OVPsim and Imperas CpuManager User Guide

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

Author:	Imperas Software Limited
Version:	2.3.15
Filename:	OVPsim_and_CpuManager_User_Guide.doc
Project:	OVPsim and CpuManager User Guide
Last Saved:	Tuesday, 04 August 2015
Keywords:	ICM API

Table of Contents

1 Introduction

This document describes use of the *Innovative CpuManager Interface* (ICM) API to implement simulation models of platforms containing any number of processor models communicating using shared memory. Platforms created using the ICM interface can be simulated using either the freely-available OVPsim simulation environment or the Imperas commercial CpuManager product.

1.1 What are CpuManager and OVPsim?

CpuManager and OVPsim are dynamic linked libraries (`.so` suffix on Linux, `.dll` suffix on Windows) implementing Imperas simulation technology. The shared objects contain implementations of the ICM interface functions described later in this document. These ICM functions enable instantiation, interconnection and simulation of complex multiprocessor platforms containing arbitrary shared memory topologies.

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

CpuManager is the commercial product available from Imperas. OVPsim is the freely-available (for Non-Commercial usage) version of this product. Which one to use is determined at runtime by the IMPERAS_RUNTIME environment variable. If it is not set or is set to OVPsim then the OVPsim library (which requires an OVP license) is dynamically linked at runtime. If it is set to CpuManager then the CpuManager library (which requires an Imperas license) will be used. To see the differences, refer to section 31.

A C++ version of the ICM API is available as source, which can be compiled using your own C++ compiler. Is has been tested with g++ (4.0.3) and Microsoft Visual C++ (MSCV 2015 Express). This is located in
`Imperas/ImpPublic/include/host/icm/icmCpuManager.hpp`

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

1.2 Use of ICM with Imperas tools

A program using ICM can be linked with the ICM RuntimeLoader to perform runtime dynamic loading of either the CpuManager or OVPsim dynamic linked libraries, to produce a stand-alone executable. Alternatively, it can be linked to create a dynamic link library, which can itself be loaded into the Imperas simulator (`imperas.exe`) or the

Imperas debug and verification environment. The Imperas simulator provides many benefits in addition to the basic OVP features, including:

- Multiprocessor debug – the simulator's extended GDB control allows simultaneous debug of all processor and PSE models in the platform, including processor specific stepping and breakpoints, temporary freezing of selected processors, and all the usual features of GDB.
- External control of simulation features. Even if you can't recompile the platform, the Imperas simulator can turn on tracing, diagnostics and other simulation features, many during a simulation run.
- Loading of extension libraries for analysis and verification. Even if you can't recompile the platform, the Imperas simulator can load additional intercept objects from its command line.

1.3 Compiling Examples Described in this Document

This documentation is supported by C code samples in an `Examples` directory, available either to download from the www.ovpworld.org website or as part of an Imperas installation.

GCC Compiler Versions

Linux32	4.5.2	i686-nptl-linux-gnu (Crosstool-ng)
Linux64	4.4.3	x86_64-unknown-linux-gnu (Crosstool-ng)
Windows32	4.4.7	mingw-w32-bin_i686-mingw
Windows64	4.4.7	mingw-w64-bin_i686-mingw

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

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

2 Imperas ICM Header Files

The ICM API, used by both CpuManager and OVPsim, is defined by several header files within the Imperas tool release tree or freely-available download from www.ovpworld.org :

Common Definitions

Standard types `Imperas/ImpPublic/include/impTypes.h`

ICM API Definitions

Formatted output & C API function
`Imperas/ImpPublic/include/host/icm/icmCpuManager.h`

For clarity this file now includes several headers which can be included according to the functionality required:

`icmConstruct.h`	Platform construction.
`icmDebugger.h`	Functions for use by an integrated debugger.
`icmDestruct.h`	Functions for clean up after a simulation.
`icmObjectfiles.h`	Reading and loading application processor object files.
`icmQuery.h`	Interrogation and exploration of an existing platform.
`icmRuntime.h`	Interacting with the platform during simulation.
`icmSimulatorSession.h`	Changing the way the simulator runs.
`icmText.h`	Writing text to the output stream, redirection of text.
`icmTrace.h`	Instruction tracing.
`icmTypes.h`	Types required by all icm functions.
`icmVersion.h`	The ICM interface version.

C++ API functions
`Imperas/ImpPublic/include/host/icm/icmCpuManager.hpp`

TLM2.0 Interfaces

TLM2.0 generic
`Imperas/ImperasLib/source/ovpworld.org/modelSupport/<>/1.0/tlm2.0/*.h`
TLM interface
`Imperas/ImperasLib/source/<v>/<l>/<n>/<v>/tlm2.0/*.hpp`

`<v>/<l>/<n>/<v>` are the vendor, library, name and version of the model, and are the same references used to locate the model in the library.

3 Simple Single-Processor Platforms

A simple program can be made that runs a single-processor platform using just five calls from the ICM API:

- **`icmInitPlatform`**
  `icmInitPlatform` initializes the simulation environment prior to a simulation run: it should be the first ICM routine called in any application. It names the platform, specifies attributes to control some aspects of the simulation to be performed, and also specifies how a debugger should be connected to the application if required.
- **`icmNewProcessor`**
  `icmNewProcessor` is used to create a new processor instance.
- **`icmLoadProcessorMemory`**
  Once a processor has been instantiated by `icmNewProcessor`, this routine is used to load an object file into the processor memory. Currently accepted formats are ELF and TI-COFF.
- **`icmSimulatePlatform`**
  `icmSimulatePlatform` is used to run simulation of the processor and program, for a specified duration.
- **`icmTerminate`**
  `icmTerminate` **must** be called at the end of simulation. It's function is to:
  - Free memory.
  - Return any licenses to the license server.
  - Ensure all tools have finished writing their results files.
  - Print the simulation statistics.

  CpuManager installs an 'atexit' handler which checks that icmTerminate has been called but does not call it automatically.

Later sections in this document will describe the arguments to these functions in more detail.

3.1 Simple Single-Processor Example

A simple single-processor platform example is available in the directory:

```
$IMPERAS_HOME/Examples/Platforms/simple
```

This uses the freely-available OR1K processor (see http://www.opencores.org/projects.cgi/web/or1k/architecture).

The following sections describe the main operations being performed

3.1.1 Initialization

The simulator is initialized by calling `icmInitPlatform`:

```
    icmInitPlatform(ICM_VERSION, 0, 0, 0, “platform”);
```

This function takes five arguments. The first argument is the version string from the ICM header files, which should be supplied as shown, `simAttrs`, is a bitmask controlling aspects of simulation behavior (for example, whether to emit verbose output giving simulated MIPS rate). Arguments 3 & 4 are used when processor debug is required, discussed in a later section. Argument 5 is an optional name for the platform.

3.1.2 Selecting Models from the Library

The components, including processors, peripherals and memories, are supplied in a Vendor, Library, Name and Version (VLNV) format library structure. An API function `icmGetVlnvString` is available to help construct the path to access these models.

```
    // select library components
    const char *vlnvRoot = 0; // when null, use the default library
    const char *model = icmGetVlnvString(
        vlnvRoot, "ovpworld.org", "processor", "or1k", "1.0", "model"
    );
    const char *semihosting = icmGetVlnvString(
        vlnvRoot, "ovpworld.org", "modelSupport", "imperasExit", "1.0", "model"
    );
```

With the first argument set to NULL the default Imperas library is used. This is specified by the environment variable IMPERAS_VLNV and by default would point to $IMPERAS_HOME/lib/$IMPERAS_ARCH/ImperasLib

3.1.3 Creation of a Model Instance

A single instance of a processor is defined by calling `icmNewProcessor`:

```
    icmProcessorP processor = icmNewProcessor(
        "cpu1",             // CPU name
        “or1k”,             // CPU type
        0,                  // CPU cpuId
        0,                  // CPU model flags
        32,                 // address bits
        model,              // model file
        0,                  // not used
        0,                  // enable tracing etc
        0,                  // user-defined attributes
        semihosting,        // semi-hosting file
        0                   // not used
    );
```

The arguments to `icmNewProcessor` are as follows:

- `name`: this is an instance name to give the instance, which must be unique in the design.
- `type`: this is a type name for the instance, in this case specified as `“or1k”` in the makefile.

- `cpuId`: every processor has an id number, specified by this argument.
- `cpuFlags`: this is a bitmask that can be accessed from within the processor model to change its behavior (for example, to turn on debug modes). In normal usage, pass 0.
- `addressBits`: this specifies the default data and instruction bus widths for the model (typically 32, though ICM supports addresses up to 64 bits wide).
- `modelFile`: this is the path to the dynamic load library (`.dll` or `.so`) implementing the processor model. The file extension can be ignored. If the path is a directory rather than a file, the file model.so or model.dll is assumed. For this example, the path is specified in the Makefile as follows (relative to the Imperas installation):
  `ImperasLib/ovpworld.org/processor/or1k/1.0/model.dll`
- (unused argument)
- `procAttrs`: this is a bitmask controlling some aspects of processor behavior. The options available here will be covered in later sections of this document.
- `userAttrs`: this argument specifies a list of application-specific attributes for the processor. In this example, the instance has no attributes.
- `semiHostFile` specify the *semihosting library* for the processor instance; this is described in the next subsection.
- Unused argument

If a platform has many processors, it is sometimes convenient to record user-data on the processor instance. This data can then be retrieved.

`icmNewProcessorWithHandle` is a variant of `icmNewProcessor` with and extra argument 'handle'
`icmGetProcessorHandle` returns the handle.

3.1.3.1 Defining SemiHosting

Imperas *semihosting* allows the default behavior of specified functions or instructions to be modified using a semihosting shared object library that is loaded by the simulator in addition to the processor model. In this case, we have defined a global label, `exit`, on the last instruction of the assembler test:

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

This label can be used in conjunction with a standard Imperas semihosting shared object library, located at the following location in the Imperas installation:

```
ImperasLib/ovpworld.org/modelSupport/imperasExit/1.0/model.dll
```

.

This semihosting library terminates simulation immediately after any instruction labeled `exit`. To use the semihosting library, `platform/platform.c` includes the semihosting object file name, specified by `semihosting`, defined in the platform makefile.

`semihosting` refers to the name of the .so or .dll file implementing the semihosting.

> ⇒ This simple example makes no specific mention of any processor memory configuration, other than to say that the processor address bus width is 32 bits. In the absence of any other specific information about memory configuration, CpuManager / OVPsim will create a single fully-populated RAM memory attached to both the processor data and instruction busses.

3.1.4 Including the Command Line Parser

By including the Command Line Parser access is given to all the standard platform arguments.

the commonly used arguments for this example are
--*program* to specify the program elf file to load
--*gdbconsole* to start debugging the application running on the processor

This is the same set of arguments that are available using the Control File. Please see the document 'OVP_Control_File_User_Guide' for further information.

```
static Bool cmdParser(int argc, const char *argv[]);

int main(int argc, const char *argv[])
{
    // Check arguments and ensure application to load specified
    if(!cmdParser(argc, argv)) {
        icmMessage("E", "platform", "Command Line parser error");
        icmExitSimulation(1);
        return 1;
    }
...
```

```
    static Bool cmdParser(int argc, const char *argv[]) {
        icmCLPP parser = icmCLParser("platform", ICM_AC_ALL);
    char message[1024];
    sprintf(message, "Basic Usage\n  platform.IMPERAS_ARCH.exe
                                  --program <elf file>\n");
        icmCLParserUsageMessage(parser, (const char *)message);

        Bool ok = icmCLParseArgs(parser, argc, argv);

        if (!icmCLParseArgUsed (parser,"program")) {
                icmMessage("E", "program",
            "Argument '--program' must be used to specify application elf file to load");
                ok = False;
        }
        return ok;
}
```

As well as adding the standard set of command line arguments, it is possible for the user to add their own, for example the following shows the addition of a new Boolean type of argument that can be used in the platform to change some operation.

```
    icmCLParserAdd(parser, "enable", 0 , 0, "user platform config", ICM_AT_BOOLVAL,
                            &options.enable, "enable my option", 0x0, 0, 1);
```

3.1.5 Loading the Application Executable

Once a processor instance has been created, an object file can be loaded into the processor memory using `icmLoadProcessorMemory`:

```
    icmLoadProcessorMemory(processor, argv[1], ICM_LOAD_DEFAULT, False, True);
```

The first argument is the processor for which to load memory.

The second argument is the application object file name. In this example the application file name is passed as the first argument to the program when the platform is run. note that ELF and TI COFF format files are accepted.

The third argument is an enumerated type allowing control over how the program is loaded. This allows:

a) The use of physical rather than virtual addresses when loading.
b) Verbose output that reports each section loaded.
c) The zeroing of the BSS section.
d) Set the processor initial PC to the entry address of the file.
e) Only load the symbols and do not modify memory.

See the definition of the icmLoaderAttrs type in impTypes.h for the values to be used.

The fourth argument enables verbose output showing the location of sections in the loaded object file. This argument overrides the setting of argument 3.

The fifth argument specifies whether the processor should start execution from the start address specified in the object file (if `True`) or whether it should start at the model-specific boot address (if `False`). This argument overrides the setting of argument 3.

⇒ NOTE: This example uses the command line parser which allows the program to loaded to be specified on the command line using the --*program* argument. This performs the same operation as the *icmLoadProcessorMemory()* function shown above.

3.1.6 Running the Simulation

Once the processor has been instantiated and an application program loaded, the program can be simulated to completion using:

```
    icmSimulatePlatform();
```

This routine simulates the entire platform using the CpuManager / OVPsim default scheduler, which (for multiprocessor platforms) runs each processor for a number of instructions in a time slice before advancing time to run the next time slice.

There is a also a routine available, `icmSimulate`, which can be used to simulate a specific processor for a precise number of instructions. This second function is useful in situations when CpuManager / OVPsim is being used as a subsystem of a larger simulation implemented in another environment (for example, SystemC).

3.1.7 Clean Up

Finally, `icmTerminate` is used to clean up simulation data structures and delete all simulation objects created since the previous `icmInitPlatform` call.

⇒ Why should `icmTerminate` always be called at the end of simulation?
Some platform components may perform significant actions in their destructors. For example, processor models may have a mode in which they print processor-specific information about the simulation run (e.g. whether certain execution units were used or not). If you don't call `icmTerminate` then these actions will not be performed.

3.2 Text Output

Use `icmPrintf` and `icmMessage` to communicate with the user of the platform.
`icmPrintf (const char *format, ...)` is similar to the C library function `printf`, but sends its output to the simulator output stream and log file (if one is active).

`icmMessage(const char *severity, const char *prefix, const char *format, ...)` also sends its output to the simulator output stream and log file (if one is active). The `severity` string should be one of:

- "I" for information only.
- "W" warning of a non-fatal problem.
- "E" An error has occurred.
- "F" A fatal error has occurred (this value will terminate the session).

The `prefix` string should be a unique, short code associated with this platform e.g. "MY_CCT". Prefix strings make post-processing the log file easier.

3.3 Running the Example

Take a copy of the example:

```
cp –r $IMPERAS_HOME/Examples/Platforms/simple .
```

3.3.1 Compiling the CpuManager Platform

The test platform can be compiled to produce an executable, `platform.${IMPERAS_ARCH}.exe`, by using this command in the `platform` directory:

```
make –C platform
```

3.3.2 Creating an Executable

A test case must be created using the processor tool chain. Because the OR1K processor is supported by Imperas tools and shipped as an example, there is already an encapsulated tool chain that you can use to compile test cases for it.

Within the `platform` directory is a simple assembler test, `application/asmtest.c`, which simply performs a few instructions and exits. The application can be compiled using the following command in the `platform` directory:

```
make –C application
```

The result is an ELF format file for the OR1K called `asmtest.OR1K.elf`.

⇒ The target `make all` is also present in the example makefile that creates both the application and the platform in a single step.

3.3.3 Running the Simulation

Having compiled the test platform and application, you are now ready to run a simulation. Do this by running the following in the `platform` directory:

```
Platform/platform.<ARCH[1]>.exe --program application/asmtest.OR1K.elf
```

⇒ The environment variable `IMPERAS_RUNTIME` determines the simulation library (OVPsim or CpuManager) that is used at runtime. If not specified, it will to default to OVPsim. To use CpuManager instead (which requires a license for the Imperas Professional Tools) do:|

```
export IMPERAS_RUNTIME=CpuManager
```

You should see the following output:

```
Processor ‘cpu1’ terminated at ‘exit’, address 0x10000bc
```

This message is printed by the *imperasExit* semihosting library as the processor executes the first instruction at `exit` in the application.

[1] ARCH is the Host machine architecture, for example Windows32, Linux32

3.3.4 Dynamic link library

Also built is a dynamic link library, a .dll or .so version of the platform for use in the Imperas simulator (see section 1)

The dynamic library can now be simulated in the Imperas simulator:

```
linux> imperas.exe --icmobject platform/model.so
```

If your platform has a `main()` which accepts arguments e.g.:

```
int main(char *argv[], int argc) {
    char *executable = argv[1];
    char *options    = argv[2];
    …
}
```

The Imperas simulator can pass arguments to the platform:

```
linux> imperas.exe --icmobject platform/model.so  \
                   --icmargv --program application/asmtest.OR1K.elf  option1 option2
```

See the *Imperas Simulation Guide* for more details.

4 Simulation Options with ICM Attributes

Predefined attributes can be applied to the platform using the `simAttrs` argument to `icmInitPlatform`() function. This is a bitwise-or of any of the following:

Attribute	Definition
`ICM_STOP_ON_CTRLC`	Install a Ctrl-C signal handler
`ICM_NO_OPTIMIZE`	Turn off aggressive JIT code optimization.
`ICM_GDB_CONSOLE`	Start a gdb console
`ICM_MPD_CONSOLE`	Start an MPD console
`ICM_SUPPRESS_BANNER`	Suppress the banner
`ICM_VERBOSE`	Output more information
`ICM_ENABLE_IMPERAS_INTERCEPTS`	Intercept special Imperas functions.
`ICM_WALLCLOCK`	Prevent the simulator running faster than real-time when it is inactive.
`ICM_NO_RSP_WAIT`	Do not wait for an debugger connection.
`ICM_PRINT_USER_ATTRIBUTES`	Print list of model defined attributes then exit
`ICM_PRINT_COMMANDS`	Print a list of model commands then exit

Predefined attributes can be applied to a processor model when it is instantiated, using the `procAttrs` argument to `icmNewProcessor`(). This a bitwise-or of any of the following:

Attribute	Definition
`ICM_ATTR_TRACE`	enable instruction tracing
`ICM_ATTR_TRACE_ICOUNT`	print instruction count with trace
`ICM_ATTR_TRACE_CHANGE`	write changed registers with trace
`ICM_ATTR_TRACE_REGS_BEFORE`	dump registers with trace before execution
`ICM_ATTR_TRACE_REGS_AFTER`	dump registers with trace after execution
`ICM_ATTR_TRACE_BUFFER`	maintain 256-instruction trace buffer
`ICM_ATTR_SIMEX`	simulate exceptions
`ICM_ATTR_FETCH_VALIDATE`	validate the address of each instruction fetch (processor model validation)
`ICM_ATTR_NOTRACE_ANNUL`	don't trace annulled instructions.

In this section, the attributes controlling instruction tracing will be covered. Details of other attributes are given later in this document.

4.1 Model Tracing Operations

Using the `ICM_ATTR_TRACE` attribute enables instruction-by-instruction tracing for that processor instance using the disassembler built in to the processor model.

The `ICM_ATTR_TRACE_CHANGE` attribute writes the value of all modified registers when tracing is enabled by `ICM_ATTR_TRACE`. Changed values are detected by maintaining a record of all values readable using the register access API (see `icmGetNextReg` and related functions) at the completion of every instruction.

The `ICM_ATTR_TRACE_REGS_BEFORE` attribute dumps the current processor register state, again using the model-specific register dump format, when tracing is enabled by `ICM_ATTR_TRACE`. The order of events for each instruction is:

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

The `ICM_ATTR_TRACE_REGS_AFTER` attribute dumps the current processor register state, again using the model-specific register dump format, when tracing is enabled by `ICM_ATTR_TRACE`. The order of events for each instruction is:

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

4.1.1 Example: Simulation Tracing

The test platform for this example is in the `tracing` directory.

```
$IMPERAS_HOME/Examples/Platforms/tracing
```

File `platform/platform.c` has been changed as follows:

```
#define SIM_FLAGS (ICM_ATTR_TRACE | ICM_ATTR_TRACE_REGS_AFTER)

// create a processor
icmProcessorP processor = icmNewProcessor(
    "cpu1",             // CPU name
    “or1k”,             // CPU type
    0,                  // CPU cpuId
    0,                  // CPU model flags
    32,                 // address bits
    model,              // model file
    0,                  // not used
    SIM_FLAGS,          // enable tracing of register values
    0,                  // user-defined attributes
    semihosting,        // semi-hosting file
    0                   // not used
);
```

We have added the value `ICM_ATTR_TRACE and ICM_ATTR_TRACE_REGS` to the procAttrs argument of `icmNewProcessor`. This enables dumping of the processor state and registers before each instruction is executed.

⇒ Note that in a multiprocessor system, processor instances do not all have to have the same attributes. This means that you can enable tracing only for specific processors, for example.

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

```
make -C platform
make –C application
```

To run the simulation, in the `tracing` directory, do:

```
./platform/platform.${IMPERAS_ARCH}.exe --program application/asmtest.OR1K.elf
```

You should see the following output:

```
Info 'cpu1', 0x0000000001000074: l.addi   r1,r0,0x0
Info 'cpu1' REGISTERS
--------------- --------------- --------------- ---------------
 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
--------------- --------------- --------------- ---------------

Info 'cpu1', 0x0000000001000078: l.addi   r2,r0,0x1
Info 'cpu1' REGISTERS
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000000   R2 : 00000001   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 : 0100007c   SR : 00008001   ESR: deadbeef   EPC: deadbeef
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:0 CF:0 OF:0
--------------- --------------- --------------- ---------------

Info 'cpu1', 0x000000000100007c: l.addi   r3,r0,0xffffffff
Info 'cpu1' REGISTERS
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000000   R2 : 00000001   R3 : ffffffff
 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 : 01000080   SR : 00008001   ESR: deadbeef   EPC: deadbeef
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:0 CF:0 OF:0
--------------- --------------- --------------- ---------------

Info 'cpu1', 0x0000000001000080: l.addi   r4,r0,0x800
Info 'cpu1' REGISTERS
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000000   R2 : 00000001   R3 : ffffffff
 R4 : 00000800   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 : 01000084   SR : 00008001   ESR: deadbeef   EPC: deadbeef
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:0 CF:0 OF:0
--------------- --------------- --------------- ---------------

...  etc ...

Info 'cpu1', 0x00000000010000ac: l.addi   r1,r2,0x0
Processor 'cpu1' terminated at 'exit', address 0x10000ac
Info 'cpu1' REGISTERS
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000001   R2 : 00000001   R3 : ffffffff
 R4 : 00000800   R5 : 00400000   R6 : 00100000   R7 : 000007ff
 R8 : ffffffff   R9 : 00000000   R10: 00000000   R11: 00000000
 R12: 00000000   R13: 00000000   R14: 00000000   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 : 010000b0   SR : 00008001   ESR: deadbeef   EPC: deadbeef
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:0 CF:0 OF:0
--------------- --------------- --------------- ---------------
```

The initial instructions of the application initialize registers R1-R14 of the OR1K processor, mostly using `l.addi` instructions. As this happens, we see each register value change from `0xdeadbeef` (set in the processor constructor) to `0x00000000`.

There is a line of trace output for every instruction that is executed. Each trace line gives the instruction address (starting with `0x1000074`, the start address specified in the ELF file) and the instruction disassembly, produced using the disassembler of the OR1K model.

4.1.2 Controlling tracing during simulation

These functions can be used to control tracing when the simulator is stopped during a session:

function	**use**
icmTraceOnAfter	Turn tracing on after this many more instructions
icmTraceOffAfter	Turn tracing off after this many more instructions
icmEnableTraceBuffer	Keep rolling record of last 254 instructions (slight speed penalty)
icmDisableTraceBuffer	Stop the rolling record.
icmDumpTraceBuffer	Dump the contents of the rolling record.

Due to the amount of data produced, tracing has a heavy speed penalty. The trace buffer has a lesser cost so can be used to record a short history which can be printed out when desired, typically when a breakpoint or watchpoint has been hit.

4.1.3 Command Line

The same tracing operations can also be controlled from the command line when the command line parser is included into the platform.

4.2 Simulating Exceptions

By default, simulation will stop (`icmSimulatePlatform` will return) if a processor exception occurs. Some examples of processor exceptions are:

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

⇒ When `icmSimulatePlatform` returns for any reason other than end-of-simulation, it returns an `icmProcessorP` object that is the handle of the processor that was executing when the termination condition occurred. To find the exact reason why simulation stopped, use:

```
icmStopReason icmGetStopReason(icmProcessorP processor);
```

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

Instead of stopping simulation on a simulated exception, it is possible to specify that the processor should perform its usual exception actions instead (typically, enter kernel mode and jump to a kernel exception handler). This is done using the `ICM_ATTR_SIMEX` processor instance attribute.

4.2.1 Example: Simulating an Unaligned Access Exception

This example is in the `exception` directory.

```
$IMPERAS_HOME/Examples/Platforms/exceptions
```

The test platform file `platform/platform.c`, has been changed as follows:

```
    #ifdef SIMEXCEPTIONS
        #define SIM_FLAGS (ICM_ATTR_SIMEX | ICM_ATTR_TRACE)
    #else
        #define SIM_FLAGS (ICM_ATTR_TRACE)
    #endif

    // create a processor
    icmProcessorP processor = icmNewProcessor(
        "cpu1",              // CPU name
        “or1k”,              // CPU type
        0,                   // CPU cpuId
        0,                   // CPU model flags
        32,                  // address bits
        model,               // model file
        0,                   // not used
        SIM_FLAGS,           // instance attributes
        0,                   // user-defined attributes
        semihosting,         // semi-hosting file
        0                    // not used
    );
```

We have added the value `ICM_ATTR_SIMEX` to the attributes passed to the processor instance under control of the makefile.

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

```
make –C platform EXCEPTIONS=0
make –C application
```

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

```
./platform/platform.${IMPERAS_ARCH}.exe --program application/asmtest.OR1K.elf
```

You should see the following output:

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

The two lines:

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

show that the load from address `0x00000001` has been detected as an unaligned load. When this happens, `icmSimulatePlatform` returns the processor object handle (to indicate that simulation stopped abnormally).

⇒ Calling `icmGetStopReason` on the processor handle returned by `icmSimulatePlatform` would return `ICM_SR_RD_ALIGN` in this case.

Now recompile the platform with simulation of exceptions enabled using the following command in the `exception` directory:

```
make –C platform clean
make –C platform EXCEPTIONS=1
```

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

```
./platform/platform.${IMPERAS_ARCH}.exe --program application/asmtest.OR1K.elf
```

You should see the following output:

```
Info 'cpu1', 0x0000000000010000: l.addi   r1,r0,0x0
Info 'cpu1', 0x0000000000010004: l.addi   r2,r0,0x1
Info 'cpu1', 0x0000000000010008: l.lwz    r3,0x0(r2)
Info 'cpu1', 0x0000000000000200: l.j      0x00010024
Info 'cpu1', 0x0000000000000204: l.nop    0x0
Info 'cpu1', 0x0000000000010024: l.addi   r1,r2,0x0
Processor 'cpu1' terminated at 'exit', address 0x10024
```

In this example we see the load instruction is executed:

```
Info 'cpu1', 0x0000000000010008: l.lwz    r3,0x0(r2)
```

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

```
Info 'cpu1', 0x0000000000000200: l.j      0x00010024
```

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

5 Attributes

Model instances may be configured by *user-defined attributes* to control any implementation-dependent details of the model, such as the endianness or the number of processors in a multi-core CPU. Attributes in the platform are referred to as parameters in a model.

5.1 Setting Attributes

Each processor instance can be given a list of attributes. This list is created by the function `icmNewAttrList`:

```
icmAttrListP icmNewAttrList(void);
```

Once a list has been created, named attributes can be added to the list using the functions:

```
void icmAddBoolAttr  (icmAttrListP attrs, const char *name, Bool value);
void icmAddDoubleAttr(icmAttrListP attrs, const char *name, double value);
void icmAddPtrAttr   (icmAttrListP attrs, const char *name, void *value);
void icmAddStringAttr(icmAttrListP attrs, const char *name, const char *value);
void icmAddUns32Attr (icmAttrListP attrs, const char *name, Uns32 value);
void icmAddUns64Attr (icmAttrListP attrs, const char *name, Uns64 value);
```

These functions allow the addition of a 64-bit unsigned attribute, a double attribute, a string attribute or a native host pointer to a previously-created attribute list. This attribute list is then passed as one of the arguments on the model instantiation call (e.g. `icmNewProcessor`). See section 5.3.1 for an example.

5.2 Attribute Definitions

The precise attributes supported by each processor model vary. For models provided with OVP, documentation of the attributes supported for each model may be found in the README.txt file in the same directory in the VLNV tree that contains the model file. The VLNV tree may be found at:

```
    $IMPERAS_HOME/lib/$IMPERAS_ARCH/ImperasLib
```

5.3 Special Attributes

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

5.3.1 MIPS Attribute

All processors support a double attribute called *mips*, used to specify *the nominal processor speed in millions of instructions per second*. This nominal mips rate is used to apportion run time between processors in a multiprocessor simulation. The default nominal mips rate for each processor is 100. Section 5.4 shows an example of instantiating a processor with a nominal mips rate of 200 MIPS instead.

5.3.2 Endian Attribute

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

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

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

See Section 5.4 for an example of setting the *endian* attribute on a processor instantiation..

5.4 User-Defined Attribute Example

The following is an example of code that overrides the *mips* and *endian* user-defined attributes when instantiating an instance of a processor:

```
// create a user attribute object
icmAttrListP userAttrs = icmNewAttrList();

// add a double attribute to set mips to 200
icmAddDoubleAttr(userAttrs, "mips", 200.0);

// add a string attribute to set endian to big
icmAddStringAttr(userAttrs, "endian", “big”);

// add a native host pointer attribute
icmAddPtrAttr(userAttrs, "dataPtr", &data);

// create a processor
icmProcessorP processor = icmNewProcessor(
    "cpu1",             // CPU name
    “or1k”,             // CPU type
    0,                  // CPU cpuId
    0,                  // CPU model flags
    32,                 // address bits
    model,              // model file
    0,                  // not used
    0,                  // simulation attributes
    userAttrs,          // user-defined attributes
    semihosting,        // semi-hosting file
    0                   // not used
);
```

5.5 Querying Attributes

The function `icmIterAllUserAttributes` will iterate over all user-defined attributes in the platform. If it is called after loading all processor models and loading and initializing all PSEs, it will include all attributes set by the platform AND all attributes tested for by models.

5.6 Overriding Attributes

Attributes can be overridden by several mechanisms:

- The -override command line argument to the Imperas simulator `imperas.exe`.
- The -override command line argument in an Imperas control file.
- The `icmOverride()` function.

```
icmOverride(const char *path, const char *value)
```

The `icmOverride() function` can be used repeatedly to create a list of overrides in the simulator before constructing the platform. This list is then consulted during platform construction and entries are applied to model instances if they match. This mechanism allows the separation of platform construction from command line parsing and platform configuration. The *path* argument is the hierarchical path to a model parameter. The *value* argument is a string representation of the value and will be converted to the required type.

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

    // parse the command line, or read configuration data
    // The override must be set before the platform is constructed
    icmOverride("plat1/cpu1/variant", "VARIANT_A");


    // Start the platform
    icmInitPlatform(ICM_VERSION, 0, 0, 0, "plat1");

    // create a processor
    icmProcessorP processor = icmNewProcessor(
        "cpu1",             // CPU name
        ...
    );

    // load the processor object file
    icmLoadProcessorMemory(processor, argv[1], ICM_LOAD_DEFAULT, False, True);

    icmSimulatePlatform();
    icmTerminate();

    return 0;
}
```

6 Custom Tracing using Model Access Functions

In section 4.1, 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 an ICM platform program is being used to generate trace output to compare with the output from another tool, comparison is much easier if the format from the ICM platform can be made to exactly match the other tool.

The ICM 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, `icmSimulate`, which will be used in this example.

6.1 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).

To access the current program counter, use `icmGetPC`:

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

The return value from `icmGetPC` 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)icmGetPC(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 `icmGetPCDS`:

```
Uns8  delaySlotOffset;
Uns32 branchPC = icmGetPCDS(processor, &delaySlotOffset);
```

`icmGetPCDS` 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 `icmGetPCDS` 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 `icmReadReg`, which fills a byref argument `buffer` with the current value of a named register:

```
Bool icmReadReg(icmProcessorP processor, const char *name, void *buffer);
```

To write a processor register, there is a similar function `icmWriteReg`:

```
Bool icmWriteReg(icmProcessorP processor, const char *name, void *buffer);
```

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

```
Uns32 regR1;
icmReadReg(processor, “R1”, regR1);
regR1 = regR1 & REG_FLAG_MASK;
icmWriteReg(processor, “R1”, regR1);
```

⇒ It is the responsibility of the ICM application 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 function icmSetPC 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 an ICM platform to iterate over all the registers in a processor instance to determine their names and sizes (in bits) using three functions: `icmGetNextReg`, `icmGetRegInfoName` and `icmGetRegInfoBits`.

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

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

while((info=icmGetNextReg(info))) {
    . . .
}
```

Given an `icmRegInfoP` pointer, the *name* of the register it corresponds to can be found using `icmGetRegInfoName` and the *register size in bits* can be found using `icmGetRegInfoBits`:

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

while((info=icmGetNextReg(info))) {

    const char *name = icmGetRegInfoName(info);
    Uns32       bits = icmGetRegInfoBits(info);

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

The name returned by `icmGetRegInfoName` can be used if required to identify the register to read or write using `icmReadReg` or `icmWriteReg`.

`icmGetRegInfoUsage` returns an enumeration describing if the register has special use.

6.2 Generating Disassembly Output

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

```
icmPrintf("%s", icmDisassemble(processor, icmGetPC(processor)));
```

6.3 Dumping Registers

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

```
icmDumpRegisters(processor);
```

6.4 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 ICM function `icmGetProcessorICount`; for example, to print the number of instructions executed at the end of simulation:

```
icmPrintf(
    "Simulation finished, "FMT_64u" instructions executed...\n",
    icmGetProcessorICount(processor)
);
```

⇒ 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`.

6.5 Simulating for One Instruction

Previous examples have used the function `icmSimulatePlatform`, which simulates a platform using a built-in scheduling algorithm that simulates each processor for many instructions before returning to the ICM platform[2]. 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 `icmSimulate`:

```
icmStopReason icmSimulate(processorP processor, Uns64 instructions);
```

`icmSimulate` 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:

```
typedef enum icmStopReasonE {
  ICM_SR_SCHED      = 0x00, ///< Scheduler expired.
  ICM_SR_YIELD      = 0x01, ///< Yield encountered.
  ICM_SR_HALT       = 0x02, ///< CPU is halted.
  ICM_SR_EXIT       = 0x03, ///< CPU has exited.
  ICM_SR_FINISH     = 0x04, ///< Simulation finish.
  ICM_SR_RD_PRIV    = 0x05, ///< Read privilege exception.
  ICM_SR_WR_PRIV    = 0x06, ///< Write privilege exception.
  ICM_SR_RD_ALIGN   = 0x07, ///< Read align exception.
  ICM_SR_WR_ALIGN   = 0x08, ///< Write align exception.
  ICM_SR_FE_PRIV    = 0x09, ///< Fetch privilege exception.
  ICM_SR_ARITH      = 0x0a, ///< Arithmetic exception.
  ICM_SR_INTERRUPT  = 0x0b, ///< Interrupt simulation.
  ICM_SR_FREEZE     = 0x0c, ///< Frozen (by icmFreeze).
  ICM_SR_WATCHPOINT = 0x0d, ///< Memory watchpoint is pending.
  ICM_SR_BP_ICOUNT  = 0x0e, ///< Instruction count breakpoint is pending.
  ICM_SR_BP_ADDRESS = 0x0f, ///< Address breakpoint is pending.
  ICM_SR_RD_ABORT   = 0x10, ///< Read abort exception.
  ICM_SR_WR_ABORT   = 0x11, ///< Write abort exception.
  ICM_SR_FE_ABORT   = 0x12, ///< Fetch abort exception.
  ICM_SR_INVALID    = 0x13  ///< (invalid entry).
} icmStopReason;
```

The three most common return codes are:

- `ICM_SR_SCHED`
  processor successfully simulated the required number of instructions and returned
- `ICM_SR_EXIT`
  processor has exited (but in a multiprocessor platform, other processors may still be running)
- `ICM_SR_FINISH`
  simulation has finished

In practice, it is usually sufficient to continue simulation while the return code from `icmSimulate` is `ICM_SR_SCHED`, for example:

```
while(icmSimulate(processor, 1)==ICM_SR_SCHED) {
    . . .
}
icmPrintf(
```

[2] In fact, `icmSimulatePlatform` simulates for a time duration, which can be specified by `icmSimulationStopTime`. The actual number of instructions executed up to this stop time is the processor nominal mips rate x 1e6 x `stopTime`.

```
    "Simulation finished, “FMT_64u” instructions executed\n",
    icmGetProcessorICount(processor)
);
```

6.6 Example

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

This example is found in the `access` directory.

```
$IMPERAS_HOME/Examples/Platforms/access
```

The test platform file, `platform/platform.c`, is as follows:

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

    // check for the application program name argument
    if(argc!=2) {
        icmPrintf("%s: expected application name argument\n", argv[0]);
    }

    // initialize CpuManager
    icmInitPlatform(ICM_VERSION, 0, 0, 0, “platform”);

    // create a processor
    icmProcessorP processor = icmNewProcessor(
        "cpu1",             // CPU name
        “or1k”,             // CPU type
        0,                  // CPU cpuId
        0,                  // CPU model flags
        32,                 // address bits
        model,              // model file
        0,                  // not used
        0,                  // enable tracing or register values
        0,                  // user-defined attributes
        semihosting,        // semi-hosting file
        0                   // not used
    );

    // load the processor object file
    icmLoadProcessorMemory(processor, argv[1], ICM_LOAD_DEFAULT, False, True);

    Bool done = False;

    while(!done) {

        Uns32 currentPC = (Uns32)icmGetPC(processor);

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

        // execute one instruction
        icmPrintf("** Instruction Execution\n");
        done = (icmSimulate(processor, 1) != ICM_SR_SCHED);

        // dump registers
        icmPrintf("** Register Dump\n");
        icmDumpRegisters(processor);
    }

    // print number of instructions executed at end of simulation
```

.

```
    icmPrintf(
        "Simulation finished, "FMT_64u" instructions executed\n",
        icmGetProcessorICount(processor)
    );

    // free simulation data structures
    icmTerminate();

    return 0;
}
```

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

```
make –C platform
make –C application
```

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

```
./platform/platform.${IMPERAS_ARCH}.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
-------------- -------------- -------------- --------------

** Instruction Disassemble
0x01000078 : l.addi   r2,r0,0x1
** Instruction Execution
** Register Dump
-------------- -------------- -------------- --------------
 R0 : 00000000   R1 : 00000000   R2 : 00000001   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 : 0100007c   SR : 00008001   ESR: deadbeef   EPC: deadbeef
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:0 CF:0 OF:0
-------------- -------------- -------------- --------------

** Instruction Disassemble
0x0100007c : l.addi   r3,r0,0xffffffff
** Instruction Execution
** Register Dump
-------------- -------------- -------------- --------------
 R0 : 00000000   R1 : 00000000   R2 : 00000001   R3 : ffffffff
 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 : 01000080   SR : 00008001   ESR: deadbeef   EPC: deadbeef
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:0 CF:0 OF:0
--------------- --------------- --------------- ---------------

… etc …
```

7 Semihosting

We saw the use of the intercept library `imperasExit` earlier which allows simple assembler tests to be constructed and exit cleanly. In this example we will show how to use a more general semihosting library so that, for example, calls to `printf` in the application code can be intercepted and performed on the host machine.

Construction of semihosting libraries is covered in detail in the *OVP Processor Modeling Guide*, in the chapter entitled *Function Address Semihosting*. Here, we will show how to instantiate a standard Imperas semihosting shared object library to intercept the system calls in the *newlib* library for the OR1K processor. The semihosting shared library is found using `icmGetVlnvString`, as follows:

```
icmGetVlnvString(vlnvRoot, "ovpworld.org", "semihosting", "or1kNewlib", "1.0", "model");
```

⇒ Why is semihosting not part of the processor model, but specified separately? This enables processor models to be “pure” instruction-accurate models, entirely independent of the environment in which they are to be used, and for multiple, incompatible, environments to be supported simply by specifying a semihosting library for each one.

7.1 Example

The following example uses the semihosting shown above to allow a simple ‘hello’ application to print to the standard output.

This example is found in the `semihosting` directory.

```
$IMPERAS_HOME/Examples/Platforms/semihosting
```

```
//
// Main simulation routine
//
int main(int argc, char ** argv) {

    // check for the application program name argument
    if(argc!=2) {
        icmPrintf("%s: expected application name argument\n", argv[0]);
    }

    // select library components
    const char *vlnvRoot = NULL; // When NULL use default library
    const char *model    = icmGetVlnvString(
        vlnvRoot ,
        "ovpworld.org",
        "processor",
        "or1k",
        "1.0",
        "model"
     );
    const char *semihosting = icmGetVlnvString(
        vlnvRoot,
        "ovpworld.org",
        "semihosting",
        "or1kNewlib",
        "1.0",
        "model
    );

    // initialize CpuManager
    icmInitPlatform(ICM_VERSION, 0, 0, 0, “platform”);

    // create a processor with semihosting
    icmProcessorP processor = icmNewProcessor(
        "cpu1",             // CPU name
        "or1k",             // CPU type
        0,                  // CPU cpuId
        0,                  // CPU model flags
        32,                 // address bits
        model,              // model file
        0,                  // not used
        0,                  // enable tracing etc
        0,                  // user-defined attributes
        semihosting,        // semi-hosting file
        0                   // not used
    );

    // load the processor object file
    icmLoadProcessorMemory(processor, argv[1], ICM_LOAD_DEFAULT, False, True);

    // run simulation
    icmSimulatePlatform();

    // terminate simulation
    icmTerminate();

    return 0;
}
```

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

```
make –C platform
make –C application
```

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

```
./platform/platform.${IMPERAS_ARCH}.exe --program application/application.OR1K.elf
```

You should see the following output:

```
Hello
```

Linked into the application are calls to standard operating system functions (for example `open` and `close`). With the appropriate semihosting library installed these low level calls are *intercepted* by the simulator and the functionality to implement them is provided by the native host instead.

7.2 Additional intercept objects

Intercept libraries can be added to an existing processor instance using `icmAddInterceptObject()` (only available in Imperas Professional products).

```
...
    icmProcessorP processor = icmNewProcessor(
        "cpu1",             // CPU name
        “or1k”,             // CPU type
        0,                  // CPU cpuId
        0,                  // CPU model flags
        32,                 // address bits
        model,              // model file
        0,                  // not used
        0,                  // simulation attributes
        0,                  // user-defined attributes
        semihosting,        // semi-hosting file
        0,                  // not used
    );

    icmAddInterceptObject(
        processor,                 // processor handle
        "intercept1",              // intercept library instance name
        "/home/library/intercept", // path to intercept library shared object
        0,                         // not used
        0                          // optional user defined attribute list
    );
```

8 Memory Operations

8.1 Accessing Processor Model Memory

When a new processor instance is created, by default an implicit RAM memory that covers the entire address space that can be accessed by the processor type is also created.

This memory can be directly accessed with the ICM platform using the functions: `icmReadProcessorMemory` and `icmWriteProcessorMemory`. The function `icmLoadProcessorMemory` can be used to load an object file. In addition, the functions `icmDebugReadProcessorMemory` and `icmDebugWriteProcessorMemory` can be used to read and write memory without causing side effect in the processor model, or in any TLM2.0 models connected to the processor.

8.1.1 Loading object files

As we have previously seen an object file can be loaded into processor memory using the `icmLoadProcessorMemory` function:

```
icmImagefileP icmLoadProcessorMemory(
    icmProcessorP processor,
    const char   *objectFile,
    icmLoadAttrs  attrs,
    Bool          verbose,
    Bool          useEntry
);
```

`icmLoadAttrs` are defined as:

```
ICM_LOAD_DEFAULT:       0x00
ICM_LOAD_PHYSICAL:      0x01   Use object file physical addresses if available
ICM_LOAD_VERBOSE:       0x02   Report each section as it is loaded
ICM_ZERO_BSS:           0x04   Zero the extent of the BSS section if present
ICM_SET_START:          0x08   Set the PC to the code start address
ICM_LOAD_SYMBOLS_ONLY   0x10   Read the symbols but do not load the code or data
ICM_ELF_USE_VMA         0x20   Load ELF files using VMA addresses instead of LMA.
```

Specifying `True` for the `verbose` argument has the same effect as setting `ICM_LOAD_VERBOSE` in the `attrs` argument. Specifying `True` for the `useEntry` argument has the same effect as setting `ICM_SET_START` in the `attrs` argument. This is done to preserve backwards compatibility with previous versions of the API.

For ELF files, the Load Memory Address (LMA) is used as the load address by default. Setting `ICM_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* using physical address information, and the PC will be set to the entry address defined in the object file:

```
icmLoadProcessorMemory(processor, “hello.or1k”, ICM_LOAD_PHYSICAL, False, True);
```

.

The algorithm used is as follows:
1. Find the processor passed in argument #1.
2. Find the bus connected to the instruction port 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 `icmLoadBus.`

The functions `icmLoadProcessorMemory` and `icmLoadBus` return an `icmImagefileP` which can be interrogated using `icmGetImagefileEndian`, `icmGetImagefileElfcode` and `icmGetImagefileEntry` to find respectively the endianness, the 16-bit processor architecture code and the executable start address.

8.1.1.1 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

8.1.1.2 Loading Symbols in object files

When `icmLoadProcessorMemory` 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 icmAddInterceptObject)

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 `icmLoadSymbols can be used` to associate symbols with 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:

create platform

```
    // initialize CpuManager
    icmInitPlatform(ICM_VERSION, 0, 0, 0, “platform”);

    // create a processor
    icmProcessorP processor = icmNewProcessor(
```

.

```
       "proc1",
       "or1k",
       0,
       0,
       32,
       modelPath,
       0
       ICM_ATTR_TRACE,
       0,
       0,
       0
    );
```

load and run simulation

```
    // load the boot loader. This program will load the contents of
    ///mainprogram.elf by some other means.
    icmLoadProcessorMemory(processor, "bootloader.elf", ICM_LOAD_DEFAULT, False. False);

    // load the symbols from the other program so that they are known to
    // the simulator.
    icmLoadSymbols(processor, "program.elf", False);

    icmSimulatePlatform();

    icmTerminate();
```

⇒ icmLoadSymbols reads the same format files as icmLoadProcessorMemory.

8.1.2 Reading and Writing Data

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

8.1.2.1 Reading and Writing Data Example

This example is found in the `hexLoader` directory.

```
$IMPERAS_HOME/Examples/Platforms/hexLoader
```

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 platform. 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 ‘#’.

```
#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 `platform/platform.c`, that takes the name of the file containing the data to be loaded and a switch to control the byte swapping.

```
static int loadHexFile(icmProcessorP processor, char *fileName, Bool swap) {

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

    fp = fopen(fileName, "r");

    if (!fp) {
        printf ("Failed to open Memory Initialization File %s\n", fileName);
        return -1;
    }

    icmPrintf("\nLoading Hex file %s\n", fileName);

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

        if ( inBuf[0] == '#') {
            // ignore header
        } else {
            if ( sscanf(inBuf, "%08x-%08x %08x ;", &address, &endAddress, &data) != 3 ) {
                sscanf(inBuf, "%08x %08x ;", &address, &data);
                endAddress = address;
            }

            if (swap) {  //byte swap
                data = (data & 0x000000ff) << 24 |
                       (data & 0x0000ff00) <<  8 |
                       (data & 0x00ff0000) >>  8 |
                       (data & 0xff000000) >> 24 ;
            }

            do {
                //
                // Access the memory through the processor memory space
                //
                icmWriteProcessorMemory(processor,      // processor
                                        address,        // memory address
                                        &data,          // data buffer of data to write
                                        4);             // number of bytes to write

                icmReadProcessorMemory(processor, address, &dataCheck, 4);

                if(data != dataCheck) {
                    icmPrintf("Failed Data Read Back at 0x%08\n", address);
                    return -1;
                }

                icmPrintf("  0x%08x <= 0x%08x\n", address, data);
            } while ( address++ < endAddress);
        }
    }

    icmPrintf("Load Complete\n\n");

    if (fclose(fp)!=0) {
        printf ("Failed to close Memory Initialization File\n");
```

.

```
        return -1;
    }

    return 0;
}
```

⇒ Note if the processor uses virtual addressing the address of the **icmWriteProcessorMemory** and **icmReadProcessorMemory** 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 two regions of memory. The memory is loaded by a call to the hexLoader routine that has been described above.

```
    // Load Hex file into Simulator Memory
    if (loadHexFile(processor, argv[1], False)) {
        printf("Hex File Load of %s Failed\n", argv[1]);
        return -1;
    }
```

To run the example, compile the test platform using the following command in the `hexLoader` directory:

```
 make –C platform
```

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

```
 ./platform/platform.${IMPERAS_ARCH}.exe --program application/asmtest.hex
```

You should see output similar to the following:

```

OVPsim (32-Bit) v20150205 Open Virtual Platform simulator from www.OVPworld.org.
Copyright (C) 2005-2015 Imperas Ltd.  Contains Imperas Proprietary Information.
Licensed Software, All Rights Reserved.
Visit www.imperas.com for multicore debug, verification and analysis solutions.
OVPsim  started: Thu Mar 12 11:54:16 2015


BUS MASTERS: 2
   PORT 'DATA' of 'cpu1'
   PORT 'INSTRUCTION' of 'cpu1'
BUS SLAVES: 2
   0x00000000:0x003fffff: PORT 'mp1' of 'mem1'
   0x00400000:0x00400fff: xxxxxxxx unmapped xxxxxxxx
   0x00401000:0xffffffff: PORT 'mp2' of 'mem2'

Loading Hex file application/asmtest.hex
  0x01000074 <= 0x0000209c
  0x01000078 <= 0x0100409c
  0x0100007c <= 0xffff609c
  0x01000080 <= 0x0008809c
  0x01000084 <= 0x000884b0
  0x01000088 <= 0x0000a484
  0x0100008c <= 0xffffa3a0
  0x01000090 <= 0x0000a4a0
  0x01000094 <= 0x0100a5a0
  0x01000098 <= 0x002804d4
  0x0100009c <= 0x0000209c
```

```
Load Complete

0x01000074 : l.addi   r1,r0,0x0
--------------- --------------- --------------- ---------------
 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
--------------- --------------- --------------- ---------------

    … lines deleted …

0x01000088 : l.lwz    r5,0x0(r4)
Processor Exception (PC_PRX) Processor 'cpu1' 0x1000088: l.lwz    r5,0x0(r4)
Processor Exception (PC_RPX) No read access at 0x400000
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000000   R2 : 00000001   R3 : ffffffff
 R4 : 00400000   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 : 01000088   SR : 00008001   ESR: deadbeef   EPC: deadbeef
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:0 CF:0 OF:0
--------------- --------------- --------------- ---------------



OVPsim  finished: Thu Mar 12 11:54:16 2015
Visit www.imperas.com for multicore debug, verification and analysis solutions.
OVPsim (32-Bit) v20150205 Open Virtual Platform simulator from www.OVPworld.org.
```

Note that:

1. The load from address 0x00400000 causes an exception – this is because there is no memory mapped at this address.
2. The simulator reports the exception (and returns `ICM_SR_RD_PRIV` from `icmSimulate`, although this isn't explicitly shown in this example). The `ICM_ATTR_SIMEX` instance attribute could be used to cause the exception to be simulated instead – see section 4.2 for more information about this.

8.1.3 Reading and writing memory without side-effects.

The functions `icmDebugReadProcessorMemory and icmDebugWriteProcessorMemory` are for use with a debugger, rather than as part of a platform model. This table compares their behavior:

Function	Endian	TLM2.0	Effect on tlb	Bad access
`icmDebugReadProcessorMemory`	selectable	transport_dbg	none	returns 'False'
`icmDebugWriteProcessorMemory`	selectable	transport_dbg	none	returns 'False'
`icmReadProcessorMemory`	target	b_transport	might update	bus err if supported
`icmWriteProcessorMemory`	target	b_transport	might update	bus err if supported

.

Their prototypes are:

```
icmDebugReadProcessorMemory(
    icmProcessorP processor,  // processor context
    Addr          simAddress, // address in processor's address space
    void         *buffer,     // pointer to destination host memory
    Uns32         objectSize, // size of each object, in bytes
    Uns32         objects,    // number of objects
    icmHostEndian endian      // byte swap behavior
);

icmDebugWriteProcessorMemory(
    icmProcessorP processor,  // processor context
    Addr          simAddress, // address in processor's address space
    void         *buffer,     // pointer to destination host memory
    Uns32         objectSize, // size of each object, in bytes
    Uns32         objects,    // number of objects
    icmHostEndian endian      // byte swap behavior
);
```

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

endian	**Effect**
ICM_HOSTENDIAN_HOST	Byte swapped, if necessary, to be host endian
ICM_HOSTENDIAN_TARGET	No swapping; result will be same as target processor
ICM_HOSTENDIAN_BIG	Byte swapped, if necessary, to be big endian
ICM_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.

8.2 Adding Memory Callbacks

Adding callbacks across memory regions allows memory *watchpoints*, amongst other features, to be implemented. A callback is executed whenever there is either a read or a write access to a specified range of memory addresses.
The callbacks are created using `icmAddReadCallback` and `icmAddWriteCallback functions.`

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

// watch write accesses to the address range 0x01000000:0x01000fff
icmAddWriteCallback(processor, 0x01000000, 0x01000fff, bufferWriteCallBack, 0);
```

Watchpoints allow the monitoring of memory access behavior of a processor as it runs an application.

8.2.1.1 Example Adding a Memory Callback

This example is found in the `watchpoint` directory.

```
$IMPERAS_HOME/Examples/Platforms/watchpoint
```

This example shows the use of a watchpoint to trap a write to a specific address.

In the main function of the platform 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.

```
    //
    // Create a watchpoint
    // Invoke callback on write accesses to the address 0x00400000-0x00400003
    //
    icmAddWriteCallback(
        processor,          // processor
        0x00400000,         // low address
        0x00400003,         // high address
        watchWriteCB,       // callback to invoke
        "watch termination" // user data passed to callback
    );
```

The callback functions used for a read or a write should be defined using the macros `ICM_MEM_WATCH_FN`. The memory callback function is defined below.

```
//
// Callback for memory writes to defined external area
//
static ICM_MEM_WATCH_FN(watchWriteCB) {

    icmPrintf(
        "WATCHPOINT '%s': Writing to 0x%08x : Finish Simulation\n",
        (Uns8 *)userData,
        (Int32)address
    );
    icmFinish(processor, -7);
}
```

This function reports the write and then makes a call to `icmFinish`. This ICM function is used to terminate the simulation at the start of the next instruction; the next instruction is not executed. The second argument of `icmFinish` is an integer status code: this is of no significance to the simulator but can be used to communicate information to the simulation harness. In this case, the status code is printed in a message just before the end of simulation:

```
    icmPrintf("Simulation finished with status %d\n", icmGetStatus());
```

To run the example, compile the test platform and application using the following commands in the `watchpoint` directory:

```
make –C platform
make –C application
```

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

```
./platform/platform.${IMPERAS_ARCH}.exe --program application/asmtest.OR1K.elf
```

You should see output similar to the following:

```
OVPsim (32-Bit) v20150205 Open Virtual Platform simulator from www.OVPworld.org.
Copyright (C) 2005-2015 Imperas Ltd.  Contains Imperas Proprietary Information.
Licensed Software, All Rights Reserved.
Visit www.imperas.com for multicore debug, verification and analysis solutions.
OVPsim started: Thu Mar 12 13:17:39 2015


BUS MASTERS: 2
   PORT 'DATA' of 'cpu1'
   PORT 'INSTRUCTION' of 'cpu1'
BUS SLAVES: 2
   0x00000000:0x003fffff: PORT 'mp1' of 'mem1'
   0x00400000:0xffffffff: PORT 'mp2' of 'mem2'

Starting Simulation ...
0x01000074 : l.addi   r1,r0,0x0
0x01000078 : l.addi   r2,r0,0x1
0x0100007c : l.addi   r3,r0,0xffffffff
0x01000080 : l.addi   r4,r0,0x800
0x01000084 : l.muli   r4,r4,0x800
0x01000088 : l.sw     0x0(r4),r5
WATCHPOINT 'watch termination': Writing to 0x00400000 : Finish Simulation
Simulation finished with status -7


OVPsim  finished: Thu Mar 12 13:17:39 2015
Visit www.imperas.com for multicore debug, verification and analysis solutions.
OVPsim (32-Bit)  v20150205 Open Virtual Platform simulator from www.OVPworld.org.
```

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 `icmFinish` is printed just before the simulation exits.

8.3 Explicit Local and External Memory

Until now, all examples have 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. It is also possible to specify that certain address ranges will be modeled by callback functions in the ICM platform itself, which is useful for modeling simple memory-mapped devices such as uarts[3].

[3] But note that in general, it is much better to use Imperas *PSE* objects to model peripherals, instead of coding them directly in ICM, 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;

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 `icmNewBus`, which takes a bus name and bit width as arguments, for example:

```
icmBusP bus = icmNewBus(“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 using `icmConnectProcessorBusses`, which takes a processor and two busses, the instruction bus and the data bus, as arguments (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:

```
icmConnectProcessorBusses(processor, bus, bus);
```

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

```
icmMemoryP memory1 = icmNewMemory(“mem1”, ICM_PRIV_RWX, 0x003fffff);
```

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 `icmPriv` in `icmCpuManager.h`:

```
typedef enum icmPrivE {
    ICM_PRIV_NONE=0x0,              // no access permitted
    ICM_PRIV_R   =0x1,              // read permitted
    ICM_PRIV_W   =0x2,              // write permitted
    ICM_PRIV_RW  =0x3,              // read & write permitted
    ICM_PRIV_X   =0x4,              // execute permitted
    ICM_PRIV_RX  =0x5,              // read & execute permitted
    ICM_PRIV_WX  =0x6,              // write & execute permitted
    ICM_PRIV_RWX =0x7,              // read, write & execute permitted
} icmPriv;
```

⇒ Note that the last argument to `icmNewMemory` 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*.

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.

Once a memory has been created, it can be connected to a bus using `icmConnectMemoryToBus`, which takes a bus object, a memory port name, a memory object and a bus address as arguments, for example:

```
icmConnectMemoryToBus(bus, “mp1”, memory1, 0x10000);
```

This example connects a memory to a bus using port *mp1* of the memory (memories may be multiport, and be connected to several busses using different port names). The memory is connected with memory address `0` mapped to bus address `0x10000`.

Memories defined with `icmNewMemory` 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 in the ICM platform instead. This is done using `icmMapExternalMemory`:

8.4 Mapping an address region to a callback

In this example, an address region is mapped to read and write callbacks supplied by 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;

ICM_MEM_READ_FN(extMemReadCB) {
    extMemDesc *p = userData;
    memcpy(value, p->localSource, bytes)
}

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

ICM_MEM_WRITE_FN(extMemWriteCB) {
    memcpy(p->localSink, value, bytes)
}

static extMemDesc extMem;

icmMapExternalMemory(
    bus, “external”, ICM_PRIV_R, 0x00400000, 0x00400fff,
    extMemReadCB, extMemWriteCB, &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 ICM platform callback functions, `extMemReadCB` and `extMemWriteCB`. These callback functions are specified using the `ICM_MEM_READ_FN` and `ICM_MEM_WRITE_FN` macros. Any time a simulated processor or device performs a memory read or write in this address range, the appropriate platform callback function will be called. The write callback will be passed the value being written in the `value` argument. The read callback should fill the `value` buffer with `bytes` bytes of data (the required contents for a read at the passed address).

8.4.1 Invalid access

During the read or write callback the client might decide that the access cannot be completed. To signal this, either function `icmAbortRead` or `icmAbortWrite` should be called by the client. Use `icmAbortRead` when the callback was initiated by either of these functions:

```
icmReadProcessorMemory
icmReadBus
```

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

```
icmWriteProcessorMemory
icmWriteBus
```

In no other context should `icmAbortRead` or `icmAbortWrite` 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.

8.4.2 Debugging Bus Connections

When there are many connections to a bus, visualizing the connections can be difficult, to help, the ICM interface defines a useful debugging function:

```
void icmPrintBusConnections(icmBusP bus);
```

Given a bus, this 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'
```

8.4.3 Example

This example is found in the `memory` directory.

```
$IMPERAS_HOME/Examples/Platforms/memory
```

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

```
    // initialize CpuManager
    icmInitPlatform(ICM_VERSION, 0, 0, 0, “platform”);

    // create a processor
    icmProcessorP processor = icmNewProcessor(
        "cpu1",             // CPU name
        “or1k”,             // CPU type
        0,                  // CPU cpuId
        0,                  // CPU model flags
        32,                 // address bits
        model,              // model file
        0,                  // not used
        0,                  // simulation attributes
        0,                  // user-defined attributes
        semihosting,        // semi-hosting file
        0,                  // not used
    );

    // create the processor bus
    icmBusP bus = icmNewBus("bus", 32);

    // connect the processor busses
    icmConnectProcessorBusses(processor, bus, bus);

    // create two simulated memories for low and high regions
    icmMemoryP memory1 = icmNewMemory("mem1", ICM_PRIV_RWX, 0x003fffff);
    icmMemoryP memory2 = icmNewMemory("mem2", ICM_PRIV_RWX, 0xffffffff-0x00401000);

    // map the address range 0x00400000:0x00400fff externally to the processor,
    // read only
    icmMapExternalMemory(
        bus, "external", ICM_PRIV_R, 0x00400000, 0x00400fff,
        extMemReadCB, extMemWriteCB, 0
    );

    // connect memories to bus
    icmConnectMemoryToBus(bus, "mp1", memory1, 0);
    icmConnectMemoryToBus(bus, "mp2", memory2, 0x00401000);
```

```
    // show the bus connections
    icmPrintBusConnections(bus);

    Bool done = False;

    while(!done) {

        Uns32 currentPC = (Uns32)icmGetPC(processor);

        // disassemble instruction at current PC
        icmPrintf(
            "0x%08x : %s\n", currentPC,
            icmDisassemble(processor, currentPC)
        );

        // execute one instruction
        done = (icmSimulate(processor, 1) != ICM_SR_SCHED);

        // dump registers
        icmDumpRegisters(processor);
    }

    // free simulation data structures
    icmTerminate();
```

The callbacks are defined using the macros in the ICM API as:

```
static ICM_MEM_READ_FN(extMemReadCB) {

    Int32 data = 0xcefaedfe;
    *(Int32 *)value = data;

    icmPrintf(
        "EXTERNAL MEMORY: Reading  0x%08x from 0x%08x\n",
        data, (Int32)address
    );
}

static ICM_MEM_WRITE_FN(extMemWriteCB) {

    icmPrintf(
        "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 platform
make –C application
```

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

```
./platform/platform.${IMPERAS_ARCH}.exe --program application/asmtest.OR1K.elf
```

You should see the following output:

```
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'

. . . lines deleted . . .

0x01000084 : l.muli   r4,r4,0x800
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000000   R2 : 00000001   R3 : ffffffff
 R4 : 00400000   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 : 01000088   SR : 00008001   ESR: deadbeef   EPC: deadbeef
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:0 CF:0 OF:0
--------------- --------------- --------------- ---------------

0x01000088 : l.lwz    r5,0x0(r4)
EXTERNAL MEMORY: Reading  0xcefaedfe from 0x00400000
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000000   R2 : 00000001   R3 : ffffffff
 R4 : 00400000   R5 : feedface   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 : 0100008c   SR : 00008001   ESR: deadbeef   EPC: deadbeef
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:0 CF:0 OF:0
--------------- --------------- --------------- ---------------

. . . lines deleted . . .

0x01000094 : l.addic  r5,r5,0x1
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000000   R2 : 00000001   R3 : ffffffff
 R4 : 00400000   R5 : 00400002   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 : 01000098   SR : 00008001   ESR: deadbeef   EPC: deadbeef
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:0 CF:0 OF:0
--------------- --------------- --------------- ---------------

0x01000098 : l.sw     0x0(r4),r5
Processor Exception (PC_PRX) Processor 'cpu1' 0x1000098: l.sw     0x0(r4),r5
Processor Exception (PC_WPX) No write access at 0x400000
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000000   R2 : 00000001   R3 : ffffffff
 R4 : 00400000   R5 : 00400002   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 : 01000098   SR : 00008001   ESR: deadbeef   EPC: deadbeef
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:0 CF:0 OF:0
--------------- --------------- --------------- ---------------
```

Note that:

1. The load to the external memory region is performed correctly but the store causes an exception – this is because the external region was specified to have read access permission only.
2. 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.
3. The simulator reports the exception (and returns `ICM_SR_WR_PRIV` from `icmSimulate`, although this isn't explicitly shown in this example). The `ICM_ATTR_SIMEX` instance attribute could be used to cause the exception to be simulated instead – see section 4.2 for more information about this.

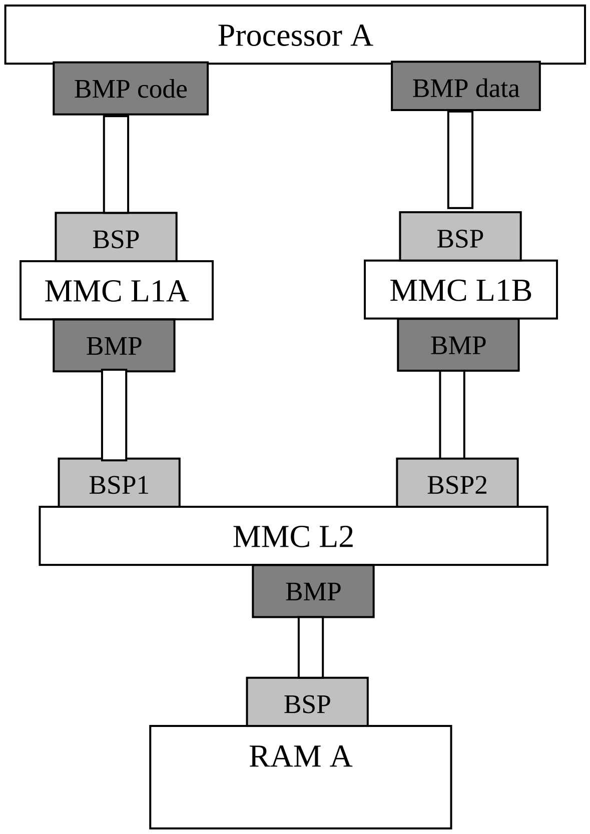
8.4.4 Processor Instruction Execution

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 icmProcessorP 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.

9 Caches

A cache, active memory device or external memory management unit can be modeled using a *Memory Model Component* (MMC). An MMC fits between a bus master such as a processor or a peripheral with bus mastership capability, and a bus slave such as a RAM, ROM or peripheral with a bus slave port. MMCs can also be cascaded to model, for example, multi-level caches.



Please refer to the *OVP VMI Memory Model Component Function Reference* for details of writing an MMC.

Note that since every bus access through an MMC causes at least one function to be called, use of an MMC will impact simulation performance.

9.1 Transparent or Full MMC Models

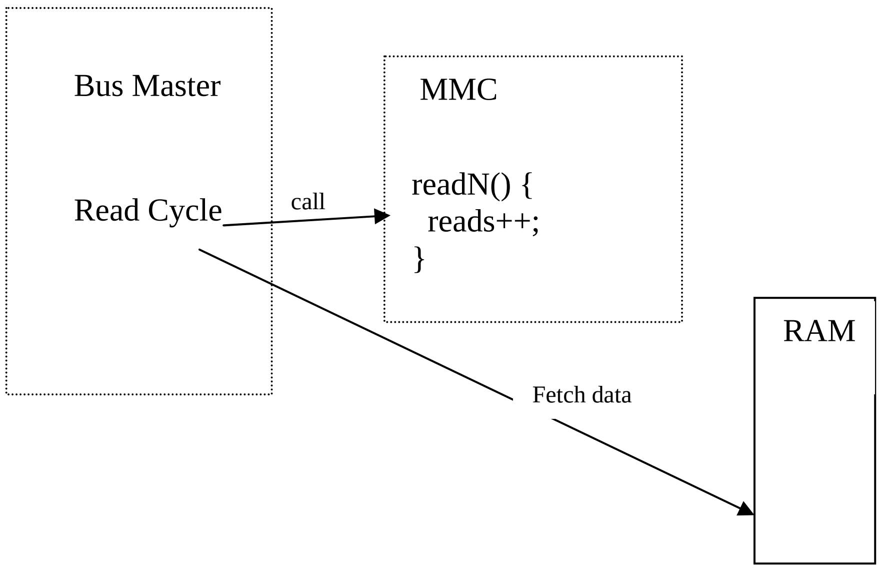
An MMC operates in one of two possible modes, *transparent* or *full*. An MMC can be written to support one or either mode. *Full* models implement storage and so can be used to accurately model components such as caches that are incoherent with main memory. *Transparent* models do not implement storage (so cannot be incoherent) but can be used to create very fast performance monitors. As an example, a transparent cache model would model only the cache tags and use this information to count hits and misses.

9.2 MMC Operation

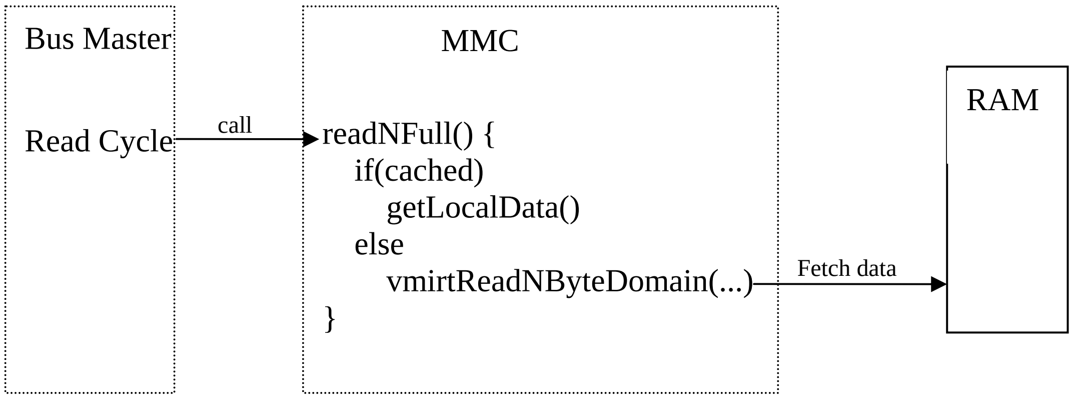
A *full* MMC model has one or more master ports and one or more slave ports. A *transparent* MMC model must have *exactly one master port* and one or more slave ports. Transparent MMCs have only one master port because during construction busses connected to the MMC slave ports are connected straight through to the master port.

In operation, a bus cycle instigated by another bus master in the system activates the MMC via one of its slave ports. This causes an activation function to be called in the MMC model. In a transparent MMC the activation function will perform some calculation and then return, allowing the simulator to propagate the effect of the bus cycle to the next component. In a full MMC the activation function might also instigate a bus cycle on a bus connected to one of its master ports.

9.3 Transparent Model



9.3.1 Full Model



9.4 Creating and connecting an MMC

An MMC is created using `icmNewMMC`. It is connected to a bus using `icmConnectMMCBus`.

9.4.1 Transparent MMC Example

An example of a transparent MMC is available at

```
$IMPERAS_HOME/Examples/Platforms/transparentMMC
```

This has a platform file containing:

```
    // select library components
    const char *vlnvRoot = 0; // when null use default library
    const char *model = icmGetVlnvString(
        vlnvRoot, "ovpworld.org", "processor", "or1k", "1.0", "model"
    );
    const char *semihosting = icmGetVlnvString(
        vlnvRoot, "ovpworld.org", "semihosting", "or1kNewlib", "1.0", "model"
    );
    const char *mmc_model = icmGetVlnvString(
        vlnvRoot, "ovpworld.org", "mmc", "wb_1way_32byteline_2048tags", "1.0",
        "model"
    );

    // initialize CpuManager
    icmInitPlatform(ICM_VERSION, 0, 0, 0, “platform”);

    // create a processor
    icmProcessorP cpu1h = icmNewProcessor(
        "cpu1",             // CPU name
        "or1k",             // CPU type
        0,                  // CPU cpuId
        0,                  // CPU model flags
        32,                 // address bits
        model,              // model file
        0,                  // not used
        0,                  // simulation attributes
        0,                  // user-defined attributes
        semihosting,        // semi-hosting file
        0,                  // not used
    );
    // create transparent MMCs
    icmMmcP mmci = icmNewMMC("mmci", mmc_model, "modelAttrs", 0, 0, True);
    icmMmcP mmcd = icmNewMMC("mmcd", mmc_model, "modelAttrs", 0, 0, True);

    // create the processor instruction bus and data bus
    icmBusP ibus = icmNewBus("ibus", 32);
    icmBusP dbus = icmNewBus("dbus", 32);

    // create the processor main bus
    icmBusP mbus = icmNewBus("mbus", 32);

    // connect processor ports to their buses
    icmConnectProcessorBusses(cpu1h, ibus, dbus);

    // connect MMCs to buses
    icmConnectMMCBus(mmci, ibus, "sp1", False);
    icmConnectMMCBus(mmcd, dbus, "sp1", False);

    // connect master ports of MMC to main bus
    icmConnectMMCBus(mmci, mbus, "mp1", True);
    icmConnectMMCBus(mmcd, mbus, "mp1", True);

    // create two simulated memories for low and high regions
    icmMemoryP memory1 = icmNewMemory("mem1", ICM_PRIV_RWX, 0x003fffff);
    icmMemoryP memory2 = icmNewMemory("mem2", ICM_PRIV_RWX, 0xffffffff-0x00401000);

    // connect memories to main bus
    icmConnectMemoryToBus(mbus, "mp1", memory1, 0);
    icmConnectMemoryToBus(mbus, "mp2", memory2, 0x00401000);
```

```
    // run until exit
    icmSimulatePlatform();

    // free simulation data structures
    icmTerminate();
```

This example instantiates a generic cache model from the ovpworld.org library. This cache model is available as source, so it can be used as-is or modified if required. In transparent mode, the cache model counts the number of accesses to hypothetical cache lines, given a particular cache configuration in terms of number of ways, line size and cache size. Example output is as follows:

```
Running example
Compiling Application hello.OR1K
Linking Application hello.OR1K.elf
2x transparent MMC

cacheConstructor called for platform/mmci
  --------------------------------------------
  Ways      : 1
  Line bits : 5
  Tag bits  : 11
  --------------------------------------------
  Tags      : 2,048
  Line bytes: 32
  Size      : 65,536
  Tag mask  : ................11111111111.....
  Key mask  : 111111111111111111111111111.....
  --------------------------------------------


cacheConstructor called for platform/mmcd
  --------------------------------------------
  Ways      : 1
  Line bits : 5
```

.

```
  Tag bits  : 11
  -------------------------------------------
  Tags      : 2,048
  Line bytes: 32
  Size      : 65,536
  Tag mask  : ................11111111111.....
  Key mask  : 111111111111111111111111111.....
  -------------------------------------------


cacheLink called for platform/mmci

cacheLink called for platform/mmcd
Hello world

cacheDestructor called for platform/mmci

READ ACCESSES:
  HITS        :           2,003
  MISSES      :             230
  1-byte      :               0
  2-byte      :               0
  4-byte      :           2,233
  8-byte      :               0
  N-byte      :               0 (0 bytes, average size=0.0 bytes)
  TOTAL READ  :           2,233
  TOTAL BYTES:            8,932

cacheDestructor called for platform/mmcd

READ ACCESSES:
  HITS        :             343
  MISSES      :              14
  1-byte      :              30
  2-byte      :              30
  4-byte      :             297
  8-byte      :               0
  N-byte      :               0 (0 bytes, average size=0.0 bytes)
  TOTAL READ  :             357
  TOTAL BYTES:            1,278

WRITE ACCESSES:
  HITS        :             282
  MISSES      :              24
  1-byte      :              12
  2-byte      :              12
  4-byte      :             282
  8-byte      :               0
  N-byte      :               0 (0 bytes, average size=0.0 bytes)
  TOTAL WRITE:              306
  TOTAL BYTES:            1,164
Done
```

9.4.2 Full MMC Example

An example of a full MMC is available at:

```
$IMPERAS_HOME/Examples/Platforms/fullMMC
```

The platform file is almost identical to that shown previously for transparent MMCs. The only significant difference is in the MMC instantiation lines:

```
    // create transparent MMCs
    icmMmcP mmci = icmNewMMC("mmci", mmc_model, "modelAttrs", 0, 0, False);
    icmMmcP mmcd = icmNewMMC("mmcd", mmc_model, "modelAttrs", 0, 0, False);
```

The final argument to `icmNewMMC` specifies whether the MMC is *transparent* or *full*. In full mode, content as well as tags are modeled, so it is possible for the system to demonstrate incoherency effects.

9.4.3 Cascaded MMC Example

Both transparent and full MMC models can be instantiated in a *cascaded* fashion, where master ports of MMCs nearer the processor are connected to slave ports of MMCs nearer the memory subsystem. This allows structures such as cache hierarchies to be easily modeled.

An example of a platform with cascaded MMCs is available at:

```
$IMPERAS_HOME/Examples/Platforms/cascadedTransparentMMC
```

This has a platform file containing:

```
    const char *vlnvRoot = 0; // when null use default library
    const char *model = icmGetVlnvString(
        vlnvRoot, "ovpworld.org", "processor", "or1k", "1.0", "model"
    );
    const char *semihosting = icmGetVlnvString(
        vlnvRoot, "ovpworld.org", "semihosting", "or1kNewlib", "1.0", "model"
    );
    const char *mmc_model = icmGetVlnvString(
        vlnvRoot, "ovpworld.org", "mmc", "wb_1way_32byteline_2048tags", "1.0",
        "model"
    );

    // initialize CpuManager
    icmInitPlatform(ICM_VERSION, 0, 0, 0, “platform”);

    // create a processor
    icmProcessorP cpu1h = icmNewProcessor(
        "cpu1",             // CPU name
        "or1k",             // CPU type
        0,                  // CPU cpuId
        0,                  // CPU model flags
        32,                 // address bits
        model,              // model file
        0,                  // not used
        0,                  // simulation attributes
        0,                  // user-defined attributes
        semihosting,        // semi-hosting file
        0                   // not used
    );

    // create transparent MMCs
    icmMmcP mmcL1I = icmNewMMC("mmcL1I", mmc_model, "modelAttrs", 0, 0, True);
    icmMmcP mmcL1D = icmNewMMC("mmcL1D", mmc_model, "modelAttrs", 0, 0, True);
    icmMmcP mmcL2  = icmNewMMC("mmcL2",  mmc_model, "modelAttrs", 0, 0, True);

    // create busses
    icmBusP PIbus  = icmNewBus("PIbus",  32);
    icmBusP PDbus  = icmNewBus("PDbus",  32);
    icmBusP L1Ibus = icmNewBus("L1Ibus", 32);
    icmBusP L1Dbus = icmNewBus("L1Dbus", 32);
    icmBusP mbus   = icmNewBus("mbus",   32);

    // connect processor busses
    icmConnectProcessorBusses(cpu1h, PIbus, PDbus);

    // connect L1 MMCs
    icmConnectMMCBus(mmcL1I, L1Ibus, "mp1", True);
    icmConnectMMCBus(mmcL1I, PIbus,  "sp1", False);
    icmConnectMMCBus(mmcL1D, L1Dbus, "mp1", True);
    icmConnectMMCBus(mmcL1D, PDbus,  "sp1", False);

    // connect L2 MMC
    icmConnectMMCBus(mmcL2, mbus,   "mp1", True);
    icmConnectMMCBus(mmcL2, L1Ibus, "sp1", False);
    icmConnectMMCBus(mmcL2, L1Dbus, "sp2", False);

    // create two simulated memories for low and high regions
    icmMemoryP memory1 = icmNewMemory("mem1", ICM_PRIV_RWX, 0x003fffff);
    icmMemoryP memory2 = icmNewMemory("mem2", ICM_PRIV_RWX, 0xffffffff-0xf0000000);

    // connect memories to bus
    icmConnectMemoryToBus(mbus, "mp1", memory1, 0);
    icmConnectMemoryToBus(mbus, "mp2", memory2, 0xf0000000);
```

```
    // run until exit
    icmSimulatePlatform();

    // free simulation data structures
    icmTerminate();
```

This example defines three MMC objects representing L1 instruction cache, L1 data cache and L2 shared cache. In the example as written, all three caches are modeled as *transparent*, but it is possible to have combinations of transparent and full models in the same simulation, with the restriction that *transparent models must be closer to the processor than full models*. For example, all of these are legal combinations:

1. L1 instruction, L1 data and L2 all *transparent*;
2. L1 instruction, L1 data and L2 all *full*;
3. L1 instruction and L1 data *transparent*; L2 *full*.

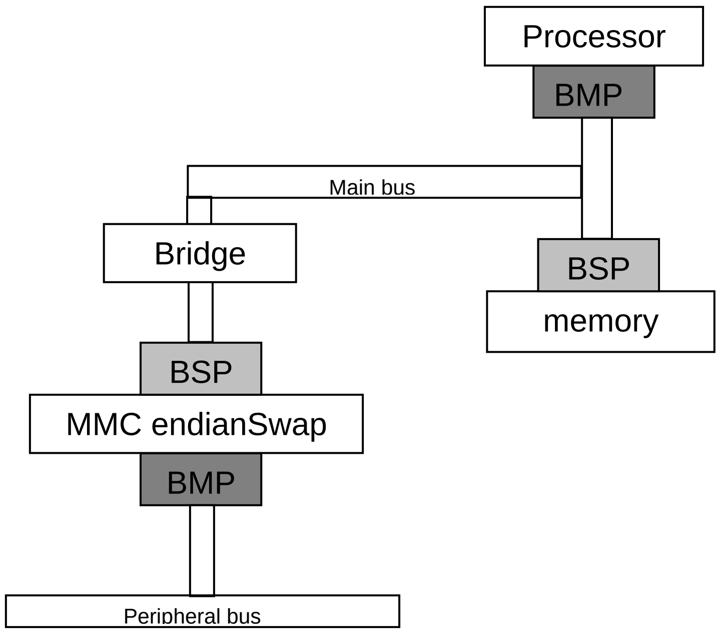
It is however not legal to try to model either L1 cache as a *full* model when the L2 cache is *transparent*.

10 Byte Swapping (Endian Correction)

A bus controller in a real platform might have the ability to perform byte-swapping on each bus cycle. This allows, for example, a big-endian processor to communicate with a little-endian peripheral component. CpuManager supports byte swapping through the use of an MMC. The bus is broken into two and an MMC inserted between the two parts.

10.1 Bus Connections

An MMC creates a one-way connection between two busses, accepting bus cycles from one bus and passing them to another. An MMC cannot perform address decoding so is activated by accesses to all addresses. If the swapping function is required for a limited address range, a bus bridge is used to decode the required range, and its output passed to the MMC.



This diagram illustrates the example in

```
$IMPERAS_HOME/Examples/Platforms/byteSwapper
```

The OR1K processor uses two RAMs (one shown) for program and stack. The bridge maps a limited address range from the main bus onto an intermediate bus which is connected to the MMC model *endianSwap* which can be found in the ovpworld.org mmc library. A simple peripheral model (not shown) is connected to the peripheral bus.

Thus, the processor has direct access to its memory without byte-swapping, but a 32-bit access (read or write) to the peripheral will have its bytes reversed.

Note that in this design, a bus master on the peripheral bus will be unable to access the processor memory.

The platform is constructed in

```
$IMPERAS_HOME/Examples/Platforms/byteSwapper/platform/platform.c
```

10.2 Bus bridge

Construction of the processor, memory and peripheral components has already been covered. The function *icmNewBusBridge* creates a bus bridge:

```
icmNewBusBridge(
    icmBusP_1,       // connection to incoming bus
    icmBusP_2,       // connection to outgoing bus
   “bridge1”,        // name of this bridge
   "decoder_sp1",    // name of slave port
   "decoder_mp1",    // name of master port
   0,                // low address of mapped region on the outgoing bus
   0xf,              // high address of mapped region on the outgoing bus
   0x80000000        // base address of mapped region on the incoming bus
);
```

The bus bridge is a generic component (it does not exist in a library) which maps part or all of the address space of one bus to the address space of another. Note that in this example *incoming* refers to the bus which is connected to the bus master, *outgoing* is the bus which is connected to the slaves. The port names are for documentation only, but should be unique on their respective busses.

10.2.1 Aliasing

A bus bridge can be used to alias a region of an address space to another region on the same bus. This example models the effect of not connecting the most significant address bit of a 32-bit bus: addresses in the top half of the address space are mapped to the bottom half.

```
icmNewBusBridge(
   bus1,             // connection to incoming bus
   bus1,             // connection to outgoing bus (the same bus)
   “bridge1”,        // name of this bridge
   "sp1",            // name of slave port
   "mp1",            // name of master port
   0,                // low address of mapped region on the outgoing bus
   0x7fffffff,       // high address of mapped region on the outgoing bus
   0x80000000        // base address of mapped region on the incoming bus
);
```

10.3 Performance considerations

In the simulator, byte swapping converts a memory access to a function call, hence a byte-swapper model should be used with care; a byte-swapper placed between a processor and its main memory (program or data) will severely restrict its performance.

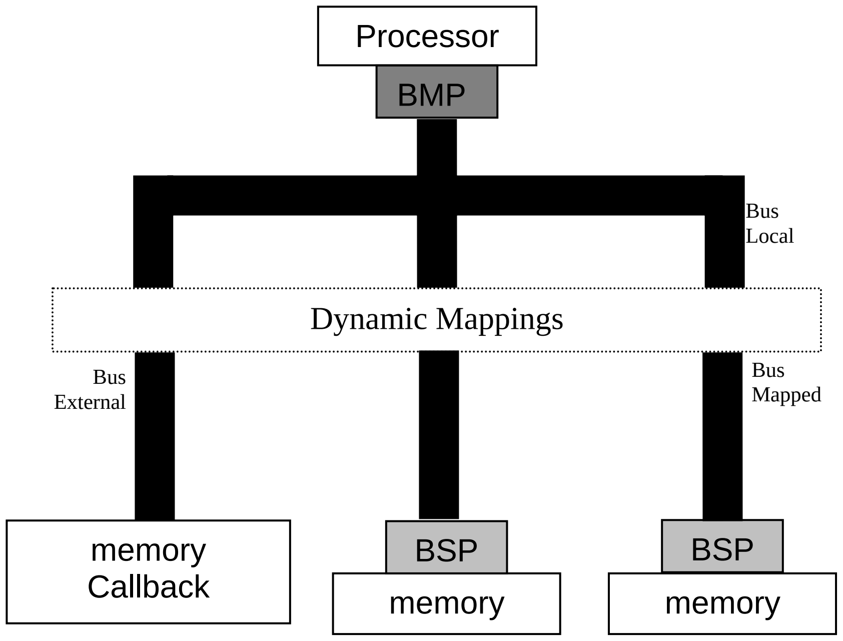
However, putting a byte-swapper between a processor and a peripheral model will cause minimal effect when the peripheral is itself modeled by function calls.

11 Dynamic Bus Bridges

A dynamic bus bridge is used in a similar way as a bus bridge, described in the previous section, but allows dynamic changes to the address space visible on a bus to be created. It is a generic component (it does not exist in a library) which maps part or all of the address space of one bus to the address space of another.

A dynamic bus bridge creates a mapping between two busses that, essentially, makes the region on the slave bus appear directly connected onto the master bus at the address range specified.

Any previously bridged addresses within a new mapped region are removed. However, the underlying memory of a mapping is not affected so that a subsequent mapping back onto an address region will make the same memory visible once again.



This diagram illustrates the example in

```
$IMPERAS_HOME/Examples/Platforms/dynamicBridges
```

The bridge is initially used to map the full extent of the processor address map to the ‘mapped’ bus. As the program executes the buses are dynamically re-mapped so that an address region accessed by the program is

1. mapped from the ‘mapped’ bus to the ‘external’ bus
2. mapped back from the ‘external’ bus to the ‘mapped’ bus, allowing previous values to be accessed.
3. unmapped, so that an access to the region will create a memory fault.

The platform is constructed in

```
$IMPERAS_HOME/Examples/Platforms/dynamicBridges/platform/platform.c
```

The function *icmBridgebuses* creates a dynamic bus bridge:

```
icmBridgebuses(
   busLocal,     // mapping on master bus, incoming bus
   busMapped,    // connection to slave, outgoing bus
   0x00400000,  // low address of mapped region on the outgoing bus
   0x0040000f,  // high address of mapped region on the outgoing bus
   0x00400000   // base address of mapped region on the incoming bus
);
```

Note that in this example *incoming* refers to the bus which is connected to the bus master, *outgoing* is the bus which is connected to the slaves.

12 Attaching a Debugger

It is possible to attach a debugger that uses the gdb RSP protocol to a processor in a CpuManager or OVPsim simulation. CpuManager offers more functionality than OVPsim:

Simulator	**Features**
OVPsim	Single gdb connection. If platform has more than 1 processor, must use `icmDebugThisProcessor` to specify which to debug
CpuManager	Up to 8 gdb connections. Need not use `icmDebugThisProcessor` if less than 8. Connections are offered in instance order.

In order to use RSP, `icmInitPlatform` must be passed the debug protocol (currently, only "rsp" is supported) and port number as arguments. The port number can be specified by giving a number greater than zero, or the allocation can be left to the host operating system by specifying a port number of zero.

If using OVPsim, or using CpuManager with more than 8 processors, or using CpuManager and connecting the debugger to the processors in other than instance order, then use `icmDebugThisProcessor`:

For example:

```
icmInitPlatform(ICM_VERSION, attributes, "rsp", portNum, “plat1”);
....
icmProcessorP processor9 = icmNewProcessor(
        "cpu9",              // CPU name
        “or1k”,              // CPU type
        9,                   // CPU cpuId
        0,                   // CPU model flags
        32,                  // address bits
        model,               // model file
        0,                   // not used
        0,                   // CPU attributes
        0,                   // user-defined attributes
        semihosting,         // semi-hosting file
        0                   // not used
);

icmDebugThisProcessor(processor9);  // specify this processor
....
```

When the ICM executable is started, it will wait for a debugger to connect on the specified port.

It is of course required to have a version of gdb specific to the target processor. The OVPWorld web site can supply a gdb for most processor models available there.

12.1 Example of attaching to GDB

This example is found in this directory:

```
$IMPERAS_HOME/Examples/Platforms/debugWithGDB
```

The following shows the program platform/platform.c:

```
    // initialize OVPsim
    unsigned int icmAttrs = ICM_INIT_DEFAULT;

    icmInitPlatform(ICM_VERSION, icmAttrs, 0, 0, “plat1”);

    // select library components
    const char *vlnvRoot = 0; // when null use default library
    const char *model = icmGetVlnvString(
        vlnvRoot, "ovpworld.org", "processor", "or1k", "1.0", "model"
    );
    const char *semihosting = icmGetVlnvString(
        vlnvRoot, "ovpworld.org", "semihosting", "or1kNewlib", "1.0", "model"
    );

    // create a processor
    icmProcessorP processor = icmNewProcessor(
        "OR1K",             // processor name
        "or1k",             // CPU type
        0,                  // processor cpuId
        0,                  // processor model flags
        32,                 // address bits
        model,              // model file
        0,                  // not used
        0,                  // no processor attributes
        0,                  // no user-defined attributes
        semihosting,        // semi-hosting file
        0                   // not used
    );
```

```
     // The simulator pauses here until connected to gdb
    icmSimulatePlatform();

    // terminate simulation
    icmTerminate();
```

The icmAttrs should include *ICM_GDB_CONSOLE* and the third argument *"rsp"* to open a GDB port connection or the command line argument --*gdbconsole* should be used.

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

```
make –C platform
make –C application
```

To start the simulation, in the `debugWithGDB` directory, run:

```
./platform/platform.${IMPERAS_ARCH}.exe \
              --program application/asmtest.OR1K.elf \
              --gdbconsole
```

You should see the following output:

```
Info (GDBT_PORT) Host: <hostname>, Port: <port number>
Info (GDBT_WAIT) Waiting for remote debugger to connect...
```

A console will be started and a connection made to the simulator debug port.

.

The simulator will display the following output after connection of the debugger.

```
Info (GDBT_CONNECTED) Client connected
```

We now have the debugger connected to the simulation and can carry out normal debugging commands supported by gdb – for example, try setting a breakpoint at main, continuing and disassembling.

For more detailed information on debugging with gdb, refer to the *Debugging Applications with GDB User Guide*.

12.2 Attaching to the remote multiprocessor debugger

As well as being integrated into the Imperas simulator, the Imperas multiprocessor debugger is available as a stand-alone program that connects to the simulator via an RSP connection, in the same way as gdb.

The procedure is identical to the previous example except that there is no need to call `icmDebugThisProcessor()`.

You may modify and re-compile the platform file to change from automatically starting the GDB console to starting the MPD console. This can be done by adding *ICM_MPD_CONSOLE* to the *icmAttrs* (and re-compiling) or by adding the command line argument --*mpdconsole*

Or if you wish you may start the simulation, in the `debugWithGDB` directory, and attach remotely without modifying the platform, run:

```
./platform/platform.${IMPERAS_ARCH}.exe \
             --program application/asmtest.OR1K.elf \
             --port 0
```

(The value 0 tells the OS to allocate a port number from its pool).
You should see the following output:

```
Info (GDBT_PORT) Host: <hostname>, Port: 5555
Info (GDBT_WAIT) Waiting for remote debugger to connect...
```

(The value 5555 was the port number selected in this run).

Run the multiprocessor debugger in a separate shell using the following command in the `debugWithGDB` directory:

```
${IMPERAS_HOME}/bin/${IMPERAS_ARCH}/mpd.exe -port 5555
```

The following output will be seen

```

                      MPD (32-Bit) version <version>
```

```
        Copyright (c) 2005-2015 Imperas Software Ltd.
                     ALL RIGHTS RESERVED

This program is proprietary and confidential information of
Imperas Software Ltd. and may be used and disclosed only as authorized
in a license agreement controlling such use and disclosure.

Info (MPD_SCS) Connecting
Info (GDBT_CONNECTED) Client connected
Info (MPD_SC) Socket connected
Info (MPD_VC) Server is compatible
idebug (OR1K) >
```

The debugger is now ready for use. Please refer to Imperas_Debugger_User_Guide.

12.3 Automatic startup of remote debuggers.

As we have seen in the previous example, the simulator can start gdb or MPD (if this option has been purchased) in a separate window. Set the simulator attributes ICM_GDB_CONSOLE or ICM_MDP_CONSOLE.

```
    // initialize CpuManager with MPD console

    icmInitPlatform(ICM_VERSION, ICM_GDB_CONSOLE, “rsp”, 0, “plat1”);
    …
or

    icmInitPlatform(ICM_VERSION, ICM_MPD_CONSOLE, “rsp”, 0, “plat1”);
    …
```

Alternatively both options may be applied using the command line parser using the arguments --*gdbconsole* and --*mpdconsole* respectively.

When simulation starts, a new window will appear with the multiprocessor debugger (or gdb) already connected to the simulator. On Windows, the console will be on the local desktop, on Linux, the console (an x-term) will follow the setting of the DISPLAY environment variable.

12.3.1 Using the control file

If your product supports use of the Imperas Control File, then the multiprocessor debugger can be invoked in one of two modes, using the simulator control file.

12.3.1.1 Integrated debugger

If the platform has access to a console (and the platform is not using the console itself) then the integrated debugger can be started. The debugger banner and prompt will appear in the console and can be used from there. Use the control file entry:

```
--idebug
```

12.3.1.2 Remote debugger

If no console is available or if the platform is using the console for other purposes, then the remote debugger must be used. A window will appear and the debugger can be used from there. Use the control file entry:

```
--mpdconsole
```

The multiprocessor debugger can also be started in TCL mode and can execute a startup script if required. Please refer to the Imperas Control File User Guide and the Imperas Debugger User Guide for more details.

12.4 Selecting the GDB

When simulating a platform in the Imperas simulator, a GDB executable can be associated with each processor type to give full symbolic debug capabilities.
Two methods of association can be used:

- Each model in the OVP processor model library contains a reference to the gdb to be used by default. If the installation includes the gdb, this will be selected automatically.
- If this gdb is not available or if another has to be used, the function `icmSetProcessorGdbPath` can set the path per instance:

```
// set the gdb path
icmSetProcessorGdbPath(
    icmProcessorP processor,    // handle to the processor instance
    const char   *path,         // full path to the GDB executable
    const char   *flags         // any flags to be appended to the GDB invocation
);
```

13 Multiprocessor Support

Any number of processors can be instantiated within an ICM platform. Shared memory resources and callbacks on mapped memory regions are used to allow communication between them.

The following section shows a simple multiprocessor platform created using bus and memory objects first introduced in section 8.3.

13.1 Example

This example is found in the `multiprocessor` directory.

```
$IMPERAS_HOME/Examples/Platforms/multiprocessor
```

The following shows the instantiation of two processors and a memory shared between them. Each processor also has a small amount of local memory for stack.

Two processors are instantiated with individual names and Id numbers.

```
    // create a processor
    icmProcessorP processor0 = icmNewProcessor(
        "cpu1",             // CPU name
        “or1k”,             // CPU type
        0,                  // CPU cpuId
        0,                  // CPU model flags
        32,                 // address bits
        model,              // model file
        0,                  // not used
        SIM_ATTRS,          // simulation attributes
        0,                  // user-defined attributes
        semihosting,        // semi-hosting file
        0                   // not used
    );

    icmProcessorP processor1 = icmNewProcessor(
        "cpu2",             // CPU name
        “or1k”,             // CPU type
        1,                  // CPU cpuId
        0,                  // CPU model flags
        32,                 // address bits
        model,              // model file
        0,                  // not used
        SIM_ATTRS,          // simulation attributes
        0,                  // user-defined attributes
        semihosting,        // semi-hosting file
        0                   // not used
    );
```

Two busses are created, one for each processor, and connected to the processors:

```
    // create the processor busses
    icmBusP bus1 = icmNewBus("bus1", 32);
    icmBusP bus2 = icmNewBus("bus2", 32);

    // connect the processor busses
    icmConnectProcessorBusses(processor0, bus1, bus1);
    icmConnectProcessorBusses(processor1, bus2, bus2);
```

This example needs three memories: a local stack memory for each processor and some shared memory. These are created and connected to the processor busses:

```
    // create memories
    icmMemoryP local1 = icmNewMemory("local1", ICM_PRIV_RWX, 0x0fffffff);
    icmMemoryP local2 = icmNewMemory("local2", ICM_PRIV_RWX, 0x0fffffff);
    icmMemoryP shared = icmNewMemory("shared", ICM_PRIV_RWX, 0xefffffff);

    // connect memories
    icmConnectMemoryToBus(bus1, "mp1", shared, 0x00000000);
    icmConnectMemoryToBus(bus2, "mp2", shared, 0x00000000);
    icmConnectMemoryToBus(bus1, "mp1", local1, 0xf0000000);
    icmConnectMemoryToBus(bus2, "mp1", local2, 0xf0000000);
```

Memory maps for multiprocessor systems can be very complex, so it is often useful to be able to show the bus connections using `icmPrintBusConnections`:

```
    // show the bus connections
    icmPrintf("\nbus1 CONNECTIONS\n");
    icmPrintBusConnections(bus1);
    icmPrintf("\nbus2 CONNECTIONS\n");
    icmPrintBusConnections(bus2);
    icmPrintf("\n");
```

The full memory map of each processor is mapped onto the shared memory object, except for a small section of local memory for each stack. The program is loaded onto both processors using the --*program* command line argument.

The platform is then simulated to completion using `icmSimulatePlatform`:

```
    // run simulation
    icmSimulatePlatform();
```

The `cpuId` defined when the processor instance is created can be accessed from within application code using the Imperas intercepted function `impProcessorId`, in order that a processor can identify itself. This is shown in the following code and used to identify messages printed from the application and also the mode in which this test application is running, reading or writing.

To enable standard Imperas function intercepts like `impProcessorId`, `ICM_ENABLE_IMPERAS_INTERCEPTS` must be passed using the 2nd argument of `icmInitPlatform`. We also set `ICM_VERBOSE` in this example, which enables simulation runtime statistics at the end of simulation:

```
    // initialize CpuManager - require Imperas intercepts because the
    // application uses impProcessorId() to get processor id
    icmInitPlatform(ICM_VERSION, ICM_VERBOSE|ICM_ENABLE_IMPERAS_INTERCEPTS, 0, 0, 0);
```

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

```
make –C platform
make –C application
```

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

```
./platform/platform.${IMPERAS_ARCH}.exe --program application/application.OR1K.elf
```

You should see the following output as the two processors execute the application. *cpu0* is generating the Fibonacci series with *cpu1* reading the results from the shared memory:

```
bus1 CONNECTIONS
BUS MASTERS: 2
   PORT 'DATA' of 'cpu0'
   PORT 'INSTRUCTION' of 'cpu0'
BUS SLAVES: 2
   0x00000000:0xefffffff: PORT 'mp1' of 'shared'
   0xf0000000:0xffffffff: PORT 'mp1' of 'local1'

bus2 CONNECTIONS
BUS MASTERS: 2
   PORT 'DATA' of 'cpu1'
   PORT 'INSTRUCTION' of 'cpu1'
BUS SLAVES: 2
   0x00000000:0xefffffff: PORT 'mp2' of 'shared'
   0xf0000000:0xffffffff: PORT 'mp1' of 'local2'

CPU 1 starting...
CPU 0 starting...
CPU 0: fib(0) = 0
CPU 1: munge(0) = 0
CPU 0: fib(1) = 1
CPU 1: munge(1) = 0
CPU 0: fib(2) = 1
CPU 1: munge(1) = 0
CPU 0: fib(3) = 2
CPU 1: munge(2) = 1
CPU 0: fib(4) = 3
CPU 1: munge(3) = 3
CPU 0: fib(5) = 5
CPU 1: munge(5) = 10

... etc ...

CPU 0: fib(33) = 3524578
CPU 1: munge(3524578) = 800566737
CPU 0: fib(34) = 5702887
CPU 1: munge(5702887) = 711033285
processor0 has executed 1658900172 instructions
processor1 has executed 1658997966 instructions
Info
Info ------------------------------------------------
Info CPU 'cpu1' STATISTICS
Info   Type                  : or1k
Info   Nominal MIPS          : 100
Info   Final program counter : 0x1dcc
Info   Simulated instructions: 1,658,997,966
Info   Simulated MIPS        : 781.2
Info CPU 'cpu0' STATISTICS
Info   Type                  : or1k
Info   Nominal MIPS          : 100
Info   Final program counter : 0x1dcc
Info   Simulated instructions: 1,658,900,172
Info   Simulated MIPS        : 781.2
Info TOTAL
Info   Simulated instructions: 3,317,898,138
Info   Simulated MIPS        : 1562.4
Info ------------------------------------------------
Info
Info ------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 16.59 seconds
```

.

```
Info   User time              : 2.12 seconds
Info   System time            : 0.00 seconds
Info   Elapsed time           : 2.12 seconds
Info   Real time ratio        : 7.81x faster
Info -------------------------------------------------
```

13.2 Verbose Output

Note that the log from example `multiprocessor` includes output at the end giving statistics about the number of instructions executed by each processor, the simulated MIPS rate for each processor, and the total instructions and MIPS rate. This information is present because `ICM_VERBOSE` was specified as an option to `icmInitPlatform`:

```
    icmInitPlatform(ICM_VERSION, ICM_VERBOSE|ICM_ENABLE_IMPERAS_INTERCEPTS, 0, 0, “plt1”);
```

The actual performance reported may vary and depends on the performance of the native host. In this example (run on a 3.4Ghz Dell Core i7-3770 desktop machine) the overall simulation speed is about 1560 simulated OR1K MIPS, approximately half for each processor.

13.2.1 Simulation Time Statistics

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

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

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

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

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

```
Info CPU 'cpu1' STATISTICS
Info   Type                   : or1k
```

.

```
Info   Nominal MIPS           : 100
Info   Final program counter : 0x1dcc
Info   Simulated instructions: 1,658,997,966
Info   Simulated MIPS         : 781.2
```

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

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

13.3 Standard Multiprocessor Scheduling Algorithm

This example used the standard multiprocessor scheduling algorithm built-in to the simulator under `icmSimulatePlatform`. This 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:

13.3.1 Changing the Time Slice Size

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

```
Bool icmSetSimulationTimeSlice(icmTime newSliceSize);
```

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

13.3.2 Changing Processor Nominal MIPS Rate

The nominal MIPS rate for each processor can be set with a user attribute. See section 5 for an example of this.

13.3.3 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 `icmSimulate` for each processor. This function will simulate a specified processor for an exact number of instructions.

13.4 Many Core Example

This example shows the instantiation of many (default 24) processors in a platform and is found in the `manycore` directory.

```
$IMPERAS_HOME/Examples/Platforms/manycore
```

The following shows the instantiation of many processors in a loop as part of a sub-system with local memory.

The processors are instantiated with individual names and Id numbers.

```
   for (i=0; i<PROCESSOR_COUNT; i++ ) {
      // create processor cpu<i>
      sprintf(name, "cpu%d", i);
      processor[i] = icmNewProcessor(
            name,               // CPU name
            "or1k",             // CPU type
            i,                  // CPU cpuId
            0,                  // CPU model flags
            32,                 // address bits
            model,              // model file
            "modelAttrs",       // morpher attributes
            SIM_ATTRS,          // simulation attributes. enable tracing etc
            0,                  // user-defined attributes
            semihosting,        // semi-hosting file
            "modelAttrs"        // semi-hosting attributes
      );
```

The busses are created, one for each processor, and connected to the processors:

```
   // create the processor busses
   sprintf(name, "bus%d", i);
   bus[i] = icmNewBus(name, 32);

   // connect the processor busses
   icmConnectProcessorBusses(processor[i], bus[i], bus[i]);
```

This example has a single memory for each sub-system. These are created and connected to the processor busses:

```
   // create memory
   sprintf(name, "memory%d", i);
   memory[i] = icmNewMemory(name, ICM_PRIV_RWX, 0xffffffff);

   // connect memory
   icmConnectMemoryToBus(bus[i], "mp1", memory[i], 0x00000000);
```

We load the program into each processor’s memory and set the start address to the entry indicated in the program file loaded. If the program is not loaded successfully we print an error message and exit.

```
   icmLoaderAttrs loadAttrs = ICM_LOAD_VERBOSE|ICM_SET_START;
   if(!icmLoadProcessorMemory(processor[i], argv[1], loadAttrs, False, True)
   ) {
      icmMessage("E", "PLATFORM_LOAD", "Failed to load %s onto processor %d", argv[1], i);
      // terminate simulation and free simulation data structures
      icmTerminate();
      return -1;
   }
```

The platform is then simulated to completion using `icmSimulatePlatform`:

```
    // run simulation
    icmSimulatePlatform();
```

Compile the test platform and application (fibonacci) as before using the following commands in the `manycore` directory:

```
make –C platform
make –C application
```

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

```
./platform/platform.${IMPERAS_ARCH}.exe --program application/fibonacci.OR1K.elf
```

You should see the following output as the processors execute the application.

Each processor sub-system bus is printed

```
BUS[0] CONNECTIONS
BUS MASTERS: 2
   PORT 'DATA' of 'platform/cpu0'
   PORT 'INSTRUCTION' of 'platform/cpu0'
BUS SLAVES: 1
   0x00000000:0xffffffff: PORT 'mp1' of 'memory0'
```

Each processor loads the program into its local memory in its sub-system.

```
Info (OR_OF) Target 'platform/cpu0' has object file read from
'application/fibonacci.OR1K.elf'
Info (OR_PH) Program Headers:
Info (OR_PH) Type           Offset     VirtAddr   PhysAddr   FileSiz    MemSiz     Flags
Align
Info (OR_PD) LOAD           0x00002000 0x00000000 0x00000000 0x0000e080 0x0000e194 RWE
2000
```

Each processor starts executing the program as it is scheduled (only partial output is shown).

```

CPU 23 starting...
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
... snip ...
fib(13) = 233
CPU 21 starting...
```

```
fib(0) = 0
fib(1) = 1
fib(2) = 1
... snip ...
fib(12) = 144
fib(13) = 233
CPU 20 starting...
fib(0) = 0
fib(1) = 1
... snip ...
fib(12) = 144
fib(13) = 233
CPU 19 starting...
... snip ...
fib(18) = 2584
fib(18) = 2584
fib(18) = 2584
fib(18) = 2584
fib(18) = 2584
fib(18) = 2584
fib(18) = 2584
fib(18) = 2584
fib(18) = 2584
fib(18) = 2584
fib(18) = 2584
fib(18) = 2584
fib(18) = 2584
fib(18) = 2584
fib(18) = 2584
fib(18) = 2584
fib(18) = 2584
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(19) = 4181
fib(20) = 6765
fib(20) = 6765
fib(20) = 6765
fib(20) = 6765
fib(20) = 6765
fib(20) = 6765
fib(20) = 6765
fib(20) = 6765
fib(20) = 6765
fib(20) = 6765
fib(20) = 6765
... snip ...
fib(31) = 1346269
CPU 23 finishing...
fib(31) = 1346269
CPU 22 finishing...
```

```
fib(31) = 1346269
CPU 21 finishing...
fib(31) = 1346269
... snip ...
fib(31) = 1346269
CPU 0 finishing...
```

Output statistics are shown for each processor with the overall platform simulation statistics shown last.

```
Info ---------------------------------------------------
Info CPU 'platform/cpu23' STATISTICS
Info   Type                   : or1k
Info   Nominal MIPS           : 100
Info   Final program counter : 0x1ab4
Info   Simulated instructions: 353,720,711
Info   Simulated MIPS         : 30.8
Info ---------------------------------------------------
Info
Info ---------------------------------------------------
Info CPU 'platform/cpu22' STATISTICS
Info   Type                   : or1k
Info   Nominal MIPS           : 100
Info   Final program counter : 0x1ab4
Info   Simulated instructions: 353,720,711
Info   Simulated MIPS         : 30.8
Info ---------------------------------------------------
... snip ...
Info
Info ---------------------------------------------------
Info CPU 'platform/cpu0' STATISTICS
Info   Type                   : or1k
Info   Nominal MIPS           : 100
Info   Final program counter : 0x1ab4
Info   Simulated instructions: 353,719,241
Info   Simulated MIPS         : 30.8
Info ---------------------------------------------------
Info
Info ---------------------------------------------------
Info TOTAL
Info   Simulated instructions: 8,489,280,698
Info   Simulated MIPS         : 738.2
Info ---------------------------------------------------
Info
Info ---------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time         : 3.54 seconds
Info   User time              : 11.49 seconds
Info   System time            : 0.01 seconds
Info   Elapsed time           : 11.78 seconds
Info ---------------------------------------------------
```

14 QuantumLeap Multiprocessor Support

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.

The following section shows how to enable the QuantumLeap algorithm and describes how performance and results are affected.

14.1 Example

This example is again found in the `multiprocessor` directory.

```
$IMPERAS_HOME/Examples/Platforms/multiprocessor
```

Refer to section 13 for a detailed description of the application and platform. Compile the test platform and application as before using the following commands in the `multiprocessor` directory:

```
make –C platform
make –C application
```

Enable the QuantumLeap algorithm using a *control file* (control files are described in section 23):

```
echo '–parallel' > control.ic
export IMPERAS_TOOLS=control.ic
export IMPERAS_RUNTIME=CpuManager
```

Note that QuantumLeap is only supported as a licensed feature of the Imperas Professional Tools; contact Imperas for details.

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

```
./platform/platform.${IMPERAS_ARCH}.exe --program application/application.OR1K.elf
```

You should see the following output as the two processors execute the application. *cpu0* is generating the Fibonacci series with *cpu1* reading the results from the shared memory:

```
CpuManagerMulti Parallel started: Fri Nov 29 09:32:32 2013

. . .

bus1 CONNECTIONS
BUS MASTERS: 2
   PORT 'DATA' of 'cpu0'
   PORT 'INSTRUCTION' of 'cpu0'
BUS SLAVES: 2
   0x00000000:0xefffffff: PORT 'mp1' of 'shared'
   0xf0000000:0xffffffff: PORT 'mp1' of 'local1'

bus2 CONNECTIONS
BUS MASTERS: 2
```

```
   PORT 'DATA' of 'cpu1'
   PORT 'INSTRUCTION' of 'cpu1'
BUS SLAVES: 2
   0x00000000:0xefffffff: PORT 'mp2' of 'shared'
   0xf0000000:0xffffffff: PORT 'mp1' of 'local2'

CPU 1 starting...
CPU 0 starting...
CPU 0: fib(0) = 0
CPU 1: munge(0) = 0
CPU 0: fib(1) = 1
CPU 1: munge(1) = 0
CPU 0: fib(2) = 1
CPU 1: munge(1) = 0
CPU 0: fib(3) = 2
CPU 1: munge(2) = 1
CPU 0: fib(4) = 3
CPU 1: munge(3) = 3
CPU 0: fib(5) = 5
CPU 1: munge(5) = 10

... etc ...

CPU 0: fib(33) = 3524578
CPU 1: munge(3524578) = 800566737
CPU 0: fib(34) = 5702887
CPU 1: munge(5702887) = 711033285
processor0 has executed 1658900172 instructions
processor1 has executed 1658997966 instructions
Info
Info ------------------------------------------------
Info CPU 'cpu1' STATISTICS
Info   Type                  : or1k
Info   Nominal MIPS          : 100
Info   Final program counter : 0x1dcc
Info   Simulated instructions: 1,660,597,974
Info   Simulated MIPS        : 1300.3
Info CPU 'cpu0' STATISTICS
Info   Type                  : or1k
Info   Nominal MIPS          : 100
Info   Final program counter : 0x1dcc
Info   Simulated instructions: 1,660,600,176
Info   Simulated MIPS        : 1300.3
Info TOTAL
Info   Simulated instructions: 3,321,198,150
Info   Simulated MIPS        : 2600.7
Info ------------------------------------------------
Info
Info ------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 16.61 seconds
Info   User time             : 2.19 seconds
Info   System time           : 0.35 seconds
Info   Elapsed time          : 1.28 seconds
Info   Real time ratio       : 13.00x faster
Info ------------------------------------------------

CpuManagerMulti Parallel finished: Fri Nov 29 09:32:33 2013
```

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

14.2 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 Core i7-3770 desktop machine) the overall simulation speed is about 2600 simulated OR1K MIPS, approximately half for each processor. *This is almost twice as fast as the same application run without QuantumLeap in section 13.*

14.3 QuantumLeap Scheduling Algorithm

The QuantumLeap scheduling algorithm is similar in many respects to the standard multiprocessor scheduling algorithm described in section 13.3. 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 sometime 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.*

14.4 QuantumLeap Options

Control file arguments `--parallelopt`, `--parallelthreads` and `--parallelmax` can be used to control details of the simulation, as described below.

14.4.1 Option `--parallelopt`

QuantumLeap algorithm behavior can be modified using option `--parallelopt` in a control file. 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 native core to avoid costs involved in synchronizing caches that can occur when native processes are moved from one native core to another. Setting this bit disables affinity fixing so that simulated core processes can migrate between native cores.

The default value of `–parallelopt` is 1, specifying *nice mode simulation* and *fixed affinities*.

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

```
echo '–parallel' > control.ic
echo '–parallelopt 2' >> control.ic
export IMPERAS_TOOLS=control.ic
export IMPERAS_RUNTIME=CpuManager
./platform/platform.${IMPERAS_ARCH}.exe --program application/application.OR1K.elf
```

14.4.2 Option `--parallelthreads`

QuantumLeap option `-parallelthreads` can be used in a control file to specify *the maximum number of parallel threads that should simulate 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.

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

```
echo '–parallel' > control.ic
echo '–parallelthreads 3' >> control.ic
export IMPERAS_TOOLS=control.ic
export IMPERAS_RUNTIME=CpuManager
./platform/platform.${IMPERAS_ARCH}.exe --program application/application.OR1K.elf
```

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

14.4.3 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.

Example

To run a simulation using maximum parallelization:

```
echo '–parallel' > control.ic
echo '–parallelmax' >> control.ic
export IMPERAS_TOOLS=control.ic
export IMPERAS_RUNTIME=CpuManager
./platform/platform.${IMPERAS_ARCH}.exe --program application/application.OR1K.elf
```

15 Limiting Performance to Wall Clock Time

In the above example, we saw that a pair of OR1K processors with a nominal speed of 100 MIPS could be made to run at over 800 MIPS (combined). Although it is usually a benefit to have better-than-real-time simulation performance, there are some occasions when this is undesirable: for example, when simulating an OS such as Linux, processors are almost entirely idle when waiting at a login prompt. Unless told otherwise, the simulator will move simulated time rapidly forward when processors are idling. The effect of this is that it is impossible to log in interactively to the simulated Linux, because the log in times out instantly as simulated time shoots forward.

It is possible to restrict maximum performance to any multiple of the real time clock using the function `icmSetWallClockFactor`:

```
void icmSetWallClockFactor(double factor);
```

The factor specifies the maximum multiple of real time at which the simulator should run. For example, a value of 3.0 implies no more than three times real time, and a value of 0.5 specifies no more than half real time. The following example uses this function to restrict the multiprocessor platform performance to a fixed factor of real time.

15.1.1 Example 2 – Wallclock Simulation

This example is found in the `wallclock` directory.

```
$IMPERAS_HOME/Examples/Platforms/wallclock
```

The following example is exactly the same as the previous one except that after initialization a new call restricts the simulation performance to no more than two times real time:

```
    // limit performance to no more than 2x nominal speed
    icmSetWallClockFactor(2);
```

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

```
make –C platform
make –C application
```

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

```
./platform/platform.${IMPERAS_ARCH}.exe --program application/application.OR1K.elf
```

The output should be similar to this:

```
processor0 has executed 1658900172 instructions
processor1 has executed 1658997966 instructions
Info
Info -------------------------------------------------
Info CPU ‘cpu1’ STATISTICS
Info   Type                   : or1k
Info   Nominal MIPS           : 100
Info   Final program counter : 0x1dcc
Info   Simulated instructions: 1,658,997,966
Info   Simulated MIPS         : 199.9
Info CPU ‘cpu0’ STATISTICS
Info   Type                   : or1k
Info   Nominal MIPS           : 100
Info   Final program counter : 0x1dcc
Info   Simulated instructions: 1,658,900,172
Info   Simulated MIPS         : 199.9
Info TOTAL
Info   Simulated instructions: 3,317,898,138
Info   Simulated MIPS         : 399.9
Info -------------------------------------------------
Info
Info -------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time         : 16.59 seconds
Info   User time              : 6.53 seconds
Info   System time            : 0.00 seconds
Info   Elapsed time           : 8.30 seconds
Info   Host utilization       : 49.0% (wallclock enabled)
Info -------------------------------------------------
```

Note that each processor is now running at almost exactly 200 simulated MIPS (i.e. twice the specified nominal MIPS). In the simulation time statistics, elapsed time has increased, because the simulation had to spend some time waiting in order not to exceed the specified maximum multiple of real time. A new *host utilization* line indicates how heavily the host processor was used by the simulation process. In this example, a utilization of 49.0% indicates that the simulation process spent approximately half of its time waiting.

16 Interrupting Simulation

Normally, `icmSimulate` and `icmSimulatePlatform` will run until they have completed the requested number of simulated instructions (for `icmSimulate`) or time has advanced until the time specified by `icmSetSimulationStopTime` (for `icmSimulatePlatform`), or until a processor model has performed some explicit action that terminates the simulation loop early (for example, halting or exiting).

16.1 Cntrl-C Handler

Occasionally, it may be required that the `icmSimulate` call be terminated early by some external event. For example, the platform 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 `icmInterrupt` 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) {
    icmInterrupt();
}

//
// 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 as this example is running, a call to `icmInterrupt` will be generated. This will cause any active `icmSimulate` or `icmSimulatePlatform` call to return, and the `stopReason` for the processor that stops will be set to `ICM_SR_INTERRUPT`. This needs to be handled in the main routine, for example:

```
    icmProcessorP stoppedProcessor;

    // simulate until done or ctrl-C
    while((stoppedProcessor=icmSimulatePlatform())) {
       if(icmGetStopReason(stoppedProcessor)==ICM_SR_INTERRUPT) {
           icmPrintf(
               "%s: interrupt after " FMT_64u " instructions...\n",
               icmGetProcessorName(stoppedProcessor),
               icmGetProcessorICount(stoppedProcessor)
           );
        } else {
           break;
       }
    }
```

In this example, when an interrupt occurs, the platform prints a message and continues simulation by calling `icmSimulatePlatform` 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 ICM API provides a method that does not require OS-specific signal handler code: simply specify `ICM_STOP_ON_CTRLC` as an attribute in `icmInitPlatform`:

```
icmInitPlatform(
    ICM_VERSION,
    ICM_VERBOSE|ICM_ENABLE_IMPERAS_INTERCEPTS|ICM_STOP_ON_CTRLC,
    0,
    0,
    “plat1”
);
```

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

16.2 Causing Processor to Yield

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

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

```
static ICM_MEM_WRITE_FN(watchWriteCB) {


    icmPrintf(
        "WATCHCALLBACK '%s': Writing to 0x%08x : Interrupt\n",
        (Uns8 *)userData,
        (Int32)address
    );

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

16.3 Example

This example is found in the `interruptSimulation` directory.

```
$IMPERAS_HOME/Examples/Platforms/interruptSimulation
```

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

```
make –C platform
make –C application
```

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

```
./platform/platform.${IMPERAS_ARCH}.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 icmInterrupt() in the platform. This will cause the simulator to return and a message will be generated of the form:

```
CPU 0: Watchpoint Trigger at 18
WATCHCALLBACK 'cpu0': Writing to 0xeffffff0 : Interrupt
/cpu0: interrupt after 2042123 instructions...
```

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

```
    cpu0: yield after <number> instructions…
```

or:

```
    cpu1: yield after <number> instructions…
```

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

16.4 Important Notes

16.4.1 ICM 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 icmSimulate or icmSimulatePlatform functions the icmInterrupt may be used. However, it is important that no other ICM API calls should be made from within a handler of this type. To do so may result in unexpected behavior.

16.4.2 icmInterrupt Usage

One important point about `icmInterrupt` is that it is *not intended to be asynchronously thread-safe*. In other words, it is not appropriate to asynchronously call `icmInterrupt` when the simulation thread is not suspended. In ICM applications 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 `icmInterrupt` 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.

17 Interrupting a Specific 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 a port (perhaps called *reset*).

Events are signaled to processor models using nets, which can be created by `icmNewNet` and connected to processor instances using `icmConnectProcessorNet`. A value can be written to a net using `icmWriteNet`.

The following example shows how a processor reset signal can be stimulated using nets.

17.1 Example

17.1.1 Reset Processor

This example is found in the `interruptProcessor` directory.

```
$IMPERAS_HOME/Examples/Platforms/interruptProcessor
```

The following CpuManager / OVPsim code creates a platform that instantiates a single processor running an application. The application is run for 100,000 instructions and then a reset by writing to a net.

The main routine is as follows (with sections relevant to this example in bold):

```
    // create processor cpu0
    icmProcessorP processor0 = icmNewProcessor(
        "cpu0",             // CPU name
        "or1k",             // CPU type
        0,                  // CPU cpuId
        0,                  // CPU model flags
        32,                 // address bits
        model,               // model file
        0,                  // not used
        SIM_ATTRS,          // simulation attributes
        0,                  // user-defined attributes
        semihosting,        // semi-hosting file
        0                   // not used
    );

    // load the processor object file
    icmLoadProcessorMemory(processor0, argv[1], ICM_LOAD_DEFAULT, False, True);

    // create a reset net and connect it to the reset port of processor0
    icmNetP resetNet = icmNewNet("resetNet");
    icmConnectProcessorNet(processor0, resetNet, "reset", ICM_INPUT);
```

```
    // simulate for one simulated millisecond
    icmSetSimulationStopTime(0.001);
    icmSimulatePlatform();

    // write to the processor reset signal
    icmWriteNet(resetNet, 1);
    icmPrintf(
        "processor0 reset after " FMT_64u " instructions\n",
        icmGetProcessorICount(processor0)
    );

    // simulate until completion
    icmSimulatePlatform();

    // report the total number of instructions executed
    icmPrintf(
        "processor0 has executed " FMT_64u " instructions\n",
        icmGetProcessorICount(processor0)
    );
```

The example creates a net object and connects it to the *reset* input port of the OR1K processor instance as follows:

```
    icmNetP resetNet = icmNewNet("resetNet");
    icmConnectProcessorNet(processor0, resetNet, "reset", ICM_INPUT);
```

We then simulate for one simulated millisecond:

```
    icmSetSimulationStopTime(0.001);
    icmSimulatePlatform();
```

The reset net is then stimulated (note that this signal is level-sensitive for the OR1K model, so the value is written high to activate the reset and lowered afterwards):

```
    icmWriteNet(resetNet, 1);
    icmWriteNet(resetNet, 0);
```

Finally, we then simulate again until the processor terminates:

```
    icmSimulatePlatform();
```

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

```
make –C platform
make –C application
```

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

```
./platform/platform.${IMPERAS_ARCH}.exe --program application/application.OR1K.elf
```

You should see output as follows:

```
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
fib(10) = 55
fib(11) = 89
fib(12) = 144
fib(13) = 233
processor0 reset after 100000 instructions
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
fib(10) = 55
fib(11) = 89
fib(12) = 144
fib(13) = 233
fib(14) = 377
fib(15) = 610
fib(16) = 987
fib(17) = 1597
fib(18) = 2584
fib(19) = 4181
processor0 has executed 1258786 instructions
```

Note that after the processor has executed 100,000 instructions, it resets and begins calculation of the fibonacci series again.

17.1.2 Startup Reset

This example is found in the `resetControl` directory.

```
$IMPERAS_HOME/Examples/Platforms/resetControl
```

The following CpuManager / OVPsim code creates a platform that instantiates two processors each running the same application. Each processor is attached to an independent reset line. Before the simulation is started one reset line is release and the second is asserted. The application is run for 0.01 seconds and the second reset line is released; the simulation is then run for a further 0.01 seconds.

The main routine is as follows (with sections relevant to this example in bold):

```
    // create a processor
    processor[0]= icmNewProcessor (
        "cpu0",             // CPU name
        "or1k",             // CPU type
        0,                  // CPU cpuId
        0,                  // CPU model flags
        32,                 // address bits
        model,              // model file
        0,                  // not used
        SIM_FLAGS,          // enable tracing etc
        userAttrs,          // user-defined attributes
        semihosting,        // semi-hosting file
        0                   // not used
    );
    // create a processor
    processor[1] = icmNewProcessor (
        "cpu1",             // CPU name
        "or1k",             // CPU type
        1,                  // CPU cpuId
        0,                  // CPU model flags
        32,                 // address bits
        model,              // model file
        0,                  // not used
        SIM_FLAGS,          // enable tracing etc
        userAttrs,          // user-defined attributes
        semihosting,        // semi-hosting file
        0                   // not used
    );


    icmBusP bus0 = icmNewBus("bus0", 32);
    icmBusP bus1 = icmNewBus("bus1", 32);

    icmConnectProcessorBusses(processor[0], bus0, bus0);
    icmConnectProcessorBusses(processor[1], bus1, bus1);

    icmMemoryP memLow0 = icmNewMemory("memLow0", ICM_PRIV_RWX, 0x0fffffff);
    icmConnectMemoryToBus(bus0,"sp1",memLow0, 0x00000000);
    icmMemoryP memLow1 = icmNewMemory("memLow1", ICM_PRIV_RWX, 0x0fffffff);
    icmConnectMemoryToBus(bus1,"sp1",memLow1, 0x00000000);

    icmMemoryP memHigh0 = icmNewMemory("memHigh0", ICM_PRIV_RWX, 0xdfffffff);
    icmConnectMemoryToBus(bus0,"sp1",memHigh0, 0x20000000);
    icmMemoryP memHigh1 = icmNewMemory("memHigh1", ICM_PRIV_RWX, 0xdfffffff);
    icmConnectMemoryToBus(bus1,"sp1",memHigh1, 0x20000000);

    // load the processor object file
    if(!icmLoadProcessorMemory(processor[0], argv[1], ICM_LOAD_DEFAULT, False, True)){
        icmPrintf("Application %s not loaded by processor0\n", argv[1]);
        return -1;
    }
    if(!icmLoadProcessorMemory(processor[1], argv[1], ICM_LOAD_DEFAULT, False, True)){
        icmPrintf("Application %s not loaded by processor1\n", argv[1]);
        return -1;
    }


    icmWriteNet(reset0, 0);
    icmWriteNet(reset1, 1);
```

```
    // run simulation

    // Set to stop at 0.001 seconds simulation time
    icmSetSimulationStopTime(0.01);

    icmSimulatePlatform();

    icmPrintf("Simulation time: %f\n",(float)icmGetCurrentTime());

    icmWriteNet(reset1, 0);

    // Set to stop at 0.002 seconds simulation time
    icmSetSimulationStopTime(0.02);

    icmSimulatePlatform();
```

The example creates net objects and connects them to the *reset* input ports of each of the two OR1K processor instance as follows:

```
    icmNetP reset0 = icmNewNet("reset0");
    icmConnectProcessorNet(processor[0], reset0, "reset", ICM_INPUT);
    icmNetP reset1 = icmNewNet("reset1");
    icmConnectProcessorNet(processor[1], reset1, "reset", ICM_INPUT);
```

We hold processor one in reset

```
    icmWriteNet(reset1, 1);
```

and release the reset to processor 0.

```
    icmWriteNet(reset0, 0);
```

We then simulate for 0.01 simulated seconds:

```
    icmSetSimulationStopTime(0.01);
    icmSimulatePlatform();
```

The reset net to processor one is then written to release the reset:

```
    icmWriteNet(reset1, 0);
```

Finally, we then simulate again for a further 0.01 simulated seconds:

```
    icmSetSimulationStopTime(0.02);
    icmSimulatePlatform();
```

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

```
make –C application CROSS=OR1K
make –C platform CROSS=OR1K
```

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

```
./platform/platform.${IMPERAS_ARCH}.exe --program application/application.OR1K.elf
```

You should see output as follows:

```
CPU 0: Starting ...
CPU 0: Hello World 0
CPU 0: Hello World 1
CPU 0: Hello World 2
CPU 0: Hello World 3
Simulation time: 0.010000
CPU 1: Starting ...
CPU 1: Hello World 0
CPU 0: Hello World 4
CPU 1: Hello World 1
CPU 0: Hello World 5
CPU 1: Hello World 2
CPU 0: Hello World 6
CPU 1: Hello World 3
Simulation time: 0.020000
```

Note that while processor ‘CPU 0’ has executed for 0.01 simulated seconds there is no output from processor ‘CPU 1’. After 0.01 seconds of simulated time has expired the reset line to processor ‘CPU 1’ is released and it starts execution. The simulator schedules execution on both processors ‘CPU 0’ and ‘CPU 1’ and output from both is generated.

18 Integration with Client Debuggers

It is a common requirement to be able to integrate ICM 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.

18.1 Memory Access

The functions icmDebugReadProcessorMemory and icmDebugWriteProcessorMemory should be used to examine or modify memory without causing side effects. See the section on Memory Operations.

18.2 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 18.9.1 for more information about watchpoints). The supported processor registers can be found using the processor register iterator:

```
icmRegInfoP icmGetNextReg(icmProcessorP processor, icmRegInfoP previous);
```

The iterator should be passed `NULL` as the `previous` 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* icmGetRegInfoName(icmRegInfoP regInfo);
Uns32 icmGetRegInfoBits(icmRegInfoP regInfo);
icmRegUsage icmGetRegInfoUsage(icmRegInfoP regInfo);
icmRegGroupP icmGetRegInfoGroup(icmRegInfoP regInfo);
```

The *register group* (`icmRegGroupP`) allows allocation of registers into model-specific sets, to ease presentation for processors that contain many registers (see the next section).

18.3 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:

```
icmRegGroupP icmGetNextRegGroup(icmProcessorP processor, icmRegGroupP previous);
```

The iterator should be passed `NULL` as the `previous` 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 *icmGetRegGroupName(icmRegGroupP group);
```

The registers within a group can be found using the by-group register iterator:

```
icmRegInfoP icmGetNextRegInGroup (
    icmProcessorP processor,
    icmRegGroupP  group,
    icmRegInfoP   previous
);
```

Like the other iterators, the iterator should be passed `NULL` as the `previous` argument on the first call. On subsequent calls, it should be passed the value returned on the previous call.

18.4 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 18.9.1 for more information about watchpoints). The supported processor modes can be found using the processor mode iterator:

```
icmModeInfoP icmGetNextMode(icmProcessorP processor, icmModeInfoP previous);
```

The iterator should be passed `NULL` as the `previous` 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 and processor-specific code can be found using these two functions:

```
const char *icmGetModeInfoName(icmModeInfoP modeInfo);
Uns32 icmGetModeInfoCode(icmModeInfoP modeInfo);
```

The *current* processor mode description can be found using:

```
icmModeInfoP icmGetMode(icmProcessorP processor);
```

18.5 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 18.9.1 for more information about watchpoints). The supported processor exceptions can be found using the processor exception iterator:

```
icmExceptionInfoP icmGetNextException(
    icmProcessorP     processor,
    icmExceptionInfoP previous
);
```

The iterator should be passed `NULL` as the `previous` 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 and processor-specific code can be found using these two functions:

```
const char *icmGetExceptionInfoName(icmExceptionInfoP exceptionInfo);
Uns32 icmGetExceptionInfoCode(icmExceptionInfoP exceptionInfo);
```

The *current* processor exception description can be found using:

```
icmExceptionInfoP icmGetException(icmProcessorP processor);
```

18.6 Processor Freezing

Two routines allow specific processors in a multiprocessor platform to be frozen and unfrozen:

```
void icmFreeze(icmProcessorP processor);
void icmUnfreeze(icmProcessorP processor);
```

When in a *frozen* state, a processor in a multiprocessor simulation will not be scheduled when `icmSimulatePlatform` 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 icmIsFrozen(icmProcessorP processor);
```

18.7 Address Breakpoints

Two routines allow breakpoints to be set and cleared for a specific processor and address:

```
void icmSetAddressBreakpoint(icmProcessorP processor, Addr simAddress);
void icmClearAddressBreakpoint(icmProcessorP processor, Addr simAddress);
```

When a breakpoint has been set for a specific address, any attempt by the processor to execute at that address will cause `icmSimulatePlatform` or `icmSimulate` to return with the processor's `stopReason` set to `ICM_SR_BP_ADDRESS`.

18.8 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 icmSetICountBreakpoint(icmProcessorP processor, Uns64 delta);
void icmClearICountBreakpoint(icmProcessorP processor);
```

Once the specified number of instructions has elapsed, `icmSimulatePlatform` or `icmSimulate` will return with the processor's `stopReason` set to `ICM_SR_BP_ICOUNT`.

18.9 Memory, Bus and Processor Watchpoints

A powerful watchpoint API is implemented specifically aimed at debugger integration.

18.9.1 Watchpoint Creation and Deletion

Three routines are available to set *read*, *write* or *access* (either read or write) watchpoints on a range of memory addresses in a *memory*:

```
icmWatchPointP icmSetMemoryReadWatchPoint(
    icmMemoryP       memory,
    Addr             low,
    Addr             high,
    void            *userData,
    icmMemNotifierFn notifierCB
);
```

```
icmWatchPointP icmSetMemoryWriteWatchPoint(
    icmMemoryP       memory,
    Addr             low,
    Addr             high,
    void            *userData,
    icmMemNotifierFn notifierCB
);

icmWatchPointP icmSetMemoryAccessWatchPoint(
    icmMemoryP       memory,
    Addr             low,
    Addr             high,
    void            *userData,
    icmMemNotifierFn notifierCB
);
```

Three more routines allow watchpoints to be specified on a *bus* range:

```
icmWatchPointP icmSetBusReadWatchPoint(
    icmBusP          bus,
    Addr             low,
    Addr             high,
    void            *userData,
    icmMemNotifierFn notifierCB
);

icmWatchPointP icmSetBusWriteWatchPoint(
    icmBusP          bus,
    Addr             low,
    Addr             high,
    void            *userData,
    icmMemNotifierFn notifierCB
);

icmWatchPointP icmSetBusAccessWatchPoint(
    icmBusP          bus,
    Addr             low,
    Addr             high,
    void            *userData,
    icmMemNotifierFn notifierCB
);
```

Three routines allow watchpoints to be specified on a *processor* address range. For each, an `isPhysical` argument specifies whether the address range is in processor *physical* memory (if `True`) or *virtual* memory (if `False`)[4]:

```
icmWatchPointP icmSetProcessorReadWatchPoint(
    icmProcessorP    processor,
    Bool             isPhysical,
    Addr             low,
    Addr             high,
    void            *userData,
    icmMemNotifierFn notifierCB
);

icmWatchPointP icmSetProcessorWriteWatchPoint(
    icmProcessorP    processor,
    Bool             isPhysical,
    Addr             low,
    Addr             high,
    void            *userData,
    icmMemNotifierFn notifierCB
);
```

[4] See the section 18.11.2 for a definition of what exactly *virtual* and *physical* mean in this context.

.

```
icmWatchPointP icmSetProcessorAccessWatchPoint(
    icmProcessorP    processor,
    Bool             isPhysical,
    Addr             low,
    Addr             high,
    void            *userData,
    icmMemNotifierFn notifierCB
);
```

One routine allows a watchpoint to be established on a *register* in a processor:

```
icmWatchPointP icmSetRegisterWatchPoint(
    icmProcessorP    processor,
    icmRegInfoP      regInfo,
    void            *userData,
    icmRegNotifierFn notifierCB
);
```

One routine allows a watchpoint to be established on a processor *mode switch*:

```
icmWatchPointP icmSetModeWatchPoint(
    icmProcessorP    processor,
    void            *userData,
    icmRegNotifierFn notifierCB
);
```

Finally, one routine allows a watchpoint to be established on a processor *exception*:

```
icmWatchPointP icmSetExceptionWatchPoint(
    icmProcessorP    processor,
    void            *userData,
    icmRegNotifierFn notifierCB
);
```

Each function returns an `icmWatchPointP` 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 icmDeleteWatchPoint(icmWatchPointP 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 ICM_MEM_NOTIFIER_FN(_NAME) Bool _NAME( \
        icmProcessorP  processor,   \
        icmWatchPointP watchpoint,  \
        Addr           PA,          \
        Addr           VA,          \
        Uns32          bytes,       \
        void          *userData,    \
        const void    *value        \
    )

typedef ICM_MEM_NOTIFIER_FN((*icmMemNotifierFn));
```

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 ICM_REG_NOTIFIER_FN(_NAME) Bool _NAME( \
        icmWatchPointP watchpoint,  \
        icmProcessorP  processor,   \
        void          *userData     \
    )

typedef ICM_REG_NOTIFIER_FN((*icmRegNotifierFn));
```

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 `stopReason` `ICM_SR_WATCHPOINT`. Otherwise, if the notifier returns `True`, the triggering processor will not stop but instead continue executing normally.

18.9.2 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 icmWatchpointTypeE {
  ICMWP_MEM_READ,      // Memory read watchpoint.
  ICMWP_MEM_WRITE,     // Memory write watchpoint.
  ICMWP_MEM_ACCESS,    // Memory access watchpoint.
  ICMWP_REGISTER,      // Register watchpoint.
  ICMWP_MODE,          // Mode change watchpoint.
  ICMWP_EXCEPTION      // Exception watchpoint.
} icmWatchpointType;

icmWatchpointType icmGetWatchPointType(icmWatchPointP watchpoint);
```

The client data pointer that was associated with the watchpoint when it was created can be found using:

```
void *icmGetWatchPointUserData(icmWatchPointP watchpoint);
```

For memory address range watchpoints, the bounding addresses can be found using:

```
Addr icmGetWatchPointLowAddress(icmWatchPointP watchpoint);
Addr icmGetWatchPointHighAddress(icmWatchPointP watchpoint);
```

These functions return zero for other watchpoint types. For processor register watchpoints, the register which is being watched can be found using:

```
icmRegInfoP icmGetWatchPointRegister(icmWatchPointP watchpoint);
```

This function returns `NULL` for other watchpoint types. For processor register and mode change watchpoints, there are query functions which return pointers to the *current* and *previous value* of the register being watched, or the current and previous mode (a pointer of type `icmModeInfoP`):

```
void *icmGetWatchPointCurrentValue(icmWatchPointP watchpoint);
void *icmGetWatchPointPreviousValue(icmWatchPointP watchpoint);
```

18.9.3 Handling Triggered Watchpoints

When a watchpoint triggers (because a processor does a read or write to the address range over which it is sensitive, or because the processor register or mode it is watching changes, or an exception occurs), `icmSimulatePlatform` or `icmSimulate` will return with the processor's `stopReason` set to `ICM_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, a function is available that returns the *first* triggered watchpoint:

```
icmWatchPointP icmGetNextTriggeredWatchPoint(void);
```

Once the first triggered watchpoint has been handled by the debugger, it must be reset using:

```
void icmResetWatchPoint(icmWatchPointP watchpoint);
```

Then a subsequent call to `icmGetNextTriggeredWatchPoint` will return the next triggered watchpoint that has not been reset, and so on until all watchpoints have been handled by the client debugger. For each triggered watchpoint, the processor which triggered it can be found using:

```
icmProcessorP icmGetWatchPointTriggeredBy(icmWatchPointP watchpoint);
```

18.10 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 `stopReason` `ICM_SR_BP_ICOUNT` before the instruction is executed;
2. When simulation is resumed by `icmSimulate` or `icmSimulatePlatform`, the address breakpoint is triggered next, causing the processor to be stopped for `stopReason` `ICM_SR_BP_ADDRESS`, again before the instruction is executed;
3. When simulation is resumed by `icmSimulate` or `icmSimulatePlatform`, the instruction completes. After completion, the processor is stopped for `stopReason` `ICM_SR_WATCHPOINT`, at which point the triggered watchpoints can be found and reset using `icmGetNextTriggeredWatchPoint` and `icmResetWatchPoint`.

18.11 Debugger Examples

There are two examples using the address breakpoint, instruction count breakpoint and watchpoint constructs in the `debuggerIntegration` directory:

```
$IMPERAS_HOME/Examples/Platforms/debuggerIntegration
```

The first example uses the two-processor Fibonacci application first seen in the multiprocessor example. The platform file, `platform1.c`, has been extensively modified to exercise the breakpoint and memory watchpoint debugger integration commands (see section 18.11.1).

The second example uses a simpler single-processor assembler example to exercise the mode change and exception watchpoints (see section 18.11.2).

18.11.1 Debugger Integration Example 1

18.11.1.1 Establishing Watchpoints

The platform for this example is `platform1.c` in the example `platforms` directory.

Once processor memory has been loaded, the platform establishes watchpoint addresses on the shared memory between the two processors as follows:

```
    applyWatchpoints(shared);
```

The platform also establishes some register watchpoints in `processor0` only using:

```
    applyRegWatchpoints(processor0);
```

Function `applyWatchpoints` is as follows:

```
static void applyWatchpoints(icmMemoryP memory) {

    Uns32 address;
    Uns32 i;
    Uns32 id = 0;

    for(i=0; (address=watchpoints[i].address); i++) {

        icmWatchPointP rwp = icmSetMemoryReadWatchPoint(
            memory, address, address+watchpoints[i].size-1, (void *)(id++), 0
        );

        icmWatchPointP wwp = icmSetMemoryWriteWatchPoint(
            memory, address, address+watchpoints[i].size-1, (void *)(id++), 0
        );

        icmPrintf("READ watchpoint is %u\n", getWatchpointId(rwp));
        icmPrintf("WRITE watchpoint is %u\n", getWatchpointId(wwp));
    }
}
```

The function iterates across a static array of watchpoint objects creating a separate read and write watchpoint object for each one (obviously in a real debugger product the list would not be static and perhaps an access watchpoint would be used). The list of watchpoints actually contains just a single entry:

```
typedef struct watchpointS {
    Uns32 address;
    Uns32 size;
} watchpoint;
```

```
const static watchpoint watchpoints[] = {
    {0xe3b4, 4},    // fibres
    {0}
};
```

The watchpoint is made on a four-byte address that happens to correspond to the address of the `fibres` static in the application. In a real debugger, the address and object size would of course be found by the debugger from the object file.

Function `applyRegWatchpoints` is as follows:

```
static void applyRegWatchpoints(icmProcessorP processor) {

    icmWatchPointP rwp1 = icmSetRegisterWatchPoint(
        processor, icmGetRegByName(processor, "r3"), (void *)(id++), 0
    );

    icmWatchPointP rwp2 = icmSetRegisterWatchPoint(
        processor, icmGetRegByName(processor, "r9"), (void *)(id++), 0
    );

    icmWatchPointP rwp3 = icmSetRegisterWatchPoint(
        processor, icmGetRegByUsage(processor, ICM_REG_SP), (void *)(id++), 0
    );

    icmPrintf("REGISTER watchpoint 1 is %u\n", getWatchpointId(rwp1));
    icmPrintf("REGISTER watchpoint 2 is %u\n", getWatchpointId(rwp2));
    icmPrintf("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(icmProcessorP processor) {

    icmPrintf("%s REGISTERS\n", icmGetProcessorName(processor, "/"));

    icmRegGroupP group = NULL;

    while((group=icmGetNextRegGroup(processor, group))) {

        icmPrintf("  GROUP %s\n", icmGetRegGroupName(group));

        icmRegInfoP reg = NULL;

        while((reg=icmGetNextRegInGroup(processor, group, reg))) {
            icmPrintf("    REGISTER %s\n", icmGetRegInfoName(reg));
        }
    }
}
```

18.11.1.2 Running the Simulator

The simulator is run in a loop which calls `icmSimulatePlatform`. 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;
    icmProcessorP stopProcessor = NULL;
    for(;;) {

        if(stepOver) {
            icmSetICountBreakpoint(stopProcessor, 1);
            stopProcessor = icmSimulatePlatform();
            stepOver = False;
        } else {
            applyBreakpoints(processor0);
            applyBreakpoints(processor1);
            stopProcessor = icmSimulatePlatform();
            clearBreakpoints(processor0);
            clearBreakpoints(processor1);
        }

        … 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:

```
            icmSetICountBreakpoint(stopProcessor, 1);
            stopProcessor = icmSimulatePlatform();
            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);
            applyBreakpoints(processor1);
            stopProcessor = icmSimulatePlatform();
            clearBreakpoints(processor0);
            clearBreakpoints(processor1);
```

In this simple example, all address breakpoints are applied to both processors, though there is no reason why this has to be the case: each processor can have a distinct set of breakpoints. The routines to set and clear breakpoints are as follows:

```
static void applyBreakpoints(icmProcessorP processor) {

    Uns32 i;

    for(i=0; breakpoints[i]; i++) {
        icmSetAddressBreakpoint(processor, breakpoints[i]);
    }
}

static void clearBreakpoints(icmProcessorP processor) {

    Uns32 i;

    for(i=0; breakpoints[i]; i++) {
        icmClearAddressBreakpoint(processor, breakpoints[i]);
    }
}
```

The breakpoint addresses for this simple example are specified in a static list, and correspond to the addresses of routines in the application. Again, a real debugger would read these from the application ELF file and not rely on fixed addresses:

```
const static Uns32 breakpoints[] = {
    0x0fdc,         // munge
    0x1330,         // main
    0x108c,          // writer
    0x1204,          // reader
    0                // terminator
};
```

Each time `icmSimulatePlatform` returns, the loop decides what to do next depending on the system state.

1. If a `NULL` processor was returned, the simulation has terminated.
2. Otherwise, if the `stopReason` was `ICM_SR_BP_ICOUNT` an instruction count breakpoint has been hit (the debugger is single-stepping over an address breakpoint location):
```
            case ICM_SR_BP_ICOUNT:
                icmPrintf(
                    "Processor %s icount %u stopped at icount\n",
                     icmGetProcessorName(stopProcessor, "/"),
                    (Uns32)icmGetProcessorICount(stopProcessor)
                );
                break;
```
3. Otherwise, if the `stopReason` was `ICM_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 ICM_SR_BP_ADDRESS:
                icmPrintf(
                    "Processor %s icount %u stopped at address 0x%08x\n",
                    icmGetProcessorName(stopProcessor, "/"),
                    (Uns32)icmGetProcessorICount(stopProcessor),
                    icmGetPC(stopProcessor)
                );
                stepOver = True;
                break;
```
4. Otherwise, if the `stopReason` was `ICM_SR_WATCHPOINT` a watchpoint has triggered. In this case, the triggered watchpoints are scanned and reported:
```
            case ICM_SR_WATCHPOINT:
                icmPrintf(
                    "Processor %s icount %u stopped at watchpoint\n",
                     icmGetProcessorName(stopProcessor, "/"),
                    (Uns32)icmGetProcessorICount(stopProcessor)
                );
                handleWatchpoints();
                break;
```
5. Otherwise, the `stopReason` is reported and simulation continues (no other `stopReasons` are expected in this simulation).

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(void) {

    icmWatchPointP wp;

    while((wp=icmGetNextTriggeredWatchPoint())) {

        Uns32         id        = getWatchpointId(wp);
        icmProcessorP processor = icmGetWatchPointTriggeredBy(wp);
```

It uses the watchpoint type 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(icmGetWatchPointType(wp)) {

            case ICMWP_REGISTER: {

                // a register watchpoint was triggered
                icmRegInfoP reg       = icmGetWatchPointRegister(wp);
                Uns32      *newValueP = icmGetWatchPointCurrentValue(wp);
                Uns32      *oldValueP = icmGetWatchPointPreviousValue(wp);

                // indicate old and new value of the affected register
                icmPrintf(
                    "  watchpoint %u (processor %s:%s) triggered 0x%08x->0x%08x\n",
                    id,
                    icmGetProcessorName(processor, "/"),
                    icmGetRegInfoName(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) {
                icmDeleteWatchPoint(wp);
            } else {
                icmResetWatchPoint(wp);
            }
```

(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 one, information about the address range is printed and the watchpoint reset:

```
            case ICMWP_MEM_READ:
            case ICMWP_MEM_WRITE:
            case ICMWP_MEM_ACCESS:

                // a memory watchpoint was triggered
                icmPrintf(
                    "  watchpoint %u (range 0x%08x:0x%08x) triggered by processor %s\n",
                    id,
                    (Uns32)icmGetWatchPointLowAddress(wp),
                    (Uns32)icmGetWatchPointHighAddress(wp),
                    icmGetProcessorName(processor, "/")
                );

                icmResetWatchPoint(wp);

                break;
```

The `userData` associated with a watchpoint is used to record an arbitrary watchpoint id number:

```
static Uns32 getWatchpointId(icmWatchPointP watchpoint) {
    return (Uns32)icmGetWatchPointUserData(watchpoint);
}
```

18.11.1.3 Compiling and Running the Example

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

```
make –C platform SRC=platform1.c
make –C application
```

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

```
./platform/platform1.${IMPERAS_ARCH}.exe --program application/application.OR1K.elf
```

You should see the following output

```
READ watchpoint is 0
WRITE watchpoint is 1
REGISTER watchpoint 1 is 2
REGISTER watchpoint 2 is 3
REGISTER watchpoint 3 is 4
/cpu0 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 EXCPT
Processor /cpu1 icount 45 stopped at address 0x00001330
Processor /cpu1 icount 46 stopped at icount
CPU 1 starting...
Processor /cpu1 icount 2136 stopped at address 0x00001204
Processor /cpu1 icount 2137 stopped at icount
Processor /cpu0 icount 2 stopped at watchpoint
  watchpoint 2 (processor /cpu0:R3) triggered 0xdeadbeef->0x00000000
Processor /cpu0 icount 8 stopped at watchpoint
  watchpoint 3 (processor /cpu0:R9) triggered 0xdeadbeef->0x00000000
Processor /cpu0 icount 31 stopped at watchpoint
  watchpoint 4 (processor /cpu0:R1) triggered 0x00000000->0xffff0000
Processor /cpu0 icount 32 stopped at watchpoint
  watchpoint 4 (processor /cpu0:R1) triggered 0xffff0000->0xfffffffc
Processor /cpu0 icount 37 stopped at watchpoint
  watchpoint 4 (processor /cpu0:R1) triggered 0xfffffffc->0xffffffec
Processor /cpu0 icount 44 stopped at watchpoint
  watchpoint 3 (processor /cpu0:R9) triggered 0x00000000->0x00001434
Processor /cpu0 icount 44 stopped at address 0x00001330
Processor /cpu0 icount 45 stopped at watchpoint

. . . many similar lines deleted . . .

Processor /cpu0 icount 526 stopped at watchpoint
  watchpoint 2 (processor /cpu0:R3) triggered 0x00000000->0x0000e4d0
Processor /cpu0 icount 542 stopped at watchpoint
  watchpoint 3 (processor /cpu0:R9) triggered 0x00003e40->0x000040d0
Processor /cpu0 icount 543 stopped at watchpoint
  watchpoint 4 (processor /cpu0:R1) triggered 0xfffff8b0->0xfffff8a8
```

```
Processor /cpu0 icount 552 stopped at watchpoint
  watchpoint 3 (processor /cpu0:R9) triggered 0x000040d0->0x0000d66e
Processor /cpu0 icount 601 stopped at watchpoint
  watchpoint 2 (processor /cpu0:R3) triggered 0x0000e4d0->0x00000000
CPU 0 starting...
Processor /cpu0 icount 1551 stopped at address 0x0000108c
Processor /cpu0 icount 1552 stopped at icount
CPU 0: fib(0) = 0
Processor /cpu0 icount 4026 stopped at watchpoint
  watchpoint 1 (range 0x0000e3b4:0x0000e3b7) triggered by processor /cpu0
Processor /cpu1 icount 100018 stopped at watchpoint
  watchpoint 0 (range 0x0000e3b4:0x0000e3b7) triggered by processor /cpu1
Processor /cpu1 icount 100034 stopped at address 0x00000fdc
Processor /cpu1 icount 100035 stopped at icount
CPU 1: munge(0) = 0
CPU 0: fib(1) = 1
Processor /cpu0 icount 102435 stopped at watchpoint
  watchpoint 1 (range 0x0000e3b4:0x0000e3b7) triggered by processor /cpu0
Processor /cpu1 icount 200021 stopped at watchpoint
  watchpoint 0 (range 0x0000e3b4:0x0000e3b7) triggered by processor /cpu1
Processor /cpu1 icount 200037 stopped at address 0x00000fdc
Processor /cpu1 icount 200038 stopped at icount
CPU 1: munge(1) = 0
CPU 0: fib(2) = 1
Processor /cpu0 icount 202442 stopped at watchpoint
  watchpoint 1 (range 0x0000e3b4:0x0000e3b7) triggered by processor /cpu0
Processor /cpu1 icount 300017 stopped at watchpoint
  watchpoint 0 (range 0x0000e3b4:0x0000e3b7) triggered by processor /cpu1
Processor /cpu1 icount 300033 stopped at address 0x00000fdc
Processor /cpu1 icount 300034 stopped at icount
CPU 1: munge(1) = 0
CPU 0: fib(3) = 2

. . . many similar lines deleted . . .

CPU 1: munge(89) = 3916
CPU 0: fib(12) = 144
Processor /cpu0 icount 1204648 stopped at watchpoint
  watchpoint 1 (range 0x0000e3b4:0x0000e3b7) triggered by processor /cpu0
Processor /cpu1 icount 1300016 stopped at watchpoint
  watchpoint 0 (range 0x0000e3b4:0x0000e3b7) triggered by processor /cpu1
Processor /cpu1 icount 1300032 stopped at address 0x00000fdc
Processor /cpu1 icount 1300033 stopped at icount
CPU 1: munge(144) = 10296
CPU 0: fib(13) = 233
Processor /cpu0 icount 1304695 stopped at watchpoint
  watchpoint 1 (range 0x0000e3b4:0x0000e3b7) triggered by processor /cpu0
Processor /cpu1 icount 1400011 stopped at watchpoint
  watchpoint 0 (range 0x0000e3b4:0x0000e3b7) triggered by processor /cpu1
Processor /cpu1 icount 1400027 stopped at address 0x00000fdc
Processor /cpu1 icount 1400028 stopped at icount
CPU 1: munge(233) = 27028
CPU 0: fib(14) = 377
Processor /cpu0 icount 1404712 stopped at watchpoint
  watchpoint 1 (range 0x0000e3b4:0x0000e3b7) triggered by processor /cpu0
Processor /cpu1 icount 1500016 stopped at watchpoint
  watchpoint 0 (range 0x0000e3b4:0x0000e3b7) triggered by processor /cpu1
Processor /cpu1 icount 1500034 stopped at address 0x00000fdc
Processor /cpu1 icount 1500035 stopped at icount
CPU 1: munge(377) = 70876
```

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 /cpu1 icount 45 stopped at address 0x00001330
```

Instruction count breakpoints are reported with lines of this form:

```
Processor /cpu1 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 /cpu0 icount 2 stopped at watchpoint
  watchpoint 2 (processor /cpu0:R3) triggered 0xdeadbeef->0x00000000
```

Memory watchpoints are reported by a pair of lines of this form:

```
Processor /cpu0 icount 4037 stopped at watchpoint
  watchpoint 1 (range 0x0000e3b4:0x0000e3b7) triggered by processor /cpu0
```

18.11.2 Semantics of Physical and Virtual Watchpoints

The functions `icmSetProcessorReadWatchPoint`, `icmSetProcessorWriteWatchPoint` and `icmSetProcessorAccessWatchPoint` each take an argument `isPhysical` which indicates whether the watch point should be physical or virtual. The semantics of these are as follows:

18.11.2.1 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`.

18.11.2.2 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 `icmSetProcessor*WatchPoint` 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*.

18.11.3 Debugger Integration Example 2

18.11.3.1 Establishing Watchpoints

The platform for this example is `platform2.c` in the example `platforms` directory. The platform has a similar structure to `platform1.c`, but instances only a single processor.

Once processor memory has been loaded, the platform establishes processor *mode change* watchpoints as follows:

```
    applyModeWatchpoints(processor);
```

It also establishes exception watchpoints as follows:

```
    applyExceptionWatchpoints(processor);
```

Function `applyModeWatchpoints` is as follows:

```
static void applyModeWatchpoints(icmProcessorP processor) {
    icmWatchPointP mwp = icmSetModeWatchPoint(processor, (void *)(id++), 0);
    icmPrintf("MODE watchpoint 1 is %u\n", getWatchpointId(mwp));
}
```

The function creates a single mode change watchpoint with an arbitrary id number.

Function `applyExceptionWatchpoints` is as follows:

```
static void applyExceptionWatchpoints(icmProcessorP processor) {
    icmWatchPointP ewp = icmSetExceptionWatchPoint(processor, (void *)(id++), 0);
    icmPrintf("EXCEPTION watchpoint 1 is %u\n", getWatchpointId(ewp));
}
```

This also creates a single exception watchpoint with an arbitrary id number.

18.11.3.2 Running the Simulator

The simulator loop is similar to that in platform1.c. 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(void) {

    icmWatchPointP wp;

    while((wp=icmGetNextTriggeredWatchPoint())) {

        Uns32         id        = getWatchpointId(wp);
        icmProcessorP processor = icmGetWatchPointTriggeredBy(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 old and new mode, and the watchpoint is reset:

```
        switch(icmGetWatchPointType(wp)) {

            case ICMWP_MODE: {

                // a mode switch watchpoint was triggered
                icmModeInfoP *oldValueP = icmGetWatchPointPreviousValue(wp);
                icmModeInfoP *newValueP = icmGetWatchPointCurrentValue(wp);
                icmModeInfoP  oldValue  = *oldValueP;
                icmModeInfoP  newValue  = *newValueP;

                icmPrintf(
                    "  watchpoint %u (processor %s:mode) triggered %s->%s\n",
                    id,
                    icmGetProcessorName(processor, "/"),
                    icmGetModeInfoName(oldValue),
                    icmGetModeInfoName(newValue)
                );

                icmResetWatchPoint(wp);

                break;
            }
```

If this is an exception watchpoint, information about the exception name is printed and the watchpoint reset. Note that exception watchpoints have no notion of previous and current value:

```
            case ICMWP_EXCEPTION: {

                icmExceptionInfoP exception = icmGetException(processor);

                // an exception watchpoint was triggered
                icmPrintf(
                    "  watchpoint %u (processor %s:exception) triggered ->%s\n",
                    id,
                    icmGetProcessorName(processor, "/"),
                    icmGetExceptionInfoName(exception)
                );

                icmResetWatchPoint(wp);

                break;
            }
```

18.11.3.3 Compiling and Running the Example

Compile the test platform and application using the following commands in the `debuggerIntegration` directory:

```
make –C platform SRC=platform2.c
make –C application
```

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

```
./platform.platform2.${IMPERAS_ARCH}.exe --program application/asmtest.OR1K.elf
```

You should see the following output

```
MODE watchpoint 1 is 0
EXCEPTION watchpoint 1 is 1
Info 'cpu0', 0x0000000000010000(_start): l.ori    r30,r0,0x0
Info 'cpu0', 0x0000000000010004(_start+4): l.ori    r31,r0,0x0
Info 'cpu0', 0x0000000000010008(_start+8): l.mtspr  r0,r0,32
Info 'cpu0', 0x000000000001000c(_start+c): l.ori    r1,r0,0x2
Info 'cpu0', 0x0000000000010010(loop1): l.mfspr  r2,r0,32
Info 'cpu0', 0x0000000000010014(loop1+4): l.addi   r2,r2,0x1
Info 'cpu0', 0x0000000000010018(loop1+8): l.mtspr  r0,r2,32
Info 'cpu0', 0x000000000001001c(loop1+c): l.addi   r1,r1,0xffffffff
Info 'cpu0', 0x0000000000010020(loop1+10): l.sfeqi  r1,0x0
Info 'cpu0', 0x0000000000010024(loop1+14): l.bnf    0x00010010
Info 'cpu0', 0x0000000000010028(loop1+18): l.nop    0x0
Info 'cpu0', 0x0000000000010010(loop1): l.mfspr  r2,r0,32
Info 'cpu0', 0x0000000000010014(loop1+4): l.addi   r2,r2,0x1
Info 'cpu0', 0x0000000000010018(loop1+8): l.mtspr  r0,r2,32
Info 'cpu0', 0x000000000001001c(loop1+c): l.addi   r1,r1,0xffffffff
Info 'cpu0', 0x0000000000010020(loop1+10): l.sfeqi  r1,0x0
Info 'cpu0', 0x0000000000010024(loop1+14): l.bnf    0x00010010
Info 'cpu0', 0x0000000000010028(loop1+18): l.nop    0x0
Info 'cpu0', 0x000000000001002c(loop1+1c): l.jal    0x00010048
Info 'cpu0', 0x0000000000010030(loop1+20): l.nop    0x0
Info 'cpu0', 0x0000000000010048(incEPC): l.mfspr  r2,r0,32
Info 'cpu0', 0x000000000001004c(incEPC+4): l.addi   r2,r2,0x1
Info 'cpu0', 0x0000000000010050(incEPC+8): l.mtspr  r0,r2,32
Info 'cpu0', 0x0000000000010054(incEPC+c): l.jr     r9
Info 'cpu0', 0x0000000000010058(incEPC+10): l.nop    0x0
Info 'cpu0', 0x0000000000010034(loop1+24): l.mtspr  r0,r0,17
Processor /cpu0 icount 26 stopped at watchpoint
  watchpoint 0 (processor /cpu0:mode) triggered SUPERVISOR->USER
Info 'cpu0', 0x0000000000010038(loop1+28): l.jal    0x00010048
Info 'cpu0', 0x000000000001003c(loop1+2c): l.nop    0x0
Info 'cpu0', 0x0000000000010048(incEPC): l.mfspr  r2,r0,32
Processor /cpu0 icount 29 stopped at watchpoint
  watchpoint 0 (processor /cpu0:mode) triggered USER->SUPERVISOR
  watchpoint 1 (processor /cpu0:exception) triggered ->ILL
Info 'cpu0', 0x0000000000000700(.text+700): l.addi   r30,r30,0x1
Info 'cpu0', 0x0000000000000704(.text+704): l.sw     0xfffffffc(r31),r1
Info 'cpu0', 0x0000000000000708(.text+708): l.mfspr  r1,r0,32
Info 'cpu0', 0x000000000000070c(.text+70c): l.addi   r1,r1,0x4
Info 'cpu0', 0x0000000000000710(.text+710): l.mtspr  r0,r1,32
Info 'cpu0', 0x0000000000000714(.text+714): l.lwz    r1,0xfffffffc(r31)
Info 'cpu0', 0x0000000000000718(.text+718): l.rfe
Processor /cpu0 icount 36 stopped at watchpoint
  watchpoint 0 (processor /cpu0:mode) triggered SUPERVISOR->USER
Info 'cpu0', 0x000000000001004c(incEPC+4): l.addi   r2,r2,0x1
Info 'cpu0', 0x0000000000010050(incEPC+8): l.mtspr  r0,r2,32
Processor /cpu0 icount 38 stopped at watchpoint
  watchpoint 0 (processor /cpu0:mode) triggered USER->SUPERVISOR
  watchpoint 1 (processor /cpu0:exception) triggered ->ILL
Info 'cpu0', 0x0000000000000700(.text+700): l.addi   r30,r30,0x1
Info 'cpu0', 0x0000000000000704(.text+704): l.sw     0xfffffffc(r31),r1
Info 'cpu0', 0x0000000000000708(.text+708): l.mfspr  r1,r0,32
Info 'cpu0', 0x000000000000070c(.text+70c): l.addi   r1,r1,0x4
Info 'cpu0', 0x0000000000000710(.text+710): l.mtspr  r0,r1,32
Info 'cpu0', 0x0000000000000714(.text+714): l.lwz    r1,0xfffffffc(r31)
Info 'cpu0', 0x0000000000000718(.text+718): l.rfe
Processor /cpu0 icount 45 stopped at watchpoint
  watchpoint 0 (processor /cpu0:mode) triggered SUPERVISOR->USER
Info 'cpu0', 0x0000000000010054(incEPC+c): l.jr     r9
Info 'cpu0', 0x0000000000010058(incEPC+10): l.nop    0x0
Info 'cpu0', 0x0000000000010040(loop1+30): l.rfe
Processor /cpu0 icount 48 stopped at watchpoint
  watchpoint 0 (processor /cpu0:mode) triggered USER->SUPERVISOR
  watchpoint 1 (processor /cpu0:exception) triggered ->ILL
Info 'cpu0', 0x0000000000000700(.text+700): l.addi   r30,r30,0x1
Info 'cpu0', 0x0000000000000704(.text+704): l.sw     0xfffffffc(r31),r1
Info 'cpu0', 0x0000000000000708(.text+708): l.mfspr  r1,r0,32
Info 'cpu0', 0x000000000000070c(.text+70c): l.addi   r1,r1,0x4
Info 'cpu0', 0x0000000000000710(.text+710): l.mtspr  r0,r1,32
Info 'cpu0', 0x0000000000000714(.text+714): l.lwz    r1,0xfffffffc(r31)
Info 'cpu0', 0x0000000000000718(.text+718): l.rfe
Processor /cpu0 icount 55 stopped at watchpoint
  watchpoint 0 (processor /cpu0:mode) triggered SUPERVISOR->USER
Info 'cpu0', 0x0000000000010044(exit): l.nop    0x0
Processor 'cpu0' terminated at 'exit', address 0x10044
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000000   R2 : 00000004   R3 : deadbeef
 R4 : deadbeef   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : 00010040   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: 00000003   R31: 00000000
 PC : 00010048   SR : 00008000   ESR: 00008000   EPC: 00010044
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:0 CF:0 OF:0
--------------- --------------- --------------- ---------------

processor has executed 56 instructions
```

Each mode change watchpoint that is encountered is reported with lines of this form:

```
Processor /cpu0 icount 26 stopped at watchpoint
  watchpoint 0 (processor /cpu0:mode) triggered SUPERVISOR->USER
```

Exception watchpoints are reported with lines of this form:

```
Processor /cpu0 icount 29 stopped at watchpoint
  watchpoint 0 (processor /cpu0:mode) triggered USER->SUPERVISOR
  watchpoint 1 (processor /cpu0:exception) triggered ->ILL
```

Note that exception watchpoints always occur at the same time as mode change watchpoints in this example, as each exception requires a switch from user to supervisor mode.

18.12 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 `icmSimulate` or `icmSimulatePlatform` are made in code that is not accessible to the debugger - for example from a SystemC platform. Use the function icmSetDebugNotifiers to install callbacks on these actions. The callbacks are:

Call	**Meaning**
icmStartSimFn	Simulation is about to begin, but no peripherals or processors have run yet.
icmEndConstructorsFn	Peripheral constructors have run, application processors are about to start.
icmEndSchedFn	Called after each application processor has finished a slice. Should return true if simulation is to continue, false to finish.
icmEndSchedFn	Called after each PSE has finished executing. Only use this if you wish to debug a PSE. Should return true if simulation is to continue, false to finish.
icmTimeAdvanceFn	Called when a scheduler advances time. Should return true if simulation is to continue, false to finish.
icmFinishFn	Called when simulation is about to finish

The example in Examples/Debugger/threads shows the use of this function to allow a platform and a debugger to run in separate thread, but in the same program. This is pseudo code extracted from the example:

```
static ICM_START_SIM_FN(startSim) {
    // called once:
    //   Before the peripheral constructors run
    ...
}

static ICM_END_CONSTRUCTORS_FN(endConstructors) {
    //  called once
    //    After the peripheral constructors run
    //    Before the first instructions are executed on application processors
    ...
}


static ICM_END_SCHED_FN(debugProc) {
    // called after each core has executed instructions.
    icmStopReason reason = icmGetStopReason(processor);
    if (debuggerNeedsToActOnThisReason(reason)) {
        ...
    }
    if(debuggerWantsToFinish()) {
        return False;
    } else {
        return True;
    }
}

static ICM_END_SCHED_FN(debugPSE) {
    icmStopReason reason = icmGetStopReason(processor);
    if (debuggerNeedsToActOnThisReason(reason)) {
        ...
    }
    if(debuggerWantsToFinish()) {
        return False;
    } else {
        return True;
    }
}
```

```
static ICM_TIME_ADVANCE_FN(advanceTime) {
    // called when (and only when) time is advanced
    ...
    return True;
}

static ICM_FINISH_FN(finishSim) {
    // called when no more instructions to execute, but before destruction.
    ...
}

int main(...) {

    icmDebugNotifier notify = {
        .start     = startSim,
        .endCons   = endConstructors,
        .sched     = debugProc,
        .schedPSE  = debugPSE,
        .advance   = advanceTime,
        .finish    = finishSim,
        .userData  = myPointer
    };

    // request callbacks. Note: this can be called before icmInitPlatform()
    icmSetDebugNotifiers(&notify);

    buildPlatform();  // using icmInitPlatform etc
    ...

    // Run the simulation
    icmSimulatePlatform();
}
```

`icmSetSchedFn` can be called before or after `icmInitPlatform` but must be called before simulation begins. 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 `icmSimulatePlatform()` or `icmSimulate(processor)` is 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 `debugPSE` is required only if you wish to debug PSE code. Leave the callback pointer null if not required. The userData field will be passed to the callback. 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.

19 Peripherals

ICM supports the inclusion of multiple peripherals in a platform. A peripheral is modeled using a Peripheral Simulation Engine (PSE) and/or an Intercept library. The creation of a PSE is described in detail in the *OVP Peripheral Modeling Guide.*

This section provides information on how a peripheral is includes and used within an ICM platform.

19.1 Adding a Peripheral

The peripheral is instantiated in the platform and connected to the bus using one or more ports. The name of the port, used when connecting onto the bus, must match the name used when the port is created within the peripheral model. Within the peripheral a port has a size but has no address, it is only when it is connected to the bus that it is given an address.

19.1.1 Instantiating a Peripheral

The peripheral is instantiated using the `icmNewPSE` or `icmNewPSEWithHandle` function calls. This function returns a handle to the PSE instance that is used by other functions.

```
    // instantiate the peripheral
    icmPseP vga = icmNewPSE("vga", vgaPse, vgaAttrs, NULL, NULL);
```

19.1.2 Attach a peripheral to a Bus

A peripheral can be connected at a fixed address on a bus or it can be attached dynamically.

19.1.2.1 Fixed Bus

When connecting to a fixed bus the peripherals port is connected as either a master or slave port between a specific address range. The port name and its address range must match that defined in the peripheral model

```
    icmConnectPSEBus(vga, bus, "config", False, 0x180003b0, 0x180003df);
```

19.1.2.2 Dynamic Bus

A port can be dynamically connected to a bus. This is the case when modeling a peripheral device that connects to a dynamic bus, such as PCI, that is configured at run time.

The address to which the peripheral will respond is coded within the behavioral code of the peripheral model and not within the platform.

```
    icmConnectPSEBusDynamic(vga, bus, "memory", False);
```

19.2 Enabling Diagnostics

A peripheral can be designed 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).

19.2.1 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 `icmSetPSEdiagnosticLevel` 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
4	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.

```
  icmSetPSEdiagnosticLevel(vga, 4);
```

19.2.2 Intercept Library Diagnostics

When the peripherals 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)

...

icmSetPSEdiagnosticLevel(vga, INT_DIAG_HIGH | PSE_DIAG_HIGH);
```

19.2.3 PSE Debugger Support

When simulating a platform in the Imperas simulator, a GDB executable can be associated with each PSE so that full symbolic debug is available for PSE.

Two methods of association are used:

- The PSE model in the Imperas component library specifies a gdb path. This will work if the Imperas simulator is supplied with its standard component library.
- The function `icmSetPSEGdbPath` can set the path

```
// set the gdb path
icmSetPSEGdbPath(
    icmPseP pse,          // handle to the PSE
    const char *path,     // full path to the GDB executable
    const char *flags     // any flags to be appended to the GDB invocation
);
```

19.3 Passing Attributes

A configurable peripheral model uses attributes passed from the platform to the peripheral model to change its behavior. There are three types of attributes that can be passed to and read by the peripheral model, unsigned 64 bit, double and string. They are passed as an attribute list and added to that declared list using different functions.

An empty attribute list is created using the `icmNewAttrList` function and then populated with attributes using the three functions `icmAddUns64Attr`, `icmAddUns64Attr` and `icmAddStringAttr`. The list is then added to the PSE instance when it is instantiated.

```
    icmAttrListP vgaAttrs = icmNewAttrList();
    icmAddUns64Attr(vgaAttrs, "scanDelay", 50000);
    icmAddUns64Attr(vgaAttrs, "PCIslot", 18);
    icmAddUns64Attr(vgaAttrs, "noGraphics", noGraphics);
    icmAddStringAttr(vgaAttrs, "title", "OVPsim MIPS32 Malta");
    icmPseP vga = icmNewPSE("vga", vgaPse, vgaAttrs, NULL, NULL);
```

19.4 Simulating a Platform

19.4.1 Default Scheduling Algorithm

The platforms created using the ICM API would normally make use of the default scheduling algorithm. By using the default algorithm the ICM platform can be directly imported into the Imperas professional tools without any modification.

The default scheduling is performed by a call to the `icmSimulatePlatform` function. This runs all processor and peripheral instances in the platform.

19.4.2 Custom Scheduling Algorithm

A custom scheduling algorithm is created using the `icmSimulate` and `icmAdvanceTime` functions in place of the `icmSimulatePlatform` function.

The function `icmSimulate` is applied to only one processor instance. To simulate the platform all processors and peripherals in the platform must be scheduled. The function `icmSimulate` 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 `icmAdvanceTime` 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. `icmAdvanceTime` returns `False` if the new time is at or beyond a requested stop time (see `icmSetSimulationStopTime`).

```
#define INST_PER_SECOND      100000000
#define TIME_SLICE           0.01
#define INST_PER_TIME_SLICE (INST_PER_SECOND*TIME_SLICE)

...

    icmTime myTime;
    icmStopReason rtnVal = ICM_SR_SCHED;
    Bool outOftime = False;
    for(myTime=TIME_SLICE;rtnVal==ICM_SR_SCHED || rtnVal==ICM_SR_HALT;myTime+=TIME_SLICE){
        rtnVal= icmSimulate(processor, INST_PER_TIME_SLICE);
        outOfTime = !icmAdvanceTime(myTime);
    }
...
```

19.5 Adding an Extension Intercept Library

Intercept libraries may be used to extend the functionality of a peripheral model. This allows some of the model behavior to be modeled using native host code. This used when the peripheral model makes use of host features, such as Ethernet, keyboard, graphics or USB connections.

The intercept library is loaded automatically by the peripheral model, though this function can be overridden by specifying the path to the extension library as the forth argument to `icmNewPSE` or `icmNewPSEWithHandle`.

The following is the instantiation of a VGA peripheral model that uses an intercept library for part of its behavior. The variable `vgaIntercept` is a string providing the path to the library to be loaded.

```
Const char *vgaExtension = “localDir/model”;  // the .so or .dll is assumed.
```

```
    icmPseP vga = icmNewPSE("vga", vgaPse, vgaAttrs, vgaExtension, 0);
```

19.5.1 Adding more Intercept Libraries

Intercept libraries can be added to an existing PSE instance using `icmAddPseInteceptObject` (only available in Imperas Professional products).

```
icmPseP vga = icmNewPSE("vga", vgaPse, vgaAttrs, NULL, NULL,);
icmAddPseInterceptObject(
    vga,                              // PSE handle
    "intercept1",                     // intercept library instance name
    "/home/library/intercept",        // path to shared object or DLL
    0,                                // not used
    0                                 // optional list of user defined attributes
                                      // used to control the intercept library
);
```

19.5.2 Example platform

An example of a peripheral that uses an intercept library for implementing native behavior may be found in following directory:

```
$IMPERAS_HOME/Examples/Models/Peripherals/creatingDMAC/5.nativeBehaviour
```

The example shows a peripheral with a native semihost instantiated into the platform in the file platform/platform.c:

```
...
    ////////////////////////////////////////////////////////////////////////////
    // DMAC Peripheral
    ////////////////////////////////////////////////////////////////////////////

    // instantiate the peripheral
    icmAttrListP config = icmNewAttrList();
    icmAddUns64Attr(config, "enableNative", 1);

    icmPseP dmac = icmNewPSE("dmac", "peripheral/pse/pse.pse", config, 0, 0);
...
```

In the peripheral's source file, peripheral/pse/dmac.attrs.igen.c, the semihost library to be loaded is defined with the extension member of the ppmModelAttr structure:

```
...
ppmModelAttr modelAttrs = {

    .versionString = PPM_VERSION_STRING,
    .type          = PPM_MT_PERIPHERAL,

    .busPortsCB    = nextBusPort,
    .netPortsCB    = nextNetPort,
    .paramSpecCB   = nextParameter,

    .vlnv          = {
        .vendor  = "ovpworld.org",
        .library = "peripheral",
        .name    = "dmac",
        .version = "1.0"
    },

    .family    = "ovpworld.org",
    .extension = "../model/model"

};
```

Compile the test platform and application as before using the following commands in the 5.nativeBehaviour directory:

```
make -C peripheral/pse   NOVLNV=1
make -C peripheral/model NOVLNV=1
make -C platform         NOVLNV=1
make -C application
```

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

```
./platform/platform.${IMPERAS_ARCH}.exe --program application/dmaTest.elf
```

You should see the following output (some repetitive parts have been removed):

```

OVPsim started: Thu Mar 26 22:09:51 2015


Info (ICM_AL) Found attribute symbol 'modelAttrs' in file
'/Imperas/lib/Linux32/ImperasLib/ovpworld.org/semihosting/or1kNewlib/1.0/model.so'
Info (ICM_AL) Found attribute symbol 'modelAttrs' in file
'/Imperas/lib/Linux32/ImperasLib/ovpworld.org/processor/or1k/1.0/model.so'
Info (OR_OF) Target 'platform/CPU1' has object file read from 'application/dmaTest.elf'
Info (OR_PH) Program Headers:
Info (OR_PH) Type           Offset     VirtAddr   PhysAddr   FileSiz    MemSiz     Flags
Align
Info (OR_PD) LOAD           0x00002000 0x00000000 0x00000000 0x00012544 0x00012664 RWE
2000
Info (ICM_AL) Found attribute symbol 'modelAttrs' in file
'/home/straus/tmp/5.nativeBehaviour/peripheral/model/model.so'
Info (INIT) platform/dmac: Use Native data on channel 1
Info (PP_STUBS) platform/dmac: User initialization
Info (INIT) 'MWRITE', hi 0 lo ffffffff master 1 dynamic 0
Info (INIT) platform/dmac: Semihost Initialized
Info (DMAC) platform/dmac:  ch 0 waiting
Info (DMAC) platform/dmac:  ch 1 waiting
Info (DMAC) platform/dmac: configWr 0x02 (2) burst size 4
TEST DMA: initial dst1 '11111111111111111111111111111111111' dst2
'22222222222222222222222222222222222'
TEST DMA: dmaBurst ch:0  bytes:13
Info (DMAC) platform/dmac: regWr32 0xffffefe4 (4294963172)
Info (DMAC) platform/dmac: regWr32 0xffffdfe4 (4294959076)
Info (DMAC) platform/dmac: regWr32 0x0000000d (13)
Info (DMAC) platform/dmac: configCh0Wr 0x00008001 enable 1 interrupts 1 halt 0
Info (DMAC) platform/dmac:  ch 0 running
Info (DMAC) platform/dmac:  ch 0 13 bytes burst size 4 from ffffefe4 to ffffdfe4
Info (DMAC) platform/dmac:  ch 0 access 4 bytes from ffffefe8 to ffffdfe8
TEST DMA: dmaBurst ch:1  bytes:35
Info (DMAC) platform/dmac: regWr32 0xffffcfe4 (4294954980)
Info (DMAC) platform/dmac: regWr32 0xffffbfe4 (4294950884)
Info (DMAC) platform/dmac: regWr32 0x00000023 (35)
Info (DMAC) platform/dmac: configCh1Wr 0x00008001 enable 1 interrupts 1 halt 0
Info (DMAC) platform/dmac:  ch 1 running
Info (DMAC) platform/dmac:  ch 1 35 bytes burst size 4 from ffffcfe4 to ffffbfe4
Info (PERIPHERAL_SEMI) Send native data 0xffffcfe4 to 0xffffbfe4 (4 bytes)
Info (DMAC) platform/dmac:  ch 1 access 4 bytes from ffffcfe8 to ffffbfe8
```

```
TEST DMA: Waiting for interrupts
Info (DMAC) platform/dmac:  ch 0 access 4 bytes from ffffefec to ffffdfec
Info (PERIPHERAL_SEMI) Send native data 0xffffcfe8 to 0xffffbfe8 (4 bytes)
Info (DMAC) platform/dmac:  ch 1 access 4 bytes from ffffcfec to ffffbfec

...

Info (DMAC) platform/dmac: ch 1 status=0x3
Info (DMAC) platform/dmac:  ch 1 waiting
TEST DMA: Interrupt
Info (DMAC) platform/dmac: regRd32 0x01800000 (25165824)
Info (DMAC) platform/dmac: regRd32 0x01800000 (25165824)
TEST DMA: Interrupt ch0 0x8001
Info (DMAC) platform/dmac: configCh0Wr 0x00000000 enable 0 interrupts 0 halt 0
Info (DMAC) platform/dmac: regRd32 0x01800000 (25165824)
Info (DMAC) platform/dmac: regRd32 0x01800000 (25165824)
TEST DMA: Interrupt ch1 0x8001
Info (DMAC) platform/dmac: configCh1Wr 0x00000000 enable 0 interrupts 0 halt 0
TEST DMA: 2 interrupts received
TEST DMA: DMA result dst1 'Hello world.' dst2 'The whole world spread before you.'

OVPsim finished: Thu Mar 26 22:09:52 2015
```

19.6 Selecting using VLNV

A peripheral model can be uniquely referenced by including the path in the platform description at compile time but this builds a platform that is not portable. Much more useful is to be able to specify the model using Vendor Library Name and Version (VLNV) information and then at run-time reference the root of a library to use.

The `icmGetVlnvString` function allows a path to be generated using VLNV information and an environment variable, `IMPERAS_VLNV`, to specify the root of the library and select a specific model.

The arguments to `icmGetVlnvString` are:

`char *rootDirectory`	`NULL`, root directory or directories
`char vendor`	vendor directory or `"*"`
`char library`	library directory or `"*"`
`char name`	name directory or `"*"`
`char version`	version directory or `"*"`
`char file`	filename or `"pse"` or `"model"`

Setting `rootDirectory` to `NULL` specifies the model directory from the product installation. Alternatively, a string in the format of the `PATH` environment variable can be used; this is a ':' separated list of paths on Linux, a ';' separated list on Windows. `icmGetVlnvString()` will search the directories in order.

Setting `vendor`, `library` or `name` to an empty string (not `NULL`) or to "*" specifies a wild-card for this directory.

Setting `version` to an empty string (not `NULL`) or to "`*`" specifies the latest numeric version. Note that "`1.1.1`" is later than "`1.1`" and that "`zzz`" is later than any number.

`file` should be set to the exact filename expected or to the special strings "`pse`" or "`model`" to make the search platform independent. Note that Imperas peripheral models are always named "`pse.pse`" and processor and intercept models are names "`model.so`" on Linux or "`model.dll` on Windows.

`icmGetVlnvString` will return the expanded string if a match is found, or `NULL` if there is zero or more than one match. If NULL is returned, an error message is printed.

If you wish to try several VLNV searches, use `icmTryVlnvString`. This is identical to icmGetVlnvString but does not produce an error message (or change the simulator's error status).

19.6.1 Example VLNV searches

```
// On Linux find the latest PSE model called uart in any library by chipco.com
s1 = icmGetVlnvString("/home/models:/home/users", "chipco.com", "", "uart", "", "pse");

// On Windows find the latest OVP or1k processor model – note that OS-dependent suffix
// (.so or .dll) is omitted for portability
s2 = icmGetVlnvString("c:\ovp\models;c:\local", "ovpworld.org", "", "or1k", "", "model");

// Find the non-intrusive code coverage module
s3 = icmGetVlnvString(NULL, "imperas.com", "intercept", "icov", "", "model");

// select a 16450 UART version 1.7 from the default ovpworld.org peripheral library.
s4 = icmGetVlnvString(NULL, "national.ovpworld.org", "peripheral", "16450", "1.7", "pse");


// Try several places without producing an error
if (!(s = icmTryVlnvString(path, v1, l1, m1, "", "model"))) {
    s = icmTryVlnvString(path, v2, l2, m2, "", "model);
}
if(!s) {
    icmPrintf("Could not find your model in the usual places\n");
    return;
}
```

20 Nets connecting models

A single- or multi-bit wire can be modeled using a net. Interrupts and reset signals are typically implemented this way. Note that although it is possible to construct a signal-level model of a bus using ICM, this style of modeling is discouraged; no attempt has been made to optimize the net object, so the performance of a signal-level model would be disappointing.

Nets are created then connected to processor and peripheral models. The arguments to icmConnectProcessorNet and icmConnectPSENet are as follows:

```
icmConnectProcessorNet
    icmProcessorP   processor     Handle returned when creating the processor instance.
    icmNetP         net           Handle returned when created the net.
    const char      *portName     Name of the port as it appears in the model
    icmDirection    direction     ICM_INPUT or ICM_OUTPUT.
```

```
icmConnectPSENet
    icmPseP       pse          Handle returned when creating the PSE instance.
    icmNetP       net          Handle returned when created the net.
    const char   *portName     Name of the port as it appears in the model
    icmDirection  direction    ICM_INPUT or ICM_OUTPUT.
```

20.1 Example

This example shows the connection of the interrupt output from a UART to the interrupt input of the OR1K processor.

```
…
icmNetP     intNet = icmNewNet("int1");    // UART

const char *uartPse = icmGetVlnvString(NULL, "ovpworld.org", "", "16450", "", "pse");
icmPseP     uart1   =  icmNewPSE("uart1", uartPse, NULL, NULL, NULL);

const char *or1kModel  = icmGetVlnvString(vlnvRoot, "ovpworld.org","","or1k","","model");

icmProcessorP processor = icmNewProcessor(
    "cpu1",            // CPU name
    "or1k",            // CPU Type
    0,                 // CPU cpuId
    0,                 // CPU model flags
    32,                // address bits
    or1kModel,         // model file
    0,                 // not used
    0,                 // simulation attributes
    0,                 // user-defined attributes
    0,                 // semi-hosting file
    0                  // not used
);

icmConnectProcessorNet(processor, intNet, "intr0", ICM_INPUT);

icmConnectPSENet(uartPSE, intNet, "intOut", ICM_OUTPUT);
…
```

21 Detecting changes on a net

Code can be triggered when a net is written by a model. A function is declared using the `NET_WRITE_FN` prototype macro, then connected to a net using `icmAddNetCallback.` Note that the trigger occurs if the net is written. It is up to the application to decide if the net value has changed.

21.1 Example

Any change on an interrupt port is made to print a line to the simulator log stream. Note the use of the userData field to pass a pointer to the recorded old value.

```
NET_WRITE_FN(intNetWritten) {
    Uns32 *old = userData;
    if(value != *old) {
        icmPrintf(“Net changed to %d\n”, value);
        *old = value;
    }
}

void construct() {
    . . .

    // create a net and connect it to the interrupt port of a peripheral
    Uns32 oldValue = 0;
    icmNetP intNet = icmNewNet("intNet");
    icmConnectPSENet(pse10, intNet, "interrupt", ICM_OUTPUT);

    icmAddNetCallback(intNet, intNetWritten, &oldValue);

    // simulate for one simulated second
    icmSetSimulationStopTime(1.0);
    icmSimulatePlatform();

    . . .
}
```

22 Packetnets

Models that communicate with Ethernet, USB CAN, GSM etc. can use the packetnet abstraction of a packet based network. A packet transaction is modeled as an instantaneous event; network speed and latency must be modeled in the transmitting or receiving devices. A packetnet communicates by callbacks and shared memory. The transmitting model creates a packet in its local memory then calls the transmit function. This causes a notification function to be called in each receiving model in turn, passing a pointer to and number of bytes in the packet. The notification function can modify the data if required. When every notification function has returned, the transmit function returns, then the transmitting model can examine the packet if required.

Note that peripheral models each occupy their own address spaces. Therefore the simulator copies the data as and when required, so the models must not rely on pointers in the data. The contents of a received packet should not be used after the notification function has returned.

The order that the connected models receive a packet is determined by the order of construction in the ICM code, but should not be relied on.

The peripheral model API can send and receive through the packetnet interface. The ICM API is used to create connections during platform construction, but can also be a test-bench by sending and receiving packets.

22.1 Packetnet Direction

A packetnet is bidirectional; a model can send and receive from the same packetnet (though it does not have to).

22.2 Packetnet ports

A named packetnet port represents the connection between a packetnet and a peripheral model instance. The ICM test-bench does not use packetnet ports.

22.3 Recursion

Common to several methods of communication between models, it is possible by carelessly connecting packetnets to create a loop so that a call in one model results in a call back into the same function in that model. The simulator detects and prevents deep recursion on any packetnet.

A peripheral model will not receive notification for a packet that it is sending, an ICM notifier function in the test-bench will not receive packets sent by the test-bench.

22.4 Packet size

Physical networks have a maximum packet size. Larger data are broken into smaller units handled by the protocol stack. A peripheral model must specify the maximum number of bytes to be sent in one packet when it connects to a packetnet, though it can send fewer bytes if needed. All peripheral models on one packetnet must define the same maximum size. The ICM interface can connect models together without knowing the maximum size, but it is an error for the test-bench to transmit a packet larger than the size set by peripherals on the packetnet.

22.5 Packetnet functions

Create an instance of a packetnet in the platform (returns a handle to the new packetnet):

```
icmPacketnetP icmNewPacketnet(const char *packetnetName)
```

Connect a packetnet to a packetnet port on an instance of a peripheral:

```
icmConnectPSEPacketnet(icmPSEP pse, icmPacketnetP net, const char *portName)
```

Send a packet to all receivers on a packetnet:

```
void icmWritePacketnet(icmPacketnetP net, void *data, Uns32 bytes)
```

packetnet notification callback definition using macro defining the prototype, and function to bind the callback to a packetnet instance.

```
static ICM_PACKETNET_NOTIFIER_FN(cb) {
   ...
}

void icmAddPacketnetCallback(icmPacketnetP net, icmPacketnetNotifierFn cb, void *userData)
```

22.6 Example

An example using a packetnet is in:

```
$IMPERAS_HOME/Examples/Models/Peripherals/packetnet
```

The example uses imperas igen to construct most of the code for the platform and peripheral models. Take a copy of the example:

```
cp $IMPERAS_HOME/Examples/Models/Peripherals/packetnet .
```

Construct and compile the platform and peripheral models and compile the application:

```
make all
```

The packetnet is created and connected to the peripheral model instances in code generated by igen here:

```
platform/platform.constructor.igen.h    (look for the CONNECTIONS comment)
```

The packetnet notifier function, its installer, and code to send a packet over the packetnet is in hand-written code here:

```
platform/platform.c    (look for pktGotData and writePkt)
```

Run the example:

```
make simulate
```

Referring to the output:

The application code starts running

```
Info PACKETNET TEST Application
```

Writing to the tx register in peripheral pktmodel1 calls the callback txWrite (see peripheral/user.c) which makes pktModel1 send a packet using ppmPacketnetWrite. The packet is received by pktModel2 and the callback pktGotData in the platform. Then the txWrite function in pktModel1 returns:

```
Info (PKT_PSETXS) testpacketnet/pktModel1: PSE to packetnet START {77} {PSE} {00}
Info (PKT_PSERXD) testpacketnet/pktModel2: Peripheral PKT model Trigger ...
Info (PKT_ICMRXD) testpacketnet/top: Platform PKT testbench Trigger ...
Info (PKT_PSETXE) testpacketnet/pktModel1: PSE to packetnet DONE  {77} {PSE} {02}
```

Then the application makes pktModel2 transmit in the same way:

```
Info (PKT_PSETXS) testpacketnet/pktModel2: PSE to packetnet START {88} {PSE} {00}
Info (PKT_PSERXD) testpacketnet/pktModel1: Peripheral PKT model Trigger ...
Info (PKT_ICMRXD) testpacketnet/top: Platform PKT testbench Trigger  ...
Info (PKT_PSETXE) testpacketnet/pktModel2: PSE to packetnet DONE  {88} {PSE} {02}

Info PACKETNET TEST Application DONE
```

When the application finishes, the test-bench sends a packet which is received by the peripherals.

```
Info (PKT_ICMTXS) testpacketnet/top: writePkt packetbus1 START {01}, {ICM}, {00}
Info (PKT_PSERXD) testpacketnet/pktModel1: Peripheral PKT model Trigger ...
Info (PKT_PSERXD) testpacketnet/pktModel2: Peripheral PKT model Trigger ...
Info (PKT_ICMRXD) testpacketnet/top: Platform PKT testbench Trigger ...
Info (PKT_ICMTXE) testpacketnet/top: writePkt packetbus1 DONE {01}, {ICM}, {03}
```

23 Simulator Control Files

Simulator control files are supported by CpuManager and OVPsim.

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

For details of the control file refer to the OVP_Control_File_User_Guide.

Control files are loaded using `icmAddControlFile()` which must be invoked before calling icmInitPlatform().

```
...

icmAddControlFile( "control1.ic" );
icmAddControlFile( "control2.ic" );

icmInitPlatform(...);

icmSimulatePlatform();
```

Control files can also be specified using the environment variable `IMPERAS_TOOLS`. Filenames are separated by `:` (Linux) or `;` (Windows) e.g.

```
shell> export IMPERAS_TOOLS="controlfile1.ic;controlfile2.ic"

shell> mySystemC.exe    # simulator using CpuManager.so
```

24 Encapsulating Models for use in other Environments

An essential purpose of the ICM API is to allow Imperas simulation models to be exported to other environments (for example, SystemC).

24.1 SystemC

The ICM API allows the Imperas models to be exported into a SystemC environment. There are two levels at which the ICM API can be used: C and C++. It is the C++ API that is utilized in the SystemC environment. Once exported, an Imperas model can be controlled from the SystemC interface by, for example, allowing it to be clocked one instruction at a time.

The following code example illustrates a few of the basic principles when using the ICM API to encapsulate a model for use in a SystemC environment. Specifically, we cover how to:

- Create a processor instance
- Attach external memory
- Register callbacks

The example code is found in the directory `systemC.` It consists of an application to be run on the processor model (in the *application* directory) and a SystemC platform (in the *platform* directory).

The example instantiates a single processor and, depending upon build commands, an external memory modeled as a SystemC module.

24.1.1 Create Processor Instance

The processor is created using constructs in the file *platform/impProcessor.hpp* that create a SystemC module using the SC_MODULE macro.

This file contains the processor constructor and destructor:

```
/////
//
// Constructor
//
theProcessor(sc_module_name instname, sc_clock &ck)
{
   _init = false;
   _proc = NULL;
   _inclk(ck);

   SC_METHOD(runner);
   sensitive << _inclk.neg();
   dont_initialize();
}

/////
//
// Destructor
//
~theProcessor()
{
   if (_init && _proc) {
      delete _proc;
     _proc = NULL;
      _init = false;
   }
}
```

An initialization routine is created separately from the constructor. This allows it to be explicitly called and better error handling.

As part of the initialization routine a *simulation stop* callback is registered, using the C++ API function `icmRegisterSimStopHandler`. Without this function added the SystemC environment would not terminate and continue to clock the CpuManager model even after the simulation has finished. Now when simulation finishes CpuManager makes a call into the supplied function which can be used to cleanly terminate the SystemC simulation.

```
/////
//
// Init function
//
bool init( const char *procName,
           const char *typeName,
           int         cpuId,
           const char *morpherFile,
           const char *morpherSymbl,
           const char *semiHostFile,
           const char *semiHostSymbol )
{
   bool retcode = false;

   char busName[strlen(procName)+10];
   sprintf(busName, "%s_bus", procName);

   _bus  = new icmBusObject(
                  busName,       // bus name
                  32             // address bits
               );

   _proc = new icmProcessorObject (
                  procName,      // processor name
                  typeName,      // processor type
                  cpuId,         // processor ID
                  0,             // processor model flags
                  32,            // address bits
                  morpherFile,   // processor model file
                  morpherSymbol, // morpher attributes
                  0,             // processor attributes
                  0,             // user-defined attributes
                  semiHostFile,  // semi-hosting library file
                  semiHostSymbol // semi-hosting attributes
                );

   _proc->connectBusses(*_bus, *_bus);

   if (_proc) {
      // register a handler for the simulator stop conditions
      _proc->icmRegisterSimStopHandler(_shandle);

      retcode = true;
   }

   return retcode;
}
```

The application code is loaded by overloading the << operator.

```
   /////
   //
   // Load the object file into processor memory
   //
   void objfile(const char *objfile)
   {
      // If processor was successfully created then
      // load the program image into its memory
      if (_proc) {
         // load object file
         (*_proc) << objfile;
      }
   }
```

The processor model is stepped using the overloaded << operator. The number to the right specifies the number of instructions to step.

```
   /////
   //
   // The SC_METHOD connected to the clock
   //
   void runner(void)
   {
     // step processor
     (*_proc) << 1;
   }
```

An external memory creation function (`extMem`) is also in this file. This function calls the ICM C++ API function `mapExternalMemory` to map the external SystemC memory region into the address space of the processor. This actually attaches two callback functions, `rcb` and `wcb`, that are used by the simulator to perform the access into the SystemC memory, for reads and writes respectively.

```
   /////
   //
   // Maps a memory range to a memory manager outside the simulator.
   //
void extMem(Addr loaddr, Addr hiaddr, icmMemWriteFn wcb, icmMemReadFn rcb, impMemory *mem)
   {
      _bus->mapExternalMemory(
        "external",
               ICM_PRIV_RW,
               loaddr,
               hiaddr,
               wcb,
               rcb,
               mem
        );
   }
```

24.1.2 External SystemC Memory

In this section we are defining a SystemC memory and its access functions used by the ICM platform. For convenience we are defining our own SystemC memory but any SystemC memory could have been used.

The memory is defined using constructs in the file `platform/impMemory.hpp` that create a SystemC module using the `SC_MODULE` macro.

This file contains the memory constructor and destructor functions. Note that the memory has a SystemC clock but this is not used in this example. The memory model could be extended using the SystemC clock to provide latency generation on the memory accesses.

```
   impMemory(sc_module_name instname, sc_clock &ck)
   {
      _init = false;
      _mem  = NULL;

      _inclk(ck);
   }

   ~impMemory()
   {
      if (_init && _mem) {
         delete _mem;
         _mem = NULL;
         _init = false;
      }
   }
```

The memory initialization routine declares a byte array to be used for storage and defines the configuration (address and size) of the memory block.

```

   bool init( Addr baseAddr, Uns32 size)
   {
      bool retcode = false;

      _mem = new char[size];
      if (_mem) {
         _init                = true;
         _base                 = baseAddr;
         _size                 = size;
         retcode      = true;
      }

      return retcode;
   }
```

The read and writes to the memory are contained within a single function; separate read and write functions could have been used. Within this function is also included some basic range checking for the accesses. The data size (bytes) is transferred between the local memory byte array and the pointer passed as `void *value`.

```
   bool inline inrange(Addr a)

      return ( (a>=_base) && (a<=(_base+_size-1)) ) ? true: false;
   }

   typedef enum { tWrite, tRead } trans_t;

   bool transaction(trans_t tr, Addr address, void *value, Uns32 bytes)
   {
      bool retcode = false;

      if (_init) {
         if ( inrange(address) && inrange(address+bytes-1) ) {
            Addr adr = address - _base;
            char *src, *dst;
            switch (tr) {
               case tWrite:
                  src = (char*)value;
                  dst = &(_mem[adr]);
                  break;
               case tRead:
                  src = &(_mem[adr]);
                  dst = (char*)value;
                  break;
               default:
                  assert(0);
                  break;
            }

            memcpy(dst, src, bytes);

            retcode = true;
         }
      }

      return retcode;
   }
```

The ICM platform accesses these callback functions whenever a read or write access is made to an address which falls in a region that has been mapped as external using the `mapExternalMemory` function we saw earlier. The simulator uses the callback functions rather than using the local memory space.

```
using namespace icmCpuManager;

extern "C" {

static ICM_MEM_WRITE_FN(memWriteCB)
{
   if (processor) {
      impMemory *m = (impMemory*) userData;
      if (m) m->transaction(impMemory::tWrite, address, value, bytes);
   }
}

static ICM_MEM_READ_FN(memReadCB)
{
   if (processor) {
      impMemory *m = (impMemory*) userData;
      if (m) m->transaction(impMemory::tRead, address, value, bytes);
   }
}
```

24.1.3 Example Platform

The example platform uses the functions defined in the previous sections to create a system containing: one OR1K processor instance and one SystemC memory instance. An application binary executed on the processor initializes an array in the external memory and then uses the values in this array as indexes for accessing a local dictionary of words to create an output sentence. The SystemC clock is used to control the operation.

The following code provides an example of the instantiation of a single OR1K processor that is clocked using a SystemC sc_clock object. There is a single memory in the SystemC environment that is mapped into a region of external memory and accessed via callback functions.

A SystemC clock is created with a 1uS period. In this example the clock frequency is arbitrary and has no effect on the simulation.

```
        // define clock period
        const sc_time clk_period(1, SC_US);

        systemClock = new sc_clock("SYSCLK", clk_period);
```

Create a new processor named ‘cpu1’ with the SystemC clock attached. Perform initialization of the processor.

```
        proc = new theProcessor("cpu1", *systemClock);

        retCode = proc->init(
            "CPU1",
            “or1k”,
            0,
            model,
            0,
            semihosting,
            0
         );
```

If the processor was created successfully we then create a memory and register the callback functions so that it can be accessed by the simulator. There is code here to allow the memory to be either specified using SystemC memory or using simulator memory, depending on whether `EXTMEM_BASEADDR` is non-zero:

```
        if (retCode && EXTMEM_BASEADDR) {

           // Create a memory object
           extMemory = new impMemory("EXTMEM", *systemClock);
           retCode = extMemory->init(EXTMEM_BASEADDR, EXTMEM_SIZE);

           icmPrintf(
               "SYSTEMC: Creating memory at %08x, size %d bytes\n",
               EXTMEM_BASEADDR, EXTMEM_SIZE
            );

             if (retCode) {
                 icmPrintf("SYSTEMC: Callback routines registered\n");
                 proc->localMem("local1", 0, EXTMEM_BASEADDR-1);
                 proc->localMem("local2", EXTMEM_BASEADDR+EXTMEM_SIZE, 0xffffffff);
                 proc->extMem(
                     EXTMEM_BASEADDR,
                            (EXTMEM_BASEADDR+EXTMEM_SIZE-1),
                            memReadCB,
                            memWriteCB,
                            extMemory
                     );
             }

         } else {

             proc->localMem("local", 0, 0xffffffff);
         }
```

Next, we load the application code to be executed.

(Note: If any of the application code resides in the external memory, the simulator will used the callback functions to access the memory and perform the program initialization.)

```
         if (retCode) {
            proc->objfile(PROC_OBJFILE);
         }
```

Simulation is started using the SystemC `sc_start()` function. This will cause the simulation to run continuously while there are events being generated. As the platform is generating clocks this simulation would not stop, even after the application program had completed, without the `simStopHandler` having been registered.

```
         // Processor and memory ready to simulate
         if (retCode) {
            icmPrintf("\nSYSTEMC: Starting Simulation ... \n");
            sc_start();
         }
```

24.1.3.1 Running the SystemC Example

The example is found in the `systemC` directory.

```
$IMPERAS_HOME/Examples/Platforms/systemC
```

This example require the availability of a systemC installation pointed to by an environment variable SYSTEMC_HOME

24.1.3.2 Platform with local Memory

Compile the systemC platform and the test application using the following command in the systemC directory

```
make –C application
make
```

Run the systemC platform using the following command in the systemC directory

```
impSimulation.exe --program application/application.OR1K.elf
```

An output similar to the following should be seen:

```

             SystemC 2.3.0-ASI --- Aug  1 2012 13:30:30
        Copyright (c) 1996-2012 by all Contributors,
        ALL RIGHTS RESERVED

OVPsim (32-Bit) v20150205 Open Virtual Platform simulator from www.OVPworld.org.
Copyright (c) 2005-2015 Imperas Software Ltd.  Contains Imperas Proprietary Information.
Licensed Software, All Rights Reserved.
Visit www.IMPERAS.com for multicore debug, verification and analysis solutions.

OVPsim started: Wed Dec 12 17:35:46 2012


Simulation started with the following parameters:
MORPHER_FILE:Imperas/lib/Linux32/ImperasLib/ovpworld.org/processor/or1k/1.0/model.so
SEMIHOST_FILE:Imperas/lib/Linux32/ImperasLib/ovpworld.org/semihosting/or1kNewlib/1.0/model
.so
APPLICATION_OBJFILE:   application/application.OR1K.elf


SYSTEMC: Starting Simulation ...

1) Internal memory allocated for string

Array 'extIndex' defined at address 0xeff0 with size 0x14

2) Copied indices from internal memory to 'extIndex'

3) Use data from 'extIndex' to index internal dictionary

extIndex[0]=3
extIndex[1]=1
extIndex[2]=2
extIndex[3]=4
extIndex[4]=0

String from memory: 'Imperas: Multicore design simplified'

EXIT instructions 25400

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

OVPsim finished: Wed Dec 12 17:35:46 2012


OVPsim (32-Bit) v20150205 Open Virtual Platform simulator from www.OVPworld.org.
Visit www.IMPERAS.com for multicore debug, verification and analysis solutions.
```

24.1.3.3 Platform with SystemC Memory

When the previous example is run, the application prints out the location of the buffer "extIndex" to which the string is written.

```
Array 'extIndex' defined at address 0xeffc with size 0x14
```

In this example a systemC memory will be created at this address and this region mapped as external memory in the platform.

In the Makefile find the section that defines membase and memsize. Uncomment the fields and add the information found from the previous run of the application.

```
# Makefile Options
ifeq (1,${EXTMEMORY})
   # !! Change membase and memsize if you are building with EXTMEMORY=1
   #membase = 0xeffc
   #memsize = 0x14
```

The SystemC memory is created in the file, platform/impSimulation.cpp and the region mapped as external.

```
       // If PROC was successfully created then attach some external memory to it
        if (retCode && EXTMEM_BASEADDR) {

            // Create a memory object
            extMemory = new impMemory("EXTMEM", *systemClock);
            retCode = extMemory->init(EXTMEM_BASEADDR, EXTMEM_SIZE);

            icmPrintf(
                "SYSTEMC: Creating memory at %08x, size %d bytes\n",
                EXTMEM_BASEADDR, EXTMEM_SIZE
            );

            if (retCode) {
                icmPrintf("SYSTEMC: Callback routines registered\n");
                proc->localMem("local1", 0, EXTMEM_BASEADDR-1);
                proc->localMem("local2", EXTMEM_BASEADDR+EXTMEM_SIZE, 0xffffffff);
                proc->extMem(
                    EXTMEM_BASEADDR,
                           (EXTMEM_BASEADDR+EXTMEM_SIZE-1),
                           memReadCB,
                           memWriteCB,
                           extMemory
                    );
            }
```

The same application will work if the internal array 'extIndex' is now mapped to external systemC memory. After the array address and size have been specified in the Makefile rebuild the simulator.

```
make clean
make EXTMEMORY=1
```

That will build a SystemC simulation that uses external memory that can be run using the command

```
impSimulation.exe --program application/application.OR1K.elf
```

The following output should now be seen

```
             SystemC 2.3.0-ASI --- Aug  1 2012 13:30:30
        Copyright (c) 1996-2012 by all Contributors,
        ALL RIGHTS RESERVED

OVPsim (32-Bit) v20150205 Open Virtual Platform simulator from www.OVPworld.org.
Copyright (c) 2005-2015 Imperas Software Ltd.  Contains Imperas Proprietary Information.
Licensed Software, All Rights Reserved.
Visit www.IMPERAS.com for multicore debug, verification and analysis solutions.

OVPsim started: Wed Dec 12 17:35:46 2012


Simulation started with the following parameters:
MORPHER_FILE:Imperas/lib/Linux32/ImperasLib/ovpworld.org/processor/or1k/1.0/model.so
SEMIHOST_FILE:Imperas/lib/Linux32/ImperasLib/ovpworld.org/semihosting/or1kNewlib/1.0/model
.so
APPLICATION_OBJFILE:  application/application.OR1K.elf


SYSTEMC: Starting Simulation ...

SYSTEMC: Creating memory at 0000effc, size 20 bytes
SYSTEMC: Callback routines registered


1) Internal memory allocated for string

Array 'extIndex' defined at address 0xeffc with size 0x14

2) Copied indices from internal memory to 'extIndex'

3) Use data from 'extIndex' to index internal dictionary

extIndex[0]=3
extIndex[1]=1
extIndex[2]=2
extIndex[3]=4
extIndex[4]=0

String from memory: 'Imperas: Multicore design simplified'
```

25 Using Native Memory

Some applications of ICM require that simulated memory be allocated by the application rather than using `icmNewMemory`. The function `icmMapNativeMemory` enables such use of native memory in a platform.

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

25.1 Example

```
icmBusP    bus    = icmNewBus("bus", 32);

// a very large (1Gb) ICM memory.
icmMemoryP mem1   = icmNewMemory("mem1", ICM_PRIV_RW, 0x40000000);

// A small (64K) area of native memory
Uns32      msize = 0x1000;
void      *mem2  = malloc(msize);

// connect both memories
icmConnectMemoryToBus(bus, "p1", mem1, 0x80000000);
icmMapNativeMemory(bus, 0, msize-1, mem2);

// A processor connected to 'bus' will see memory mapped at
// 0x0-0xFFF and 0x80000000-0xBFFFFFFF
```

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

The simulator's object loader can be used to load external memory using `icmLoadNativeMemory:`

```
icmBusP       bus  = icmNewBus(...);
icmProcessorP proc = icmNewProcessor(...);

// (64K) area of native memory
Uns32      msize = 0x1000;
void      *mem  = malloc(msize);
Addr       base = 0x0;

// connect memory to bus

icmMapNativeMemory(bus, base, msize-1, mem);

// load it with a program
icmLoadNativeMemory(mem, msize, base, "myprog.elf", 0, True, 0, proc);
```

25.2 Combining External 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 `icmMapExternalMemory` and `icmMapNativeMemory`). 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 `icmMapExternalNativeMemory` can be used:

```
void icmMapExternalNativeMemory (
    icmBusP       bus,
    const char*   portName,
    icmPriv       priv,
    Addr          lowAddr,
    Addr          highAddr,
    icmMemReadFn  readCB,
    icmMemWriteFn writeCB,
    void*         memory,
    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`.

26 Simulation Optimization

A simulator using the ICM 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 ICM API call (e.g.. `icmWriteProcessorMemory()`), then the simulator must discard and re-generate its optimizations. If however, the code is stored in memory created using `icmMapNativeMemory()` and modified by an agent external to ICM, the simulator will not know its code is invalid. The Programmer must therefore notify the simulator using `icmFlushProcessorMemory()`.

26.1 Example

```
icmBusP       bus   = icmNewBus("bus", 32);
icmProcessorP proc  = icmNewProcessor(……);

// A 64K area of native memory
Uns32      msize = 0x1000;
void      *mem2  = malloc(msize);

// connect memory
icmMapNativeMemory(bus, 0, msize-1, mem2);

// connect processor
icmConnectProcessorBusses(proc, bus, bus)
…
// . . . during simulation . . .

memcpy(mem2, newData, msize);
icmFlushProcessorMemory(proc, 0, msize-1);

// When proc is allowed to continue, it will re-optimize any code that has been
// executed from this memory.
```

27 Plugin Commands

A processor model or an intercept object or plugin 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 plugin or to extract analysis data that the model or plugin 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 plugin using `vmiosAddCommand()`. See OVP_VMI_OS_Support_Function_Reference.doc, and `icmAddInterceptObject()` in this document.

Commands are called using `icmCallCommand`(). 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 plugins are recommended to install their commands in their constructors, in which case the earliest 'safe' time to call a command is immediately before `icmSimulate`() or `icmSimulatePlatform`().
This example calls two commands; before simulation and before shutdown.

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

    const char *vlnvRoot  = NULL;
    const char *model     = icmGetVlnvString(vlnvRoot,
                            "ovpworld.org", "processor", "or1k", "1.0", "model");
    const char *intercept = icmGetVlnvString(vlnvRoot,
                            "myco.org", "intercepts", "countThings", 0, "model");

    icmInitPlatform(ICM_VERSION, 0, 0, 0, “platform”);

    icmProcessorP processor = icmNewProcessor(
        "cpu1",             // CPU name
        "or1k",             // CPU type
        0,                  // CPU cpuId
        0,                  // CPU model flags
        32,                 // address bits
        model,              // model file
        0,                  // not used
        0,                  // enable tracing etc
        0,                  // user-defined attributes
        0,                  // no semihost or intercept library
        0                   // not used
    );

    icmAddInterceptObject(
        processor,
        "plugin1",
        intercept,
        0,                   // not used
        0
    );

    icmLoadProcessorMemory(processor, "program.elf", ICM_LOAD_DEFAULT, False, True);
    // Call this command before simulation
    char *argv[2] = { "logging", "-on" };

    icmCallCommand(
        "cpu1",      // instance that has the command
        0,           // it's in the processor, not in its plugin
        argv[0]      // name of the command
        2,           // number of arguments
        argv         // argument array (by convention including the command)
    );

    icmSetSimulationStoptime(0.37);  // stop after this many seconds
    icmSimulatePlatform();

    // stopped again after 0.37 secs
    argv[0] = "showResults";
    argv[1] = "-all";

    // Call this command during simulation
    icmCallCommand(
        "cpu1",      // instance
        "plugin1",   // name of plugin containing the command
        argv[0]      // name of the command
        2,           // number of arguments
        argv         // argument array (by convention including the command)
    );

    icmTerminate();

    // do NOT try calling a command here

    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/Platforms/callCommand.
```

This uses the MIPS32 model commands.

In platform/platform.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", "16", "0"};
```

When calling the command this array is passed to the icmCallCommand function.

```
    result = icmCallCommand("cpu1", NULL, cmd1Argv[0], 3, &cmd1Argv[0]);
```

The result returned from the icmCallCommand 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.

This example also shows setting the “pluginInstanceName” argument to NULL when the command is on a model itself rather than on a plugin loaded onto a model.

27.1 Discovering Installed Commands

An ICM application can discover what commands are available, for the whole platform or for a particular processor model.

An example of discovering commands that are created within a processor model is in

```
Imperas/Examples/Platforms/callCommand.
```

This uses the MIPS32 model commands.

A function is declared using the provided prototype macro. Arguments to the function are: platform name (if supplied), model name, plugin name (if from a plugin), command name short help string (if supplied) and user data.

```
static ICM_INSTALLED_COMMAND_FN(printCommand) {
...
}
```

The function printCommands will be called for every installed command, in the platform:

```
 icmGetAllPlatformCommands(printCommand, userData);
```

... or in a particular processor:

```
icmProcessorP proc = icmFindProcessorByName(procName);

icmGetAllProcessorCommands(proc, printCommand, userData);
```

28 Multicore (SMP) Support

Multicore processors have information at their 'root' level - accessed from the handle returned during construction, and also on processor objects under the root level - accessed through handles obtained by iterator or callback functions. Each sub-processor has a unique name and a description which indicates its function within the multicore model. This is a summary of the functions used to access a multicore processor:

function	use
icmGetSMPParent	Return the parent of the given processor.
icmGetSMPChild	Return the first child of the given processor.
icmGetSMPPrevSibling	Return the previous sibling of the given processor
icmGetSMPNextSibling	Return the next sibling of the given processor
icmGetSMPIndex	Return the index number of the given processor
icmSMPIsLeaf	Return true if the given processor has no children
icmIterAllChildren	Call the given function on each child of the given processor; finds all processor at one level.
icmIterAllDescendants	Call the given function on each descendant of the given processor. From the root, this will find the whole hierarchy, except the root
icmIterAllProcessors	Call the given function on each descendant of the given processor. From the root, this will find the whole hierarchy, including the root
icmGetProcessorDesc	Return a string describing the type of the given processor. Can be null.
icmGetProcessorName	Return the (unique) name of the current processor

28.1.1 Controlling an SMP

This is a summary of operations that can be applied to the root of an SMP and to sub-processors (if they exist):

operation	allowed on root	allowed on children
icmTraceOnAfter, etc.	y (applies to all children)	y
icmDebugThisProcessor	y	n
icmSetGdbPath	y	n
icmReadProcessorMemory etc	y (uses current TLB state)	y (uses current TLB state)
icmReadReg, icmSetPC etc.	y (if register is present)	y (if register is present)
Freeze, unfreeze etc.	n	y
set/clear breakpoints	y (applies to all children)	y

29 Processor Information

The ICM API can be used to find information that is embedded in a processor model. This information is typically used by:

- a debugger to present information to the user
- on-line documentation generators

All these functions require a handle to a processor instance. Any parameters passed to the instance when it was created (such as the variant) will be reflected in the model instance.

There is an example in:

```
$(IMPERAS_HOME)/Examples/Platforms/queryProcessor
```

The example in `platform/platform.c` creates instances of various processor models from the library (note that some of these models might not be available on your release). It then uses the functions described below to extract information from the models.

29.1 Processor Documentation

29.1.1 Documentation Nodes

Processor documentation is presented as a hierarchy of nodes. Each node can be a section title or section content and can have one or more child nodes. The 'root' node is found using icmGetProcessorDoc. The following functions are used to traverse the hierarchy:

function	**use**
icmGetProcessorDoc	Returns the root document node.
icmDocNextNode	Get the next sibling of the current node.
icmDocChildNode	Get the first child of the current node.
icmDocText	Return the text from this node
icmDocIsText	Returns True if this node is text content, False if it is a title or heading.

29.1.2 Processor Simulation Information

These functions return information from a processor model useful to a simulator or debugger:

function	**use**
icmGetProcesorVlnv	Returns information about where the model is stored in an Imperas release. This information will usually match where the model was found in the first place.
icmGetProcessorElfCode	Get 1 or more the ELF codes expected by this model. This should normally match the type code found in ELF program files loaded for execution by this model.
icmGetProcessorEndian	Get the endianness supported by this model.
icmGetProcessorGdbPath	Get the path to the gdb debugger matching this

	processor in the Imperas release.
icnGetProcessorDefaultSemihost	Returns information about where the default semihost library for the model is stored in the Imperas release.

29.2 Processor Parameters

The behavior of a processor model can be changed by setting name/value pairs in the simulator prior to simulation. These pairs are referred to as *attributes* when being set and *parameters* when being read. Processor parameter names, types and descriptions can be discovered from the ICM API. Iterators and accessor use icmParamInfoP, which is a handle to a unique parameter of a processor.

function	**use**
icmGetNextParameterInfo	Return the handle of the first or subsequent parameter of this processor.
icmGetParamName	Return the parameter name.
icmGetParamType	Return the parameter type as an enumeration.
icmGetParamTypeString	Return the parameter type as a string.
icmGetParamDesc	Return a short description of the parameter.

Parameter types are as follows:

enumeration	**string**	**use**
ICM_PT_BOOL	Boolean	True if set to 1,y or t; false otherwise
ICM_PT_INT32	Int32	32 bit signed integer
ICM_PT_UNS32	Uns32	32 bit unsigned integer
ICM_PT_UNS64	Uns64	64 bit unsigned integer, also used for addresses.
ICM_PT_DOUBLE	Double	Double precision floating point.
ICM_PT_STRING	String	Arbitrary text string
ICM_PT_ENUM	Enumeration	String whose value must be one of a specified set.
ICM_PT_ENDIAN	Endian	enumeration with values 'big' or 'little'

The enumeration names and values of a parameter of type ICM_PT_ENUM can be obtained using the icmParamEnumP handle. Note that the mapping of the string to an integer is private to the model.

function	**use**
icmGetNextParamEnum	Return the handle of the 1st or subsequent enumeration
icmGetParamEnumName	Return the name (string value) of the enumeration
icmGetParamEnumValue	Return the integer value of the enumeration.
icmGetParamEnumDesc	Return a short a description of the enumeration.

29.3 Processor Ports

Processor bus, net and FIFO ports can be discovered from the ICM API. The three port types are distinct and are accessed through different handle types.

handle	use
icmBusPortInfoP	handle to a bus port description
icmNetPortInfoP	handle to a net port description
icmFifoPortInfoP	handle to a FIFO port description

29.3.1 Bus ports

These functions are used to query a model's bus ports. Note that most processor models have two bus ports; INSTRUCTION and DATA; DSP and other special processors might have others.

function	use
icmGetNextBusPortInfo	Return the next bus port information handle
icmGetBusPortName	Return the name of the bus port.
icmGetBusPortType	Return the type of bus port as an enumeration
icmGetBusPortTypeString	Return the type of bus port as a string
icmGetBusPortDomainType	Return the domain type of bus port as an enumeration
icmGetBusPortDesc	Return a short description of the bus port
icmGetBusPortAddrBits	Return the width of the address bus, in bits.

Types of bus port:

type	string	description
ICM_BPT_MASTER	Bus master	creates bus transactions
ICM_BPT_SLAVE	Bus slave	responds to bus transactions
ICM_BPT_MASTERSLAVE	Bus master/slave	creates and responds

Types of bus domain:

type	string	description
ICM_DOM_CODE	Code domain	Used to fetch program code
ICM_DOM_DATA	Data domain	Used to read write data
ICM_DOM_OTHER	Other domain	Other uses.

29.3.2 Net ports

These functions are used to query a model's net ports. The most common use of net ports are for reset and for interrupt inputs.

function	**use**
icmGetNextNetPortInfo	Return the next net port information handle
icmGetNetPortName	Return the name of the net port.
icmGetNetPortType	Return the type of net port as an enumeration
icmGetNetPortTypeString	Return the type of net port as a string
icmGetNetPortDesc	Return a short description of the net port

Types of Net port:

type	**string**	**description**
ICM_NPT_INPUT	Input	Input pin
ICM_NPT_OUTPUT	Output	Output pin
ICM_NPT_INOUT	Inout	Bidirectional pin

29.3.3 FIFO ports

Some processors have serial interconnections typically used to build large processor arrays. The connections are modeled using FIFOs which can be queried using these functions:

function	**use**
icmGetNextFifoPortInfo	Return the next FIFO port information handle
icmGetFifoPortName	Return the name of the FIFO port.
icmGetFifoPortWidth	Number of bits in each word read or written
icmGetFifoPortType	Return the type of FIFO port as an enumeration
icmGetFifoPortTypeString	Return the type of FIFO port as a string
icmGetFifoPortDesc	Return a short description of the FIFO port

Types of FIFO port:

type	**string**	**description**
ICM_FPT_INPUT	Input	FIFO Input
ICM_FPT_OUTPUT	Output	FIFO Output

29.3.4 Fetching information from an SMP

This is a summary of data that can be found on the root of an SMP and on the sub-processors (if they exist):

data	**appears on the root**	**appears on sub-processors**
document node	y	n
Bus port	y	n
Net Port	y	n
FIFO port	y	n
Parameter	y	n
Name	y	y
Description	y	y
Index	y	y
Registers	y	y

30 Custom Object Readers

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

A new reader is installed using `icmInstallObjectReader()`. An example of a custom reader is in:

```
$IMPERAS_HOME/Examples/Platforms/objectReader.
```

In platform/platform.c the custom reader is installed before the processor memory is loaded (otherwise the reader will not be available).

```
    // install the new object reader
    icmInstallObjectReader("loader/model.so");
```

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

30.1 Writing a custom reader

This is covered in Custom_Object_Reader.pdf.

31 Command Line Parsing

A command line parser is available in the ICM API. This provides:

- The ability to read simulator flags from the user’s command line.
- A standard mechanism for parsing flags with boolean, string and numeric values.

31.1 Simulator Flags

In its simplest form, call icmCLParseStd() from main() BEFORE calling icmInitPlatform:

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

    // Create and use the standard argument parser
    icmCLPP parser = icmCLParser("myPlatform ", ICM_AC_ALL);
    cmdParserAddUserArgs(parser);

    icmCLParseArgs(parser, argc, argv);

    // initialize CpuManager
    icmInitPlatform(ICM_VERSION, 0, 0, 0, “myPlatform”);

    …
```

With this call, the program will parse most arguments available in the simulator control file (see Imperas Control File User Guide), and most arguments accepted by imperas.exe; –help shows all available flags. The function is specified as follows:

```
icmCLParse (
    const char      *executableName,    // for use in error reports
    icmCLPArgClass   use,               // choose which arguments to parse
);
```

The use argument is set as follows:

value	**meaning**
ICM_AC_NONE	No simulator arguments
ICM_AC_BASIC	Basic arguments
ICM_AC_EXT1	Extended argument set
ICM_AC_EXT2	Further extended argument set
ICM_AC_ALL	All arguments (available in some products)

To set the default help and usage message (perhaps an overview of the platform and its controls) use:

```
icmCLParseUsageMessage(
    icmCLPP parser,           // Existing parser
    const char      *message  // default message
);
```

This will appear as part of the –help output or when a command line error is detected.

31.2 User-specified Flags

To parse arguments specific to the platform use the following idiom:

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

    Bool goVal;

    // Create the standard argument parser
    icmCLPP parser = icmCLParser("myPlatform ", ICM_AC_ALL);

    // Add your arguments here
    icmCLParserAdd(parser, “go”, “G”, 0, ICM_AT_BOOLVAL, &goVal, “Say GO”, 0, userData);

    // Parse the arguments
    icmCLParseArgs(parser, argc, argv);

    // initialize CpuManager
    icmInitPlatform(ICM_VERSION, 0, 0, 0, “myPlatform”);

    …
```

In this example, if the program command line includes –go then the CLP sets the value ‘goVal’.

The function to add a custom argument is:

```
icmCLParserAdd (
    icmCLPP          parser,       // parser handle
    const char      *name,         // full name for the flag (without the ‘-‘)
    const char      *shortName,    // optional short name for the flag (without the ‘-‘)
    const char      *argDesc,      // Describe the value that follows the flag (if reqd)
    const char      *group,        // argument group
    icmCLPArgType    type,         // Enumeration (see below)
    void            *ptr,          // Ptr to value or function
    const char      *description,  // Description for –help
    Uns32            mask,         // Set to zero
    Uns32            userData,     // 2nd argument to callback (if required)
    Bool             mandatory     // If set, this argument _must_ be specified
 );
```

Note that the flag name and short name must not clash with existing names; a run-time error will be produced if they do.

The `group` argument allows command line arguments to be grouped to clarify the output of –help – all arguments of the same group appear together. If `group` is null, the argument will appear in the `platform` group.

The `type` argument to icmCLParserAdd specifies the argument type. Options are:

Enumeration	**Flag is followed by**	**Action**
ICM_AT_BOOLVAL	nothing	Sets a boolean variable
ICM_AT_INT32VAL	+ve or –ve integer	Sets a 32bit integer variable
ICM_AT_UNS64VAL	+ve 64 bit integer	Sets a 64bit integer variable
ICM_AT_DOUBLEVAL	floating point number	Sets a double variable
ICM_AT_STRINGVAL	a string	Sets a string variable
ICM_AT_STRINGLIST	a list of strings *	Adds to a list of strings
ICM_AT_PAIRLIST	a list of pairs of the form name=value	Adds to a list of name,value pairs

ICM_AT_FC_BOOLEAN	nothing	Calls a function
ICM_AT_FC_INT32	+ve or –ve integer	Calls a function taking an Int32
ICM_AT_FC_UNS64	+ve 64 bit integer	Calls a function taking an Uns64
ICM_AT_FC_DOUBLE	floating point number	Calls a function taking a double
ICM_AT_FC_STRING	string	Calls a function taking a string
ICM_AT_FC_ARGV	one or more strings +	Calls a function taking an array
ICM_AT_FC_ARGV_PAIR	one or more pairs of the form name=value	Calls a function taking two arrays

* The ICM_AT_STRINGLIST type can be specified on the command line more than once. Each occurrence adds to a linked list of strings.
+ The ICM_AT_FC_ARGV type can be followed by one or more space-separated values (which cannot begin with ‘-‘).

If a callback action is specified the callback should be declared using the CLPA_xxxx prototype macros defined in icmContruct.h.

If `mandatory` is true, the parser will report an error if the new argument is not specified.

32 Differences between CpuManager and OVPsim

These are the differences between CpuManager and OVPsim:

32.1 Heterogeneous Platforms

CpuManager allows the simulation of heterogeneous platforms i.e. platforms that contain processors of different types.
The user will therefore notice that heterogeneous platforms will run faster in CpuManager than in OVPsim.

32.2 Control Files

CpuManager and OVPsim can read a simulation control file which allows modifications to the simulation without recompiling the platform. Some features available in CpuManager are not available in OVPsim. Put –help in the control file to list the available features.

32.3 Debug Interface

A remote debugger can be connected to just one processor in an OVPsim platform. CpuManager allows many simultaneous debugger connections. CpuManager can also be used with the Imperas Multi-Processor debugger.

32.4 Verification and Analysis

OVPsim does not support the loading of extension libraries that provide tools. OVPsim can load one ‘semihost’ library onto each processor that is used to intercept system calls to provide semihosting.
CpuManager can load many intercept libraries onto each processor, providing the opportunity for comprehensive analysis and verification of application code without recompiling.

32.5 ICM API Support

OVPsim implements a subset of the Imperas CpuManager (ICM) API.

Some functions in OVPsim are only implemented as stubs which will cause the simulator to exit with a warning message. OVPsim can simulate exactly the same platforms as CpuManager. However if you require comprehensive debug and analysis, you will need CpuManager.

32.6 Simulator Feature summary

Feature	**OVPsim**	**CpuManager**
Speed	Fast	Fast
Multiple processors	Yes	Yes
Heterogeneous	No	Yes

Control files	Yes	Yes
Semihost intercept	Yes	Yes
Verification and Analysis	No	many libraries
GDB Debug Interface	Single GDB	Multiple GDBs
Tracing (disassembly)	Instructions only	Instruction and code labels
API:		
Platform Construction	Yes	Yes
Simulator Control	Yes	Yes
Remote debug	Single gdb	Multiply gdb or Imperas MPD or user multi-debug
Integrated Debugger	No	Imperas MPD
Command Parser	fewer built-in commands	Yes

32.7 OVPsim unsupported API functions summary

The following CpuManager functions are NOT implemented in OVPsim:

Finding and calling model commands
icmCallCommand
icmPrintCommands
icmGetAllPlatformCommands
icmGetAllProcessorCommands

Passing control to a debugger
icmSetSchedFn

Freezing and releasing individual processors
icmFreeze
icmUnfreeze
icmIsFrozen

Breakpoints and watchpoints
icmSetAddressBreakpoint
icmClearAddressBreakpoint
icmSetICountBreakpoint
icmClearICountBreakpoint
icmSetMemoryReadWatchPoint
icmSetMemoryWriteWatchPoint
icmSetMemoryAccessWatchPoint
icmSetBusReadWatchPoint
icmSetBusWriteWatchPoint
icmSetBusAccessWatchPoint
icmSetProcessorReadWatchPoint
icmSetProcessorWriteWatchPoint
icmSetProcessorAccessWatchPoint

Breakpoints and watchpoints
icmSetRegisterWatchPoint
icmSetModeWatchPoint
icmSetExceptionWatchPoint
icmGetWatchPointType
icmDeleteWatchPoint
icmGetNextTriggeredWatchPoint
icmResetWatchPoint
icmGetWatchPointLowAddress[5]
icmGetWatchPointHighAddress[4]
icmGetWatchPointRegister[4]
icmGetWatchPointCurrentValue[4]
icmGetWatchPointPreviousValue[4]
icmGetWatchPointUserData[4]
icmGetWatchPointTriggeredBy[4]

Callback after a delay
icmTriggerAfter
icmCancelTrigger

Add a user-defined object file reader
icmInstallObjectReader

Add disassembler
icmAddSymbol

Verification and Analysis
icmAddInterceptObject

Save and Restore State
icmProcessorSaveState
icmProcessorRestoreState
icmProcessorSaveStateFile
icmProcessorRestoreStateFile
icmMemorySaveState
icmMemoryRestoreState
icmMemorySaveStateFile
icmMemoryRestoreStateFile

Debug Control
icmSetDebugMode

##

[5] Although these functions can be called there is no underlying watch point available to interrogate

.