iGen Platform and Module Creation User Guide

This document describes the use of the
Imperas Model Generator ***iGen*** to
create a virtual platform using OVP APIs.

Imperas Software Limited

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

Author:	Imperas Software Limited
Version:	2.1.1
Filename:	iGen_Platform_and_Module_Creation_User_Guide.doc
Last Saved:	Tuesday, 02 June 2020
Keywords:	iGen Platform and Module Creation User Guide

Copyright Notice

Right to Copy Documentation

Destination Control Statement

Disclaimer

Table of Contents

1 Preface 6
1.1 Notation 6
1.2 Related Documentation 6
1.3 Glossary / Terminology 7
2 Introduction 9
2.1 Prerequisites 9
2.2 Obtaining & installing iGen 9
2.3 Compiling Examples described in this Document 9
2.4 Shared Objects and executables 10
2.5 Platforms, Modules and Shared Objects 10
2.6 Simulating Modules using harness.exe 10
2.7 Writing a bespoke C test harness 11
3 Describing a hardware design (a virtual platform) 12
3.1 Design hierarchy 12
4 Creating and simulating a first virtual platform 14
4.1 A quick run through 14
4.2 The simple application - Hello World 16
4.2.1 How printf works using semihosting 16
4.3 Defining the platform using iGen 16
4.3.1 Creating a new module 16
4.3.2 Adding a bus 16
4.3.3 Adding a processor instance and connecting it up 17
4.3.4 Adding a memory and connecting it up 17
4.3.5 Full iGen listing for simple single core module 17
4.3.6 Getting help on iGen module creating functions 17
4.4 Running iGen 18
4.5 iGen generated module files 19
4.6 Running the simulation using harness.exe 19
4.6.1 The simulation command line 20
5 Creating a module with a peripheral (a UART) 21
5.1 Module overview 21
5.2 Quick run through 22
5.3 The application - writing to the UART 23
5.4 Defining the platform using iGen 24
5.4.1 Adding nets to the module 24
5.4.2 Adding a peripheral instance (a UART) 24
5.4.3 Getting help on iGen module creating functions 25
5.4.4 Full iGen listing for simpleCpuMemoryUart module 25
5.5 Running iGen 26
5.6 Running the simulation using harness.exe 26
5.6.1 The simulation command line 27
6 Changing the processor being used (From OR1K to ARM) 29
7 Adding Bus Bridges, Aliasing address regions 31

7.1 Static Bus bridges 31
7.1.1 Aliasing 32
7.2 Dynamic Bus Bridges 34
7.2.1 Introduction 34
7.2.2 Running the simulation 36
8 Byte Swapping (Endian Correction) 39
8.1 Bus Connections 39
8.2 Performance considerations 40
9 Two processors with shared memory 42
9.1 Adding Hardware Elements 42
9.2 Making Connections 44
9.3 The example encrypt and decrypt applications 45
9.4 Running the Example 48
10 Caches (using Memory Model Components (MMC)) 50
10.1 Transparent or Full MMC Models 50
10.2 MMC Operation 51
10.2.1 Transparent Model 51
10.2.2 Full Model 52
10.3 Creating and connecting an MMC 52
10.3.1 Transparent MMC Example 52
10.3.2 Full MMC Example 54
10.3.3 Cascaded MMC Example 56
11 Using Module Hierarchy in Virtual Platforms 61
11.1 A two level platform: simpleHierarchy 61
11.2 Creating the sub module 62
11.2.1 Compiling the submodule 63
11.3 Creating the top module 63
11.3.1 Instancing a sub module 63
11.3.2 Compiling the topmodule 64
11.4 Application0 - writing to the UART 64
11.5 Application1 - reading from the UART 64
11.6 Running the hierarchical platform simulation 65
12 Hierarchy and Connectivity in Modules 67
12.1 The top level 67
12.2 The processor sub system 69
12.3 The memory sub system 70
12.4 The Test Application 70
12.5 Building the Example 70
12.6 Running the simulation 71
13 Passing Parameters down module hierarchy 73
13.1 Creating the sub module 74
13.2 Creating the top module 75
13.3 The application 75
13.4 Running the simulation 76
14 Directory structure: VLNV or direct paths 78
14.1 A hierarchical design using a VLNV directory structure 78

14.2 The directory structure 79
14.3 Changing the controlling scripts 79
14.4 The module instances 80
15 Loading programs into the design 81
16 Loading symbols into the simulator 82
17 Setting Model Parameters 83
18 Advanced Information & Usage of iGen 84
18.1 Overview of detailed platform construction 84
18.1.1 Harnesses and Modules 84
18.1.2 The contents of a module 84
18.1.3 The Contents of a Harness or Test Bench 85
18.1.4 Module Parameterization 86
18.1.5 Efficiency 87
18.2 Order of Platform construction 87
18.3 Editing the C of a module 87
18.4 Writing out a testbench / harness 88
19 iGen Module related Error Messages 89
20 (Deprecated) Creating ICM platforms with iGen 90
20.1 Generated files 90
20.1.1 User file : platform.c 90
20.1.2 Constructor file: platform.constructor.igen.h 91
20.1.3 Options file: platform.options.igen.h 91
20.1.4 Handles file: platform.handles.igen.h 91
20.1.5 Command line parser file: platform.clp.igen.h 91
20.2 Repeated use of iGen 91
20.3 Adding a copyright header 91
20.4 Checking the Platform 91

1 Preface

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

This document describes the use of the iGen Model Generator, which executes scripts making calls to the iGen Command API. The scripts use TCL (Tool Control Language) as input and iGen creates C templates for simulation models and plugins, and creates interfaces for SystemC TLM2 simulation and creates virtual platforms, testbenches and modules using the OP C API and SystemC TLM2.

This document specifically describes how iGen is used to create platforms and modules for use with Imperas and OVP virtual platform simulators and tools.

1.1 Notation

`code`	Text representing code, a command or output from *iGen* or other program.
keyword	A word with special meaning.

1.2 Related Documentation

There are several documents available as PDF:

Getting Started
- Imperas Installation and Getting Started Guide

Interface and API
- OVP Peripheral Modeling Guide
- OVPsim Using OVP Models in SystemC TLM2.0 Platforms

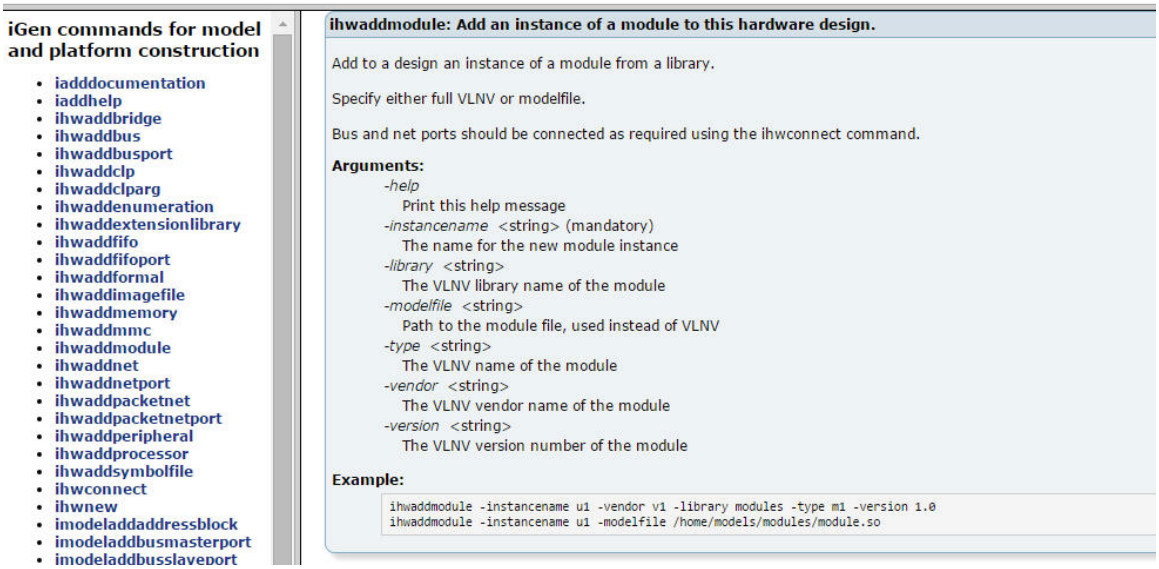
References to specific uses of iGen
- iGen Model Generator Introduction
- iGen Platform and Module Creation User Guide
- Imperas Peripheral Generator Guide

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

Also, in your installation there is the online iGen Function API Command reference documentation. This is Doxygen-like API documentation available at:

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

For example:



1.3 Glossary / Terminology

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

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

OVPsim - Simulator for Open Virtual Platforms that executes platforms and models coded in the OVP APIs.

CpuManager - Imperas commercial simulator that executes platforms and models coded in the OVP APIs.

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

Testbench / Harness - program in C/C++ making calls into OP API to connect and control OVP components. Normally linked to the simulator to provide an .exe binary that can be executed. Used to instantiate one or more platforms/modules and controls their execution. The main difference, from a platform/module, is that a testbench or harness

includes a call to the function main(), may include a command line parser and is linked to create an executable binary (.exe) file.

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

2 Introduction

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

iGen is an Imperas productivity tool that has a powerful script based function API that is used to create C/C++/SystemC platform models and component templates.

2.1 Prerequisites

Since models and platforms for use with Imperas and OVP tools are written in C, an important prerequisite is that you must be proficient in the C language.

iGen uses the TCL scripting language, so it is beneficial to have some basic understanding of TCL.

2.2 Obtaining & installing iGen

iGen is available as part of the Imperas DEV and SDK packages. So it is assumed you have downloaded one of these from the Imperas website and have installed it on the host machine.

2.3 Compiling Examples described in this Document

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

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

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

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

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

2.4 Shared Objects and executables

The shared objects referred to in this document are either Linux shared objects, with suffix *.so* or Windows dynamic link libraries with suffix *.dll*.

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

2.5 Platforms, Modules and Shared Objects

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

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

2.6 Simulating Modules using harness.exe

For most simple iGen constructed platforms and modules the `harness.exe` program can be used.

To see the commands that can be used with the program, type:

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

The simplest execution is to load a module with an application loaded onto the processor(s) in the module:

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

The argument --*modulefile* <*filename*> states which shared object module[1] to simulate.

The argument --*program* <*filename*> states which program binary (elf) file to load to run on the processor(s)[2] contained within the module.

Look at some of the examples in Examples/PlatformConstruction.

1 The module could also be loaded from a VLNV library using the --modulevendor, --modulelibrary, --modulename and --moduleversion arguments in place of the --*modulefile* argument.

2 When no processor is specified the program elf file is loaded onto ALL the processors in the module. To specify programs to load onto a specific processor use --program modulename/processorname=app.elf

2.7 Writing a bespoke C test harness

If you require a more complex test harness, where you wish to control the simulator (for instance single step processor execution), or add monitors or other test related capabilities, then please refer to the Simulation Control of Platforms and Modules User Guide.

3 Describing a hardware design (a virtual platform)

There are three phases to describing a platform:

1. Creation of the design, which involves specifying the hardware components it contains along with the connections between those components.
2. Validation of the design, which involves ensuring that all the library elements which have been referenced are available in the library and that the connections are valid.
3. Use of that design with software programs designed to run on it in an appropriate test environment which allows the verification of both hardware and associated software. Each of these processes will be described in the sections that follow.

The first phase of the process of building a platform is to determine which hardware is to be included. A typical platform consists of a combination of modules, processors, buses, memory and other components.

The models for these are stored within the libraries. A library model does not need to exist within a library to be included in a platform, so a platform developer can work in parallel with a library model developer, however, the library elements must exist before verification of the design can take place. The exceptions to this are basic memories (RAM and ROM), FIFOs, packetnets, and buses for which models are provided with the tools.

The second phase is validating that the platform elaborates and connects up correctly - this is done simply by starting to simulate the platform but not executing instructions. This can be done by using the harness program, *harness.exe*, or it can be accomplished by using your own created harness / test bench written in C using the OP simulator control API.

The third phase is to use a test bench / harness to simulate the platform with appropriate application software and Operating Systems loaded.

3.1 Design hierarchy

With the use of the OP platform API, a platform can be constructed from a hierarchy of modules. A module is separately compiled and is a collection of component instances and connections. A module is like a sub-system. It can be used as a component in other modules or simulated directly by a harness. Connections can be made to modules using ports, and parameters can be passed in to configure some aspects of the module or its contained components, busses etc.

The term module instance is used to refer to the instantiation of a module in another module.

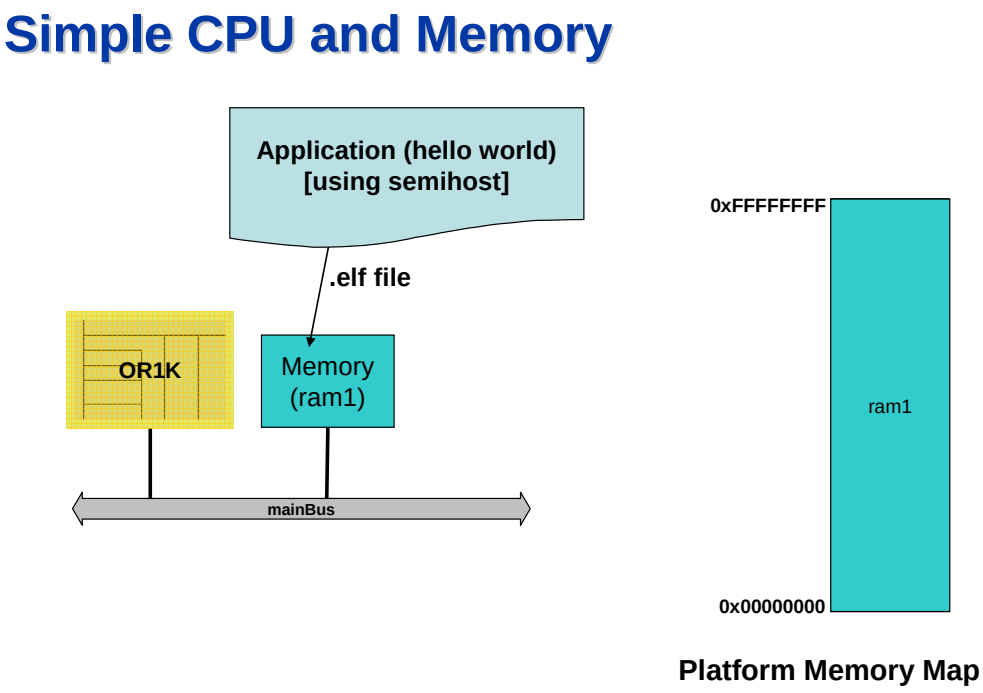
When connecting a bus, net, FIFO or packetnet to a module external port it is exported to the containing module that instances it. When not connected to a module external port it remains private within the created module.

Modules can instance modules.

4 Creating and simulating a first virtual platform

Let us first walk through a simple example of using iGen to create a module of a simple CPU and memory, then use the provided harness.exe program to execute it with a cross compiled simple application program running on the CPU.

This platform has an OR1K CPU, RAM memory, bus and no hierarchy, as shown here:



Platform Memory Map

It is available in the directory:

```
$IMPERAS_HOME/Examples/PlatformConstruction/simpleCpuMemory
```

Take a copy of the example directory tree:

```
> cp –r $IMPERAS_HOME/Examples/PlatformConstruction/simpleCpuMemory .
```

4.1 A quick run through

First, we need to cross compile the application. If you do not have your own cross compiler tool chains, you can download pre-built tools for processors provided by OVP from the OVP website or Imperas user areas. For more information read the 'Installing Additional Tools' section of the Imperas Installation and Getting Started Guide.

Compile the application using the provided Makefile and specify the cross compile target to use:

```
> cd simpleCpuMemory/application
> make CROSS=OR1K
```

This uses Make to cross compile and link the application:

```
# Compiling application.c
# Linking application.OR1K.elf
```

Then make the module:

```
> cd ../module
> make
```

This uses Make to a) run iGen to take the input TCL file and create the output .c/.h files, and then b) host compile and link the platform.

If you do not have iGen installed, you will see the error message:

```
> make
# iGen Create OP MODULE module
make: igen.exe: Command not found
```

If you get that message, you need to download and install iGen (see section 2.2 above).

If iGen is installed, then you will see:

```
> make
# iGen Create OP MODULE module
# Copying STUBS module.c.igen.stubs to module.c
# Host Depending obj/Linux32/module.d
# Host Compiling Module obj/Linux32/module.igen.o
# Host Linking Module object model.so
```

Which will have created all the .c and .h files, and host compiled and linked them to shared object: model.so/.dll.

Now execute the simulation of the module with the compiled application using the provided harness.exe program:

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

The simulator will run; see that the 'Hello first world' is output...

```
...
OVPsim started: Tue Oct 13 10:57:22 2015

Hello first world from iGen generated platform

OVPsim finished: Tue Oct 13 10:57:22 2015
...
```

To make things simpler, we have provided a script that performs these make and run commands:

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

4.2 The simple application - Hello World

Look at the *application/application.c* file.

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

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

    printf("Hello first world from iGen generated platform\n");

    return 0;
}
```

4.2.1 How printf works using semihosting

For details on Imperas semihosting see the Imperas Installation and Getting Started User Guide.

In this example semihosting is used to provide behavior for the low level functions without having to add anything to the platform or application code. The *write* function invoked by the *printf* function called in the application is intercepted and the functionality provided by the simulator with, in this case, the printed characters that are written to stdout appearing in the simulator's output.

4.3 Defining the platform using iGen

The input to iGen is a tcl program/script that makes calls to the iGen platform/module building functions. iGen is run in a script or makefile to write out the .c/.h files that will be compiled to create and run the platform/module.

4.3.1 Creating a new module

Look at the *module/module.op.tcl* file.

```
ihwnew -name simpleCpuMemory
```

The command *ihwnew* starts the definition of a hardware component (module) and names it *simpleCpuMemory*.

4.3.2 Adding a bus

We then add a bus with name *mainBus* with the width of the address being 32 bits.

```
ihwaddbus -instancename mainBus -addresswidth 32
```

Note that you do not specify the data width of the bus, only the byte addressable range

4.3.3 Adding a processor instance and connecting it up

A processor is added to the module with one function call:

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

This specifies an instance of the *generic* variant from the VLNV library of your Imperas installation *ovpworld.org/processor/or1k/1.0* using the semihost library of *or1kNewlib* and with an instance name of *cpu1*.

We connect it to the bus using:

```
ihwconnect -bus mainBus -instancename cpu1 -busmasterport INSTRUCTION
ihwconnect -bus mainBus -instancename cpu1 -busmasterport DATA
```

This connects the *mainBus* to both master ports on the processor instance.

4.3.4 Adding a memory and connecting it up

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

This instances a memory and connects up its port and provides the address range that it is usable at. Size is not specified as it is defined by the hi/lo byte addresses, i.e. its valid address range.

4.3.5 Full iGen listing for simple single core module

```
ihwnew -name simpleCpuMemory

ihwaddbus -instancename mainBus -addresswidth 32

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

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

So in 7 lines of input for iGen we have created a simple module including a processor instance and a memory instance and connected them up.

4.3.6 Getting help on iGen module creating functions

To see a list of module building functions, visit the online documentation at:

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

Or you could use the iGen *--apropos* command line argument to give information on a command:

```
> igen.exe --apropos ihwaddproc
    ...
NAME: ihwaddprocessor - Add to a design, an instance of a processor
    ...
ARGUMENTS:
    -instancename <string> (mandatory)
        The name for the new processor instance
    -type <string>
        The VLNV name of the processor
    -variant <string>
        The processor variant
    -vendor <string>
        The VLNV vendor name of the processor
    -library <string>
        The VLNV library name of the processor
    -version <string>
        The VLNV version number of the processor
    -semihostname <string>
        The VLNV name of a Semihost library
   ...
```

Note that for *ihwaddprocessor* there are many more arguments to set things like mips rate, start addresses, program image to load etc.

4.4 Running iGen

igen.exe is a program in the Imperas installation in the *IMPERAS_HOME/bin/IMPERAS_ARCH* directory and can be run from the command shell as illustrated above when using it to get help etc.

You can run *igen.exe* with the *--help* option to explore its command line options, but using igen.exe in a makefile or script for module generation is recommended and for our examples we will use the provided Makefile system.

The simplest way to run iGen is to use the Imperas provided Makefile system. In the examples we provide two files in the module directory, the *Makefile* and the *module.op.tcl* source file.

```
> cd module
> ls
Makefile   module.op.tcl
```

The Imperas Makefile recognizes the file *module.op.tcl* as a script file to input to iGen to generate the module in C using the OP API.

```
> make
# iGen Create OP MODULE module
# Copying STUBS module.c.igen.stubs to module.c
# Host Depending obj/Linux32/module.d
# Host Compiling Module obj/Linux32/module.o
# Host Linking Module object model.so
```

4.5 iGen generated module files

After iGen has been run you will find three files in the module directory; module.c, module.c.igen.stubs and module.igen.h.

The *module.c.igen.stubs* file is always generated and is overwritten each time iGen is run. The Make system will only invoke iGen if the input (module.op.tcl) file is newer than the generated files.

If *module.c* does not exist the stubs file, *module.c.igen.stubs*, is copied to create *module.c*.

If you look at the module.c and the module.igen.h files, you will see the OP API calls.

The file module.c can be edited and provides some empty callbacks in which you can add OP function calls to the module if you need to add more capability. Subsequent runs of iGen will write out a module.c.igen.stubs file that can be used for comparison and code merging purposes.

The *module.igen.h* file is always written and should not be modified by the user.

For details of the contents of the generated C files and the editing of them, see the Simulation_Control_of_Platforms_and_Modules_User_Guide and Advanced_Simulation_Control_of_Platforms_and_Modules_User_Guide.

4.6 Running the simulation using harness.exe

To run the simulation a harness or test bench is required. This can be written in C using OP API calls (see the Simulation Control of Platforms and Modules User Guide) or you can use the *harness.exe* program.

To use the *harness.exe* program to execute the simulation of the platform with the compiled application:

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

The simulator will run; see that the 'Hello first world' is output...

```
OVPsim started: Tue Oct 13 10:57:22 2015

Hello first world from iGen generated platform

OVPsim finished: Tue Oct 13 10:57:22 2015
```

To make things simpler, we have provided a script that performs these make and run commands:

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

You should be able to just edit the *application/application.c* and *module/module.op.tcl* files and then just run *./example.sh* to rebuild, recompile, and run.

4.6.1 The simulation command line

The Imperas simulators that are invoked by *harness.exe* and *iss.exe* have many common built-in command line arguments. Examples argument are *--modulefile* and *--program* as shown and used above, other useful arguments are *--trace*, *--verbose* as shown in the example below:

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

Use *--help* to see what each command line argument does.

Note that command line arguments like these can be added to the *example.sh* script invocation and they are passed to the simulator command line:

```
> ./example.sh --trace --verbose
> ./example.sh --help
```

5 Creating a module with a peripheral (a UART)

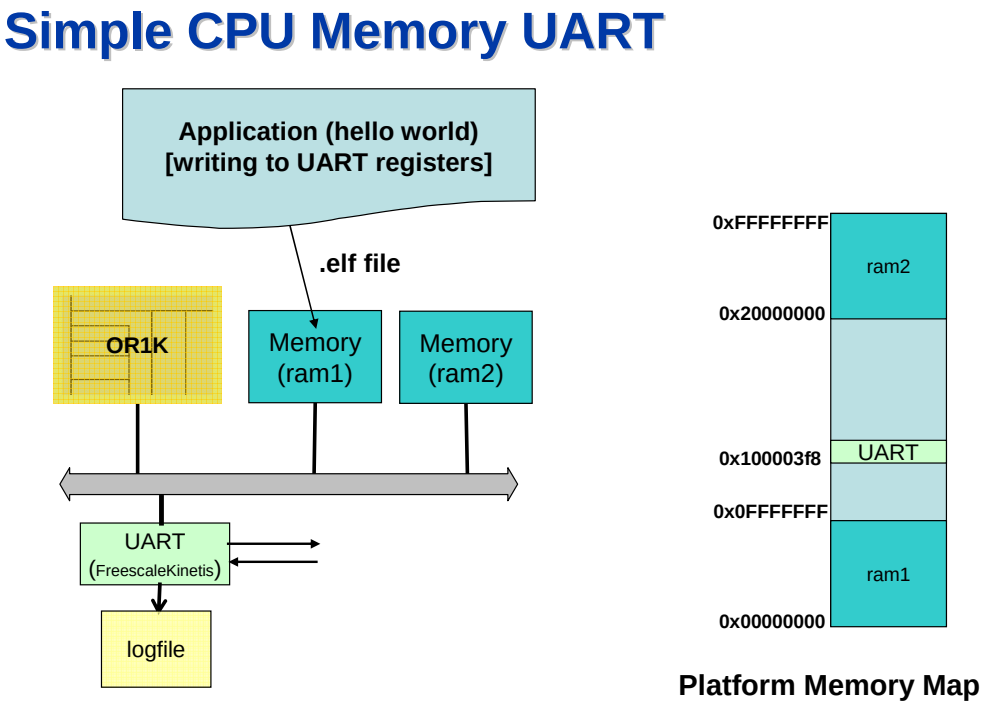
This example introduces peripherals. It builds on the module created in the previous example.

We expect you to look at the code in the example directory and in this text we will discuss those parts that are different or new when compared to the previous examples in this document.

This is an introductory example; please see later sections in this document for more detailed examples.

5.1 Module overview

If you look at the picture below, you will see there is a CPU, two memories, and an instance of a UART in the platform.



This platform is available at

```
$IMPERAS_HOME/Examples/PlatformConstruction/simpleCpuMemoryUart
```

Take a copy of the example directory tree:

```
> cp -r $IMPERAS_HOME/Examples/PlatformConstruction/simpleCpuMemoryUart .
```

5.2 Quick run through

First we need to cross compile the application. If you do not have your own cross compiler tool chains, you can download pre-built tools for processors provided by OVP from the OVP website or Imperas user areas. For more information read the 'Installing Additional Tools' section of the Imperas Installation and Getting Started Guide.

Compile the application using the provided Makefile and specify the cross compile target to use:

```
> cd simpleCpuMemoryUart/application
> make CROSS=OR1K
```

This uses Make to cross compile and link the application:

```
# Compiling application.c
# Linking application.OR1K.elf
```

Then make the module:

```
> cd ../module
> make
```

This uses Make to a) run iGen to take the input tcl file and create the output .c/.h files, and then b) host compile and link the platform.

If you do not have iGen installed, you will see the error message:

```
> make
# iGen Create OP MODULE module
make: igen.exe: Command not found
```

If you get that message, you need to download and install iGen (see section 2.2).

If iGen is installed, then you will see:

```
> make
# iGen Create OP MODULE module
# Copying STUBS module.c.igen.stubs to module.c
# Host Depending obj/Linux32/module.d
# Host Compiling Module obj/Linux32/module.igen.o
# Host Linking Module object model.so
```

Which will have created all the needed .c and .h files, and host compiled and linked them to shared object `model.so/.dll`.

Now execute the simulation of the module with the compiled application using `harness.exe`:

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

The simulator will run; you will see the initialization of the UART and the writing to UART messages:

```
OVPsim started: Mon Nov 30 16:59:41 2015

Initializing KinetisUART
Writing to uart - see log file

OVPsim finished: Mon Nov 30 16:59:41 2015
```

The output of the UART is in the file uartTTY0.log:

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

To make things simpler, we have provided a script that performs these make and run commands:

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

And don't forget you can use the verbose and other command line arguments, such as modeldiags:

```
> ./example.sh --modeldiags 0x3 --verbose
```

5.3 The application - writing to the UART

Look at the *application/application.c* file.

Here is the declaration of a function to write a character to the UART:

```
static void writeMessFreescaleKinetisUart (unsigned char *uartBase, unsigned
const char *myString) {
       volatile unsigned char *ab_S1 = uartBase + 0x4;
       volatile unsigned char *ab_D  = uartBase + 0x7;
       #define UART_S1_TDRE_MASK 0x80

       unsigned int i;
       for(i=0;i<strlen(myString);i++){
           while ((*ab_S1 & UART_S1_TDRE_MASK) == 0) {
               // Wait for TransmitRegister Empty flag
           }
              *ab_D = myString[i];
       }
}
```

The address of the UART is passed in as `uartBase` and this is used in `ab_S1` and `ab_D` declared as `volatile unsigned char` so accesses to them will not be optimized away.

And then a *main* to call it:

```
#define UART0_BASE ((unsigned char *) 0x100003f8)

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

       initFreeScaleKinetisUart(UART0_BASE);
```

```
        printf ("Writing to uart - see log file\n\n");

        writeMessFreescaleKinetisUart(UART0_BASE,
               "Hello UART0 world\n\n");

    return 0;
}
```

Note the declaration of `UART0_BASE` to define the address where the UART is located in the platform.

Note also the call to `initFreeScaleKinetisUart` which is declared in *application.c* which just executes the initialization that you need to perform on the UART.

The application is compiled with:

```
> cd application
> make
# Compiling application.c
# Linking application.OR1K.elf
```

5.4 Defining the platform using iGen

The input to iGen is a tcl program/script that makes calls to the iGen platform/module building functions. iGen is run in a script or makefile to write out the .c/.h files that will be compiled to create and run the platform/module.

Look at the *module/module.op.tcl* file. Only new iGen commands will be introduced here. Please refer to the previous iGen example (in section 4.3.5) for description of the usage of iGen and information on starting a new module, adding a bus, processor, memories and connecting them up.

5.4.1 Adding nets to the module

To add nets to a module, use *ihwaddnet*:

```
ihwaddnet -instancename directWrite
ihwaddnet -instancename directRead
```

5.4.2 Adding a peripheral instance (a UART)

Adding a peripheral instance to a module is very similar to instancing a processor:

```
ihwaddperipheral -instancename periph0 \
       -vendor freescale.ovpworld.org -library peripheral -version 1.0 \
       -type KinetisUART
```

Note that when instancing a peripheral you must specify the vendor, library, and version.

OVP peripherals are documented on the OVP website. For this peripheral look at: OVPworld->Library->Peripherals->FreescalePeripherals->FreescaleKinetisUART.

5.4.2.1 Setting a peripheral instance's parameters

Often a component you instance will have been declared with formal parameters or parameters that can be set when the component is instanced. (Check the documentation).

The FreescaleKinetisUART has many parameters that can be set including outfile. Parameters on an instance are set with `ihwsetparameter`:

```
ihwsetparameter -handle periph0 -name outfile -value uartTTY0.log –type string
```

This names the log file.

5.4.2.2 Connecting up a peripheral instance to a bus and to nets

Check the documentation of the peripheral to see what you can connect to which ports.

The FreescaleKinetisUART has connections for a bus and nets:

```
ihwconnect -instancename periph0 -busslaveport bport1 -bus mainBus \
           -loaddress 0x100003f8 -hiaddress 0x100013f7
ihwconnect -instancename periph0 -netport DirectWrite -net directWrite
ihwconnect -instancename periph0 -netport DirectRead  -net directRead
```

This connects the peripheral's slave port to the bus and connects up the two netports to the nets.

5.4.3 Getting help on iGen module creating functions

To see a list of module building functions, visit the online documentation at:

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

Or you could use the `iGen` *--apropos* command line argument to give information on a command:

```
> igen.exe --apropos iseta
```

5.4.4 Full iGen listing for simpleCpuMemoryUart module

```
ihwnew -name simpleCpuMemoryUart

ihwaddbus -instancename mainBus -addresswidth 32

ihwaddnet -instancename directWrite
ihwaddnet -instancename directRead

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

ihwaddmemory -instancename ram1 -type ram
ihwconnect -bus mainBus -instancename ram1 \
       -busslaveport sp1 -loaddress 0x0 -hiaddress 0x0fffffff
```

```
ihwaddmemory -instancename ram2 -type ram
ihwconnect -bus mainBus -instancename ram2 \
       -busslaveport sp1 -loaddress 0x20000000 -hiaddress 0xffffffff

ihwaddperipheral -instancename periph0 \
       -vendor freescale.ovpworld.org -library peripheral \
       -version 1.0 -type KinetisUART
ihwsetparameter -handle periph0 -name outfile -value uartTTY0.log –type string
ihwconnect -instancename periph0 \
       -busslaveport bport1 -bus mainBus \
       -loaddress 0x100003f8 -hiaddress 0x100013f7
ihwconnect -instancename periph0 -netport DirectWrite -net directWrite
ihwconnect -instancename periph0 -netport DirectRead  -net directRead
```

5.5 Running iGen

igen.exe is a program in the Imperas installation in the `$IMPERAS_HOME/bin/$IMPERAS_ARCH` directory and can be run from the command shell as illustrated above when using it to get help etc.

You can run *igen.exe* with the *--help* option to explore its command line options, but using igen.exe in a makefile or script for module generation is recommended and for our examples we will use the provided Makefile system.

The simplest way to run iGen is to use the Imperas provided Makefile system. In the examples we provide two files in the module directory, the *Makefile* and the *module.op.tcl* source file.

```
> cd module
> ls
Makefile   module.op.tcl
```

The Imperas make system recognizes the file *module.op.tcl* as a script file to input to iGen to generate the module in C using the OP API.

```
> make
# iGen Create OP MODULE module
# Copying STUBS module.c.igen.stubs to module.c
# Host Depending obj/Linux32/module.d
# Host Compiling Module obj/Linux32/module.o
# Host Linking Module object model.so
```

For additional information regarding the files generated by iGen for a module please refer to section 4.5 ”iGen generated module files”.

5.6 Running the simulation using harness.exe

To run the simulation a harness or test bench is required. This can be written in C using OP API calls (see the Simulation Control of Platforms and Modules User Guide), or you can use the provided *harness.exe* program.

To use the provided *harness.exe* program to execute the simulation of the platform with the compiled application:

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

The simulator will run, producing the following output:

```
OVPsim started: Tue Oct 13 10:57:22 2015

Initializing KinetisUART
Writing to uart - see log file

OVPsim finished: Tue Oct 13 10:57:22 2015
```

To make things simpler, we have provided a script that performs these make and run commands:

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

You should be able to edit the *application/application.c* and *module/module.op.tcl* files and then run *./example.sh* to rebuild, recompile, and run.

5.6.1 The simulation command line

The Imperas simulators, *harness.exe* and *iss.exe* have many common built-in command line arguments. Examples for *harness.exe* are *--modulefile* and *--program* as above, other useful arguments are:

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

Use *--help* to see what each command line argument does.

This lists many arguments (some of which will only be useful to specific simulators and models). The arguments are loosely grouped into categories - for example: control, debug, diagnostics, log, parameters, program and trace.

From the log group:

```
--verbose
```

which shows the simulation run statistics.

In the diagnostics group is:

```
--modeldiags 0x3
```

which shows the accesses to the peripheral models registers:

```
OVPsim started: Mon Nov 30 15:59:36 2015

Info (UART_UIS) top/periph0: Uart initialized in serial channel mode
```

```
Initializing KinetisUART
Info (UART_BRC) top/periph0: Baud rate changed to 19921
Info (UART_BRC) top/periph0: Baud rate changed to 19577
Info (UART_TFT) top/periph0: Transmitter fifo threshold set to 1
Info (UART_RFT) top/periph0: Receiver fifo threshold set to 1
Info (UART_UW) top/periph0: Write to Data register: data=0x0d ('')
Writing to uart - see log file

Info (UART_UW) top/periph0: Write to Data register: data=0x48 ('H')
Info (UART_UW) top/periph0: Write to Data register: data=0x65 ('e')
Info (UART_UW) top/periph0: Write to Data register: data=0x6c ('l')
Info (UART_UW) top/periph0: Write to Data register: data=0x6c ('l')
Info (UART_UW) top/periph0: Write to Data register: data=0x6f ('o')
Info (UART_UW) top/periph0: Write to Data register: data=0x20 (' ')
Info (UART_UW) top/periph0: Write to Data register: data=0x55 ('U')
Info (UART_UW) top/periph0: Write to Data register: data=0x41 ('A')
Info (UART_UW) top/periph0: Write to Data register: data=0x52 ('R')
Info (UART_UW) top/periph0: Write to Data register: data=0x54 ('T')
Info (UART_UW) top/periph0: Write to Data register: data=0x30 ('0')
Info (UART_UW) top/periph0: Write to Data register: data=0x20 (' ')
Info (UART_UW) top/periph0: Write to Data register: data=0x77 ('w')
Info (UART_UW) top/periph0: Write to Data register: data=0x6f ('o')
Info (UART_UW) top/periph0: Write to Data register: data=0x72 ('r')
Info (UART_UW) top/periph0: Write to Data register: data=0x6c ('l')
Info (UART_UW) top/periph0: Write to Data register: data=0x64 ('d')
Info (UART_UW) top/periph0: Write to Data register: data=0x0a ('')
Info (UART_UW) top/periph0: Write to Data register: data=0x0a ('')

OVPsim finished: Mon Nov 30 15:59:36 2015
```

Arguments:

```
--showbuses
--showdomains
```

show the connections to the buses and the memories.

From the trace group, examples are:

```
--trace
--trace --tracechange
```

These trace instructions in the processor with the option of tracing registers that have changed.

Note that command line arguments like the above can be added to the *example.sh* script invocation and they are passed to the simulator command line:

```
> ./example.sh --showbuses --showdomains
> ./example.sh --trace --tracechange
> ./example.sh --help
```

6 Changing the processor being used (From OR1K to ARM)

The example above as provided uses the OpenCores OR1K processor. It is easy to change this example to use another processor.

First check that the example runs unmodified.

```
> cp -r $IMPERAS_HOME/Examples/PlatformConstruction/simpleCpuMemoryUart .
> cd simpleCpuMemoryUart
> ./example.sh
> cd ..
```

Then copy the directories, and as we will be using an ARM processor name it accordingly:

```
> cp -r simpleCpuMemoryUart simpleCpuMemoryUart_ARM
> cd simpleCpuMemoryUart_ARM
```

In fact it takes 2 edits:

example.sh change the cross compilation target from:

```
CROSS=OR1K
```

to:

```
CROSS=ARM_CORTEX_A
```

module/module.op.tcl has changes to the selection of the processor model from:

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

to:

```
ihwaddprocessor -instancename cpu1 \
    -vendor arm.ovpworld.org -library processor -type arm -version 1.0 \
    -semihostname armNewlib -semihostvendor arm.ovpworld.org \
    -variant Cortex-A9UP
```

So thus we have changed the platform to use the *arm* model from *arm.ovpworld.org* as the variant *Cortex-A9UP* using semihost library *armNewlib* from vendor *arm.ovpworld.org* and changed the cross compiler make system to select the *ARM_CORTEX_A* tool chain.

Also note you will need to install the ARM toolchain (armv7.toolchain.<version>.<arch>.exe) (see installing tools chains above).

So running *./example.sh* with our edits, will compile everything and run and we will see the UART log displayed at the end:

```
./example.sh --verbose
# Compiling application.c
# Linking application.ARM_CORTEX_A.elf
# Host Compiling Platform obj/Linux32/platform.o
# Host Linking Platform platform.Linux32.exe
# Host Linking Platform object model.so

OVPsim started: Mon Nov 30 16:11:47 2015


Info (OP_AL) Found attribute symbol 'modelAttrs' in file
       ImperasLib/arm.ovpworld.org/processor/arm/1.0/model.dll'
Info (OP_AL) Found attribute symbol 'modelAttrs' in file
       ImperasLib/arm.ovpworld.org/semihosting/armNewlib/1.0/model.dll'
Info (OR_OF) Target 'simpleCpuMemoryUart/cpu1' has object file read from
       'application/application.ARM_CORTEX_A.elf'
Info (OR_PH) Program Headers:
Info (OR_PH) Type       Offset     VirtAddr   PhysAddr   FileSiz    MemSiz
Flags Align
Info (OR_PD) PROC       0x0000a34c 0x0000a34c 0x0000a34c 0x00000008 0x00000008
R--   4
Info (OR_PD) LOAD       0x00008000 0x00008000 0x00008000 0x00002358 0x00002358
R-E   8000
Info (OR_PD) LOAD       0x0000a358 0x00012358 0x00012358 0x00000854 0x00100ca8
RW-   8000

Initializing KinetisUART
Writing to uart - see log file

Info
Info ------------------------------------------------------
Info PSE SIMULATION TIME STATISTICS
Info    0.00 seconds: PSE THREAD 'simpleCpuMemoryUart/periph0'
Info    0.01 seconds: PSE 'simpleCpuMemoryUart/periph0' (and 90 terminated
callbacks)
Info ------------------------------------------------------
Info
Info ------------------------------------------------------
Info CPU 'simpleCpuMemoryUart/cpu1' STATISTICS
Info   Type                  : arm (Cortex-A9UP)
Info   Nominal MIPS          : 100
Info   Final program counter : 0x8030
Info   Simulated instructions: 2,492
Info   Simulated MIPS        : run too short for meaningful result
Info ------------------------------------------------------
Info
Info ------------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 0.00 seconds
Info   User time             : 0.04 seconds
Info   System time           : 0.00 seconds
Info   Elapsed time          : 0.04 seconds
Info ------------------------------------------------------

OVPsim finished: Mon Nov 30 16:11:47 2015

Hello UART0 world
```

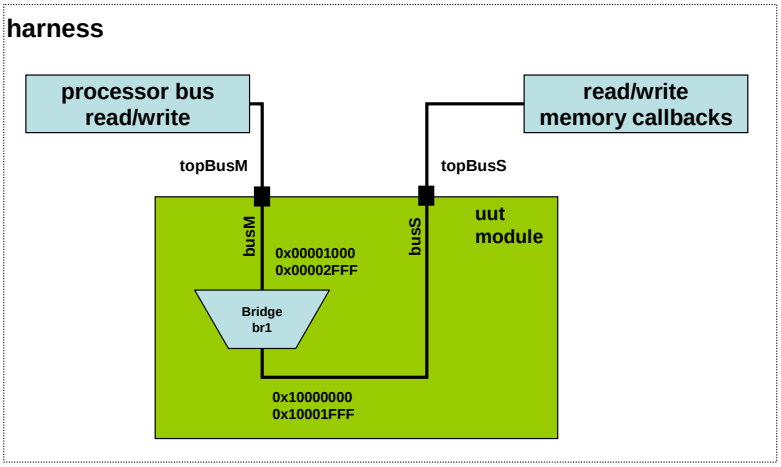
7 Adding Bus Bridges, Aliasing address regions

A 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.

7.1 Static Bus bridges

Construction of the processor, memory and peripheral components has already been covered.

A static bridge may be used to map sections of one address space to another. It is also used to create aliases of memory.



In this example a processor read/write in the address range 0x00001000 to 0x00002fff will be mapped to a memory access in the range 0x10000000 to 0x10001fff.

This example is available at

```
$IMPERAS_HOME/Examples/PlatformConstruction/busHierarchy
```

Take a copy of the example directory tree:

```
> cp -r $IMPERAS_HOME/Examples/PlatformConstruction/busHierarchy .
> cd busHierarchy
```

There are three parts to a bridge.

The instance definition:

```
ihwaddbridge -instancename br1
```

The slave port connection onto the bus containing the bus master:

```
ihwconnect -bus busM -busslaveport ps -instancename br1 \
           -loaddress 0x1000 -hiaddress 0x00002fff
```

The master port connection onto the bus containing the bus slaves:

```
ihwconnect -bus busS -busmasterport pm -instancename br1 \
           -loaddress 0x10000000 -hiaddress 0x10001fff
```

Note that in this example *busM* refers to the bus on which the bus master accesses are carried out and onto which the bridge slave port is connected and *busS* is the bus which is connected to the slaves and onto which the bridge master port is connected. The port names are for documentation only but should be unique on their respective busses.

The definition above is executed by iGen to generate a C definition in which the function *opBridgeNew* creates the bus bridge:

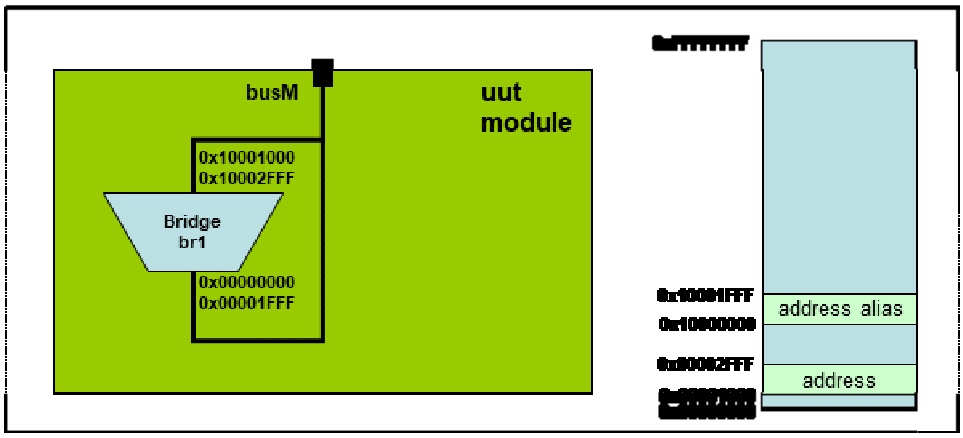
```
opBridgeNew(
  mi,                 // module
  "bridge1",          // name of this bridge
  OP_CONNECTIONS(
    OP_BUS_CONNECTIONS(
      OP_BUS_CONNECT(busM, "ps", .addrlo=0x1000, .addrHi=0x2fff, .slave=1),
      OP_BUS_CONNECT(busS, "pm", .addrlo=0x10000000, .addrHi=0x10001fff)
    )
  )
)
```

Note that the port is identified as a slave port by the setting of *slave=1*.

7.1.1 Aliasing

A bus bridge can be used to alias a region of an address space to another region on the same bus. There are two ways this could be achieved:

7.1.1.1 Using a single bus bridge to map a region back onto the same bus

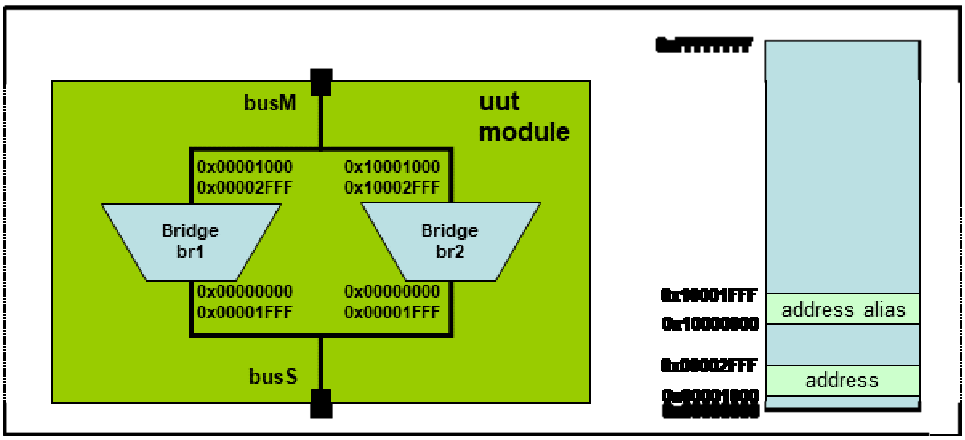



```
ihwaddbridge -instancename br1
ihwconnect -bus busM  -busslaveport  ps -instancename br1 \
```

```
           -loaddress 0x10000000 -hiaddress 0x10002fff
ihwconnect -bus busM  -busmasterport pm -instancename br1 \
           -loaddress 0x00000000 -hiaddress 0x00001fff
```

This is valid in OVP but if trying to generate a SystemC TLM platform it is not possible to map regions back onto the same bus.

7.1.1.2 Using a separate bus and two bridges to map a region



An alternate method, and recommended as this overcomes a failure in SystemC, is to create a second bus on which the components and memory are located separately to the bus masters which see the aliased memory. This requires the use of two bus bridges, one to make the original mapping and the second to make the aliased mapping.

```
ihwaddbridge -instancename br1
ihwconnect -bus busM  -busslaveport  ps -instancename br1 \
           -loaddress 0x00000000 -hiaddress 0x00001fff
ihwconnect -bus busS  -busmasterport pm -instancename br1 \
           -loaddress 0x00000000 -hiaddress 0x00001fff
ihwaddbridge -instancename br2
ihwconnect -bus busM  -busslaveport  ps -instancename br2 \
           -loaddress 0x10000000 -hiaddress 0x10001fff
ihwconnect -bus busS  -busmasterport pm -instancename br2 \
           -loaddress 0x00000000 -hiaddress 0x00001fff
```

7.1.1.3 Using bridges to model unconnected top address bit

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.

```
ihwaddbridge -instancename br1
ihwconnect -bus busM  -busslaveport  ps -instancename br1 \
           -loaddress 0x80000000 -hiaddress 0xffffffff
ihwconnect -bus busS  -busmasterport pm -instancename br1 \
           -loaddress 0x00000000 -hiaddress 0x7fffffff
```

From which iGen generates the following C function usage

```
opBridgeNew(
  mi,                // module
```

```
  "br1",              // name of this bridge
  OP_CONNECTIONS(
    OP_BUS_CONNECTIONS(
      OP_BUS_CONNECT(busM, "ps",
                     .addrlo=0x80000000, .addrHi=0xffffffff, .slave=1),
      OP_BUS_CONNECT(busM, "pm",
                     .addrlo=0x00000000, .addrHi=0x7fffffff),
  )
)
```

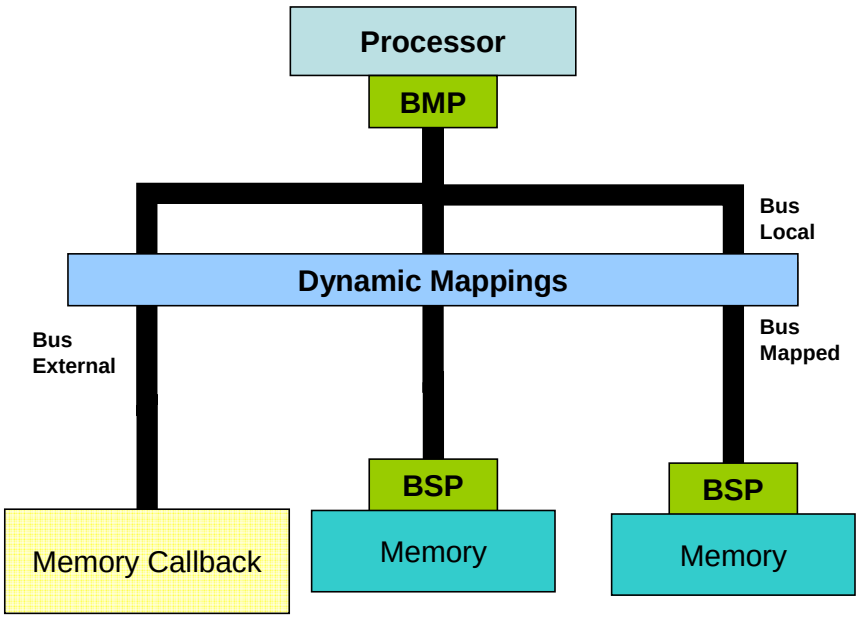
7.2 Dynamic Bus Bridges

7.2.1 Introduction

A dynamic bus bridge allows a testbench or module to make dynamic changes to the address space visible on a bus. It can map 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/SimulationControl/dynamicBridge
```

The bridge, as part of the module definition, is initially used to map the full extent of the processor address map of the 'local' bus to the 'mapped' bus. As the program executes

the testbench dynamically re-maps the busses 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 module contains a single processor and memory and is defined in

```
$IMPERAS_HOME/Examples/SimulationControl/dynamicBridge/platform/cpuSystem/module
.op.tcl
```

The processor and the memory are connected onto separate busses with an initial bus bridge mapping the full address extent from the local processor connected bus to the bus on which the memory is connected. This is created as part of the module using:

```
ihwaddbridge -instancename br1
ihwconnect -bus busLocal -busslaveport ps -instancename br1 \
           -loaddress 0x00000000 -hiaddress 0xffffffff
ihwconnect -bus busMapped -busmasterport pm -instancename br1 \
           -loaddress 0x00000000 -hiaddress 0xffffffff
```

The testbench is used to instantiate the module and then perform the re-mapping of the connections between the local and mapped busses as the application program is executed on the processor.

The ‘subSystem’ module is instantiated as ‘u1’.

```
      const char *u1_path = "platform/cpuSystem";
      optModuleP module = opModuleNew(
       mi,       // parent module
       u1_path,  // modelfile
       "u1",     // name
       0,
       0
   );
```

In order to modify the mappings from the testbench the objects must be discovered from the module.

```
    busLocal  = opObjectByName(mi, "u1/busLocal", OP_BUS_EN ).Bus;
    busMapped = opObjectByName(mi, "u1/busMapped", OP_BUS_EN ).Bus;
```

These objects may then be used to dynamically change the mappings between the busses.

Mappings may be made to connect the local bus to an alternate external test bus

```
    opDynamicBridge(object->busLocal,object->busExternal,
                    0x00400000, 0x004000ff,0x00400000);
```

or to map back to the original bus

```
    opDynamicBridge(object->busLocal,object->busMapped,
                    0x00400000, 0x0040000f,0x00400000);
```

and if required a mapping can be deleted, leaving a 'hole' in the memory address.

```
    opDynamicUnbridge(object->busLocal,
                      0x00400000, 0x004000ff);
```

7.2.2 Running the simulation

The testbench harness is compiled to an executable for simulation execution. The standard command line parser is included to allow ease of use.

To build the testbench, modules and a simple test application Makefiles are used.

For compilation of the application take a copy and build it:

```
> cp -r $IMPERAS_HOME/Examples/SimulationControl/dynamicBridge .
> cd dynamicBridge
> make -C application
```

And to use iGen to generate the platform and then to build

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

The execution of the testbench is performed with the call to the compiled executable:

```
> harness/harness.${IMPERAS_ARCH}.exe \
            --program application/asmtest.${CROSS}.elf \
            --trace --tracechange
```

So that we can see what is happening in this testbench execution optional arguments are added to the command line to turn on tracing of instruction execution *--trace* and also register change *--tracechange*.

When the example script is executed the following should be observed:

The building of the testbench and module components :

```
make: Entering directory 'application'
Compiling Application asmtest.OR1K.o
Linking Application asmtest.OR1K.elf
make: Leaving directory 'application'
make: Entering directory 'module'
# iGen Create OP MODULE module
# Copying STUBS module.c.igen.stubs to module.c
# Host Depending obj/Linux32/module.d
make: Leaving directory 'module'
make: Entering directory 'module'
# Host Compiling Module obj/Linux32/module.o
# Host Linking Module object model.so
make: Leaving directory 'module'
```

```
make: Entering directory 'harness'
# Host Depending obj/Linux32/harness.d
make: Leaving directory 'harness'
make: Entering directory 'harness'
# Host Compiling Platform obj/Linux32/harness.o
# Host Linking Platform harness.Linux32.exe
# Host Linking Platform object model.so
make: Leaving directory 'harness'
```

The execution of the testbench with application on the processor.

Testbench initialization and setup:

```
Info (harness) Setting up bus regions as callbacks on test bus (busExternal)
Info (harness) Root Module Simulate Phase
Info (harness) Find processor 'u1/cpu1'
Info (harness) Processor 'u1/cpu1': Run for 10 instructions
```

The execution of a fixed number of instructions of the test application on the processor, instruction and register change tracing. This shows the write of 0x1234 to address 0x00400000 and the subsequent read back of the same value from that address with the default mapping in place:

```
Info 'u1/cpu1', 0x0000000001000074(_start): l.addi   r1,r0,0x0
Info 'u1/cpu1', 0x0000000001000078(_start+4): l.addi   r2,r0,0x1
Info   R2 deadbeef -> 00000001
Info 'u1/cpu1', 0x000000000100007c(_start+8): l.addi   r3,r0,0x1234
Info   R3 deadbeef -> 00001234
Info 'u1/cpu1', 0x0000000001000080(_start+c): l.addi   r4,r0,0x800
Info   R4 deadbeef -> 00000800
Info 'u1/cpu1', 0x0000000001000084(_start+10): l.muli   r4,r4,0x800
Info   R4 00000800 -> 00400000
Info 'u1/cpu1', 0x0000000001000088(_start+14): l.sw     0x0(r4),r3
Info 'u1/cpu1', 0x000000000100008c(_start+18): l.nop    0x0
Info 'u1/cpu1', 0x0000000001000090(_start+1c): l.lwz    r5,0x0(r4)
Info   R5 deadbeef -> 00001234
Info 'u1/cpu1', 0x0000000001000094(_start+20): l.nop    0x0
Info 'u1/cpu1', 0x0000000001000098(_start+24): l.nop    0x0
Info (harness) Processor 'u1/cpu1' stopped for reason 'Scheduler has expired'
```

The previous instructions were completed and the testbench modifies the mapping of a region from 0x00400000 to 0x004000ff so that it now maps to a bus created in the testbench and that includes callbacks to monitor any access.

The execution of the next fixed number of instructions of the test application on the processor shows the read from address 0x00400000 trigger the testbench read callback which returns the value 0x00000000.

```
Info (harness) Processor 'u1/cpu1' stopped for reason 'Scheduler has expired'
Info (harness) Bridge region of 'busLocal' to 'busExternal'
Info (harness) Processor 'u1/cpu1': Run for 4 instructions
Info 'u1/cpu1', 0x000000000100009c(_start+28): l.nop    0x0
Info 'u1/cpu1', 0x00000000010000a0(_start+2c): l.lwz    r5,0x0(r4)
Info (harness_RCB) readCallback: busExternal, 0x400000
Info   R5 00001234 -> 00000000
Info 'u1/cpu1', 0x00000000010000a4(_start+30): l.nop    0x0
```

```
Info 'u1/cpu1', 0x00000000010000a8(_start+34): l.nop    0x0
Info (harness) Processor 'u1/cpu1' stopped for reason 'Scheduler has expired'
```

The previous instructions were completed and the testbench modifies the mapping of a region from 0x00400000 to 0x0040000f so that it now maps back to the original bus and hence the original memory content.

The execution of the next fixed number of instructions of the test application on the processor shows the read from address 0x00400000 returns the value 0x00001234.

```
Info (harness) Bridge partial region of 'busLocal' back to 'busLocal'
Info (harness) Processor 'u1/cpu1': Run for 4 instructions
Info 'u1/cpu1', 0x00000000010000ac(_start+38): l.nop    0x0
Info 'u1/cpu1', 0x00000000010000b0(_start+3c): l.lwz    r5,0x0(r4)
Info   R5 00000000 -> 00001234
Info 'u1/cpu1', 0x00000000010000b4(_start+40): l.nop    0x0
Info 'u1/cpu1', 0x00000000010000b8(_start+44): l.nop    0x0
Info (harness) Processor 'u1/cpu1' stopped for reason 'Scheduler has expired'
```

The previous instructions were completed and the testbench modifies the mapping of a region from 0x00400000 to 0x004000ff to remove all mappings.

The execution of the next fixed number of instructions of the test application on the processor shows the read from address 0x00400000 causes a read exception i.e. the memory access fails.

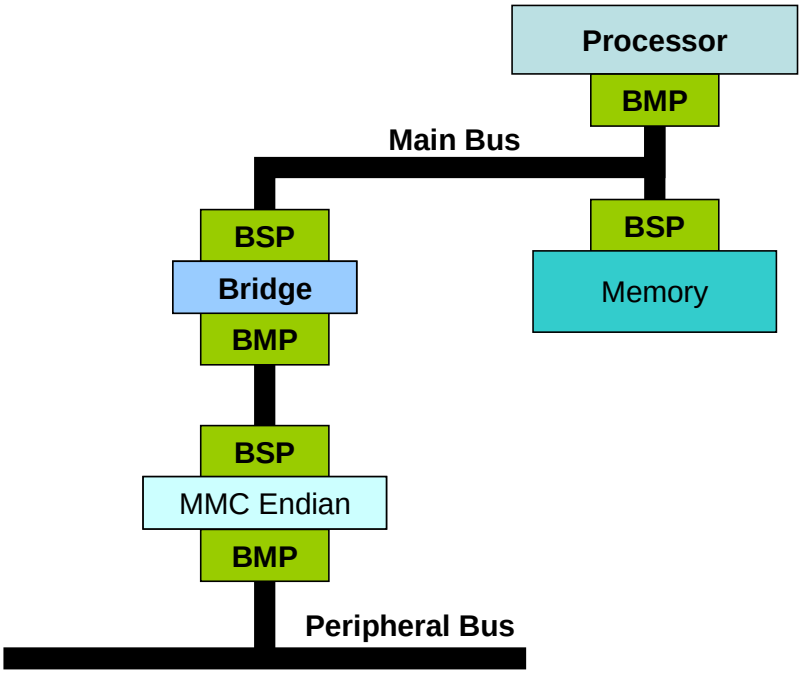
```
Info (harness) UnBridge region of 'busLocal'
Info (harness) Processor 'u1/cpu1': Run for 4 instructions
Info 'u1/cpu1', 0x00000000010000bc(_start+48): l.nop    0x0
Info 'u1/cpu1', 0x00000000010000c0(_start+4c): l.lwz    r5,0x0(r4)
Info (harness) Processor 'u1/cpu1' stopped for reason 'Read privilege exception'
Processor Exception (PC_PRX) Processor 'u1/cpu1' 0x10000c0: l.lwz    r5,0x0(r4)
Processor Exception (PC_RPX) No read access at 0x400000
```

8 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.

8.1 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 example is listed in:

```
$IMPERAS_HOME/Examples/PlatformConstruction/byteSwapperMMC
```

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.

Look at the module definition in the example in:

```
$IMPERAS_HOME/Examples/PlatformConstruction/byteSwapperMMC/module/module.op.tcl
```

This shows the instantiation of the components as shown in the above diagram. This includes the instantiation of the byteSwap MMC model from the VLNV library.

```
ihwaddmmc -instancename swap \
    -vendor "ovpworld.org" -library "mmc" -type "endianSwap" -version "1.0"
```

Which is connected between the two busses, *busInter* (which maps to a small address space in the processor memory space) and *busPeripheral* (which includes the peripheral).

```
ihwconnect -bus busInter -instancename swap -busslaveport sp1
ihwconnect -bus busPeripheral -instancename swap -busmasterport mp1
```

The module is executed using the Imperas harness by passing the module shared object (dynamic link library) and the program to be loaded (note: run example.sh to compile the application and module first):

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

The peripheral model is a simple programmer's interface model. This provides registers that can be accessed, in this case all are read only, which are initialized at reset to known values.

When executed the following should be observed

```
./example.sh
Starting
R0 = 0x01020304
R1 = 0x11121314
R2 = 0x21222324
R3 = 0x31323334
Done
Info (endianSwap) TOTAL SWAPPED BYTES:              16
```

The four registers of the peripheral have been accessed and a byte swapped version of the data displayed.

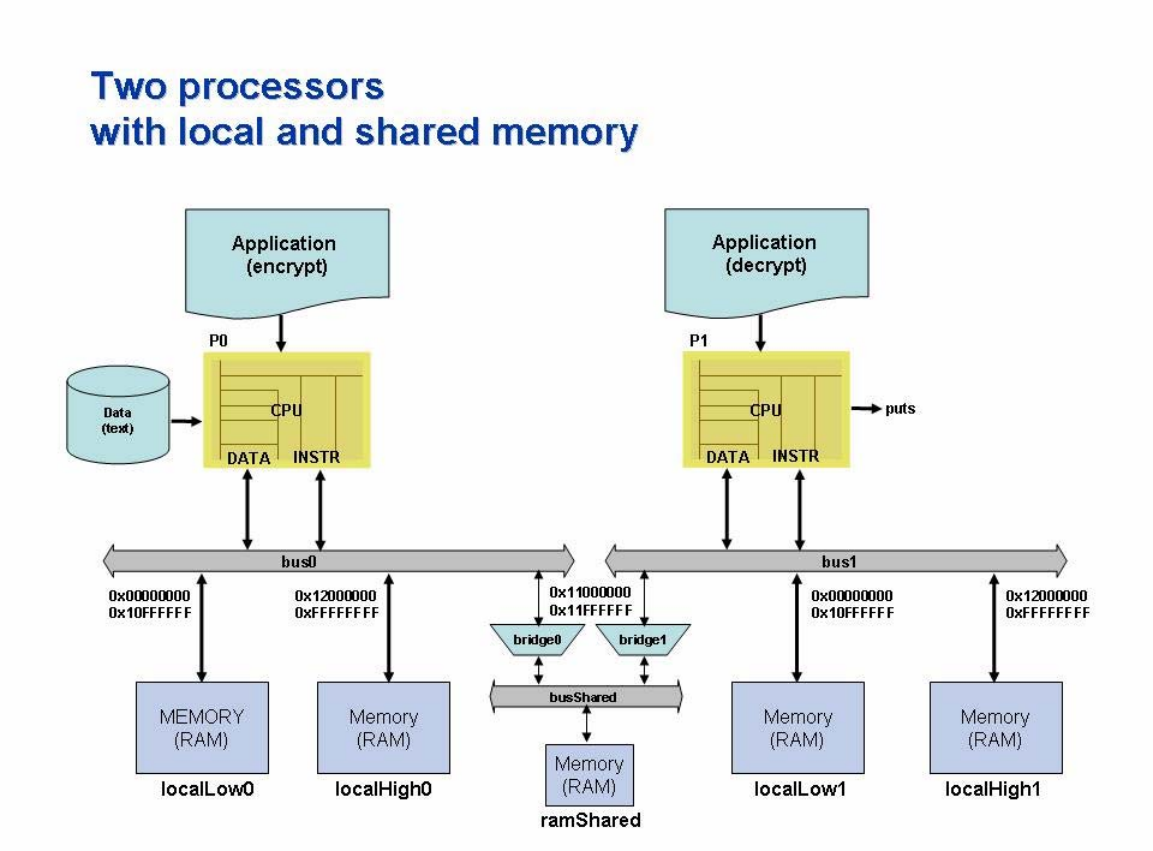
8.2 Performance considerations

In the simulator, the byte swapping MMC 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 because the peripheral behavior is itself modeled by function calls and there will be comparatively few accesses to the peripheral device.

9 Two processors with shared memory

The following diagram shows the example hardware design to be built. It is a two-processor design with both local and shared memory.



This example is documented in:

```
$IMPERAS_HOME/Examples/PlatformConstruction/twoProcessorSharedMemory
```

9.1 Adding Hardware Elements

The first step is to add the hardware elements.

Each element is added in turn. This design contains 2 processors, 5 ram elements, 2 bridge elements and the associated interconnect. These elements are in the standard library; however, no checking takes place at this stage to ensure that the library models exist. This means that it is possible to create the hardware design before the models are completed.

Components can be added in any order, though this design starts with the processors. This requires a choice of processor type – in this case an or1k – and a unique instance name by which it can be identified. The command used is *ihwaddprocessor*

This command has other options not listed here. You can use the command:

```
> igen.exe --apropos addprocessor
```

for more information. The meanings of simulation options are documented in the simulator manual. Example:

```
# add the processors
ihwaddprocessor -type or1k -instancename P0 -semihostname  or1kNewlib -variant
generic
ihwaddprocessor -type or1k -instancename P1 -semihostname  or1kNewlib -variant
generic
```

This adds two OR1K processors to the design with unique instance names P0 & P1. In this case the name of a semi-host library has been specified but this is optional.

RAM is added using the *ihwaddmemory* command as follows. This refers to the simple built-in model. There are two kinds of memory that can be added in this way; *ram* which can be read or written and *rom* which can be loaded from an imagefile, but not written by other hardware. The *ram* and *rom* components are generic so can be connected with any address range - their size adjusts accordingly and is specified by the *ihwconnect* command. Specifically, sized memories can be added to your library if it is required to restrict the design to use 'real' components.

```
# add memories (local and shared)
ihwaddmemory -type ram -instancename localLow0
ihwaddmemory -type ram -instancename localHigh0
ihwaddmemory -type ram -instancename localLow1
ihwaddmemory -type ram -instancename localHigh1
ihwaddmemory -type ram -instancename ramShared
```

Buses and Bridges are added using the *ihwaddbus* and *ihwaddbridge* commands as follows

```
# add the buses
ihwaddbus -instancename bus1     -addresswidth "32"
ihwaddbus -instancename bus0     -addresswidth "32"
ihwaddbus -instancename busShare -addresswidth "32"

# add the bus bridges
ihwaddbridge -instancename bridge0
ihwaddbridge -instancename bridge1
```

This adds three buses and two bridge elements. Again, all have unique names and, in the case of the buses, the address width has been specified.

9.2 Making Connections

Component instances need to be connected using the *ihwconnect* command. This allows the user to specify the precise details of how the elements are connected together. There are a number of different kinds of connection; some are a simple connection of one element to another, others are more complex and require the specification of address ranges.

ihwconnect is used to make all connections. In the design example shown above, the connections need to be made between each processor and its associated bus, between each bus and the memory elements and between the bus and the bridge elements.

Connections are made by reference to either the unique instance names which were specified in the *ihwadd** commands, and the name of the port being connected, or to an external port on this design.

bus0 and *bus1* have connections to:

- Master port connection to the *DATA* and *INSTRUCTION* ports of the processor
- Slave port connections to the low and high local memories
- A slave port connection to a bridge to the shared memory

busShare has connections to:

- Master port connections to each of the bus bridges
- A slave port connection to the shared memory

These connections are added as shown here:

```
# add the processor connections
ihwconnect -bus bus0 -instancename P0         -busmasterport "INSTRUCTION"
ihwconnect -bus bus0 -instancename P0         -busmasterport "DATA"
ihwconnect -bus bus0 -instancename localLow0  -busslaveport  "sp0" \
                           -loaddress "0x00000000" -hiaddress "0x10ffffff"
ihwconnect -bus bus0 -instancename bridge0    -busslaveport  "sp0" \
                           -loaddress "0x11000000" -hiaddress "0x11ffffff"
ihwconnect -bus bus0 -instancename localHigh0 -busslaveport  "sp0" \
                           -loaddress "0x12000000" -hiaddress "0xffffffff"

# add connections to bus1
ihwconnect -bus bus1 -instancename P1         -busmasterport "INSTRUCTION"
ihwconnect -bus bus1 -instancename P1         -busmasterport "DATA"
ihwconnect -bus bus1 -instancename localLow1  -busslaveport  "sp1" \
                           -loaddress "0x00000000" -hiaddress "0x10ffffff"
ihwconnect -bus bus1 -instancename bridge1    -busslaveport  "sp1" \
                           -loaddress "0x11000000" -hiaddress "0x11ffffff"
ihwconnect -bus bus1 -instancename localHigh1 -busslaveport  "sp1"
                           -loaddress "0x12000000" -hiaddress "0xffffffff"

# add connections to busShare
ihwconnect -bus busShare -instancename bridge0   -busmasterport "mp0" \
                           -loaddress "0x00000000" -hiaddress "0x00ffffff"
ihwconnect -bus busShare -instancename bridge1   -busmasterport "mp1"" \
                           -loaddress "0x00000000" -hiaddress "0x00ffffff"
ihwconnect -bus busShare -instancename ramShared -busslaveport  "sp0"" \
                           -loaddress "0x00000000" -hiaddress "0x00ffffff"
```

Slave port connections must always have an address range specifying the input addresses that will map to that port.

A bridge supports uni-directional transfers from the slave port to the master port i.e. a master on one bus accesses the slave port and this is translated to an access from the master port onto a second bus. This requires the specification of address ranges using the *–loaddress* and *–hiaddress* options for both the master and slave ports.

9.3 The example encrypt and decrypt applications

An example application is provided in the applications directories, one for each processor. The applications illustrate that accesses are being performed in the shared memory region. Each processor has its own private memory to hold the program and the stack and data sections.

The first application, encrypt, reads input from a text file, does a trivial byte by byte encryption by xor'ing each byte read with the next value returned by rand(), and places each encrypted byte into a frame buffer. When the frame buffer is full (or the end of the file is reached) the value of ENCRYPT_INDEX, which is in shared memory, is incremented.

The source of the encrypt application is:

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

#include "sharedData.h"

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

    bufferP      buffer   = SHARED_BLOCK;
    const char  *fileName = argc < 2 ? "constitution.txt" : argv[1];
    FILE        *file     = fopen(fileName, "r");
    int          findex   = 0;
    int          done     = 0;
    size_t       num;

    // seed random number generator
    srand(RAND_SEED);

    // check file can be opened
    if(!file) {
        printf("Unable to open file %s for read\n", fileName);
        return -1;
    }

    // handle each frame of data
    while(!done && (findex<NUM_FRAMES)) {

        int i;
```

```
        // read next frame of data from input file
        num = fread(&buffer->frame[findex], 1, FRAME_SIZE, file);

        // identify the last frame being sent
        // note: this assumes input data does not include a 0 (e.g. text file)
        if(num!=FRAME_SIZE) {
            buffer->frame[findex][num] = 0;
            done = 1;
        }

        // encrypt each character in the frame
        for(i=0; i<FRAME_SIZE; i++) {
            buffer->frame[findex][i] ^= rand();
        }

        // step to the next frame
        *ENCRYPT_INDEX = ++findex;
    }

    return 0;
}
```

The decrypt application watches for the ENCRYPT_INDEX value in shared memory to be incremented by the encrypt application. When it has been incremented it decrypts each byte in the next frame into a local buffer and then writes the buffer to stdout (which is semihosted to the simulator console).

The source of the decrypt application is:

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

#include "sharedData.h"

#define TIMEOUT 100000

//
// Wait until the next frame is ready
//
void waitForFrame(int findex) {

    int idleCount = 0;

    while(findex == *ENCRYPT_INDEX) {
        if (idleCount++ >= TIMEOUT) {
            printf ("Timeout waiting for frame %d\n", findex);
            exit(-1);
        }
    }
}

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

    bufferP buffer = SHARED_BLOCK;
    int     findex = 0;
    int     ch     = -1;
```

```
    char    writeBuf[FRAME_SIZE+1];

    // seed random number generator
    srand(RAND_SEED);

    // write each decrypted frame as it is ready until entire message is sent
    while(ch && (findex<NUM_FRAMES)) {

        int i;

        // spin until encrypted frame is prepared
        waitForFrame(findex);

        // decrypt each character in the frame
        for(i=0; (i<FRAME_SIZE) && ch; i++) {
            ch = writeBuf[i] = (buffer->frame[findex][i] ^ rand());
        }

        // terminate the string to write
        writeBuf[i] = 0;

        // write output
        printf("\n**** FRAME %i ****\n\n", findex);
        puts(writeBuf);
        fflush(0);

        // step to next frame
        findex++;
    }

    return 0;
}
```

The files *shared.h* and *sharedData.h* define values that are used in both the encrypt and decrypt applications:

shared.h defines the location of the shared memory:

```
// these define the shared memory range
#define SHARED_LOW      0x11000000
#define SHARED_HIGH     0x11ffffff
```

While *sharedData.h* defines the structure and the locations of shared data passed between the programs:

```
#include "shared.h"

// define 32 frames of 1024 characters each
#define NUM_FRAMES 32
#define FRAME_SIZE 1024

typedef struct bufferS {
    char frame[NUM_FRAMES][FRAME_SIZE];
} buffer, *bufferP;

// index numbers of frame being encrypted
#define ENCRYPT_INDEX   ((volatile int *)(SHARED_LOW+0))

// address of data block
#define SHARED_BLOCK    ((bufferP)(SHARED_LOW+0x1000))
```

```
// seed used for random number generation
#define RAND_SEED 0x12345678
```

Note that since only the encrypt application writes to the shared memory no semaphore is used to lock the memory, as would be used in a more complex shared memory application.

9.4 Running the Example

Take a copy of the example directory tree:

```
> cp -r $IMPERAS_HOME/Examples/PlatformConstruction/twoProcessorSharedMemory .
```

and build and run it using the provided script:

```
> cd twoProcessorSharedMemory
> ./example.sh
```

After building the applications and module the simulation may be run manually using the *harness.exe* program to execute the module. The applications and the command line arguments passed to the applications are specified on the harness.exe command line as follows:

```
> harness.exe --modulefile module/model.so \
            --program twoProcessorShared/P0=application/encrypt.OR1K.elf \
            --program twoProcessorShared/P1=application/decrypt.OR1K.elf \
            --argv application/constitution.txt
```

The output of the simulation (trimmed for brevity) is as follows:

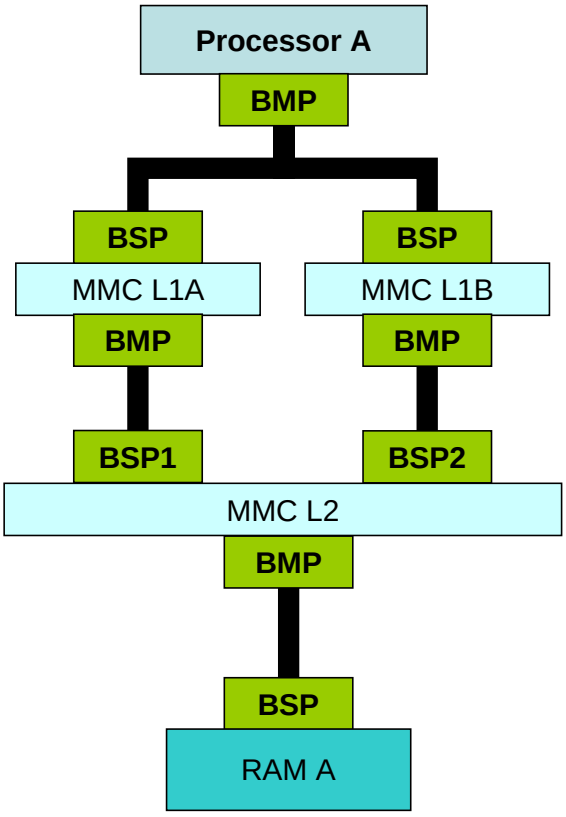
```
OVPsim (32-Bit) v20190306.0 Open Virtual Platform simulator from
www.OVPworld.org.
Copyright (c) 2005-2016 Imperas Software Ltd.
Visit www.IMPERAS.com for multicore debug, verification and analysis solutions.

OVPsim started: Fri Feb 12 20:07:32 2016

**** FRAME 0 ****

THE CONSTITUTION OF THE UNITED STATES OF AMERICA

Preamble
...
 United States, and who shall not, when elected, be an inh

**** FRAME 1 ****

abitant of that state in which he shall be chosen.
...

**** FRAME 26 ****

South Carolina: J. Rutledge, Charles Cotesworth Pinckney, Charles Pinckney,
Pierce Butler
```

```
Georgia: William Few, Abr Baldwin

OVPsim finished: Fri Feb 12 20:09:44 2016

OVPsim (32-Bit) v20190306.0 Open Virtual Platform simulator from
www.OVPworld.org.
Visit www.IMPERAS.com for multicore debug, verification and analysis solutions.
```

Thus showing the data has been passed through the shared memory from the encrypting processor to the decrypting processor.

10 Caches (using Memory Model Components (MMC))

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 (that is a bus master), 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.

10.1 Transparent or Full MMC Models

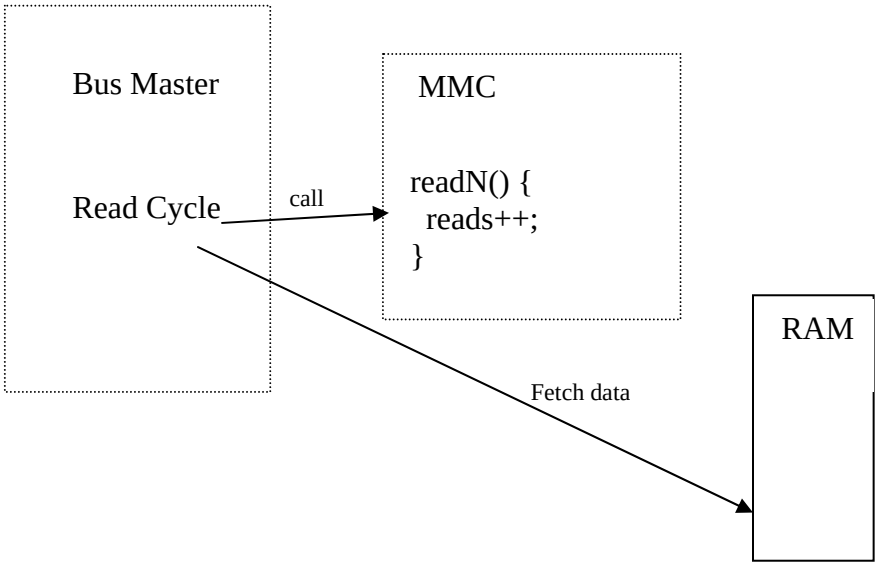
An MMC operates in one of two possible modes, *transparent* or *full*. An MMC can be written to support one or either mode. *Full* models implement storage and so can be used to accurately model components such as caches that are incoherent with main memory. *Transparent* models do not implement storage (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.

10.2 MMC Operation

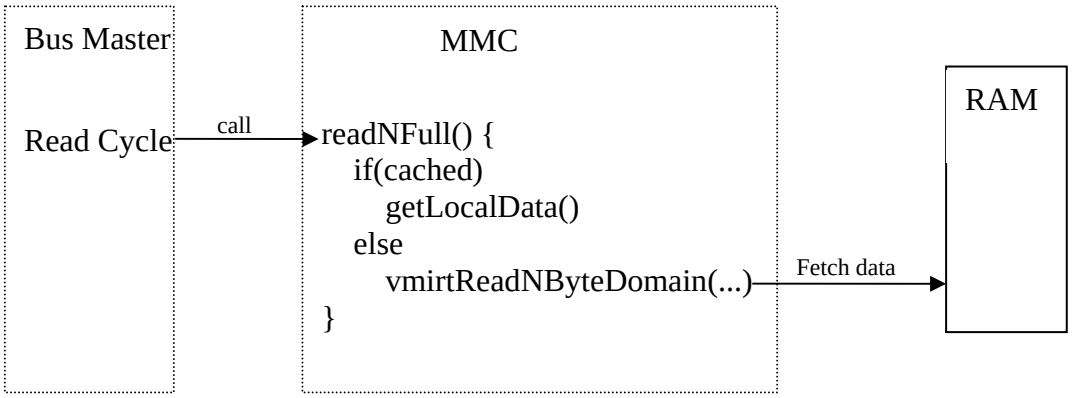
A *full* MMC model has one or more master ports and one or more slave ports. A *transparent* MMC model must have *exactly one master port* and one or more slave ports. Transparent MMCs have only one master port because during construction busses connected to the MMC slave ports are connected straight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

10.2.1 Transparent Model



10.2.2 Full Model



10.3 Creating and connecting an MMC

10.3.1 Transparent MMC Example

An example of a transparent MMC is available at:

```
$IMPERAS_HOME/Examples/PlatformContruction/transparentMMC
```

This has an iGen module definition file containing:

```
ihwnew -name transparentMMC

ihwaddbus -instancename ibus -addresswidth 32
ihwaddbus -instancename dbus -addresswidth 32
ihwaddbus -instancename mbus -addresswidth 32

#
# Add a processor to do some reading and writing
#
ihwaddprocessor -instancename cpu1 \
    -vendor ovpworld.org -library processor -type or1k -version 1.0 \
    -semihostname or1kNewlib -variant generic
ihwconnect -bus ibus -instancename cpu1 -busmasterport INSTRUCTION
ihwconnect -bus dbus -instancename cpu1 -busmasterport DATA

#
# Add MMC Instruction Cache
#
ihwaddmmc -instancename mmci \
    -vendor "ovpworld.org" -library "mmc" -type "wb_1way_32byteline_2048tags" \
    -transparent
ihwconnect -bus ibus -instancename mmci -busslaveport sp1
ihwconnect -bus mbus -instancename mmci -busmasterport mp1

#
# Add MMC Data Cache
#
ihwaddmmc -instancename mmcd \
```

```
    -vendor "ovpworld.org" -library "mmc" -type "wb_1way_32byteline_2048tags" \
    -transparent
ihwconnect -bus dbus -instancename mmcd -busslaveport sp1
ihwconnect -bus mbus -instancename mmcd -busmasterport mp1

#
# Memory on the main bus
#
ihwaddmemory -instancename ramM -type ram
ihwconnect -bus mbus -instancename ramM \
           -busslaveport sp1 -loaddress 0x00000000 -hiaddress 0x003fffff
ihwaddmemory -instancename ramMStack -type ram
ihwconnect -bus mbus -instancename ramMStack \
           -busslaveport sp1 -loaddress 0x00401000 -hiaddress 0xffffffff
```

The usage of the argument *-transparent* used when instancing an MMC specifies whether the MMC is *transparent* or *full* respectively. A *transparent* MMC does not make any changes to the accesses being performed by a master on the connected bus. A *full* MMC can modify accesses and provides additional behavior to the execution.

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.

Take a copy of the example directory tree:

```
> cp -r $IMPERAS_HOME/Examples/PlatformConstruction/transparentMMC .
```

and build and run it using the provided script:

```
> cd transparentMMC
> ./example.sh
```

Example output is as follows:

```
cacheConstructor called for transparentMMC/mmci
  ------------------------------------------
  Ways      : 1
  Line bits : 5
  Tag bits  : 11
  ------------------------------------------
  Tags      : 2,048
  Line bytes: 32
  Size      : 65,536
  Tag mask  : ................11111111111.....
  Key mask  : 11111111111111111111111111.....
  ------------------------------------------


cacheConstructor called for transparentMMC/mmcd
  ------------------------------------------
  Ways      : 1
  Line bits : 5
  Tag bits  : 11
```

```
  ------------------------------------------
  Tags      : 2,048
  Line bytes: 32
  Size      : 65,536
  Tag mask  : ................11111111111.....
  Key mask  : 11111111111111111111111111111.....
  ------------------------------------------


cacheLink called for transparentMMC/mmci

cacheLink called for transparentMMC/mmcd
Hello world

cacheDestructor called for transparentMMC/mmci

READ ACCESSES:
  HITS       :           1,996
  MISSES     :             230
  1-byte     :               0
  2-byte     :               0
  4-byte     :           2,226
  8-byte     :               0
  N-byte     :               0 (0 bytes, average size=0.0 bytes)
  TOTAL READ :           2,226
  TOTAL BYTES:           8,904

cacheDestructor called for transparentMMC/mmcd

READ ACCESSES:
  HITS       :             341
  MISSES     :              15
  1-byte     :              30
  2-byte     :              30
  4-byte     :             296
  8-byte     :               0
  N-byte     :               0 (0 bytes, average size=0.0 bytes)
  TOTAL READ :             356
  TOTAL BYTES:           1,274

WRITE ACCESSES:
  HITS       :             282
  MISSES     :              23
  1-byte     :              12
  2-byte     :              12
  4-byte     :             281
  8-byte     :               0
  N-byte     :               0 (0 bytes, average size=0.0 bytes)
  TOTAL WRITE:             305
  TOTAL BYTES:           1,160
  ------------------------------------------
```

10.3.2 Full MMC Example

An example of a full MMC is available at:

```
$IMPERAS_HOME/Examples/PlatformConstruction/fullMMC
```

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

```
ihwaddmmc -instancename mmci \
    -vendor "ovpworld.org" -library "mmc" -type "wb_1way_32byteline_2048tags"
ihwconnect -bus ibus -instancename mmci -busslaveport sp1
ihwconnect -bus mbus -instancename mmci -busmasterport mp1

#
# Add MMC Data Cache
#
ihwaddmmc -instancename mmcd \
    -vendor "ovpworld.org" -library "mmc" -type "wb_1way_32byteline_2048tags"
ihwconnect -bus ibus -instancename mmcd -busslaveport sp1
ihwconnect -bus mbus -instancename mmcd -busmasterport mp1
```

When the *-transparent* argument is not used on the mmc instance it will operate in the default full mode. In full mode, content as well as tags are modeled, so it is possible for the system to demonstrate incoherency effects.

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 full mode, the cache model has behavior to act as a basic cache with a particular cache configuration in terms of number of ways, line size and cache size.

Take a copy of the example directory tree:

```
> cp -r $IMPERAS_HOME/Examples/PlatformConstruction/fullMMC .
```

and build and run it using the provided script:

```
> cd fullMMC
> ./example.sh
```

Example output is as follows:

```
cacheConstructor called for fullMMC/mmci
  ------------------------------------------
  Ways      : 1
  Line bits : 5
  Tag bits  : 11
  ------------------------------------------
  Tags      : 2,048
  Line bytes: 32
  Size      : 65,536
  Tag mask  : ..............11111111111.....
  Key mask  : 11111111111111111111111111.....
  ------------------------------------------


cacheConstructor called for fullMMC/mmcd
  ------------------------------------------
  Ways      : 1
  Line bits : 5
  Tag bits  : 11
  ------------------------------------------
```

```
  Tags      : 2,048
  Line bytes: 32
  Size      : 65,536
  Tag mask  : ................11111111111.....
  Key mask  : 11111111111111111111111111111.....
  ------------------------------------------


cacheLink called for fullMMC/mmci

cacheLink called for fullMMC/mmcd
Hello world

cacheDestructor called for fullMMC/mmci

READ ACCESSES:
  HITS       :           1,996
  MISSES     :             230
  1-byte     :               0
  2-byte     :               0
  4-byte     :           2,226
  8-byte     :               0
  N-byte     :               0 (0 bytes, average size=0.0 bytes)
  TOTAL READ :           2,226
  TOTAL BYTES:           8,904

cacheDestructor called for fullMMC/mmcd

READ ACCESSES:
  HITS       :             341
  MISSES     :              15
  1-byte     :              30
  2-byte     :              30
  4-byte     :             296
  8-byte     :               0
  N-byte     :               0 (0 bytes, average size=0.0 bytes)
  TOTAL READ :             356
  TOTAL BYTES:           1,274

WRITE ACCESSES:
  HITS       :             282
  MISSES     :              23
  1-byte     :              12
  2-byte     :              12
  4-byte     :             281
  8-byte     :               0
  N-byte     :               0 (0 bytes, average size=0.0 bytes)
  TOTAL WRITE:             305
  TOTAL BYTES:           1,160
-------------------------------------------
```

10.3.3 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/PlatformConstruction/cascadedTransparentMMC
```

This has a module file containing:

```
ihwnew -name cascadedTransparentMMC

ihwaddbus -instancename PIbus -addresswidth 32
ihwaddbus -instancename PDbus -addresswidth 32
ihwaddbus -instancename L1Ibus -addresswidth 32
ihwaddbus -instancename L1Dbus -addresswidth 32
ihwaddbus -instancename mbus -addresswidth 32

#
# Add a processor to do some reading and writing
#
ihwaddprocessor -instancename cpu1 -vendor ovpworld.org -library processor -type
or1k -version 1.0 -semihostname or1kNewlib -variant generic
ihwconnect -bus PIbus -instancename cpu1 -busmasterport INSTRUCTION
ihwconnect -bus PDbus -instancename cpu1 -busmasterport DATA

#
# Add MMC L1 Instruction Cache
#
ihwaddmmc -instancename mmcl1i \
    -vendor "ovpworld.org" -library "mmcL1I" -type "wb_1way_32byteline_2048tags"
-version "1.0"
ihwconnect -bus PIbus -instancename mmcl1i -busslaveport sp1
ihwconnect -bus L1Ibus -instancename mmcl1i -busmasterport mp1

#
# Add MMC L1 Data Cache
#
ihwaddmmc -instancename mmcl1d \
    -vendor "ovpworld.org" -library "mmcL1D" -type "wb_1way_32byteline_2048tags"
-version "1.0"
ihwconnect -bus PDbus -instancename mmcl1d -busslaveport sp1
ihwconnect -bus L1Dbus -instancename mmcl1d -busmasterport mp1


#
# Add MMC L2 Cache
#
ihwaddmmc -instancename mmcl2 \
    -vendor "ovpworld.org" -library "mmcL2" -type "wb_1way_32byteline_2048tags"
-version "1.0"
ihwconnect -bus L1Ibus -instancename mmcl2 -busslaveport sp1
ihwconnect -bus L1Dbus -instancename mmcl2 -busslaveport sp2
ihwconnect -bus mbus   -instancename mmcl2 -busmasterport mp1
ihwsetparameter -handle mmcl2 -name numSlavePorts -value 2 -type Uns32

#
# Memory on the main bus
#
ihwaddmemory -instancename ramM -type ram
ihwconnect -bus mbus -instancename ramM -busslaveport sp1 -loaddress 0x00000000
-hiaddress 0x003fffff
ihwaddmemory -instancename ramMStack -type ram
ihwconnect -bus mbus -instancename ramMStack -busslaveport sp1 -loaddress
0x00401000 -hiaddress 0xffffffff
```

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 and model either L1 cache as a *full* model when the L2 cache is *transparent*.

Take a copy of the example directory tree:

```
> cp -r $IMPERAS_HOME/Examples/PlatformConstruction/cascadedTransparentMMC .
```

and build and run it using the provided script:

```
> cd fullMMC
> ./example.sh
```

Example output is as follows:

```
cacheConstructor called for cascadedTransparentMMC/mmcl1i
  ----------------------------------------
  Ways      : 1
  Line bits : 5
  Tag bits  : 11
  ----------------------------------------
  Tags      : 2,048
  Line bytes: 32
  Size      : 65,536
  Tag mask  : ................11111111111.....
  Key mask  : 11111111111111111111111111111.....
  ----------------------------------------


cacheConstructor called for cascadedTransparentMMC/mmcl1d
  ----------------------------------------
  Ways      : 1
  Line bits : 5
  Tag bits  : 11
  ----------------------------------------
  Tags      : 2,048
  Line bytes: 32
  Size      : 65,536
  Tag mask  : ................11111111111.....
  Key mask  : 11111111111111111111111111111.....
  ----------------------------------------


cacheConstructor called for cascadedTransparentMMC/mmcl2
  ----------------------------------------
  Ways      : 1
  Line bits : 5
  Tag bits  : 11
```

```
  ------------------------------------------
  Tags      : 2,048
  Line bytes: 32
  Size      : 65,536
  Tag mask  : ................11111111111.....
  Key mask  : 11111111111111111111111111111.....
  ------------------------------------------


cacheLink called for cascadedTransparentMMC/mmcl1i

cacheLink called for cascadedTransparentMMC/mmcl1d

cacheLink called for cascadedTransparentMMC/mmcl2
Hello world

CpuManagerMulti finished: Fri Jan 15 14:37:48 2016


CpuManagerMulti (32-Bit) v20190306.0 Open Virtual Platform simulator from
www.IMPERAS.com.
Visit www.IMPERAS.com for multicore debug, verification and analysis solutions.


cacheDestructor called for cascadedTransparentMMC/mmcl1i

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 cascadedTransparentMMC/mmcl1d

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

cacheDestructor called for cascadedTransparentMMC/mmcl2
```

```
READ ACCESSES:
  HITS       :              0
  MISSES     :            268
  1-byte     :              0
  2-byte     :              0
  4-byte     :              0
  8-byte     :              0
  N-byte     :            268 (8,576 bytes, average size=32.0 bytes)
  TOTAL READ :            268
  TOTAL BYTES:          8,576
```

11 Using Module Hierarchy in Virtual Platforms

In previous examples we saw that the platforms/modules created with the *ihwnew* command instanced buses, processors, memories, peripherals, etc. These platforms can be thought of as modules and can be used as sub-modules in other platforms or modules.

Hierarchy is created by instancing one module in another.

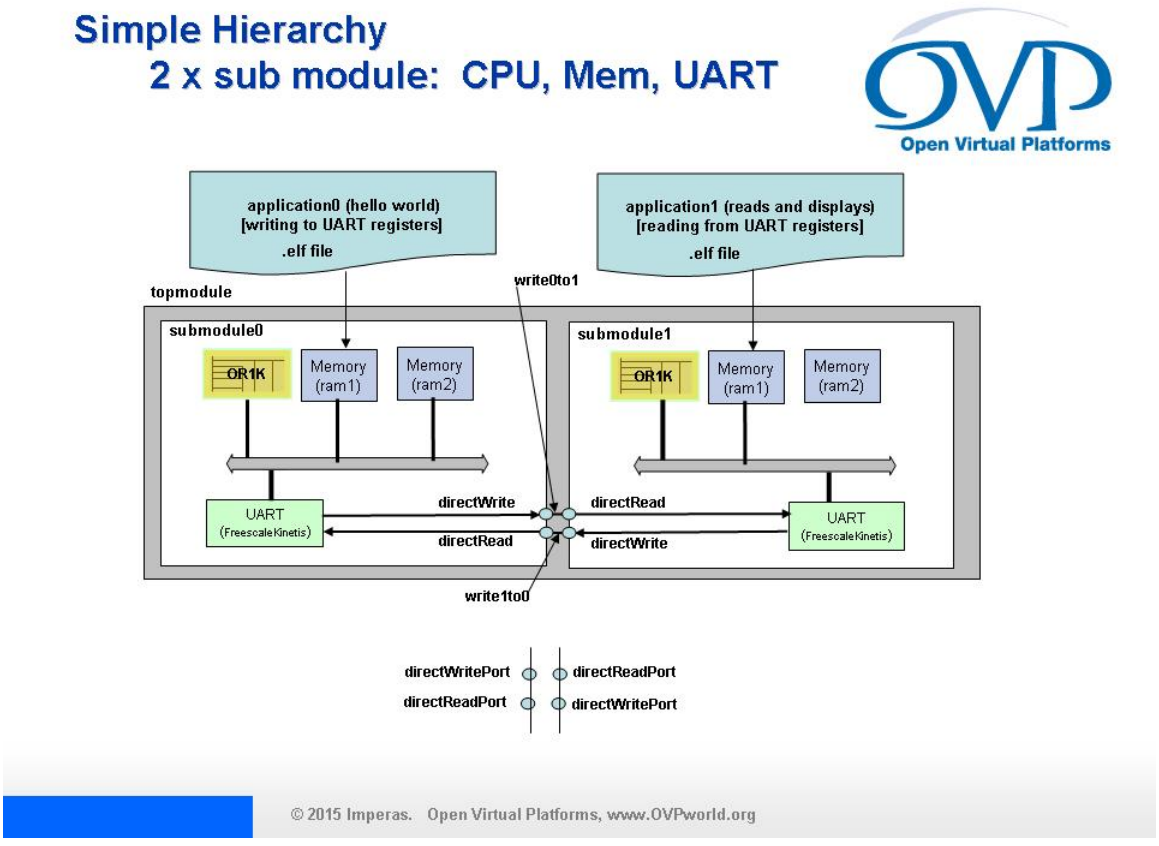
We therefore create each level of the hierarchy as a separate module in a separate directory tree, and compile it separately into a shared object - just like we did for the platforms in the previous examples.

Some modules in the hierarchy could be just instancing previously compiled module shared objects, and leave the instancing of other components such as processors, peripherals, memories etc. to lower level, and leaf modules.

When we create a hierarchy of levels we need to pass connections between levels, such as buses, and also we need to pass parameters, such as addresses.

11.1 A two level platform: simpleHierarchy

If you look at the picture below, you will see a *topmodule* that contains two instances of *submodule*, with each *submodule* containing a processor, memories, bus, and UART.



Note the nets (*directRead*, *directWrite*) and the netports (*directWritePort*, *directReadPort*) in the submodules. It is the modules' ports that allow connection (in this case of nets) between levels of hierarchy. Note also the nets (*write0to1*, *write1to0*) in the topmodule.

This platform is available at

```
$IMPERAS_HOME/Examples/PlatformConstruction/simpleHierarchy
```

Take a copy of the example directory tree:

```
> cp -r $IMPERAS_HOME/Examples/PlatformConstruction/simpleHierarchy .
> cd simpleHierarchy
> ls
application0 application1 example.sh submodule topmodule
```

As we are going to have two different, separately compiled modules, we will now have two module directories, and also there are two application directories.

11.2 Creating the sub module

The *submodule/module.op.tcl* script looks like:

```
ihwnew -name sub
iadddocumentation -name Description \
       -text "This a sub module that includes CPU Memory and UART"

ihwaddbus -instancename mainBus -addresswidth 32
ihwaddnet -instancename directWrite
ihwaddnet -instancename directRead

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

ihwconnect -bus mainBus -instancename cpu1 -busmasterport INSTRUCTION
ihwconnect -bus mainBus -instancename cpu1 -busmasterport DATA

ihwaddmemory -instancename ram1 -type ram
ihwconnect -bus mainBus -instancename ram1 \
       -busslaveport sp1 -loaddress 0x0 -hiaddress 0x0fffffff

ihwaddmemory -instancename ram2 -type ram
ihwconnect -bus mainBus -instancename ram2 \
       -busslaveport sp1 -loaddress 0x20000000 -hiaddress 0xffffffff

ihwaddperipheral -instancename uart0 -vendor freescale.ovpworld.org \
       -library peripheral -type KinetisUART -version 1.0
ihwconnect -bus mainBus -instancename uart0 \
       -busslaveport  bport1 -loaddress 0x100003f8 -hiaddress 0x100013f7
ihwconnect -net directWrite -instancename uart0 -netport  DirectWrite
ihwconnect -net directRead -instancename uart0 -netport  DirectRead
ihwsetparameter -handle uart0 -name directReadWrite -value 1 -type bool

ihwaddnetport -instancename directWritePort
ihwaddnetport -instancename directReadPort

ihwconnect -netport directWritePort -net directWrite
```

```
ihwconnect -netport directReadPort -net directRead
```

New functions used here are:

```
iadddocumentation -name Description \
       -text "This a sub module that includes CPU Memory and UART"
```

which adds a documentation field to the declaration of this module. (*iadddocumentation* adds to the previously declared/instanced item).

and:

```
ihwaddnetport -instancename directWritePort
ihwaddnetport -instancename directReadPort

ihwconnect -netport directWritePort -net directWrite
ihwconnect -netport directReadPort -net directRead
```

which declares the two netports and connects up the nets to them.

11.2.1 Compiling the submodule

To generate and compile the submodule:

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

11.3 Creating the top module

The *topmodule/module.op.tcl* script looks like:

```
ihwnew -name simpleHierarchy

# nets will be how sub modules communicate
ihwaddnet -instancename write0to1
ihwaddnet -instancename write1to0

# Create an instance 0 of the submodule
ihwaddmodule  -instancename sub0 -modelfile submodule/model
ihwconnect -net write0to1 -instancename sub0 -netport directWritePort
ihwconnect -net write1to0 -instancename sub0 -netport directReadPort

# Create an instance 1 of the submodule
ihwaddmodule  -instancename sub1 -modelfile submodule/model
ihwconnect -net write0to1 -instancename sub1 -netport directReadPort
ihwconnect -net write1to0 -instancename sub1 -netport directWritePort
```

11.3.1 Instancing a sub module

The new function used here is *ihwaddmodule*:

```
ihwaddmodule  -instancename sub0 -modelfile submodule/model
ihwconnect -net write0to1 -instancename sub0 -netport directWritePort
ihwconnect -net write1to0 -instancename sub0 -netport directReadPort
```

which adds a module instance and connects up the module's netports.

Note that the definition of the sub module being instanced is being located/referenced using a direct local path specification using the *-modelfile* argument.

Alternatively the sub module could have been located by the specification of a VLNV directory structure location (see example later below).

11.3.2 Compiling the topmodule

To generate and compile the topmodule:

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

11.4 Application0 - writing to the UART

Look at the *application0/application0.c* file. It is similar to the previous application that wrote to the UART.

In the main() is:

```
        printf ("[application0]  Writing to uart\n\n");

        writeMessFreescaleKinetisUart(UART0_BASE, "Hello UART world\n\n");
        writeMessFreescaleKinetisUart(UART0_BASE, "x\n"); // for exit
```

This will use the semihosting of *printf* to write to the simulator console the 'writing' message.

Then there are calls to *writeMessFreescaleKinetisUart* to send messages to the UART. It sends a 'hello' message and then sends 'x' which in our example is used as a message to indicate termination.

The application is compiled with:

```
> make -C application0
# Compiling application0.c
# Linking application0.OR1K.elf
```

11.5 Application1 - reading from the UART

Look at the *application1/application1.c* file. It is similar to the previous application that wrote to the UART.

After UART initialization, it sits in a loop getting characters. When it gets a character it prints it to the simulator console with the semihosted printf. If it gets an 'x' it terminates the application (and thus the simulation).

```
char uart_getchar () {
```

```
    volatile unsigned char *ab_S1 = UART0_BASE + 0x4;
    volatile unsigned char *ab_D  = UART0_BASE + 0x7;
    #define UART_S1_RDRF_MASK 0x20

    while (!(*ab_S1 & UART_S1_RDRF_MASK)) {
    }
    /* Return the 8-bit data from the receiver */
    return *ab_D;
}

int main(int argc, char **argv) {
    initFreeScaleKinetisUart(UART0_BASE);
    printf ("[application1]  Reading from uart\n\n");
    char c;
    while (c = uart_getchar ()) {
        if (c == 'x') {
            printf ("[application1]  Read termination char(x) from uart\n\n");
            break;
        }
    printf ("%c", c);
    }
    return 0;
}
```

The application is compiled with:

```
> make -C application1
# Compiling application1.c
# Linking application1.OR1K.elf
```

11.6 Running the hierarchical platform simulation

Assuming the applications and modules have all been compiled the simulation can be run using the `harness.exe` program:

```
> pwd
simpleHierarchy

> harness.exe \
       --modulefile topmodule/model.so \
       --program simpleHierarchy/sub0/cpu1=application0/application0.OR1K.elf \
       --program simpleHierarchy/sub1/cpu1=application1/application1.OR1K.elf

OVPsim started: Tue Oct 17 11:21:11 2015

[application1] Initializing KinetisUART
[application1]  Reading from uart

[application0] Initializing KinetisUART
[application0]  Writing to uart

Hello UART world

[application1]  Read termination char(x) from uart

OVPsim finished: Tue Oct 17 11:21:11 2015
```

During the simulation we can see that application1 initializes the UART in its sub module, and then waits for input from the UART. Then application0 initializes its UART

and then writes to the UART. We then get the message sent character by character and written to the simulator console as the 'Hello UART world". When application1 reads the termination character the simulation finishes.

Note that we have created the script *example.sh* for your convenience.

If we use the --verbose argument we see that the two processors have executed a similar number of instructions:

```
./example.sh --verbose
...
[application1] Initializing KinetisUART
[application1]  Reading from uart

[application0] Initializing KinetisUART
[application0]  Writing to uart

Hello UART world

[application1]  Read termination char(x) from uart
...
Info CPU 'simpleHierarchy/sub1/cpu1' STATISTICS
Info   Type                 : or1k (generic)
Info   Nominal MIPS         : 100
Info   Final program counter : 0x19d8
Info   Simulated instructions: 716,027
...
Info CPU 'simpleHierarchy/sub0/cpu1' STATISTICS
Info   Type                 : or1k (generic)
Info   Nominal MIPS         : 100
Info   Final program counter : 0x199c
Info   Simulated instructions: 714,388
...
Info TOTAL
Info   Simulated instructions: 1,430,415
...
```

Don't forget you can use other commands, for example:

```
--showbuses
--showdomains
```

12 Hierarchy and Connectivity in Modules

When modules are instanced they may need to communicate by writing to shared resources, for example memory, within other modules. This is achieved by connecting the busses through ports on modules. This example uses three modules connected by busses to allow access to a shared memory and also to a 'mailbox' memory within the processor sub-system. The diagram below shows the system connectivity.

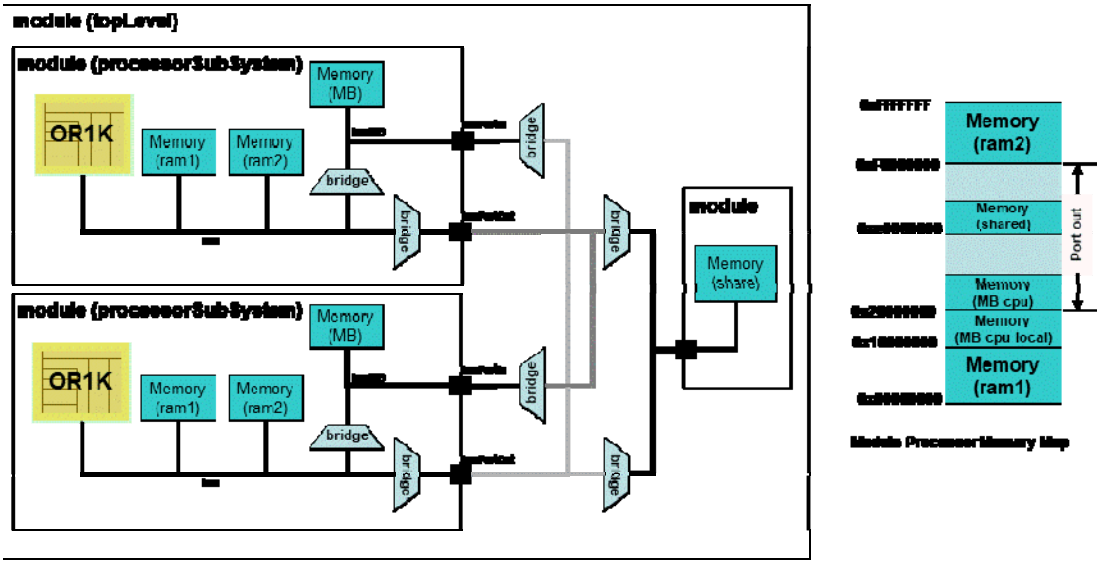
This example can be found at:

```
IMPERAS_HOME/Examples/PlatformConstruction/moduleHeirarchyAndConnectivity
```

Take a copy:

```
> cp -r $IMPERAS_HOME/Examples/PlatformConstruction/moduleHeirarchyAndConnectivity .
> cd moduleHeirarchyAndConnectivity
```

12.1 The top level



The top level instances the other modules of the design and creates the interconnecting bus. This is defined in *topLevel/module.op.tcl*:

Create the design and instance the sub-modules

```
ihwnew -name topLevel

# instance sub-modules
```

```
ihwaddmodule -instancename sys0   -modelfile processorSubSystem

ihwaddmodule -instancename sys1   -modelfile processorSubSystem

ihwaddmodule -instancename shmem  -modelfile sharedMemorySubSystem
```

Connect the sub-system ports to local bus connections

```
# connections
# Sys0
ihwaddbus -instancename busSys0In -addresswidth 32
ihwconnect -instancename sys0 -busport busPortIn -bus busSys0In

ihwaddbus -instancename busSys0Out -addresswidth 32
ihwconnect -instancename sys0 -busport busPortOut -bus busSys0Out

#Sys1
ihwaddbus  -instancename busSys1In -addresswidth 32
ihwconnect -instancename sys1 -busport busPortIn -bus busSys1In

ihwaddbus  -instancename busSys1Out -addresswidth 32
ihwconnect -instancename sys1 -busport busPortOut -bus busSys1Out

# Shared Memory
ihwaddbus -instancename busSM -addresswidth 32
ihwconnect -instancename shmem -busport busPort -bus busSM
```

Make the mapped connections between the sub-systems; connect a sub-system bus out address range to the mailbox address range of bus in of the other sub-system. Connect the sub-system bus out address range to the shared memory bus.

```
# connect out port from Sys1 to Sys0 in port for Mailbox
ihwaddbridge -instancename bridgeSys0MB
ihwconnect   -instancename bridgeSys0MB -busslaveport  "sp" -bus busSys1Out
             -loaddress "0x20000000" -hiaddress "0x20000fff"
ihwconnect   -instancename bridgeSys0MB -busmasterport "mp" -bus busSys0In
             -loaddress "0x10000000" -hiaddress "0x10000fff"
# connect out port from Sys1 to Shared Memory
ihwaddbridge -instancename bridgeSys0SM
ihwconnect   -instancename bridgeSys0SM -busslaveport  "sp" -bus busSys1Out
             -loaddress "0xe0000000" -hiaddress "0xe0000fff"
ihwconnect   -instancename bridgeSys0SM -busmasterport "mp" -bus busSM
             -loaddress "0x00000000" -hiaddress "0x00000fff"


 # connect out port from Sys0 to Sys1 in port for Mailbox
ihwaddbridge -instancename bridgeSys1MB
ihwconnect   -instancename bridgeSys1MB -busslaveport  "sp" -bus busSys0Out
             -loaddress "0x20000000" -hiaddress "0x20000fff"
ihwconnect   -instancename bridgeSys1MB -busmasterport "mp" -bus busSys1In
             -loaddress "0x10000000" -hiaddress "0x10000fff"
# connect out port from Sys0 to Shared Memory
ihwaddbridge -instancename bridgeSys1SM
ihwconnect   -instancename bridgeSys1SM -busslaveport  "sp" -bus busSys0Out
             -loaddress "0xe0000000" -hiaddress "0xe0000fff"
```

```
ihwconnect   -instancename bridgeSys1SM -busmasterport "mp" -bus busSM
             -loaddress "0x00000000" -hiaddress "0x00000fff"
```

The top level brings together two processor sub-systems and a shared memory sub-system

12.2 The processor sub system

This is defined by *processorSubSystem/module.op.tcl* as we have seen in previous examples. Shown below are the bus port connections for the module and connecting these to an intermediate bus.

```
# create and name the module's external bus ports
ihwaddbusport -instancename busPortIn
ihwaddbusport -instancename busPortOut

# busses connecting ports
ihwaddbus -instancename busPIn  -addresswidth "32"
ihwaddbus -instancename busPOut -addresswidth "32"

# connect the external mailbox bus ports to the bus
ihwconnect -busport busPortIn  -bus busPIn
ihwconnect -busport busPortOut -bus busPOut
```

The bus ports are connected using bridges to set the address ranges mapped out of the module

```
# outgoing bridge
ihwaddbridge -instancename bridgeOutMB
ihwconnect   -instancename bridgeOutMB -bus bus     \
             -busslaveport  "sp" -loaddress "0x20000000" -hiaddress "0xefffffff"
ihwconnect   -instancename bridgeOutMB -bus busPOut \
             -busmasterport "mp" -loaddress "0x20000000" -hiaddress "0xefffffff"
```

and also to allow access to only the portions of the module required, here the ‘mailbox’ memory is accessed from the port via a bridge

```
# incoming connection
ihwaddbridge -instancename bridgeIn
ihwconnect   -instancename bridgeIn -bus busPIn \
             -busslaveport "sp" -loaddress "0x10000000" -hiaddress "0x10000fff"
ihwconnect   -instancename bridgeIn -bus busMB  \
             -busmasterport "mp" -loaddress "0x00000000" -hiaddress "0x00000fff"

# internal connection
ihwaddbridge -instancename bridgeLocal
ihwconnect   -instancename bridgeLocal -bus bus   \
             -busslaveport "sp" -loaddress "0x10000000" -hiaddress "0x10000fff"
ihwconnect   -instancename bridgeLocal -bus busMB \
             -busmasterport "mp" -loaddress "0x00000000" -hiaddress "0x00000fff"
```

This creates a processor system with local memory, memory accessible from an external master and a port to access external memory regions.

12.3 The memory sub system

The memory sub-system simply comprises a memory instance connected to the lower address space. This is defined in *sharedMemorySubSystem/module.op.tcl*:

```
ihwnew -name sharedMemorySubSystem

# add memory
ihwaddmemory -instancename ram -type ram

# add bus
ihwaddbus -instancename bus   -addresswidth "32"

# add connections to bus
ihwconnect -instancename ram -bus bus  \
           -busslaveport  "sp" -loaddress "0x00000000" -hiaddress "0x00000fff"

# create and name the module's external bus port
ihwaddbusport -instancename busPort

# connect the external bus ports to the bus
ihwconnect -busport busPort -bus bus
```

12.4 The Test Application

The same test application runs on both of the processors in the design but modified to indicate, in the top byte, which sub-system generated the last write. The application reads from the local ‘mailbox’ memory, the ‘mailbox’ memory in the other processor sub-system and also from the shared memory region. The value read from each memory location is incremented and written back (including the top byte set to the sub-system number plus 1, for example sub-system 0 write 1 in the top byte).
This is a trivial example showing the accesses are available to each memory, it is not intended to show a real situation.

12.5 Building the Example

The application is compiled using the following command:

```
make -C application CROSS=${CROSS}
```

The three parts that make up the virtual platform design are compiled using the following commands:

```
# generate and compile the iGen created module
make -C processorSubSystem
make -C sharedMemorySubSystem
make -C topLevel
```

NOTE: The example shows modules in discrete directories, when systems get more complex it is recommended to use a VLNV library. This allows all components to be compiled using a provided library build system.

12.6 Running the simulation

To run the simulation, we use the standard harness.exe program:

```
> # run the module using the harness
harness.exe --modulefile topLevel/model.${IMPERAS_SHRSUF} \
            --program topLevel/sys0/cpu=application/appSys0.${CROSS}.elf \
            --program topLevel/sys1/cpu=application/appSys1.${CROSS}.elf \
            $*
```

Note

1) the harness loads the top level module shared object
2) the application program is specified for each processor

When executed you should see the following:

```
Sys1: Application Started
Sys0: Application Started
Sys1: Current Values local MB  0 (write Id 0), external MB  0 (write Id 0),
Shared Memory  0 (write Id 0). Increment all.
Sys0: Current Values local MB  1 (write Id 2), external MB  1 (write Id 2),
Shared Memory  1 (write Id 2). Increment all.
Sys1: Current Values local MB  2 (write Id 1), external MB  2 (write Id 1),
Shared Memory  2 (write Id 1). Increment all.
Sys0: Current Values local MB  3 (write Id 2), external MB  3 (write Id 2),
Shared Memory  3 (write Id 2). Increment all.
Sys1: Current Values local MB  4 (write Id 1), external MB  4 (write Id 1),
Shared Memory  4 (write Id 1). Increment all.
Sys0: Current Values local MB  5 (write Id 2), external MB  5 (write Id 2),
Shared Memory  5 (write Id 2). Increment all.
Sys1: Current Values local MB  6 (write Id 1), external MB  6 (write Id 1),
Shared Memory  6 (write Id 1). Increment all.
Sys0: Current Values local MB  7 (write Id 2), external MB  7 (write Id 2),
Shared Memory  7 (write Id 2). Increment all.
Sys1: Current Values local MB  8 (write Id 1), external MB  8 (write Id 1),
Shared Memory  8 (write Id 1). Increment all.
Sys0: Current Values local MB  9 (write Id 2), external MB  9 (write Id 2),
Shared Memory  9 (write Id 2). Increment all.
Sys1: Current Values local MB 10 (write Id 1), external MB 10 (write Id 1),
Shared Memory 10 (write Id 1). Increment all.
Sys0: Current Values local MB 11 (write Id 2), external MB 11 (write Id 2),
Shared Memory 11 (write Id 2). Increment all.
Sys1: Current Values local MB 12 (write Id 1), external MB 12 (write Id 1),
Shared Memory 12 (write Id 1). Increment all.
Sys0: Current Values local MB 12 (write Id 1), external MB 12 (write Id 1),
Shared Memory 12 (write Id 1). Increment all.
Sys1: Current Values local MB 13 (write Id 1), external MB 13 (write Id 1),
Shared Memory 13 (write Id 1). Increment all.
Sys0: Current Values local MB 14 (write Id 2), external MB 14 (write Id 2),
Shared Memory 14 (write Id 2). Increment all.
Sys1: Current Values local MB 15 (write Id 1), external MB 15 (write Id 1),
Shared Memory 15 (write Id 1). Increment all.
Sys0: Current Values local MB 16 (write Id 2), external MB 16 (write Id 2),
Shared Memory 16 (write Id 2). Increment all.
Sys1: Current Values local MB 17 (write Id 1), external MB 17 (write Id 1),
Shared Memory 17 (write Id 1). Increment all.
Sys1: Application Finished
Sys0: Current Values local MB 18 (write Id 2), external MB 18 (write Id 2),
Shared Memory 18 (write Id 2). Increment all.
Sys0: Application Finished
```

Provided is the *example* script to create, compile and run this example.

The information for the design can be obtained by using the *--showoverrides* command, which provides all of the overrides that can be applied to the components in the design and also the simulator, but also provides the components hierarchical names.

```
> example.sh --showoverrides
```

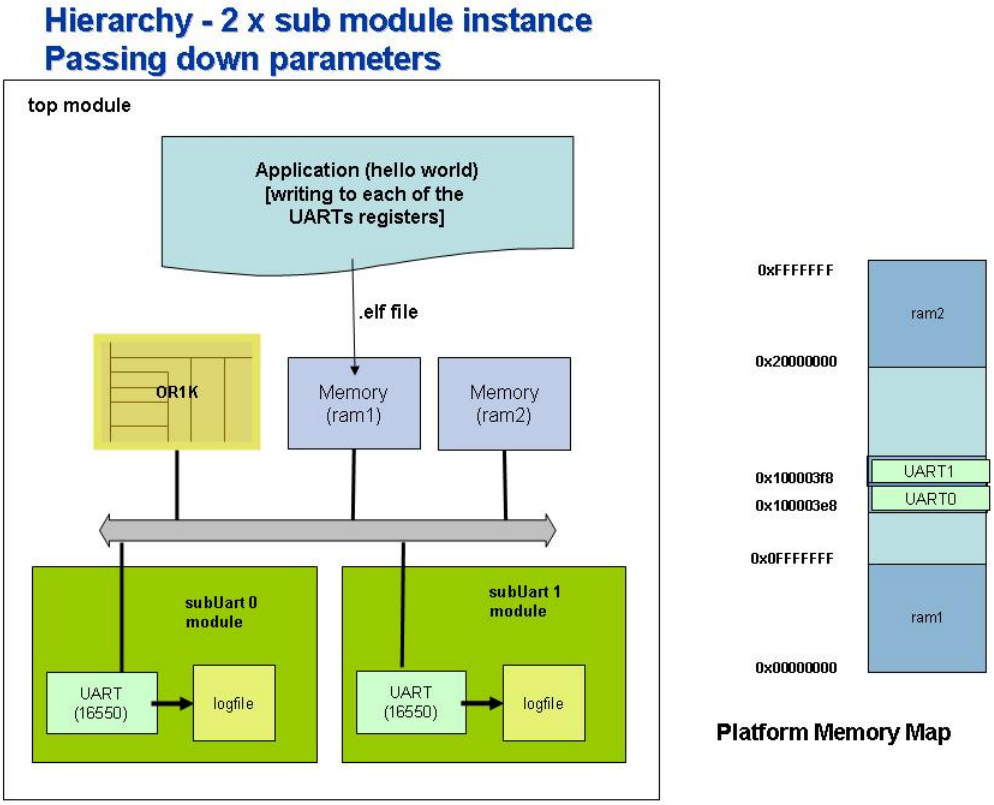
A large amount of output will be generated containing lines similar to the following, which include the cpu hierarchical name:

```
...
--override topLevel/sys0/cpu/defaultsemihost=F (Boolean) (default=F) This
processor will load its default semihost library, specified in the processor
model.
--override topLevel/sys0/cpu/enabletools=F (Boolean) (default=F) Load VAP tools
...
--override topLevel/sys1/cpu/defaultsemihost=F (Boolean) (default=F) This
processor will load its default semihost library, specified in the processor
model.
--override topLevel/sys1/cpu/enabletools=F (Boolean) (default=F) Load VAP tools
for this processor
...
```

13 Passing Parameters down module hierarchy

We have seen in the example above how module ports can be created at a module boundary and how these can be connected to in the level above. These are modeling the structural connection between the levels. Often it is necessary to pass data down from one level to another, for example to pass an address down or a file name.

This example creates a submodule that contains a UART and then instances two copies of it - and using arguments passes down the address that the UART should be located in the memory map, and also the name of the file that the UART should create to log its output.



This example can be found at:

```
IMPERAS_HOME/Examples/PlatformConstruction/moduleParameters
```

So take a copy:

```
> cp -r $IMPERAS_HOME/Examples/PlatformConstruction/moduleParameters .
> cd moduleParameters
```

and you will see the following directories: application, topmodule, submodule and the usual example script to compile and run the example.

13.1 Creating the sub module

Having a look at the *submodule/module.op.tcl*:

```
ihwnew -name subUart

ihwaddformalparameter -name baseAddress -type address
ihwaddformalparameter -name logFileName -type string

ihwaddbus -instancename mainBus -addresswidth 32

ihwaddperipheral -instancename uart0 -vendor national.ovpworld.org \
       -library peripheral -type 16550 -version 1.0

ihwconnect -bus mainBus -instancename uart0 \
       -busslaveport  bport1 -loaddress baseAddress -hiaddress {baseAddress+7}
ihwsetparameter -handle uart0 -name outfile -expression logFileName –type string

ihwaddbusport -instancename mainBusPort
ihwconnect -busport mainBusPort -bus mainBus
```

We create a module called *subUart* that has two formal arguments:

```
ihwaddformalparameter -name baseAddress -type address
ihwaddformalparameter -name logFileName -type string
```

with their types being specified.

When we connect the bus to the UART instance we provide an expression for the hiaddress:

```
ihwconnect -bus mainBus -instancename uart0 \
       -busslaveport  bport1 -loaddress baseAddress -hiaddress {baseAddress+7}
```

That uses *baseAddress*, one of the two formal arguments.

We also set the parameter *outfile* for the UART instance and the value of it is not defined at this declaration time, but will be defined before simulation starts by being passed down from above through this module's formal argument *logFileName* by being set by the level above that instances this module:

```
ihwsetparameter -handle uart0 -name outfile -expression logFileName –type string
```

Thus parameters for a module can be defined that are passed down from above and they can be used in this level as part of expressions.

Also note, the UART is a different one than the previous examples, this one being a national.ovpworld.org 16550.

We then compile the module with make as normal:

```
> make -C submodule
```

13.2 Creating the top module

Having a look at the *topmodule/module.op.tcl*:

```
ihwnew -name topmodule

ihwaddbus -instancename mainBus -addresswidth 32

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

ihwconnect -bus mainBus -instancename cpu1 -busmasterport INSTRUCTION
ihwconnect -bus mainBus -instancename cpu1 -busmasterport DATA

ihwaddmemory -instancename ram1 -type ram
ihwconnect -bus mainBus -instancename ram1 \
       -busslaveport sp1 -loaddress 0x0 -hiaddress 0x0fffffff

ihwaddmemory -instancename ram2 -type ram
ihwconnect -bus mainBus -instancename ram2 \
       -busslaveport sp1 -loaddress 0x20000000 -hiaddress 0xffffffff

# Create instance 0 of the submodule
ihwaddmodule -instancename subUart0 -modelfile submodule/model
ihwconnect -instancename subUart0 -busport mainBusPort -bus mainBus
ihwsetparameter -handle subUart0 -name baseAddress -value 0x100003e8   –type uns32
ihwsetparameter -handle subUart0 -name logFileName -value uartTTY0.log –type string

# Create instance 1 of the submodule
ihwaddmodule -instancename subUart1 -modelfile submodule/model
ihwconnect -instancename subUart1 -busport mainBusPort -bus mainBus
ihwsetparameter -handle subUart1 -name baseAddress -value 0x100003f8   –type uns32
ihwsetparameter -handle subUart1 -name logFileName -value uartTTY1.log –type string
```

We see this is similar to previous examples, though this time in the module instances we set the parameters to pass down specific values for the sub module parameters.

```
ihwsetparameter -handle subUart0 -name baseAddress -value 0x100003e8   –type uns32
ihwsetparameter -handle subUart0 -name logFileName -value uartTTY0.log –type string
```

And we set different addresses and different names for each of the two sub module instances.

We create and compile the model with:

```
> make -C topmodule
```

13.3 The application

The application is simpler than the previous UART examples as it uses a UART that has simpler initialization.

If we look at application/application.c

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

static void writeMess (unsigned char *uartBase,  unsigned const char *myString)
{
       volatile unsigned char *uartTX = uartBase + 0;
       volatile unsigned char *uartLSR = uartBase + 5;
       unsigned int i;

       for(i=0;i<strlen(myString);i++){
           while ((*uartLSR & 0x20) == 0) {
               // Wait for Tx Holding Register Empty flag
           }
              *uartTX = myString[i];
       }
}

#define UART0_BASE ((unsigned char *) 0x100003e8)
#define UART1_BASE ((unsigned char *) 0x100003f8)

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

       printf ("Writing to UARTs - see log files\n\n");

       writeMess(UART0_BASE, "Hello UART0 world\n\n");
       writeMess(UART1_BASE, "Hello UART1 world\n\n");

    return 0;
}
```

We see that we write two messages - one to each of the UART base addresses.

Again it is compiled as before:

```
> make -C application
```

13.4 Running the simulation

To run the simulation, we again use the provided harness.exe program:

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

and get the following:

```
OVPsim started: Thu Nov 26 14:15:11 2015
Writing to UARTs - see log files
OVPsim finished: Thu Nov 26 14:15:11 2015
```

And if we look at the two log files written by the two UARTS, we get:

```
> cat uartTTY0.log

Hello UART0 world
```

```
> cat uartTTY1.log
```

```
Hello UART1 world
```

We have also provided the *example* script to create, compile and run this example.

Also don't forget that you can add commands to get more information from the simulator, for example:

```
> ./example.sh --modeldiags 0x3
...
Info (16550_BRS) harness/topmodule/subUart0/uart0: baud rate=1152000  parity=N  data
bits=5  total bits=7 character delay=6usec
Info (16550_BRS) harness/topmodule/subUart1/uart0: baud rate=1152000  parity=N  data
bits=5  total bits=7 character delay=6usec

Writing to UARTs - see log files

Info (16550_UWR) topmodule/subUart0/uart0: Write to Data register: data=0x48 ('H')
Info (16550_UWR) topmodule/subUart0/uart0: Write to Data register: data=0x65 ('e')
Info (16550_UWR) topmodule/subUart0/uart0: Write to Data register: data=0x6c ('l')
Info (16550_UWR) topmodule/subUart0/uart0: Write to Data register: data=0x6c ('l')
Info (16550_UWR) topmodule/subUart0/uart0: Write to Data register: data=0x6f ('o')
Info (16550_UWR) topmodule/subUart0/uart0: Write to Data register: data=0x20 (' ')
Info (16550_UWR) topmodule/subUart0/uart0: Write to Data register: data=0x55 ('U')
Info (16550_UWR) topmodule/subUart0/uart0: Write to Data register: data=0x41 ('A')
Info (16550_UWR) topmodule/subUart0/uart0: Write to Data register: data=0x52 ('R')
Info (16550_UWR) topmodule/subUart0/uart0: Write to Data register: data=0x54 ('T')
Info (16550_UWR) topmodule/subUart0/uart0: Write to Data register: data=0x30 ('0')
Info (16550_UWR) topmodule/subUart0/uart0: Write to Data register: data=0x20 (' ')
Info (16550_UWR) topmodule/subUart0/uart0: Write to Data register: data=0x77 ('w')
Info (16550_UWR) topmodule/subUart0/uart0: Write to Data register: data=0x6f ('o')
Info (16550_UWR) topmodule/subUart0/uart0: Write to Data register: data=0x72 ('r')
Info (16550_UWR) topmodule/subUart0/uart0: Write to Data register: data=0x6c ('l')
Info (16550_UWR) topmodule/subUart0/uart0: Write to Data register: data=0x64 ('d')
Info (16550_UWR) topmodule/subUart0/uart0: Write to Data register: data=0x0a (' ')
Info (16550_UWR) topmodule/subUart0/uart0: Write to Data register: data=0x0a (' ')
Info (16550_UWR) topmodule/subUart1/uart0: Write to Data register: data=0x48 ('H')
Info (16550_UWR) topmodule/subUart1/uart0: Write to Data register: data=0x65 ('e')
Info (16550_UWR) topmodule/subUart1/uart0: Write to Data register: data=0x6c ('l')
Info (16550_UWR) topmodule/subUart1/uart0: Write to Data register: data=0x6c ('l')
Info (16550_UWR) topmodule/subUart1/uart0: Write to Data register: data=0x6f ('o')
Info (16550_UWR) topmodule/subUart1/uart0: Write to Data register: data=0x20 (' ')
Info (16550_UWR) topmodule/subUart1/uart0: Write to Data register: data=0x55 ('U')
Info (16550_UWR) topmodule/subUart1/uart0: Write to Data register: data=0x41 ('A')
Info (16550_UWR) topmodule/subUart1/uart0: Write to Data register: data=0x52 ('R')
Info (16550_UWR) topmodule/subUart1/uart0: Write to Data register: data=0x54 ('T')
Info (16550_UWR) topmodule/subUart1/uart0: Write to Data register: data=0x31 ('1')
Info (16550_UWR) topmodule/subUart1/uart0: Write to Data register: data=0x20 (' ')
Info (16550_UWR) topmodule/subUart1/uart0: Write to Data register: data=0x77 ('w')
Info (16550_UWR) topmodule/subUart1/uart0: Write to Data register: data=0x6f ('o')
Info (16550_UWR) topmodule/subUart1/uart0: Write to Data register: data=0x72 ('r')
Info (16550_UWR) topmodule/subUart1/uart0: Write to Data register: data=0x6c ('l')
Info (16550_UWR) topmodule/subUart1/uart0: Write to Data register: data=0x64 ('d')
Info (16550_UWR) topmodule/subUart1/uart0: Write to Data register: data=0x0a (' ')
Info (16550_UWR) topmodule/subUart1/uart0: Write to Data register: data=0x0a (' ')
...
```

We can see the initialization state of the UART (written out due to the UART checking the value of the `modeldiags` flag), and then the writes to the peripheral registers.

14 Directory structure: VLNV or direct paths

In all the examples above we have used a very simple directory structure where the modules and application file all live in close proximity and the scripts and module instances just use path names. This is a direct path approach. This is good for quick and simple examples, but becomes hard work for large multi-user shared projects.

There is a different way.

Imperas has adopted the VLNV (Vendor Library Name Version) style of library structure as developed by the Spirit Consortium (now part of Accellera) and used with its IP-XACT XML libraries.

Imperas provides all its processor, peripheral, modules, platforms, and intercept libraries in a VLNV library structure. It also provides a complete suite of Makefiles that make use of this VLNV structure and can build the complete library automatically.

If you do not use a VLNV directory structure, then you will need to manage and maintain the building of your components, designs etc yourself.

If you use a VLNV library structure then you can make use of all the automation that we provide to manage your library.

One of the main differences when using an Imperas VLNV approach is that the VLNV structure uses two parallel directory trees - one for source, and one for the derived/created binary files.

For details of the Imperas / OVP VLNV library structure and its management, please see Imperas Installation and Getting Started document, Appendix G.

14.1 A hierarchical design using a VLNV directory structure

For this example we have taken the previous simpleHierarchy (which has two sub modules, each with a processor, and two applications) and moved locations around to make use of a VLNV structure.

```
> cp -r $IMPERAS_HOME/Examples/PlatformConstruction/simpleHierarchyVLNV .
> cd simpleHierarchyVLNV
> ls
application0 application1 source
```

Previously, we had the modules at this level. Now, we have moved the design down into a VLNV structure. Typically it is the design modules and components that go into the VLNV. Normally the applications and run scripts are kept outside but make use of VLNV based components.

We keep the source in a directory - we have called this '*./source'*.

The application directories are as before.

14.2 The directory structure

The modules are now below the source directory in the 'vendor' directory *ovpworld.org*:

```
source/ovpworld.org/module/simpleHierarchy_top/1.0/module/Makefile
source/ovpworld.org/module/simpleHierarchy_top/1.0/module/module.op.tcl
source/ovpworld.org/module/simpleHierarchy_sub/1.0/module/Makefile
source/ovpworld.org/module/simpleHierarchy_sub/1.0/module/module.op.tcl
```

Where we have the path made up to reflect the VLNV:

vendor	ovpworld.org
library	module
name	simpleHierarchy_*
version	1.0

You can work locally with this structure, and then when ready, you can copy it into a central library structure for your project. We work locally and then copy it into the *$IMPERAS_HOME/ImperasLib/source* VLNV library.

Note we have given the 'name' as top and sub, but also included the name of the project/design too - as many different projects/designs might be modules under this vendor. An alternative would be to have a different vendor for each project, e.g.:

source/simpleHierarchy.ovpworld.org/module/top/1.0
source/simpleHierarchy.ovpworld.org/module/sub/1.0

etc.

We have to make a few changes to switch from the previous use of direct path locations to using the VLNV.

14.3 Changing the controlling scripts

The *example.sh* has to change to compile the modules in the new location:

```
...
VLNVROOT=$(pwd)/vlnvroot
VLNVSRC=$(pwd)/source
mkdir -p ${VLNVROOT}
make -C ${VLNVSRC} VLNVSRC=${VLNVSRC} VLNVROOT=${VLNVROOT} \
       -f ${IMPERAS_HOME}/ImperasLib/buildutils/Makefile.library VERBOSE=0
...
```

We define two shell variables to be the source (`VLNVSRC`) and binary (`VLNVROOT`) trees, and create the binary directory (`VLNVROOT`) if it does not exist (`mkdir -p ${VLNVROOT}`).

With the variables set, we can then just call compile our library by using provided make system. Use `VERBOSE=1` to see the details of what is taking place.

We then use the *harness.exe* and it needs to use the *--modulevendor*, *--modulelibrary* etc. commands (as opposed to locating the *model.so* file directly with *--modulefile*:

```
harness.exe \
    --vlnvroot ${VLNVROOT} \
    --modulevendor ovpworld.org --modulelibrary module \
    --modulename simpleHierarchy_top --moduleversion 1.0 \
    --program simpleHierarchy_top/sub0/cpu1=application0/application0.OR1K.elf \
    --program simpleHierarchy_top/sub1/cpu1=application1/application1.OR1K.elf \
```

Note the use of the *--vlnvroot* ${VLNVROOT} command to add this location/name to the list of VLNV paths to look in to find the binary components.

14.4 The module instances

The top module also needs to be edited to make use of the VLNV structure.
Previously, in module.op.tcl, the sub module was instanced with the iGen command:

```
ihwaddmodule -instancename sub0 -modelfile submodule/model
```

Now, using VLNV it is instanced with:

```
ihwaddmodule -instancename sub0 \
       -vendor ovpworld.org -library module \
       -version 1.0 -type simpleHierarchy_sub
```

telling the simulator where to look for the module.

15 Loading programs into the design

To simulate application software, it must be loaded into platform. Instructions for loading the software can be given to the Imperas Simulator at run-time (recommended) using the command line argument *--program* or can be embedded in the platform. If the loading instructions are to be embedded in the platform, it may be loaded as part of the processor instantiation by adding *--imagefile* argument to *ihwaddprocessor* or separately using the *ihwaddimagefile* command.

By defining the program load as part of the platform definition, the program is always loaded. For example, this could be a boot loader that is used to initialize the system and then a further program is loaded to be executed.

The following shows the program *application.OR1K.elf* being loaded onto the processor.

The first two examples show the loading of a fixed executable defined in the module instantiation and so always part of the hardware definition.

This may be done on two ways, as part of the processor instantiation:

```
ihwaddprocessor –instancename cpu1 –imagefile application/application.OR1K.elf
```

or as a separate command adding the executable to the created processor instance handle, *cpu1*:

```
ihwaddimagefile –handle cpu1 –filename application/application.OR1K.elf
```

This final example show the program loaded using the command line. This keeps the hardware definition and program load separate

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

16 Loading symbols into the simulator

Imperas simulators require symbolic information for application programs to enable debug, context-sensitive trace and some forms of semihosting.

Normally, symbols can be read directly from program files as they are loaded. However, sometimes it is necessary to load a program's symbols separately from the program itself. This may be the case when the program has either been loaded from an elf file without symbolic information or in a form that does not contain symbolic information, or by some other means unknown to the simulator, for example using a peripheral to initialize memory.

Use *ihwaddsymbolfile* to do this (see the Imperas Simulation Guide for details) as part of the hardware creation or on the command line use *–symbolfile <elf file name>*.

The first example shows the loading of an elf file containing symbolic information onto a processor instance in the module definition, so it is always part of the hardware definition, whatever the program that is being executed. If addresses in the symbolfile do not match those in the program, then simulator features such as semihosting and symbolic disassembly will not work.

```
ihwaddsymbolfile –handle cpu1 –filename application/application.OR1K.elf
```

The following example shows the symbol file loaded using the command line. This keeps the hardware definition and symbolfile loading separate.

```
harness.exe \
       --modulefile topmodule/model.so \
       --symbolfile application/application.OR1K.elf
```

The above assumes that the program is loaded separately i.e. it must be present in the hardware to be executed. Using *--program* to load an elf file will also automatically load the symbols from the elf file, if present.

In the example above the symbolfile will be associated with all the processors in the hardware definition. To target the symbolfile at one particular processor use

```
    --symbolfile simpleHierarchy_top/sub0/cpu1=application/application.OR1K.elf
```

The full hierarchical pathname of the processor should be specified. The *--showoverrides* option can be useful to determine the exact hierarchical names of all the components in a design.

17 Setting Model Parameters

The **user-defined parameter** mechanism allows the platform to configure the functionality of a model. A named parameter applied to a module or model instance can be read by the model to change its behavior.

To set a parameter, iGen requires a *handle* which is a valid the instance name, a *name* which must correspond to a formal parameter on the model, a *type* which must match the type on the model and a *value* which must be consistent with the type. Refer to the model's documentation to find the parameters accepted by the model and use *ihwsetparameter* to set the value of a parameter on an instance.

Examples might be:

```
#an integer parameter
ihwsetparameter -handle cpu1 -name hiddenTLBentries -value 1 –type Uns32

#a string parameter
ihwsetparameter -handle cpu1 -name title -value 6800 –type string
```

This can also be performed on the command line using overrides, for example

```
harness.exe \
       --modulefile topmodule/model.so \
       --override myDesign/cpu1/title=6800
```

The simulator argument *--showoverrides* can be used to show what parameters can be modified.

18 Advanced Information & Usage of iGen

iGen takes a script making calls to the tcl iGen API and creates C files that make calls to the OP API.

The OP API is one of the public Open Virtual Platforms (OVP) APIs and is implemented in the Imperas / OVP simulators. The OP API has functions for creating the structure of test benches, platforms, modules and components - and it also has a very rich API for controlling the simulations.

This chapter provides an introduction to the C used to create modules and how the simulators use the different parts of modules. It describes the C output created by iGen.

18.1 Overview of detailed platform construction

18.1.1 Harnesses and Modules

A *module* is a model that creates and connects instances of processors, peripherals, RAMs, ROMs, caches (implemented using an MMC component), and other modules. It is linked with the libRuntimeLoader library to produce a *shared object*.

The entry point to a module is the symbol *modelAttrs* which refers to a predefined table of functions and constants in the shared object.

A *harness / test bench* is a special type of module that creates the top level of the system. It is linked with the libRuntimeLoader library to produce an *executable*.

The entry point to an executable top level module is the function *main.*

A module template produced by iGen only has the *modelAttrs*.

A *harness / test bench* can have both entry points; *main* and *modelAttrs*, and this too can be generated by iGen but this is a special case.

The standard Makefiles supplied by Imperas for module creation produce a shared object.

If you are creating a *harness / test bench* you will need to create an executable.

18.1.2 The contents of a module

A module is going to be used as a level of hierarchy or as a leaf at the bottom of the hierarchy, and it must contain the *modelAttrs* table which has

- a code to identify the shared object as a module (and not a processor, for instance)
- the version of the API
- a default name of the module
- pointers to functions in the module
- classification and status of the module to be interrogated by other tools.

The *modelAttrs* table is an instance of the *optModuleAttr* type defined in *ImpPublic/include/host/op/op.h*

A module defines some or all of the following functions:

18.1.2.1 Interface Iterators
Functions to return in sequence each interface object(bus, net, packetnet, FIFO).

18.1.2.2 Parameter Iterator
A function to return in sequence each parameter accepted by the module.

18.1.2.3 Constructor
(This function must be provided) A function to construct the contents of the module including creation of component instances, buses, nets etc. and connection to components.

18.1.2.4 Initialization (pre-simulation)
Called after all constructors in the design have been called and before simulation begins for the first time.

18.1.2.5 Simulation
Called each time simulation starts or restarts.

18.1.2.6 Reporting (post-simulation)
Called when simulation has finished, but before any destructors are called.

18.1.2.7 Destructor
Called when the simulation terminates to allow models to close files, free memory etc.

18.1.3 The Contents of a Harness or Test Bench
The entry point of the top level is (like any C program) is a function called *main*.

18.1.3.1 Command Line parser
The *main* function will usually include a command line parser to read the user's options for this simulation. The OP standard command line parser gives a consistent method of parsing standard data types, while also allowing the user to specify any of the rich set of standard options accepted by the simulator. Please refer to the OVP Control File User Guide.

18.1.3.2 The root Module
The *main* function must create a root module (using *opRootModuleNew*). This root module could then either:

- contain all the components in the system (this is how a legacy ICM platform is constructed as it has no hierarchy).

- create an instance of the top level of the design and supply any test–bench functionality required to run the design. In fact it could create instances of several designs and run then independently to perform *step and compare* testing.
- Reference the *modelAttrs* of this module to make this the top level of the design. Thus, if the compiled executable is used, main is called and this becomes the top level, or if the shared object is loaded, this becomes part of a larger system simulation.

18.1.4 Module Parameterization

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

18.1.4.1 The module interface

A module connects to its model instances using the following interface abstractions. Interface objects can be connected to components within the module or, via module ports, to model instances in other modules.

18.1.4.2 Bus

A bus is a high level model of a microprocessor bus system. It represents a distinct physical address space. It allows *bus masters* such as processors or DMA engines to read or write to *bus slaves* such are memories or memory-mapped registers. A bus cannot model contention, has no facility to model the time taken for each access, and consequently has no means to model burst modes, bus locking or priority schemes.

18.1.4.3 Net

A net is used to model a wire carrying a digital value, usually zero or one. In fact a net carries a 32-bit value so can carry more information but is usually used to model resets, interrupts, and mode controls. A net can have multiple drivers and receivers but does not model contention – the current value is that set by the most recent driver. All receivers are notified of a new value by a callback function in the model.

18.1.4.4 Packetnet

A packetnet is used to model packet-based protocols such as RS232, USB, Ethernet or GSM. A packetnet can have multiple drivers and receivers but does not model contention. A packet sent by a model is received instantaneously (time does not advance during its propagation) by all connected models in the order they were connected. Receiving models can modify the packet which can be examined by the sending model at the end of the transaction.

18.1.4.5 FIFO

A FIFO is a unidirectional point to point connection between two processor models. Words of a specified width (in bits) are pushed into one end of a FIFO and popped out of the other. The depth of a FIFO and the behaviour when pushing or popping would block

the FIFO is determined by the models connected and not by the platform. Primitives allow blocking or non-blocking push and pop operations.

18.1.5 Efficiency

Modules can be assembled with arbitrary depth of hierarchy, allowing the simulation of complex systems. However, the depth of hierarchy has no effect on simulation efficiency; module ports are removed before simulation.

18.2 Order of Platform construction

This section summarizes the operation of a hierarchical platform.

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

Higher-level modules are constructed before lower modules.

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

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

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

18.3 Editing the C of a module

The topic of editing the C of a module is covered fully in the Advanced Simulation Control of Platforms and Modules User Guide.

18.4 Writing out a testbench / harness

The main difference between a harness / testbench and a normal module is that the harness / testbench has the inclusion of a main function and often a command line parser and it is compiled and linked and used as an executable.

iGen can be used to create this harness / test bench with the *main* function contained within it. To do this, add *ihwaddclp* to the iGen script.

There is the restriction that if you are adding the *ihwaddclp* to your script, then you can not instance anything but modules in that harness / test bench.

If you require a more complex test harness, where you need to have more than just module instances in it, or where you need to control the simulator, and maybe single step it, or add monitors or other test related capabilities, then it will need to be written in C using the OP API so please refer to the Simulation Control of Platforms and Modules User Guide.

19 iGen Module related Error Messages

During usage of iGen when creating modules, you might get some of the following error message and the creation of the model will fail.

Error (CN_TLM) Module with command line parser should only instance modules
You have an iGen script that is creating a module that is instancing components other than modules and it include a call to add a Command Line Parser with *ihwaddclp*. Only the highest level of a platform can include the CLP and it must only instance modules. The solution is to add a higher top level module and only instance that in your testbench/harness that has the CLP. See the advanced section of this manual for more information.

Error (MDL_IFNF) Image file 'platform/sub/model' not found
When trying to load the models, the simulator has not managed to find the model.so/.dll model object specified. It is normally because you have specified it wrong in the *--modulefile* argument to harness.exe or in the *ihwaddmodule* in an iGen module creating script. (Note if the model name in the error message has a .so or .dll on the end then it is probably from your harness.exe call!)

Warning (CM_MF) flag 'modulefile' specified more than once (last value accepted).
Normally from the call to harness.exe where you are telling it to load a module, but you have specified more than one - currently the harness.exe program can only take one modulefile as an argument - normally you only have one (your top module), and you do not need to specify the sub modules, as the top module should be instancing them. If you do need to have two top modules, then either create another level of hierarchy, or write your own C test harness.

Error (OP_PNF) Parameter 'top/sub/logFileName' has no formal parameter defined in the model.
You have instanced a module and are trying to pass in a formal parameter using -ihwsetparameter, but the sub module has no been defined as requiring an argument - check the spelling.

20 (Deprecated) Creating ICM platforms with iGen

The *iGen* command line argument --*writec* causes *iGen* to write a C program. The program can be modified by hand if required, compiled with the host's C compiler, linked with the simulator run-time library and run as a stand-alone program.

```
shell> igen.exe \
    --batch   platform.icm.tcl \                  # the tcl input file
    --writec  platform.c \
    --icm
```

iGen will produce several output files with names taken from the argument to --writec.

Note that use of the ICM API has been deprecated in favor of the OP API.

The iGen flag –*op* makes iGen create a platform using the OP API.

The iGen flag -*icm* make iGen create a platform using the ICM API instead of OP.

The default for iGen is to assume an `-op` flag being set.

The rest of this chapter describes the use of iGen when a C platform using ICM is being created. The output files written when writing OP platforms/modules are different.

20.1 Generated files

If –*writec platform.c* is specified, the following files are generated:

File	Contains	User Edited?
platform.c	Function stubs	y
platform.constructor.igen.h	Platform construction	n
platform.options.igen.h	Variables set by CLP if requested	n
platform.handles.igen.h	Handles to platform objects	n
platform.clp.igen.h	Construction of the CLP if requested	n

20.1.1 User file : platform.c

The user file contains:

- main()
- user functions to be filled in if required.
- #include of the other files

Note that since it is expected that this file will be edited, igen will not overwrite this file if it exists. Use the –overwrite command line option to force overwriting.

20.1.2 Constructor file: platform.constructor.igen.h

The constructor file will generally not require modification. It includes the following features derived from the TCL script:

- Initialization
- Platform name
- Common simulation attributes

- Component instances
- Interconnect

- Loading of extension libraries
- Loading of application code

20.1.3 Options file: platform.options.igen.h

The options file defines variables which will be set by the command line parser (if requested).

20.1.4 Handles file: platform.handles.igen.h

The handles file defines handles for instances of processors, peripherals, nets, buses etc. Access to handles is required by test-harness code which could be written in the user file.

20.1.5 Command line parser file: platform.clp.igen.h

The CLP file constructs the command line parser (if requested) and configures it by adding command line parser arguments (CLPAs).

20.2 Repeated use of iGen

The user file will not be overwritten if it already exists, unless `-overwrite` is specified on the *iGen* command line. The other files will be overwritten without warning.

20.3 Adding a copyright header

To add your own copyright header to each C file, use either the igen command line option -userheader <path to your copyright text> or add the flag –userheader <path to your copyright text> to the `ihwnew` iGen command.

20.4 Checking the Platform

A C platform can be created without referring to the models that is uses, but if the models are available, igen can load the models and check them against the platform during creation. Use –checkmodels to do this.

```
shell> igen.exe \
```

```
    --batch        platform.tcl \
    --writec       platform.c   \
    --icm                       \
    --checkmodels
```

##