x01000fff, fetchCallBack, 0);
```

12.4.1.3 Monitoring Accesses Over a Bus

An access, read, write or instruction fetch, on a specific bus can be monitored using `opBusReadMonitorAdd` , `opBusWriteMonitorAdd` and `opBusFetchMonitorAdd` functions respectively.

```
// watch read accesses to the address range 0x01000000:0x01000fff
opBusReadMonitorAdd(bus, proc, 0x01000000, 0x01000fff, readCallBack, 0);

// watch write accesses to the address range 0x01000000:0x01000fff
opBusWriteMonitorAdd(bus, proc, 0x01000000, 0x01000fff, writeCallBack, 0);

// watch instruction fetch accesses to the address range 0x01000000:0x01000fff
opBusFetchMonitorAdd(bus, proc, 0x01000000, 0x01000fff, fetchCallBack, 0);
```

The second argument (*proc*) is an optional processor argument that may be provided, if set to a processor handle the bus access monitor will only be triggered on an access by the specific processor, for monitoring any bus access this argument should be set to NULL

12.4.1.4 Monitoring Accesses to a Memory

An access, read, write or instruction fetch, on a specific memory can be monitored using `opMemoryReadMonitorAdd` , `opMemoryWriteMonitorAdd` and `opMemoryFetchMonitorAdd` functions respectively.

```
// watch read accesses to the address range 0x01000000:0x01000fff
opMemoryReadMonitorAdd(mem, proc, 0x01000000, 0x01000fff, readCallBack, 0);

// watch write accesses to the address range 0x01000000:0x01000fff
opMemoryWriteMonitorAdd(mem, proc, 0x01000000, 0x01000fff, writeCallBack, 0);

// watch instruction fetch accesses to the address range 0x01000000:0x01000fff
opMemoryFetchMonitorAdd(mem, proc, 0x01000000, 0x01000fff, fetchCallBack, 0);
```

The second argument (*proc*) is an optional processor argument that may be provided, if set to a processor handle the Memory access monitor will only be triggered on an access by the specific processor, for monitoring any Memory access this argument should be set to NULL

12.4.1.5 Example

This example is found in the monitorAccesses directory.

```
$IMPERAS_HOME/Examples/SimulationControl/monitorAccesses
```

This example shows the use of a monitor to trap a write to a specific address.
In the main function of the harness a callback on a write to a word at 0x00400000 is added. The userData field is used to pass a name of the watch point to the callback function.

```
  //
// triggered when registered access happens and prints information of access
//
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
   );
}
```

```
//
// iterate across the processors found in the module and register callbacks for read, write
and fetch
//
static void monitorProcessor(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 processors found in module
    optProcessorP processor = 0;
    while ((processor = opProcessorNext(mod, processor))) {

        Addr max = 0;
        optBusPortConnP bpc = opObjectByName(processor, "DATA",
OP_BUSPORTCONN_EN).BusPortConn;
        if(bpc) {
            optBusP bus = opBusPortConnBus(bpc);
            max = opBusMaxAddress(bus);
        }
        Addr min = 0;

        opMessage("I", PREFIX "_BM", "Add monitor for '%s' (0x" FMT_A0Nx " to 0x"
FMT_A0Nx ")\n",
                                opObjectHierName(processor), (Addr)0, max);
        opProcessorFetchMonitorAdd(processor, 0, min, max, monitorCallback, "processor-
fetch");
        opProcessorReadMonitorAdd (processor, 0, min, max, monitorCallback, "processor-
read");
        opProcessorWriteMonitorAdd(processor, 0, min, max, monitorCallback, "processor-
write");

    }
}

//
// iterate across the busses found in the module and register callbacks for read, write and
fetch
//
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 busses found in module
    optBusP bus = 0;
    while ((bus = opBusNext(mod, bus))) {

        Addr max = opBusMaxAddress(bus);
        Addr min = 0;

        if (options.busshow) {

            opBusShow(bus);     // print the bus connections for each bus found

        } else {

            opMessage("I", PREFIX "_BM", "Add monitor for '%s' (0x" FMT_A0Nx " to 0x"
FMT_A0Nx ")\n",
                                    opObjectHierName(bus), (Addr)0, max);


            opBusFetchMonitorAdd(bus, 0, min, max, monitorCallback, "bus-fetch");
            opBusReadMonitorAdd (bus, 0, min, max, monitorCallback, "bus-read");
            opBusWriteMonitorAdd(bus, 0, min, max, monitorCallback, "bus-write");
        }

    }
}

//
// iterate across the memories found in the module and register callbacks for read, write
and fetch
//
static void monitorMemory(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 memories found in module
    optMemoryP memory = 0;
    while ((memory = opMemoryNext(mod, memory))) {

        Addr max = opMemoryMaxAddress(memory);
        Addr min = 0;

        opMessage("I", PREFIX "_BM", "Add monitor for '%s' (0x" FMT_A0Nx " to 0x"
FMT_A0Nx ")\n",
                                opObjectHierName(memory), (Addr)0, max);

        opMemoryFetchMonitorAdd(memory, 0, min, max, monitorCallback, "memory-
fetch");
        opMemoryReadMonitorAdd (memory, 0, min, max, monitorCallback, "memory-
read");
        opMemoryWriteMonitorAdd(memory, 0, min, max, monitorCallback, "memory-
write");

    }
}
```

The callback functions used for a read, a write or a fetch should be defined using the macro `OP_MONITOR_FN`. The callback function is defined below.

```
//
// triggered when registered access happens and prints information of access
//
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
    );
}
```

To run the example, compile the test harness, the module and the application using the following commands in the `monitorAccess point` directory:

```
make –C harness
make –C module
make –C application CROSS=OR1K
```

To run the simulation, in the `monitorAccess` directory, run :

```
./ # run the harness
./harness/harness.Linux32.exe \
        --program application/application.OR1K.elf
```

You should see output similar to the following:

```

```

12.4.2 Adding a Memory Watchpoint

Unlike the Memory Monitor a Memory Watchpoint can cause the simulation to return to the harness and so return control.

OP_ADDR_WATCHPOINT_CONDITION_FN

Note that:

1. The load from address 0x00400000 is captured by the write callback.
2. The simulation is set to finish before the next instruction, which is not executed.
3. The status code passed as the second argument to `opModuleFinish` is printed just before the simulation exits.

13 Simulation Optimization

A simulator using the OP interface is at liberty to pre-read code that is going to simulate, then make internal optimization to run that code a quickly as possible. If that code is then modified by another processor or PSE model, or by an OP API call (e.g.. `opProcessorWrite()`), then the simulator must discard and re-generate its optimizations. If however, the code is stored in memory created using `opMemoryNativeNew()` and modified by an agent external to OP, the simulator will not know its code is invalid. The Programmer must therefore notify the simulator using `opProcessorFlush()`.

13.1 Example

In the module creation a native memory is created and connected onto the bus that has been found in the module. If multiple busses are contained in the module an alternative function, opObjectByName, could be used to find a specific bus or processor.

```
// assume single bus, so first is the one we want
optBusP bus = opBusNext(mi, NULL);
 if (!bus)
    opMessage("F", "MOD", "No bus in module '%s'", opObjectName(mi));

// An area of native memory
Uns32 memsize = 0x03E07FFF - 0x03E00000+1;
 void *nativePointer = STYPE_ALLOC_N(memsize, char);
 memset(nativePointer, 0, memsize);

// connect memory
 opMemoryNativeNew(
    mi,
     "Native",
     OP_PRIV_RWX,
    memsize-1,
    nativePointer,
    OP_CONNECTIONS(
        OP_BUS_CONNECTIONS(
            OP_BUS_CONNECT(bus1, "sp1", .slave=1,
                                    .addrLo=0x03E00000, .addrHi=0x03E07FFF)
        )
     ),
     0
 );
```

During simulation if is possible that the memory may be changed, for example an update from an external source as shown in this example code using a memcpy to modify the memory region

```
…
   // assume single processor, so first is the one we want
   optProcessorP proc = opProcessorNext(mi, NULL);
    if (!proc)
       opMessage("F", "MOD", "No processor in module '%s'", opObjectName(mi));

// copy newData into the native memory region
memcpy(nativePointer, newData, memsize);

// flush processor for the modified address range
opProcessorFlush(proc, 0x03E00000, 0x03E00000+memsize -1);
```

When the processor is now allowed to continue simulation, it will re-optimize any code that has been executed from this memory region.

14 Enabling Peripheral Diagnostics

A peripheral model can be written to provide diagnostics information during its execution. The diagnostics can be defined by the model developer using the modeling equivalent of printf within the model (model diagnostics) or provided from the simulation system (system diagnostics).

14.1 Model Diagnostics

The model diagnostics are controlled by setting the diagnostic level of the peripheral model. This is enabled in the platform after the PSE has been instantiated using the *opPeripheralDiagnosticLevelSet* function call.

It is standard to provide 3 levels of diagnostics within the model diagnostics, each higher level providing a super set of lower level diagnostics.

For a PSE based peripheral the diagnostics would be controlled by values of 0, 1, 2, or 3 being written.

0 No diagnostics
1 Low diagnostics
2 Medium diagnostics
3 High diagnostics

By setting a value higher than 16 System level diagnostics may also be enabled and displayed in conjunction with the model based diagnostics level set

16 System diagnostics. At this level (and above) the simulator automatically reports net and register callbacks, without the addition of code to the model.

The example code below would set the diagnostics to the highest level and so provide the most verbose output. This could provide details down to the individual register level.

```
opPeripheralDiagnosticLevelSet(vga, 3);
```

14.2 Intercept Library Diagnostics

When the peripheral's behavior is created using native code within an interception library the diagnostics must be passed through the PSE part of the model using the same mechanism as for the model diagnostics, mentioned earlier.

The recommended approach is to use high order bits in the diagnostic level for the intercept library. So, for example, to turn on the highest diagnostics level for both the PSE and Native elements of a peripheral model, we can use bits 0 and 1 for the PSE and 4 and 5 for the Native.

```
#define PSE_DIAG_HIGH          3
#define PSE_DIAG_MEDIUM    2
#define PSE_DIAG_LOW           1
#define INT_DIAG_HIGH        (3<<4)
#define INT_DIAG_MEDIUM   (2<<4)
#define INT_DIAG_LOW         (1<<4)

...

opPeripheralDiagnosticLevelSet(vga, INT_DIAG_HIGH | PSE_DIAG_HIGH);
```

14.3 Peripheral Debug Support

When simulating a platform with the Imperas professional simulator (CpuManager) full symbolic debug is available for each peripheral (PSE) model within the integrated debug environment, eGui.

14.4 Controlling peripheral model diagnostics

The diagnostics generated by a peripheral model can be individually controlled by the use of overrides either applied on the command line or in the harness. Alternatively, the diagnostic level can be set for all the peripheral models in the entire design on the command line.

14.4.1 From Command Line

The overrides that can be applied in a design from the command line can be found using the *--showoverrides* argument.
This will provide a listing similar to that shown (a small extract generated for the MIPS Malta platform) which will show the available overrides

```
--override MipsMalta/debugpseconstructors
--override MipsMalta/idebug
--override MipsMalta/mi
…
--override MipsMalta/PCI_PM/diagnosticlevel
--override MipsMalta/PCI_PM/trace
--override MipsMalta/PCI_PM/enabletoolspse
--override MipsMalta/PCI_PM/PCIslot
--override MipsMalta/PCI_PM/PCIfunction
--override MipsMalta/PCI_NET/diagnosticlevel
--override MipsMalta/PCI_NET/trace
…
--override MipsMalta/intCtrlMaster/diagnosticlevel
```

```
--override MipsMalta/intCtrlMaster/trace
--override MipsMalta/intCtrlMaster/enabletoolspse
--override MipsMalta/intCtrlMaster/spen
--override MipsMalta/intCtrlSlave/diagnosticlevel
--override MipsMalta/intCtrlSlave/trace
--override MipsMalta/intCtrlSlave/enabletoolspse
--override MipsMalta/intCtrlSlave/spen
…
```

These may then be applied to the command line to set the diagnostics level for one or more of the peripherals

```
./harness/harness.Linux32.exe --program application/application.OR1K.elf \
        --override MipsMalta/PCI_NET/diagnosticlevel=2 \
        --override MipsMalta/intCtrlSlave/diagnosticlevel=2
```

By using the command line argument *--modeldiags* the diagnostics level can be set for ALL the peripheral models in the design.

For example, the following will set all peripheral model diagnostic levels to 2

```
./harness/harness.Linux32.exe --program application/application.OR1K.elf \
        --modeldiags 2
```

14.4.2 From Harness

A diagnostics level may be controlled from the test harness using overrides during the construction phase using the Uns32 parameter override for the peripheral instance as shown below

```
opParamUns32Override(mi, “top/u1/peripheral1/” OP_FP_DIAGNOSTICLEVEL, 2);
```

15 Model and Intercept Object Additional Commands

A processor model or an intercept object can install its own commands to be executed as required during simulation. Commands are typically used to enable or disable functionality in the model or intercept object or to extract analysis data that the model or intercept object has been accumulating.

Commands are installed in a processor model using vmirtAddCommand(). See OVP_VMI_Run_Time_Function_Reference.doc.

Commands are installed in a intercept object using vmiosAddCommand(). See OVP_VMI_OS_Support_Function_Reference.doc, and opProcessorExtensionNew() in this document.

Commands are called, from a harness, using opCommandCall(). A command can be called any time after it has been installed and before the simulation terminates, but the user needs to be aware of when installation occurs. Models and intercept objects are recommended to install their commands in their constructors, in which case the earliest 'safe' time to call a command is immediately before opRootModuleSimulate() or opProcessorSimulate().

This example calls a command that is installed by a processor model and lists all the available commands on the processor using an iterator function

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

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

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

    opRootModulePreSimulate(mr);

    // get processor, assumes only a single processor
    optProcessorP processor = opProcessorNext(mi, NULL);

    // Run the processor for 100 instructions
    opProcessorSimulate(processor, 100);

    // Call a command that had been created in a processor model
    // Pointer to the call command result string. This could be any string
    const char *result;

    opPrintf("Call Model Commands\n\n");
```

```
    // Setup command for reading MIPS COP0 registers
    const char* cmdToExecute = "u1/cpu1/mipsCOP0";
    optCommandP cmd = opObjectByName(mr,
                                     cmdToExecute,
                                     OP_COMMAND_EN).Command;
    if(!cmd)
        opMessage("F", "harness", "Command '%s' not found", cmdToExecute);

    // Setup calling arguments to read COP0 'Config' register
    const char *cmd1Argv[] = {"0", "16", "0"};
    result = opCommandCall( cmd,
                            3,
                            cmd1Argv);

    opPrintf("Call Command result '%s'\n", result);

    // Setup calling arguments to read COP0 'PrId' register
    const char *cmd2Argv[] = {"0", "15", "0"};

    result = opCommandCall( cmd,
                            3,
                            cmd2Argv);

    opPrintf("Call Command result '%s'\n", result);

    opPrintf("Discover installed commands\n");

    // iterates over all commands on all processor instances
    opModuleCommandsShow(mi);

    const char *procName = "cpu1";
    opPrintf("Commands for %s\n", procName);
    optProcessorP proc = opObjectByName(mi, procName,
OP_PROCESSOR_EN).Processor;

    // iterates over all commands on this processor instance
    opProcessorCommandIterAll(proc, printCommand, 0);

    opPrintf("Complete Simulation\n\n");

    // run simulation to completion
    opRootModuleSimulate(mr);

    opSessionTerminate();
    return 0;
}
```

Note that although each command can use its arguments in any way, it is normal practice to use the Unix convention of passing the command name as the first argument. Thus argv[0] is the command name and argv[1] is the first true argument.

An example of calling commands that are created within a processor model is in

```
$IMPERAS_HOME/Examples/SimulationControl/callingInstalledCommands.
```

This uses the MIPS32 model commands.

In harness/harness.c an array is defined for the arguments for the command. This is a Linux like argv, argc array; with argument zero the name of the command

```
const char *cmd1Argv[] = {"mipsCOP0", “-register”, "16", “-select”, "0"};
```

When calling the command this array is passed to the opCallCommand function.

```
result = opCommandCall( cmd,
                        5,
                        cmd1Argv);
```

The result returned from the opCommandCall function is a string passed back from the command itself after execution. It can represent success or failure of the command or it can be an information string; this is command dependent.

An OP API harness can discover what commands are available, for a module or for a particular processor model.

The example uses the MIPS32 model commands.

All the installed commands can be shown for a module using

```
// iterates over all commands on all processor instances
 opModuleCommandsShow(mi);
```

This provides the output in a form that can be used on the command line to call the command, with the help message provided in brackets

```
Discover installed commands
--callcommand u1/cpu1/mipsCOP0 [query a COP0 register value using <register>
<select>]
--callcommand u1/cpu1/mipsWriteRegister [Write to a processor register using
<resource> <offset> <value>]
--callcommand u1/cpu1/mipsReadRegister [Read a processor register using <resource>
<offset>]
```

```
--callcommand u1/cpu1/mipsCacheEnable [enable tag or full cache model]
--callcommand u1/cpu1/mipsCacheDisable [Disables tag or full cache model]
--callcommand u1/cpu1/mipsCacheReport [Report current cache statistics]
--callcommand u1/cpu1/mipsCacheReset [reset the cache model]
--callcommand u1/cpu1/mipsCacheRatio [Report current hit ratio for selected cache]
--callcommand u1/cpu1/mipsCacheTrace [Control the tracing of cache accesses]
--callcommand u1/cpu1/mipsDebugFlags [Set the processor model debug flags to
<value>]
--callcommand u1/cpu1/itrace [enable or disable instruction tracing]
--callcommand u1/cpu1/isync [specify instruction address range for synchronous
execution]
```

Alternatively, the commands of a specific processor instance can be extracted and printed by an command function defined using the macro OP_COMMAND_FN. The function is declared using the provided prototype macro. Arguments to the function are the command name and user data.

```
static OP_COMMAND_FN(printCommand) {
    opPrintf("Command %s\n", opObjectHierName(command));
}
```

The function printCommand will be called for every installed command on a particular processor:

```
    optProcessorP proc = opObjectByName(mi, procName,
OP_PROCESSOR_EN).Processor;

    // iterates over all commands on this processor instance
    opProcessorCommandIterAll(proc, printCommand, 0);
```

that provides the output listing the command only

```
Commands for cpu1
Command u1/cpu1/isync
Command u1/cpu1/itrace
Command u1/cpu1/mipsCOP0
Command u1/cpu1/mipsCacheDisable
Command u1/cpu1/mipsCacheEnable
Command u1/cpu1/mipsCacheRatio
Command u1/cpu1/mipsCacheReport
Command u1/cpu1/mipsCacheReset
Command u1/cpu1/mipsCacheTrace
Command u1/cpu1/mipsDebugFlags
Command u1/cpu1/mipsReadRegister
Command u1/cpu1/mipsWriteRegister
```

16 Introspecting and Querying Platforms & Components

As we have seen in many of the previous examples the OP API provides a large set of functions to allow objects to be found in a simulation and for these objects then to be queried for information.

When wishing to obtain information from objects in a design it is possible to search for the object using its name with the opObjectByName function which, using a union of object types, allows the correct type to be returned. The following shows how a bus object named 'busMain' would be found within the module instance *mi* using this function

```
optBusP bus = opObjectByName(mi, "busMain", OP_BUS_EN).Bus;
```

An alternate way of finding the an object is to use a combination of an iterator function and the query functions. The following shows how we could iterate within the same module, mi for the bus named 'busMain'.

```
 optBusP bus = 0;
while ((bus = opBusNext(mi, bus))) {
    const char *busName = opObjectHierName(bus);
    if (strcmp(busName,"busMain") == 0) {
        opPrintf("Found bus 'busMain'\n");
        break;
    }
}
```

Each object type has its own iterator and query functions available.

16.1 Platform Introspector: Examples/PlatformConstruction/walker

The full set of iterator and query functions will not be listed in this document but the example found in ***IMPERAS_HOME/Examples/PlatformConstruction/walker*** makes use of all the introspection and query functions in the harness (harness/harness.c) to find objects for each type and fully iterate across them and print their respective information.

The harness instances a module that contains a processor, bus, peripheral and memory

```
    optModuleP mi = opRootModuleNew(0, 0, 0);
    if(!options.modulepath) {
        options.modulepath = MODULE_DIR;
    }
    opModuleNew(mi, options.modulepath, MODULE_INSTANCE, 0, 0);
```

```
    opRootModulePreSimulate(mi);
```

and then for this module it calls a 'walker' function that given the name of an object from which to start (obtained from the custom command line argument 'objectname) iterates all the objects

```
optObjectP startObject = opObjectByName(mi, options. objectname, 0);
…
        walkObject(0, startObject);
…
```

The walkObject function obtains, using the opObjectType(), the optStrType from the enumeration optStrTypeE so that it can call the correct object iterator

The enumeration of object types is

Enumeration	Object Type Description
OP_APPLICATION_EN	Application program object file.
OP_BRIDGE_EN	Instance of a bridge.
OP_BUS_EN	Bus interconnect instance.
OP_BUSSLAVE_EN	Pair of read/write callbacks on a bus region.
OP_BUSPORT_EN	Bus port specification.
OP_BUSPORTCONN_EN	Bus port instance.
OP_COMMAND_EN	Model or plugin command.
OP_EXTENSION_EN	Instance of an extension applied to a model.
OP_EXTELAB_EN	An elaborated extension.
OP_FIFO_EN	The FIFO instance.
OP_FIFOPORT_EN	FIFO port definition.
OP_FIFOPORTCONN_EN	FIFO port instance.
OP_FORMAL_EN	The formal parameter specifying a legal parameter of a model.
OP_FORMALENUM_EN	A name and value pair.
OP_MEMORY_EN	Instance of a memory.
OP_MMC_EN	Instance of an MMC.
OP_MMREGISTER_EN	Memory Mapped Register.
OP_MODULE_EN	Instance of a module.
OP_NET_EN	Net instance.
OP_NETMONITOR_EN	Net callback installed before the platform is built.
OP_NETPORT_EN	Net port definition, used when querying the model.
OP_NETPORTCONN_EN	Net port connection instance.
OP_PACKETNET_EN	Interconnect used to model packet networks such as Ethernet or CAN bus.
OP_PACKETNETMONITOR_EN	Packetnet callback installed before the platform is built.

OP_PACKETNETPORT_EN	Packetnet port definition, used when querying the model.
OP_PACKETNETPORTCONN_EN	Packetnet port instance.
OP_PARAM_EN	Parameter of module or component instance.
OP_PERIPHERAL_EN	Instance of a peripheral model.
OP_PROCESSOR_EN	Instance of a processor model.

Once the object type is known a specific function is called to iterate.

At a low level we could be walking over a bus instance as shown in the following:

```
static void walkBus(Uns32 indent, optBusP o){
    indentText(indent++);
    opPrintf("Bus: \"%s\"\n", opObjectHierName(o));
    indentText(indent);
    opPrintf("addrBits : %d\n", opBusAddrBits(o));
    walkBusSlaveList(indent, o);
    walkParamList(indent, o);
    walkFormalList(indent, o);
    walkApplicationList(indent, o);
}
```

For this bus instance we walk over and display query information for the objects that are related to the bus, for example the walkBusSlaveList will iterate across all the bus slaves connected to this bus and provide information of that object

```
static void walkBusSlaveList(Uns32 indent, optBusP o){
    optBusSlaveP ch = 0;
    while((ch = opBusSlaveNext(o,ch))) {
        walkBusSlave(indent, ch);
    }
}
```

16.2 Running the platform & component introspecting harness: walker

Take a copy of the introspection and query example in

```
$IMPERAS_HOME/Examples/SimulationControl/walker
```

The introspection and query harness and the example module can be built with the following commands

```
> make –C harness
> make –C module
```

and then executed with

```
> ./harness/harness.Linux32.exe
```

The following output has been reduced to show some of the information that would be observed:

```
Module: ""
.  path : "(null)"
.  Formal: "debugpseconstructors"
.  .  Type: "Boolean"
.  .  Desc: "Start the debugger BEFORE PSE constructors have run"
.  .  default:0

…

.  Module: "u1"
.  .  path : "module"
.  .  Bus: "u1/mainBus"
.  .  .  addrBits : 32
.  .  .  Parameter: (cursor)
.  .  Memory: "u1/ram1"
.  .  .  priv : 7
.  .  .  maxAddress : fffffff
.  .  .  BusPort: "u1/ram1/sp1"
.  .  .  .  description : "(null)"
.  .  .  .  type : 2
.  .  .  .  domainType : 0
.  .  .  .  addrHi : 0
.  .  .  .  addrBitsMin : 0
.  .  .  .  addrBitsMax : 0
.  .  .  .  mustConnect : 0
.  .  .  .  isDynamic : 0

…

.  .  Peripheral: "u1/periph0"
.  .  .  path :
"Imperas/lib/Linux32/ImperasLib/freescale.ovpworld.org/peripheral/KinetisUART/1.0/pse.pse"
.  .  .  BusPort: "u1/periph0/bport1"
…
```

16.3 Using the walkers command line

You can use the command line options to specify the module to load and the object to walk (introspect):

```
> ./harness/harness.Linux32.exe \
        --modulepath module \
        --objectname u1/periph0
```

This will load the module from the path 'module', and will start the walking from the object named 'u1/periph0' and will introspect the peripheral:

```
OVPsim started: Mon Mar 14 12:56:57 2016

Peripheral;'u1/periph0'
.  path : ImperasLib/freescale.ovpworld.org/peripheral/KinetisUART/1.0/pse.pse"
.  BusPort;'u1/periph0/bport1'
.  .  description : "(null)"
.  .  type : 2
.  .  domainType : 0
.  .  addrHi : 0xfff
.  .  addrBitsMin : 0
.  .  addrBitsMax : 0
.  .  mustConnect : 0
.  .  isDynamic : 0
.  BusPortConn;'u1/periph0/bport1'
.  .  type : 2
.  .  addrHi : 0x100013f7
.  .  addrLo : 0x100003f8
.  .  isDynamic : 0
.  .  Bus: u1/mainBus
.  NetPort;'u1/periph0/DirectWrite'
.  .  description : "(null)"
.  .  type : 2
.  .  mustConnect : 0
.  NetPort;'u1/periph0/DirectRead'
.  .  description : "(null)"
.  .  type : 1
.  .  mustConnect : 0
.  NetPort;'u1/periph0/Interrupt'
.  .  description : "(null)"
.  .  type : 2
.  .  mustConnect : 0
.  NetPort;'u1/periph0/Reset'
.  .  description : "(null)"
.  .  type : 1
.  .  mustConnect : 0
.  NetPortConn;'u1/periph0/DirectWrite'
.  .  type : 0
.  .  Net: u1/directWrite
.  NetPortConn;'u1/periph0/DirectRead'
.  .  type : 0
.  .  Net: u1/directRead
.  Parameter: (cursor)
.  Parameter: "u1/periph0/outfile"    uartTTY0.log
.  Formal: "u1/periph0/diagnosticlevel" (system)
.  .  Type: "Uns32"
.  .  Desc: "Set the peripheral model diagnostic level to this value"
```

```
.  .  min:0 max:4294967295 default:0

...

.  Formal: "u1/periph0/replay" (model)
.  .  Type: "String"
.  .  Desc: "Replay external events from this file"
.  .  max:0 default:((null))

OVPsim finished: Mon Mar 14 12:56:57 2016
```

17 Save / Restore

17.1 Introduction

The Imperas simulator supports save and restore. This allows a simulation to be run to a specific time or event in the future, stopped and the current state saved. This state may then be restored into the simulator and execution continued which allows multiple simulations to be carried out from a known state without having to re-execute any of the previous simulation.

The simulator automatically performs save and restore on known internal state of a component, for example defined register structures in processors and peripherals. Where the component has additional specific internal state callback functions are used to define how this state should be saved and restored.

17.2 Checking Supported

For any object in a design the *opObjectSaveRestoreSupported* can be used to determine if that object supports save and restore.

All components used within a design should support save and restore. Each component can be interrogated to see if it supports save and restore, for example to find the processors and peripherals status in a module

```
//
// function to report and return save / restore status of an object
//
static Bool reportSupported(optObjectP obj) {
    Bool ok = opObjectSaveRestoreSupported((optObjectP) obj);

    opMessage("I", "SAVE_RESTORE_CHECK", "'%s' ('%s/%s/%s/%s') : %s",
                            opObjectName(obj),
                                                opVLNVVendor(opObjectVLNV(obj)),
                                                opVLNVLibrary(opObjectVLNV(obj)),
                                                opVLNVName(opObjectVLNV(obj)),
                                                opVLNVVersion(opObjectVLNV(obj)),
                                                ok ? "supported" : "not supported");

    return ok;
}

//
// iterate across components and check they all support save/restore
```

```
//
static void checkSupported(optModuleP mi) {
        Bool allSupport = True;
        opMessage("I", "SAVE_RESTORE_CHECK",
                "Check all components support save/restore");

        // Processors
        optProcessorP proc = 0;
        while ( (proc = opProcessorNext(mi, proc))) {
                Bool ok = reportSupported(proc);
                allSupport &= ok;
        }
        // Peripherals
        optPeripheralP per = 0;
        while ( (per = opPeripheralNext(mi, per))) {
                Bool ok = reportSupported(per);
                allSupport &= ok;
        }

        if (!allSupport) {
                opMessage("W", "SAVE_RESTORE_CHECK",
"Not all components fully support save/restore: may not work in all operation modes!");
        }
}
```

17.3 Validating Processor Model Save and Restore

There are a set of OP API functions that can be used to verify that save and restore is correctly working in a processor model.

NOTE: These functions are not intended to be used for the save and restore of a design. They allow the state of a processor and its associated memory to be saved and restored for the purpose of verifying the save and restore capabilities of a processor module during development. The save and restore for a design is controlled using the Imperas M*SDK product interactive interfaces, see section 17.4 “Using Save and ”.

The processor and memory state can be saved and restored locally within the harness to allow a verification harness to be created using the set of functions

```
opMemoryStateSave
opProcessorStateSave
opMemoryStateRestore
opProcessorStateRestore
```

The state can also be saved to and restored from a file using the following

```
opMemoryStateSaveFile
opProcessorStateSaveFile
opMemoryStateRestoreFile
opProcessorStateRestoreFile
```

Take a copy of the processor model validation save and restore example in

```
$IMPERAS_HOME/Examples/SimulationControl/processorSaveRestoreValidation
```

The example module can be built with the following commands

```
> make –C harness
> make –C module
```

and then executed with

```
> harness/harness.Linux32.exe
```

This executes 100 instructions on the processor, saves state and then runs a further 1000 instructions. At this point the processor and memory state is restored and the simulation runs until completion. Instruction tracing is performed to verify that the instructions executed after the save/restore point are the same in each case.

The following shows an extract from the simulation output on the initial execution of instructions 101 onwards:

```
Info 99: 'top/u1/cpu1', 0x00000000000017d4(_strlen+3c): l.and    r4,r4,r5
Info 100: 'top/u1/cpu1', 0x00000000000017d8(_strlen+40): l.movhi  r5,0x8080
Simulator StopReason = Instruction count breakpoint is pending
OP_SR_BP_ICOUNT @ST1 : Save State
Info 101: 'top/u1/cpu1', 0x00000000000017dc(_strlen+44): l.ori    r5,r5,0x8080
Info 102: 'top/u1/cpu1', 0x00000000000017e0(_strlen+48): l.and    r4,r4,r5
Info 103: 'top/u1/cpu1', 0x00000000000017e4(_strlen+4c): l.sfnei  r4,0x0
Info 104: 'top/u1/cpu1', 0x00000000000017e8(_strlen+50): l.bnf    0x000017bc
Info 105: 'top/u1/cpu1', 0x00000000000017ec(_strlen+54): l.nop    0x0
Info 106: 'top/u1/cpu1', 0x00000000000017bc(_strlen+24): l.addi   r3,r3,0x4
Info 107: 'top/u1/cpu1', 0x00000000000017c0(_strlen+28): l.lwz    r4,0x0(r3)
```

Subsequently after the simulation is stopped and restored at instruction 1100 we get the same instruction execution sequence:

```
…
Info 1098: 'top/u1/cpu1', 0x00000000000019f0(___sfvwrite+1d0): l.ori    r4,r18,0x0
Info 1099: 'top/u1/cpu1', 0x00000000000019f4(___sfvwrite+1d4): l.sfgts  r7,r6
```

```
Info 1100: 'top/u1/cpu1', 0x00000000000019f8(___sfvwrite+1d8): l.bf     0x00001c0c
Simulator StopReason = Instruction count breakpoint is pending
OP_SR_BP_ICOUNT @ST2 : Restore state
Info 101: 'top/u1/cpu1', 0x00000000000017dc(_strlen+44): l.ori    r5,r5,0x8080
Info 102: 'top/u1/cpu1', 0x00000000000017e0(_strlen+48): l.and    r4,r4,r5
Info 103: 'top/u1/cpu1', 0x00000000000017e4(_strlen+4c): l.sfnei  r4,0x0
Info 104: 'top/u1/cpu1', 0x00000000000017e8(_strlen+50): l.bnf    0x000017bc
Info 105: 'top/u1/cpu1', 0x00000000000017ec(_strlen+54): l.nop    0x0
Info 106: 'top/u1/cpu1', 0x00000000000017bc(_strlen+24): l.addi   r3,r3,0x4
Info 107: 'top/u1/cpu1', 0x00000000000017c0(_strlen+28): l.lwz    r4,0x0(r3)
…
```

17.4 Using Save and Restore in simulation

The save and restore is controlled from the Imperas M*SDK command line interface using the ‘save’ and ‘restore’ commands. These commands save and restore respectively the complete state of the design to a file.

This can be illustrated using the example:

```
IMPERAS_HOME/Examples/PlatformConstruction/simpleCpuMemoryUart
```

Take a copy of the example and run the example script so that the application, harness and module are generated and compiled.

The following requires that a valid installation of M*SDK is available.

Start the platform with Imperas MPD using command line (integrated or remote) or eGui. This example will show the use of the integrated command line version with tracing enabled to show the instruction execution of the processor and also the UART console enabled to see the UART output

```
harness.exe \
    --modulefile module \
    --program application/application.OR1K.elf \
    --idebug \
    --trace –traceshowicount \
    --override simpleCpuMemoryUart/periph0/console=1
```

This will bring up the interactive command line mode with tracing enabled.

First let us set a breakpoint after 3500 instructions have been executed and run the simulation:

```
idebug (cpu1) > ::iseticountpoint -icount 3500
idebug (cpu1) > continue
```

We will see the output

```
…
CpuManagerMulti started: Mon Feb 29 14:48:19 2016
…
Info 1: 'simpleCpuMemoryUart/cpu1', 0x0000000000000100(start): l.addi   r2,r0,0x0
Info 2: 'simpleCpuMemoryUart/cpu1', 0x0000000000000104(start+4): l.addi   r3,r0,0x0
Info 3: 'simpleCpuMemoryUart/cpu1', 0x0000000000000108(start+8): l.addi   r4,r0,0x0
Info 4: 'simpleCpuMemoryUart/cpu1', 0x000000000000010c(start+c): l.addi   r5,r0,0x0
Info 5: 'simpleCpuMemoryUart/cpu1', 0x0000000000000110(start+10): l.addi   r6,r0,0x0
Info 6: 'simpleCpuMemoryUart/cpu1', 0x0000000000000114(start+14): l.addi   r7,r0,0x0
Info 7: 'simpleCpuMemoryUart/cpu1', 0x0000000000000118(start+18): l.addi   r8,r0,0x0
Info 8: 'simpleCpuMemoryUart/cpu1', 0x000000000000011c(start+1c): l.addi   r9,r0,0x0
…
Info 3498: 'simpleCpuMemoryUart/cpu1', 0x0000000000001924(_strlen+78): l.sub
r11,r3,r6
Info 3499: 'simpleCpuMemoryUart/cpu1', 0x0000000000001914(_strlen+68): l.addi
r3,r3,0x1
Info 3500: 'simpleCpuMemoryUart/cpu1', 0x0000000000001918(_strlen+6c): l.lbs
r4,0x0(r3)

Icountpoint 1 for cpu1 triggered at 3500
0x0000191c in strlen (str=0x4afb "0 world\n\n") at ../../../../../../src/gcc-
3.4.2/newlib/libc/string/strlen.c:88
88      in ../../../../../../src/gcc-3.4.2/newlib/libc/string/strlen.c
```

The console of the UART will show the text

```
Hello
```

We will now save the state of the design into a file ‘file.sav’

```
idebug (cpu1) >  save file.sav
```

and then set a new breakpoint after 5000 instructions have been executed and continue the simulation:

```
idebug (cpu1) > ::iseticountpoint -icount 5000
idebug (cpu1) > continue
```

which will execute further instructions from 3501 to the next break at 5000

```
Info 3501: 'simpleCpuMemoryUart/cpu1', 0x000000000000191c(_strlen+70): l.sfeqi  r4,0x0
Info 3502: 'simpleCpuMemoryUart/cpu1', 0x0000000000001920(_strlen+74): l.bnf    0x00001914
Info 3503: 'simpleCpuMemoryUart/cpu1', 0x0000000000001924(_strlen+78): l.sub    r11,r3,r6
```

```
Info 3504: 'simpleCpuMemoryUart/cpu1', 0x0000000000001914(_strlen+68): l.addi   r3,r3,0x1
…
Info 4997: 'simpleCpuMemoryUart/cpu1', 0x000000000000191c(_strlen+70): l.sfeqi  r4,0x0
Info 4998: 'simpleCpuMemoryUart/cpu1', 0x0000000000001920(_strlen+74): l.bnf    0x00001914
Info 4999: 'simpleCpuMemoryUart/cpu1', 0x0000000000001924(_strlen+78): l.sub    r11,r3,r6
Info 5000: 'simpleCpuMemoryUart/cpu1', 0x0000000000001914(_strlen+68): l.addi   r3,r3,0x1

Icountpoint 2 for cpu1 triggered at 5000
0x00001918 in strlen (str=0x4af4 "lo UART0 world\n\n") at ../../../../../../src/gcc-
3.4.2/newlib/libc/string/strlen.c:88
88      in ../../../../../../src/gcc-3.4.2/newlib/libc/string/strlen.c
```

And the UART console now shows

```
Hello UART0 world
```

We can now restore the simulation state back to the point at which we stopped at 3500 instructions, and then run the simulation to completion

```
idebug (cpu1) >  restore file.sav
idebug (cpu1) >  continue
```

we see that the simulation starts from instruction 3501 once again

```
Info 3501: 'simpleCpuMemoryUart/cpu1', 0x000000000000191c(_strlen+70): l.sfeqi  r4,0x0
Info 3502: 'simpleCpuMemoryUart/cpu1', 0x0000000000001920(_strlen+74): l.bnf    0x00001914
Info 3503: 'simpleCpuMemoryUart/cpu1', 0x0000000000001924(_strlen+78): l.sub    r11,r3,r6
Info 3504: 'simpleCpuMemoryUart/cpu1', 0x0000000000001914(_strlen+68): l.addi   r3,r3,0x1
Info 3505: 'simpleCpuMemoryUart/cpu1', 0x0000000000001918(_strlen+6c): l.lbs    r4,0x0(r3)
Info 3506: 'simpleCpuMemoryUart/cpu1', 0x000000000000191c(_strlen+70): l.sfeqi  r4,0x0
…
```

Until it terminates on reaching *_exit* (which was intercepted by the loaded semihosting library)

```
…
Info 5947: 'simpleCpuMemoryUart/cpu1', 0x0000000000001974(_exit+40): l.ori    r3,r10,0x0
Info 5948: 'simpleCpuMemoryUart/cpu1', 0x00000000000047f0(__exit): *** INTERCEPT *** (__exit)
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

CpuManagerMulti finished: Mon Feb 29 15:00:20 2016
```

The UART console is now showing the additional output generated when the application program is re-executed from the restored state.

```
Hello UART0 world
```

UART0 world

NOTE: In this example the save / restore is not fully implemented in the components used but the automatic simulator save and restore is sufficient.

18 Encapsulating Models for use in other Environments

An essential purpose of the OP API is to allow Imperas simulation models to be exported to other environments (for example, SystemC).

18.1 SystemC

Integration with SystemC using OP is available and is described in the document *OVPsim Using OVP Models in SystemC TLM2.0 Platforms*.

19 Integration with Client Debuggers

It is a common requirement to be able to integrate OP platforms with client debuggers. To support this requirement, additional capabilities are supported in the Imperas Professional Tools product (not OVPsim), as described in the following sections.

19.1 Memory Access

The functions `opProcessorRead` and `opProcessorWrite` with the `debugAccess` argument equal to `True` should be used to examine or modify memory without causing side effects. See section 12 on Memory Operations.

19.2 Register Query

Debuggers often need to know the processor *registers* supported, so that they can be presented to the user and watchpoints can be set on register value changes (see section 19.9.1 for more information about watchpoints). The supported processor registers can be found using the processor register iterator:

```
optRegP opProcessorRegNext (
   optProcessorP  processor,
   optRegP  reg
);
```

The iterator should be passed `NULL` as the `reg` argument on the first call. On subsequent calls, it should be passed the value returned on the previous call. For each non-`NULL` value returned, the register name, width in bits, usage and group can be found using these functions:

```
const char *opRegName (optRegP reg);

Uns32 opRegBits (optRegP reg);

optRegUsage opRegUsageEnum (optRegP reg);

const char *opRegUsageString (optRegP reg);

optRegGroupP opRegGroup (optRegP reg);
```

The *register group* (`optRegGroupP`) allows allocation of registers into model-specific sets, to ease presentation for processors that contain many registers (see the next section).

19.3 Register Group Query

Debuggers often need to know the processor *register groups* supported. Register groups are model-specific sets into which registers are allocated to ease presentational problems when a processor model contains a large number of registers. The supported processor register groups can be found using the processor register group iterator:

```
optRegGroupP opProcessorRegGroupNext (
    optProcessorP processor,
    optRegGroupP group
);
```

The iterator should be passed `NULL` as the `group` argument on the first call. On subsequent calls, it should be passed the value returned on the previous call. For each non-`NULL` value returned, the group name can be found using:

```
const char *opRegGroupName(optRegGroupP group);
```

The registers within a group can be found using the by-group register iterator:

```
optRegP opRegGroupRegNext (
    optProcessorP processor,
    optRegGroupP group,
    optRegP reg
);
```

Like the other iterators, the iterator should be passed `NULL` as the `reg` argument on the first call. On subsequent calls, it should be passed the value returned on the previous call.

19.4 Mode State Query

Debuggers often need to know the processor *modes* supported, so that they can be presented to the user and watchpoints can be set on mode changes (see section 19.9.1 for more information about watchpoints). The supported processor modes can be found using the processor mode iterator:

```
optModeP opProcessorModeNext (
    optProcessorP processor,
    optModeP      mode
);
```

The iterator should be passed `NULL` as the `mode` argument on the first call. On subsequent calls, it should be passed the value returned on the previous call. For each non-`NULL` value returned, a string name, processor-specific code and description string can be found using these functions:

```
const char *opModeName (optModeP mode);

Uns32 opModeCode (optModeP mode);

const char * opModeDescription (optModeP mode);
```

The *current* processor mode info can be found using:

```
optModeP opProcessorModeCurrent (optProcessorP processor);
```

19.5 Exception State Query

Debuggers often need to know the processor *exceptions* supported, so that they can be presented to the user and watchpoints can be set on exception events (see section 19.9.1 for more information about watchpoints). The supported processor exceptions can be found using the processor exception iterator:

```
optExceptionP opProcessorExceptionNext (
    optProcessorP processor,
    optExceptionP exception
);
```

The iterator should be passed `NULL` as the `exception` argument on the first call. On subsequent calls, it should be passed the value returned on the previous call. For each non-`NULL` value returned, , a string name, processor-specific code and description string can be found using these functions:

```
const char * opExceptionName (optExceptionP exception);

Uns32 opExceptionCode (optExceptionP exception);

const char * opExceptionDescription (optExceptionP exception);
```

The *current* processor exception description can be found using:

```
optExceptionP opProcessorExceptionCurrent (optProcessorP processor);
```

19.6 Processor Freezing

Two routines allow specific processors in a multiprocessor platform to be frozen and unfrozen:

```
void opProcessorFreeze (optProcessorP processor);
void opProcessorUnfreeze (optProcessorP processor);
```

When in a *frozen* state, a processor in a multiprocessor simulation will not be scheduled when `opRootModuleSimulate` is called. It is therefore possible to restrict simulation to a subset of processors in a multiprocessor platform by freezing those processors that should not be run. A function is also available to test the frozen state of a specific processor:

```
Bool opProcessorFrozen (optProcessorP processor);
```

19.7 Address Breakpoints

Two routines allow breakpoints to be set and cleared for a specific processor and address:

```
void opProcessorBreakpointAddrSet (
   optProcessorP processor,
   Addr          addr
);

void opProcessorBreakpointAddrClear (
   optProcessorP processor,
   Addr          addr
);
```

When a breakpoint has been set for a specific address, any attempt by the processor to execute at that address will cause `opRootModuleSimulate` or `opProcessorSimulate` to return with the processor's `optStopReason` set to `OP_SR_BP_ADDRESS`.

19.8 Instruction Count Breakpoints

Two routines allow a breakpoint to be set and cleared that causes a processor to stop executing after a specific number of instructions:

```
void opProcessorBreakpointICountSet (
   optProcessorP processor,
   Uns64         delta
);

void opProcessorBreakpointICountClear (
   optProcessorP processor
);
```

Once the specified number of instructions has elapsed, `opRootModuleSimulate` or `opProcessorSimulate` will return with the processor's `optStopReason` set to `OP_SR_BP_ICOUNT`.

There is only one active ICount breakpoint on any processor; a second call to `opProcessorBreakpointICountSet` (before the count is reached) will replace the previous value. When the ICount breakpoint is reached, the breakpoint deletes itself.

19.9 Memory, Bus and Processor Watchpoints

A powerful watchpoint API is implemented specifically aimed at debugger integration.

19.9.1 Watchpoint Creation and Deletion

Three routines are available to set *read*, *write* or *access* (both read and write) watchpoints on a range of memory addresses in a *memory*:

```
optWatchpointP opMemoryReadWatchpointNew (
   optMemoryP     memory,
   Addr           addrLo,
   Addr           addrHi,
   void*          userData,
   optAddrWatchpointConditionFn notifierCB
);

optWatchpointP opMemoryWriteWatchpointNew (
   optMemoryP     memory,
   Addr           addrLo,
   Addr           addrHi,
   void*          userData,
   optAddrWatchpointConditionFn notifierCB
);

optWatchpointP opMemoryAccessWatchpointNew (
   optMemoryP     memory,
   Addr           addrLo,
   Addr           addrHi,
   void*          userData,
   optAddrWatchpointConditionFn notifierCB
);
```

Three more routines allow watchpoints to be specified on a *bus* range:

```
optWatchpointP opBusReadWatchpointNew (
   optBusP        bus,
   Addr           addrLo,
   Addr           addrHi,
   void*          userData,
   optAddrWatchpointConditionFn notifierCB
);

optWatchpointP opBusWriteWatchpointNew (
   optBusP        bus,
   Addr           addrLo,
   Addr           addrHi,
   void*          userData,
   optAddrWatchpointConditionFn notifierCB
);
```

```
optWatchpointP opBusAccessWatchpointNew (
    optBusP       bus,
    Addr          addrLo,
    Addr          addrHi,
    void*         userData,
    optAddrWatchpointConditionFn notifierCB
);
```

Three routines allow watchpoints to be specified on a *processor* address range. For each, a `physical` argument specifies whether the address range is in processor *physical* memory (if `True`) or *virtual* memory (if `False`)[5]:

```
optWatchpointP opProcessorReadWatchpointNew (
    optProcessorP  processor,
    Bool          physical,
    Addr          addrLo,
    Addr          addrHi,
    void*         userData,
    optProcWatchpointConditionFn notifierCB
);

optWatchpointP opProcessorWriteWatchpointNew (
    optProcessorP  processor,
    Bool          physical,
    Addr          addrLo,
    Addr          addrHi,
    void*         userData,
    optProcWatchpointConditionFn notifierCB
);

optWatchpointP opProcessorAccessWatchpointNew (
    optProcessorP  processor,
    Bool          physical,
    Addr          addrLo,
    Addr          addrHi,
    void*         userData,
    optProcWatchpointConditionFn notifierCB
);
```

One routine allows a watchpoint to be established on a *register* in a processor:

```
optWatchpointP opProcessorRegWatchpointNew (
    optProcessorP  processor,
    optRegP       reg,
    void*         userData,
```

[5] See the section 19.9.2 for a definition of what exactly *virtual* and *physical* mean in this context.

```
    optProcWatchpointConditionFn notifierCB
);
```

One routine allows a watchpoint to be established on a processor *mode switch*:

```
optWatchpointP opProcessorModeWatchpointNew (
    optProcessorP  processor,
    void*          userData,
    optProcWatchpointConditionFn notifierCB
);
```

Finally, one routine allows a watchpoint to be established on a processor *exception*:

```
optWatchpointP opProcessorExceptionWatchpointNew (
    optProcessorP  processor,
    void*          userData,
    optProcWatchpointConditionFn notifierCB
);
```

Each function returns an `optWatchpointP` opaque type pointer for the watchpoint that was created. The `userData` argument allows a client-specific data pointer to be associated with the watchpoint object for later use (see below). A previously-created watchpoint can be deleted using:

```
void opWatchpointDelete (optWatchpointP watchpoint);
```

The `notifierCB` arguments to the watchpoint addition functions above allow a notifier callback function to be associated with each watchpoint that decides whether the watchpoint should be triggered or not (i.e., it allows the specification of *conditional* watchpoints).

For memory watchpoints, the prototype of the notifier is:

```
#define OP_ADDR_WATCHPOINT_CONDITION_FN(_name) \
Bool _name ( \
    optProcessorP  processor, \
    optWatchpointP watchpoint, \
    Addr           PA, \
    Addr           VA, \
    Uns32           bytes, \
    void*          userData, \
    void*          value)

typedef OP_ADDR_WATCHPOINT_CONDITION_FN ((* optAddrWatchpointConditionFn));
```

In this case, the notifier is passed the physical and virtual addresses of the memory access, the number of bytes being accessed and a pointer to a buffer containing those bytes.

For other watchpoint types, the prototype of the notifier is:

```
#define OP_PROC_WATCHPOINT_CONDITION_FN (_name) \
Bool _name ( \
    optWatchpointP watchpoint, \
    optProcessorP  processor, \
    void*          userData)

typedef OP_PROC_WATCHPOINT_CONDITION_FN ((* optProcWatchpointConditionFn));
```

In both cases, if the notifier is `NULL` or returns `False` then any processor triggering the watchpoint will stop before it executes its next instruction with `optStopReason OP_SR_WATCHPOINT`. Otherwise, if the notifier returns `True`, the triggering processor will not stop but instead continue executing normally.

19.9.2 Semantics of Physical and Virtual Watchpoints

The functions `opProcessorReadWatchpointNew`, `opProcessorWriteWatchpointNew` and `opProcessorAccessWatchpointNew` each take an argument `physical` which indicates whether the watch point should be physical or virtual. The semantics of these are as follows:

19.9.2.1 Physical Watchpoints

Physical watchpoints are created on the externally-connected processor bus. Creating a physical watchpoint is therefore equivalent to creating a bus watchpoint on the processor data bus.

When a physical memory watch point is set, it applies to the addressed physical memory *irrespective of the route by which that is accessed*. For example, if you set a physical watch point on address `0x10000`, the watch point will trigger if the processor is in a non-TLB mapped mode and accesses address `0x10000`, or if it is a TLB mapped mode where VA=`0x50000` (say) maps to `0x10000` and an access is made to VA=`0x50000`.

19.9.2.2 Virtual Watchpoints

When a virtual memory watch point is set, it applies to the *memory addressed by the virtual address range as viewed from the current processor mode*. As a contrived example:

1. Suppose that a processor is currently in TLB-mapped kernel mode, and that virtual address `0x50000` maps to physical address `0x10000`.
2. A watch point is set using `opProcessor*WatchPointNew` for *virtual* address VA=`0x50000`.

3. The watch point is triggered by any accesses to VA=0x50000 in TLB-mapped kernel mode (as expected), or any aliased access to PA=0x10000.
4. The mapping for VA=0x50000 in TLB-mapped kernel mode is changed to PA=0x20000.
5. The watch point is still triggered by any accesses to 0x50000 in TLB-mapped kernel mode (as expected). Note that the physical memory for the watch point has changed from 0x10000 to 0x20000. Accesses that change memory at PA=0x10000 by any route no longer trigger the watch point.
6. The processor enters TLB-mapped *user* mode. Say that in this mode VA=0x50000 is mapped to PA=0x60000 and VA=0x70000 is mapped to PA=0x20000.
7. The processor accesses VA=0x50000 in TLB-mapped *user* mode. The watch point does *not* trigger because VA=0x50000 maps to PA=0x60000, which does not correspond to VA=0x50000/PA=0x20000 in TLB-mapped *kernel* mode.
8. The processor accesses VA=0x70000 in TLB-mapped user mode. The watch point *triggers* because VA=0x70000 maps to PA=0x20000.

These semantics avoid much spurious watch point triggering when processors switch modes. When a user places a memory watch point at *virtual* address 0x20000, he almost always means virtual address 0x20000 *in the current mode*.

19.9.3 Watchpoint Attribute Query

There are various functions that allow watchpoint attributes to be queried. The type of a watchpoint can be found using:

```
typedef enum optWatchpointTypeE {
    OP_WP_MEM_READ    , // Memory read watchpoint
    OP_WP_MEM_WRITE   , // Memory write watchpoint
    OP_WP_MEM_ACCESS  , // Memory access watchpoint
    OP_WP_REG     ,     // Register watchpoint
    OP_WP_MODE        , // Mode change watchpoint
    OP_WP_EXCEPTION     // Exception watchpoint
} optWatchpointType;

optWatchpointType opWatchpointType (optWatchpointP watchpoint);
```

The client data pointer that was associated with the watchpoint when it was created can be found using:

```
void* opWatchpointUserData (optWatchpointP watchpoint);
```

For memory address range watchpoints, the bounding addresses can be found using:

```
Addr opWatchpointAddressLo (optWatchpointP watchpoint);
Addr opWatchpointAddressHi (optWatchpointP watchpoint);
```

These functions return zero for other watchpoint types. For processor register watchpoints, the register which is being watched can be found using:

```
optRegP opWatchpointReg (optWatchpointP watchpoint);
```

This function returns `NULL` for other watchpoint types or if the watchpoint has not been triggered.

For processor register watchpoints, there are query functions which return pointers to the current and previous value of the register being watched:

```
void* opWatchpointRegCurrentValue (optWatchpointP watchpoint);
void* opWatchpointRegPreviousValue (optWatchpointP watchpoint);
```

19.9.4 Handling Triggered Watchpoints

When a watchpoint triggers (because a read or write occurs to the address range over which it is sensitive, or because the processor register or mode it is watching changes, or an exception occurs), `opRootModuleSimulate` or `opProcessorSimulate` will return with the processor's `optStopReason` set to `OP_SR_WATCHPOINT`. Because watchpoints can be specified with overlapping ranges and on multiple registers and other events simultaneously, it is possible for *multiple watchpoints to be triggered by a single processor instruction.* To enable these all to be handled, an iterator function is available that returns the *next* triggered watchpoint or `NULL` if there are no currently triggered watchpoints:

```
optWatchpointP opRootModuleWatchpointNext (optModuleP module);
```

Once the returned watchpoint has been handled by the debugger, it must be reset using:

```
void opWatchpointReset (optWatchpointP watchp);
```

Then a subsequent call to `opRootModuleWatchpointNext` will return the next triggered watchpoint that has not been reset, and so on until all watchpoints have been handled by the client debugger and a `NULL` is returned. For each triggered watchpoint, the processor which triggered it can be found using:

```
optProcessorP opWatchpointTriggeredBy (optWatchpointP watchpoint);
```

19.10 Handling Simultaneous Debug Events

It is possible that execution of a single processor instruction could potentially cause an address breakpoint, an instruction count breakpoint and a watchpoint all to trigger. In this case, the priority order is as follows:

1. The instruction count breakpoint is triggered first, causing the processor to be stopped for `optStopReason` `OP_SR_BP_ICOUNT` before the instruction is executed;
2. When simulation is resumed by `opRootModuleSimulate` or `opProcessorSimulate`, the address breakpoint is triggered next, causing the processor to be stopped for `optStopReason` `OP_SR_BP_ADDRESS`, again before the instruction is executed;
3. When simulation is resumed by `opRootModuleSimulate` or `opProcessorSimulate`, the instruction completes. After completion, the processor is stopped for `optStopReason` `OP_SR_WATCHPOINT`, at which point the triggered watchpoints can be found and reset using `opRootModuleWatchpointNext` and `opWatchpointReset`.

19.11 Debugger Integration Examples

There are two examples using the address breakpoint, instruction count breakpoint and watchpoint constructs in the `debuggerIntegration` directory:

```
$IMPERAS_HOME/Examples/DebuggerIntegration/multiProcessor
$IMPERAS_HOME/Examples/DebuggerIntegration/modeAndException
```

The first example uses the two processor shared memory application first seen in the *iGen Platform and Module Creation User Guide.* We will add a custom harness to exercise the breakpoint and memory watchpoint debugger integration commands (see section 19.11.1).

The second example uses a simpler single processor assembler example to exercise the mode change and exception watchpoints (see section 19.11.2).

19.11.1 Multi Processor Debugger Integration Example

The following sections refer to the example which may be found in:

```
$IMPERAS_HOME/Examples/DebuggerIntegration/multiProcessor
```

19.11.1.1 Establishing Watchpoints

The simulation harness for this example is `harness.c` in the example `harness` directory.

Once the simulation module has been instantiated and the applications loaded into simulated memory, the platform establishes a write watchpoint on the first word of the shared memory between the two processors as follows:

```
applyWatchpoint(processor0, SHARED_LOW, 4, OP_WP_MEM_WRITE);
```

Function `applyWatchpoint` is as follows:

```
static void applyWatchpoint(
```

```
    optProcessorP proc,
    Addr          address,
    Addr          size,
    watchType     type
) {

    if (type == OP_WP_MEM_ACCESS) {

        // create access watchpoint
        optWatchpointP rwp = opProcessorAccessWatchpointNew(
            proc, False, address, address+size-1, (void *)(id++), 0
        );
        opPrintf(
            "%s: ACCESS watchpoint is %u\n", opObjectName(proc), getWatchpointId(rwp)
        );

    } else if (type == OP_WP_MEM_READ) {

        // create read watchpoint
        optWatchpointP rwp = opProcessorReadWatchpointNew(
            proc, False, address, address+size-1, (void *)(id++), 0
        );
        opPrintf(
            "%s: READ watchpoint is %u\n", opObjectName(proc), getWatchpointId(rwp)
        );

    } else  if (type == OP_WP_MEM_WRITE) {

        // create write watchpoint
        optWatchpointP wwp = opProcessorWriteWatchpointNew(
            proc, False, address, address+size-1, (void *)(id++), 0
        );
        opPrintf("%s: WRITE watchpoint is %u\n", opObjectName(proc),
getWatchpointId(wwp));

    }
}
```

The function adds access, read or write watchpoints for the specified address range, as specified by the `type` enum.

In our call, a write watchpoint is made on the first four bytes of shared memory which corresponds to the address specified by the `ENCRYPT_INDEX` macro in the application. Whenever this address is written control will be returned to the debugger.

In a real debugger, the address and object size would of course be found by the debugger from the object file, typically in response to a user command.

The platform also establishes some register watchpoints for `processor0` only using:

```
applyRegWatchpoints(processor0);
```

Function `applyRegWatchpoints` is as follows:

```
static void applyRegWatchpoints(optProcessorP processor) {

    optWatchpointP rwp1 = opProcessorRegWatchpointNew(
        processor, opProcessorRegByName(processor, "r3"), (void *)(id++), 0
    );

    optWatchpointP rwp2 = opProcessorRegWatchpointNew(
        processor, opProcessorRegByName(processor, "r9"), (void *)(id++), 0
    );

    optWatchpointP rwp3 = opProcessorRegWatchpointNew(
        processor, opProcessorRegByUsage(processor, OP_REG_SP), (void *)(id++), 0
    );

    opPrintf("REGISTER watchpoint 1 is %u\n", getWatchpointId(rwp1));
    opPrintf("REGISTER watchpoint 2 is %u\n", getWatchpointId(rwp2));
    opPrintf("REGISTER watchpoint 3 is %u\n", getWatchpointId(rwp3));

}
```

This function establishes register change watchpoints on three registers; two are found by name (`r3` and `r9`) and the third is found by usage (the OR1K stack register, `r1`). In a real debugger, the registers would of course be selected dynamically.

The platform also demonstrates how to query the registers by register group. Function `queryRegisters` lists all registers found on the processor, by group:

```
static void queryRegisters(optProcessorP processor) {

    opPrintf("%s REGISTERS\n", opObjectName(processor));

    optRegGroupP group = 0;

    while((group=opProcessorRegGroupNext(processor, group))) {

        opPrintf("  GROUP %s\n", opRegGroupName(group));
```

```
        optRegP reg = 0;

        while((reg=opRegGroupRegNext(processor, group, reg))) {
            opPrintf("    REGISTER %s\n", opRegName(reg));
        }
    }
}
```

19.11.1.2 Running the Simulator

The simulator is run in a loop which calls `opRootModuleSimulate`. There are two modes of operation: a normal mode (which runs to the next debug event or termination) and an instruction step mode (used to single-step past an address breakpoint):

```
    Bool         stepOver      = False;
    Bool         finished      = False;
    optProcessorP stopProcessor = 0;

    // Simulation loop
    while(!finished) {

        // Simulate the platform using the default scheduler
        // All breakpoints are set and cleared together, ala gdb behavior
        if(stepOver) {
            opProcessorBreakpointICountSet(stopProcessor, 1);
            stopProcessor = opRootModuleSimulate (root);
            stepOver = False;
        } else {
            applyBreakpoints(processor0, breakpoints0);
            applyBreakpoints(processor1, breakpoints1);
            stopProcessor = opRootModuleSimulate(root);
            clearBreakpoints(processor0, breakpoints0);
            clearBreakpoints(processor1, breakpoints1);
        }

        optStopReason sr = stopProcessor ? opProcessorStopReason(stopProcessor)
                                         : OP_SR_EXIT;
        … actions depending on stopReason here
    }
```

In the *single step* mode, an instruction count breakpoint is set for one instruction and then the platform is simulated:

```
            opProcessorBreakpointICountSet(stopProcessor, 1);
            stopProcessor = opRootModuleSimulate (root);
```

```
        stepOver = False;
```

In the *normal* mode, address breakpoints are established, the simulation is run until the next debug event or termination and then address breakpoints are removed:

```
        applyBreakpoints(processor0, breakpoints0);
        applyBreakpoints(processor1, breakpoints1);
        stopProcessor = opRootModuleSimulate(root);
        clearBreakpoints(processor0, breakpoints0);
        clearBreakpoints(processor1, breakpoints1);
```

The approach of resetting the breakpoints every time the debugger retains control used here is consistent with the approach taken by GDB.

Each processor has its own table of breakpoints which are specified in a NULL terminated static array:

```
// List of breakpoint addresses for processor 0 (encrypt)
const static Addr breakpoints0[] = {
   0xf2c,          // main
   0               // terminator
};

// List of breakpoint addresses for processor 1 (decrypt)
const static Addr breakpoints1[] = {
   0x0f2c,         // waitForFrame
   0x1000,         // main
   0               // terminator
};
```

The addresses of routines in the application were obtained by examining the objdump output for the application executable. If the application is modified these tables may need to be updated. Again, a real debugger would read these from the application ELF file and not rely on fixed addresses:

The routines to set and clear breakpoints are as follows:

```
static void applyBreakpoints(optProcessorP processor, const Addr breakpoints[]) {

   Uns32 i;

   for(i=0; breakpoints[i]; i++) {
      opProcessorBreakpointAddrSet(processor, breakpoints[i]);
   }
}

static void clearBreakpoints(optProcessorP processor, const Addr breakpoints[]) {
```

```
    Uns32 i;

    for(i=0; breakpoints[i]; i++) {
        opProcessorBreakpointAddrClear(processor, breakpoints[i]);
    }
}
```

Each time `opRootModuleSimulate` returns, the loop decides what to do next depending on the system state.

1. If the `optStopReason` was `OP_SR_EXIT` or `OP_SR_FINISH`, the simulation has terminated.

```
case OP_SR_EXIT:
case OP_SR_FINISH:
    finished = True;
    break;
```

2. Otherwise, if the `optStopReason` was `OP_SR_BP_ICOUNT` an instruction count breakpoint has been hit (the debugger is single-stepping over an address breakpoint location):

```
case OP_SR_BP_ICOUNT:
    opPrintf(
        "Processor %s icount %u stopped at icount\n",
         opObjectName (stopProcessor),
        (Uns32)opProcessorICount(stopProcessor)
    );
    break;
```

3. Otherwise, if the `optStopReason` was `OP_SR_BP_ADDRESS` an address breakpoint has been hit. In this case, the simulation switches mode to step for one instruction to get past the breakpoint address:

```
case OP_SR_BP_ADDRESS:
    opPrintf(
        "Processor %s icount %u breakpoint at address 0x%08x\n",
        opObjectName (stopProcessor),
        (Uns32)opProcessorICount(stopProcessor),
        (Uns32)opProcessorPC(stopProcessor)
    );
    stepOver = True;
    break;
```

4. Otherwise, if the `optStopReason` was `OP_SR_WATCHPOINT` a watchpoint has triggered. In this case, the triggered watchpoints are scanned and reported:

```
case OP_SR_WATCHPOINT:
    opPrintf(
        "Processor %s icount %u stopped at watchpoint\n",
         opObjectName (stopProcessor),
```

```
            (Uns32)opProcessorICount(stopProcessor)
        );
        handleWatchpoints(root);
        break;
```

5. Otherwise, the `optStopReason` is reported and simulation continues (no other `optStopReason` value is expected in this simulation).

```
    default:
        opPrintf(
            "Processor %s icount %u stopped with unexpected reason 0x%x (%s)\n",
            opObjectName (stopProcessor),
            (Uns32)opProcessorICount(stopProcessor),
            sr,
            opStopReasonString(sr)
        );
        break;
```

Function `handleWatchpoints` reports and resets all triggered watchpoints. The function iterates over all triggered but unhandled watchpoints, finding the watchpoint id and the processor that caused the watchpoint to trigger:

```
static void handleWatchpoints(optModuleP mi) {

    optWatchpointP wp;

    while((wp=opRootModuleWatchpointNext(mi))) {

        Uns32            id        = getWatchpointId(wp);
        optProcessorP    processor = opWatchpointTriggeredBy(wp);
        optWatchpointType type     = opWatchpointType(wp);
```

The `userData` associated with a watchpoint is used to record an arbitrary watchpoint id number:

```
static Uns32 getWatchpointId(optWatchpointP watchpoint) {
    return (UnsPS)opWatchpointUserData(watchpoint);
}
```

The watchpoint type is used to disambiguate the *register* and *address* watchpoint cases. If this is a register watchpoint, details about it are printed, together with the old and new values of the register:

```
        switch (type) {

            case OP_WP_REG: {
```

```
            // a register watchpoint was triggered
            optRegP reg       = opWatchpointReg(wp);
            Uns32  *newValueP = (Uns32 *)opWatchpointRegCurrentValue(wp);
            Uns32  *oldValueP = (Uns32 *)opWatchpointRegPreviousValue(wp);

            // indicate old and new value of the affected register
            opPrintf(
               "  Register watchpoint %u (processor %s:%s) triggered 0x%08x-
>0x%08x\n",
               id,
               opObjectName(processor),
               opRegName(reg),
               *oldValueP,
               *newValueP
            );
```

If register watchpoints have fired more than 100 times, any one that fires is deleted the next time it is triggered, otherwise it is reset:

```
            // delete watchpoint after 100 triggers
            if(regWatchPointCount++>100) {
               opWatchpointDelete(wp);
            } else {
               opWatchpointReset(wp);
            }

            break;
         }
```

(This behavior would not be required in a real debugger integration – it is done here simply so that the example output is not swamped by register change callback messages).

If the watchpoint is a memory read, write or access, information about the address range is printed and the watchpoint reset:

```
         case OP_WP_MEM_READ:
         case OP_WP_MEM_WRITE:
         case OP_WP_MEM_ACCESS: {

            // a memory watchpoint was triggered
            opPrintf(
               "  %s watchpoint %u (range 0x%08x:0x%08x) triggered by processor %s\n",
               (type==OP_WP_MEM_READ) ?   "Read" :
               (type==OP_WP_MEM_WRITE) ? "Write" :
                                                "Access",
               id,
```

```
            (Uns32)opWatchpointAddressLo(wp),
            (Uns32)opWatchpointAddressHi(wp),
            opObjectName(processor)
        );

        opWatchpointReset(wp);

        break;
    }
```

If the watchpoint type is not recognized then a message is issued and the watchpoint reset:

```
default: {

        opPrintf(
            "  unknown watchpoint type %u triggered by processor %s\n",
            type,
            opObjectName (processor)
        );

        opWatchpointReset(wp);

        break;
    }
```

Note that if a watchpoint is not reset then the next call to `opRootModuleWatchpointNext` will return the same watchpoint, resulting in an infinite loop.

19.11.1.3 Compiling and Running the Example

First, take a copy of the example:

```
> cp -r $IMPERAS_HOME/Examples/DebuggerIntegration/multiProcessor .
> cd multiProcessor
```

Build the test platform and application with the following commands:

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

To run the simulation:

```
> harness/harness.Linux32.exe --argv application/constitution.txt
```

Alternatively, an example.sh script (example.bat for Windows) has been provided which will do the build and run as a single command:

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

You should see the following output:

```
P0: WRITE watchpoint is 0
REGISTER watchpoint 1 is 1
REGISTER watchpoint 2 is 2
REGISTER watchpoint 3 is 3
P0 REGISTERS
  GROUP GPR
    REGISTER R0
    REGISTER R1
    REGISTER R2

    . . . many similar lines deleted . . .

    REGISTER R29
    REGISTER R30
    REGISTER R31
  GROUP System
    REGISTER PC
    REGISTER SR
    REGISTER EPCR
    REGISTER EEAR
    REGISTER ESR
    REGISTER PICMR
    REGISTER PICSR
    REGISTER TTCR
    REGISTER TTMR
  GROUP Integration_Support
    REGISTER EXCPT
Processor P1 icount 45 breakpoint at address 0x00001000
Processor P1 icount 46 stopped at icount
Processor P1 icount 90 breakpoint at address 0x00000f2c
Processor P1 icount 91 stopped at icount
Processor P0 icount 2 stopped at watchpoint
  Register watchpoint 1 (processor P0:R3) triggered 0xdeadbeef->0x00000000
Processor P0 icount 8 stopped at watchpoint
  Register watchpoint 2 (processor P0:R9) triggered 0xdeadbeef->0x00000000
Processor P0 icount 31 stopped at watchpoint

. . . many similar lines deleted . . .
```

```
  Register watchpoint 2 (processor P0:R9) triggered 0x000055f4->0x0000582c
Processor P0 icount 578 stopped at watchpoint
  Register watchpoint 3 (processor P0:R1) triggered 0xfffffe10->0xfffffe0c
Processor P0 icount 153361 stopped at watchpoint
  Write watchpoint 0 (range 0x11000000:0x11000003) triggered by processor P0

**** FRAME 0 ****

THE CONSTITUTION OF THE UNITED STATES OF AMERICA

Preamble
We the People of the United States, in Order to form a more perfect Union, establish
justice, insure domestic Tranquility, provide for the common defense, promote the

. . . many similar lines deleted . . .

No person shall be a Representative who shall not have attained to the age of twenty five
years, and been seven years a citizen of the United States, and who shall not, when
elected, be an inh
Processor P1 icount 374775 breakpoint at address 0x00000f2c
Processor P1 icount 374776 stopped at icount
Processor P0 icount 305408 stopped at watchpoint
  Write watchpoint 0 (range 0x11000000:0x11000003) triggered by processor P0
Processor P0 icount 457455 stopped at watchpoint
  Write watchpoint 0 (range 0x11000000:0x11000003) triggered by processor P0

**** FRAME 1 ****

abitant of that state in which he shall be chosen.

. . . many similar lines deleted . . .

Processor P1 icount 4523854 breakpoint at address 0x00000f2c
Processor P1 icount 4523855 stopped at icount

**** FRAME 25 ****

tution between the states so ratifying the same.

Done in convention by the unanimous consent of the states present the seventeenth day of
September in the year of our Lord one thousand seven hundred and eighty seven and of

. . . many similar lines deleted . . .


Virginia: John Blair, James Madison Jr.
```

```
North Carolina: Wm. Blount, Richd. Dobbs Spaight, Hu Williamson


Processor P1 icount 4703775 breakpoint at address 0x00000f2c
Processor P1 icount 4703776 stopped at icount

**** FRAME 26 ****

South Carolina: J. Rutledge, Charles Cotesworth Pinckney, Charles Pinckney, Pierce
Butler

Georgia: William Few, Abr Baldwin
```

The example first shows the result of the register group iterator and the by-group register iterator: there are two groups (`GPR` and `System`) containing the OR1K GPRs and system registers, respectively.

Each address breakpoint that is encountered is reported with lines of this form:

```
Processor P1 icount 45 breakpoint at address 0x00001000
```

Instruction count breakpoints are reported with lines of this form:

```
Processor P1 icount 46 stopped at icount
```

Register watchpoints are reported by a pair of lines of this form, giving the old and new values of the affected register:

```
Processor P0 icount 2 stopped at watchpoint
  Register watchpoint 1 (processor P0:R3) triggered 0xdeadbeef->0x00000000
```

Memory watchpoints are reported by a pair of lines of this form:

```
Processor P0 icount 153361 stopped at watchpoint
  Write watchpoint 0 (range 0x11000000:0x11000003) triggered by processor P0
```

19.11.2 Mode and Exception Debugger Integration Example

The following sections refer to the example which may be found in:

```
$IMPERAS_HOME/Examples/DebuggerIntegration/modeAndException
```

19.11.2.1 Establishing Watchpoints

The simulation harness for this example is `harness.c` in the example `harness` directory.

The harness has a similar structure to the previous example, `multiProcessor`, but instances only a single processor and adds watchpoints for mode changes and exceptions.

Once processor memory has been loaded, the platform establishes processor *mode change* and *exception* watchpoints as follows:

```
    // Apply watchpoints on mode switches and exceptions
    applyWatchpoint(processor, 0, 0, OP_WP_MODE);
    applyWatchpoint(processor, 0, 0, OP_WP_EXCEPTION);
```

Function `applyWatchpoint` has been enhanced from the version in the previous example to add mode and exception watchpoints as follows:

```
    switch(type) {
...
    case (OP_WP_MODE):
        wp = opProcessorModeWatchpointNew(
            proc, (void *)(id++), 0
        );
        typeName = "MODE";
        break;

    case (OP_WP_EXCEPTION):
        wp = opProcessorExceptionWatchpointNew(
            proc, (void *)(id++), 0
        );
        typeName = "EXCEPTION";
        break;
...
    }

    opPrintf("%s: %s watchpoint is %u\n", opObjectName(proc), typeName,
getWatchpointId(wp));
```

19.11.2.2 Running the Simulator

The simulator loop is similar to that in `multiProcessor`. The only significant difference is in function `handleWatchpoints`. The function once more iterates over all triggered but unhandled watchpoints, finding the watchpoint id and the processor that caused the watchpoint to trigger:

```
static void handleWatchpoints(optModuleP mi) {

    optWatchpointP wp;

    while((wp=opRootModuleWatchpointNext(mi))) {
```

```
    Uns32            id        = getWatchpointId(wp);
    optProcessorP    processor = opWatchpointTriggeredBy(wp);
    optWatchpointType type     = opWatchpointType(wp);
```

In this platform, it uses the watchpoint type to disambiguate the *mode* and *exception* watchpoint cases. If this is a mode change watchpoint, details about it are printed, together with the mode's code and name, and the watchpoint is reset:

```
    switch(type) {

      case OP_WP_MODE: {

        // a mode switch watchpoint was triggered
        optModeP    mode     = opProcessorModeCurrent(processor);
        Uns32       modeCode = mode ? opModeCode(mode) : 0;
        const char *modeName = mode ? opModeName(mode) : "none";

        // report new mode
        opPrintf(
          "  watchpoint %u (processor %s:MODE) triggered mode -> %d (%s)\n",
          id,
          opObjectName(processor),
          modeCode,
          modeName ?: ""
        );

        opWatchpointReset(wp);

        break;
      }
```

If this is an exception watchpoint, information about the exception type is printed and the watchpoint reset.:

```
      case OP_WP_EXCEPTION: {

        // an exception watchpoint was triggered
        optExceptionP except = opProcessorExceptionCurrent(processor);
        Uns32       exceptCode = except ? opExceptionCode(except) : 0;
        const char *exceptName = except ? opExceptionName(except) : "none";
        const char *exceptDesc = except ? opExceptionDescription(except) : "";

        // report current exception
        opPrintf(
```

```
                "  watchpoint %u (processor %s:EXCEPTION) triggered exception -> %d
%s: %s)\n",
                id,
                opObjectName(processor),
                exceptCode,
                exceptName ?: "",
                exceptDesc ?: ""
            );

            opWatchpointReset(wp);

            break;
        }
```

19.11.2.3 Compiling and Running the Example

First, take a copy of the example:

```
> cp -r $IMPERAS_HOME/Examples/DebuggerIntegration/multiProcessor .
> cd multiProcessor
```

Build the test platform and application with the following commands:

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

To run the simulation:

```
> harness/harness.Linux32.exe --argv application/constitution.txt
```

Alternatively, an example.sh script (example.bat for Windows) has been provided which will do the build and run as a single command:

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

You should see the following output:

```
cpu1: MODE watchpoint is 0
cpu1: EXCEPTION watchpoint is 1
Processor cpu1 icount 26 stopped at watchpoint
  watchpoint 0 (processor cpu1:MODE) triggered mode -> 1 (USER)
Processor cpu1 icount 29 stopped at watchpoint
  watchpoint 0 (processor cpu1:MODE) triggered mode -> 0 (SUPERVISOR)
  watchpoint 1 (processor cpu1:EXCEPTION) triggered exception -> 1792 ILL: Illegal
instruction)
Processor cpu1 icount 36 stopped at watchpoint
```

```
  watchpoint 0 (processor cpu1:MODE) triggered mode -> 1 (USER)
Processor cpu1 icount 38 stopped at watchpoint
  watchpoint 0 (processor cpu1:MODE) triggered mode -> 0 (SUPERVISOR)
  watchpoint 1 (processor cpu1:EXCEPTION) triggered exception -> 1792 ILL: Illegal
instruction)
Processor cpu1 icount 45 stopped at watchpoint
  watchpoint 0 (processor cpu1:MODE) triggered mode -> 1 (USER)
Processor cpu1 icount 48 stopped at watchpoint
  watchpoint 0 (processor cpu1:MODE) triggered mode -> 0 (SUPERVISOR)
  watchpoint 1 (processor cpu1:EXCEPTION) triggered exception -> 1792 ILL: Illegal
instruction)
Processor cpu1 icount 55 stopped at watchpoint
  watchpoint 0 (processor cpu1:MODE) triggered mode -> 1 (USER)
```

Each mode change watchpoint that is encountered is reported with lines of this form:

```
Processor cpu1 icount 26 stopped at watchpoint
  watchpoint 0 (processor cpu1:MODE) triggered mode -> 1 (USER)
```

Exception watchpoints are reported with lines of this form:

```
Processor cpu1 icount 29 stopped at watchpoint
  watchpoint 0 (processor cpu1:MODE) triggered mode -> 0 (SUPERVISOR)
  watchpoint 1 (processor cpu1:EXCEPTION) triggered exception -> 1792 ILL: Illegal
instruction)
```

Note that exceptions that occur in USER mode cause both an exception and mode watchpoint trigger at the same time, as the exception causes a switch from user to supervisor mode.

19.12 Scheduler Notification

Without changing the platform or the scheduler, an integrated debugger can be notified when significant actions occur that might require debugger intervention. This is useful when calls to `opRootModuleSimulate` or `opProcessorSimulate` are made in code that is not accessible to the debugger - for example from a SystemC platform.

Use the function `opSessionDebuggerNotifiersAdd` to install callbacks on these actions. This function accepts a pointer to the root module and a struct containing pointers to callbacks for specific phases of simulation:

```
void opSessionDebuggerNotifiersAdd (
    optModuleP            root,
    optDebuggerNotifiersP notifiers
);
```

The `optDebuggerNotifiers` structure is defined as:

```
typedef struct optDebuggerNotifiersS {

    // Callback into a debugger at the start of simulation
    optSimulateStartFn          start;

    // Callback into a debugger before PSE constructors have run
    optSimulateStartFn          endPeripheralCons;

    // Callback into a debugger after platform constructor has run
    optSimulateStartFn          endPlatformCons;

    // Callback into a debugger after all constructors have run
    optSimulateStartFn          endCons;

    // Callback into a debugger after processor has executed a slice
    optSimulatePostProcessorFn postProcessor;

    // Callback into a debugger after peripheral has completed a callback or thread
    optSimulatePostPeriphFn     postPeripheral;

    // Callback into a debugger when time advances
    optSimulateTimeAdvanceFn    advance;

    // Callback into a debugger at end of simulation
    optSimulateEndFn            end;

    // User data, passed to each callback
    void*                       userData;

} optDebuggerNotifiers;
```

Macros are provided that define each of the function prototypes for the callbacks:

```
#define OP_SIMULATE_START_FN(_name) \
void _name ( \
    optModuleP module, \
    void*      userData)
typedef OP_SIMULATE_START_FN((*optSimulateStartFn));

#define OP_SIMULATE_POST_PROCESSOR_FN(_name) \
Bool _name ( \
    optProcessorP processor, \
    void*         userData) typedef
OP_SIMULATE_POST_PROCESSOR_FN((*optSimulatePostProcessorFn));
```

```
#define OP_SIMULATE_POST_PERIPH_FN(_name) \
Bool _name ( \
    optPeripheralP peripheral, \
    void*          userData) typedef
OP_SIMULATE_POST_PERIPH_FN((*optSimulatePostPeriphFn));

/// opSessionDebuggerNotifiersAdd
#define OP_SIMULATE_END_FN(_name) \
void _name ( \
    optProcessorP processor, \
    void*         userData)
 typedef OP_SIMULATE_END_FN((*optSimulateEndFn));

#define OP_SIMULATE_TIME_ADVANCE_FN(_name) \
Bool _name ( \
    optAdvanceTimeInfo info, \
    void*              userData)
typedef OP_SIMULATE_TIME_ADVANCE_FN((*optSimulateTimeAdvanceFn));
```

A typical use of these callbacks might look like (note that not all callbacks need be used - unneeded ones may be set to 0):

```
static OP_SIMULATE_START_FN (startSim) {
    // called once:
    //   Before at the start of simulation
    ...
}

static OP_SIMULATE_START_FN (endConstructors) {
    //  called once
    //    after all constructors have run
    ...
}


static OP_SIMULATE_POST_PROCESSOR_FN (debugProc) {
    // called after processor has executed a slice
    optStopReason reason = opProcessorStopReason(processor);
    if (debuggerNeedsToActOnThisReason(reason)) {
        ...
    }
    if(debuggerWantsToFinish()) {
        return False;
    } else {
        return True;
```

```
    }
}

static OP_SIMULATE_TIME_ADVANCE_FN (advanceTime) {
    // called when (and only when) time is advanced
    ...
    return True;
}

static OP_SIMULATE_END_FN (finishSim) {
    // called when no more instructions to execute, but before destruction.
    ...
}

int main(...) {

    optDebuggerNotifiers notify = {
        .start     = startSim,
        .endPeripheralCons = 0,
        .endPlatformCons = 0,
        .endCons   = endConstructors,
        .postProcessor  = debugProc,
        .postPeripheral  = 0,
        .advance   = advanceTime,
        .finish    = finishSim,
        .userData  = myPointer
    };

    // Load the design
    optModuleP root = opRootModuleNew(0, 0, 0);

    // request callbacks.
    opSessionDebuggerNotifiersAdd (root, &notify);

    ...

}
```

`opSessionDebuggerNotifiersAdd` must be called during the construction phase after `opRootModuleNew` is called. There will be one call to `startSim`, then a call to `debugProc` each time a processor core stops executing. This might be because the simulator has executed all the instructions requested of this processor or it might be that a breakpoint, watchpoint or other simulator event has occurred. `debugProc` will be called if either `opRootModuleSimulate(root)` or `opProcessorSimulate(processor)` are used. If the processor has multiple cores, there will be callbacks for each core.

The function `debugProc` should return `True` if the simulation can continue after the callback or `False` if the simulation should finish, in which case end of simulation events will be triggered but no more instructions will be simulated.

The function `finishSim` will be called once, before the platform is destroyed.

The function `postPeripheral` is required only if you wish to debug PSE code. Leave the callback pointer null if not required. It should return `True` if the simulation can continue after the callback or `False` if the simulation should finish.

The function `advanceTime` will be called when the simulator moves simulated time forwards. It should return `True` if the simulation can continue after the callback or `False` if the simulation should finish.

The `userData` value set in the `optDebuggerNotifiers` struct will be passed to every callback.

##