OVP Processor Modeling Guide

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

Author:	Imperas Software Limited
Version:	7.48.1
Filename:	OVP_Processor_Modeling_Guide.doc
Project:	OVP Processor Modeling Guide
Last Saved:	Thursday, 13 January 2022
Keywords:	

Copyright Notice

Table of Contents

1 Preface

This document describes how to create processor models for use with OVPsim and Imperas simulation tools.

1.1 Notation

`Code` Code extracts

1.2 Recommended Reading

Imperas simulation technology is based on just-in-time (JIT) compiler technology. The following book provides a good introduction to the concepts involved:

Virtual Machines, by James E. Smith, Ravi Nair
ISBN 1-55860-910-5
Publisher: Morgan Kaufmann/Elsevier

1.3 Related Imperas & OVP Documents

- VMI Morph Time Function Reference
- VMI Run Time Function Reference

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 new models of processors and other platform components using interfaces and libraries supplied by Imperas. Processor models developed using this technology can be used both with Imperas simulation products and the freely-available OVPsim platform simulator.

This document describes how to use the OVP interfaces to create new processor models.

The documentation here is supported by C code samples in the `Examples` directory of your Imperas installation, and also to download from the OVPWorld website (www.ovpworld.org). The compilation makes use of Makefiles, the instructions for which indicate the use of the command *make*. On Windows systems, the MinGW *mingw32-make* command should be used in its place.

2.1 Prerequisites

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

In very rare circumstances it is beneficial to implement some highly-performance-critical routines directly in assembler. You should ensure you are familiar with the x86 instruction set and assembler usage if required.

GCC Compiler Versions

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

3 Imperas Simulation Overview

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

3.1 Simulation Environments

There are two simulation environments that can be used with models that you create:

- *OVPsim* allows processor models created using OVP modeling technology to be used in C harness or platform files to create executables that execute binaries compiled for those processor models. It can also simulate behavioral components (the subject of this guide). OVPsim can also be used in 3rd party simulation environments (for example, SystemC). It can also be used to create a test harness to help validate processor models under construction, or even to create custom simulation environments. OVPsim has less functionality than the Imperas Professional Simulator Products in some areas and has restricted commercial usage as stipulated in the OVP click-through license agreement.
- *Imperas Professional Simulator Products* enhance the basic capabilities provided by OVPsim, particularly in the areas of debugger integration, tool integration and multiprocessor simulation support (including QuantumLeap parallel simulation). Contact Imperas for more information.

3.2 Processor Models

The core simulation components are *processor models*. In order to create a new processor model, you must implement the following major components by writing C code using the Imperas *Virtual Machine Interface* (VMI) API:

- An *instruction decoder*, capable of decoding a single processor instruction. This is a required component for the *disassembler*, *morpher* and *debugger interface*, described in section 5.
- An *instruction disassembler*, capable of generating a text representation of an instruction, described in section 6.
- An *instruction morpher*, capable of describing the behavior of a single instruction, described in sections 7 - 16.
- A *debugger and register interface*, which provides functions required for the model to be debugged using gdb or the Imperas multiprocessor debugger, described in section 17. This is also a prerequisite for advanced features such as register change tracing, save/restore and instruction attributes generation.
- A *programmer's view*, which allows details of model operation to be made available to tools such as debuggers in a structured way. This is described in section 18.
- If a processor implements virtual memory, then the hardware structures that support that virtual memory (MMU and TLB, for example) should also form part of the processor model. This is described in sections 21 - 23.

- If processors are to be used with the QuantumLeap parallel simulation algorithm of the Imperas Professional Simulation products, some changes may be required (for example, to identify atomic instructions). This is described in section 24.
- Most processor models will need to model members of a family of processors (family members are referred to as *variants*). To re-use your code as far as possible it is often convenient for one processor model to support multiple variants and configuration options that can be configured from the platform. Model configuration is covered in section 26.6.

Processor models are compiled into a shared object (.so or .dll) which is then dynamically loaded by the simulation environment.

3.3 Semihosting

Semihosting allows behavior that would normally occur on a simulated system to be implemented using features of the host system instead. As a simple example, a real platform might contain a UART peripheral to receive output. When simulating this system, it is generally more convenient not to simulate the UART at all but instead to intercept any `write` call that a processor makes and redirect the output to the simulator log instead. Such behavior is specified in a *semihosting library* for a processor.

Implementation of semihosting libraries is described in section 25.

3.4 Cache and Memory Subsystem Models

Memory subsystem models such as caches can be modeled as loadable shared objects (or dynamic linked libraries on Windows) and separately instantiated. This makes it very easy to explore hardware options: what happens to the performance of this application if I double the size of the L2 cache?

Memory subsystem models can be either *full* or *transparent*. A full model implements memory contents: for example a full cache model would implement both cache tags and the cache line contents. A transparent model implements some state but not the memory contents: for example, a transparent cache model would implement the cache tags but *not* the line contents, which is useful for performance analysis models that simply count hits and misses.

Implementation of memory subsystem models is beyond the scope of this document.

4 Introduction to Processor Modeling

4.1 Prerequisites

Before starting implementation of a new processor model, we recommend that you do the following:

1. Identify the particular processor variant to be modeled (when variants exist).
2. Obtain a processor tool chain and understand how to use it for the variant you will be modeling (if a tool chain is available). Typically, you will find it useful to have an assembler, linker, object dump utility and C compiler. If you are writing a model for a completely new processor then it is possible that no supporting tools may yet exist: in this case, you will need to become familiar with the object code format of the processor and possibly implement a custom object file loader as part of the modeling project.
3. Obtain a golden reference model if possible. Validating a processor model is much easier if there is a golden reference against which comparisons can be made.

4.2 Creating a Processor Outline Model

A minimal processor outline model is available in the directory:

```
$IMPERAS_HOME/Examples/Models/Processor/1.or1kOutline
```

This model is for the freely-available OR1K processor (see http://opencores.org/or1k/Main_Page). At this point, the model implements the bare minimum functionality to create a shared object usable by the Imperas simulation tools.

Take a copy of the outline model:

```
cp -r $IMPERAS_HOME/Examples/Models/Processor/1.or1kOutline .
```

Compile the model using the `make` utility:

```
cd 1.or1kOutline
make
```

Running make compiles the model in the current directory (using `Makefile`) and links it with an Imperas stub library (`vmiStubs.static.a`) to create a shared object loadable by the Imperas tools (`model.so/model.dll`).

> The model source code covered here and in following sections refers to Imperas header files in the directory:
>
> `$IMPERAS_HOME/ImpPublic/include/host/vmi`

These header files comprise the Imperas *Virtual Modeling Interface* (VMI) API.

The outline model files are described in the following subsections.

4.2.1 Declaring Shared Functions - `or1kFunctions.h`

File `or1kFunctions.h` declares prototypes of functions used throughout the processor model. Functions that must have particular prototypes for use with the Imperas tools should be defined using macros from file `vmiAttrs.h` within the VMI header directory; for example, every processor must have constructor, which is declared as:

```
VMI_CONSTRUCTOR_FN(or1kConstructor);
```

The macro `VMI_CONSTRUCTOR_FN` is defined in `vmiAttrs.h` as:

```
#define VMI_CONSTRUCTOR_FN(_NAME) void _NAME( \
    vmiProcessorP  processor,           \
    Bool           simulateExceptions,  \
    vmiSMPContextP smpContext,          \
    void          *parameterValues      \
)
```

In other words, function `or1kConstructor` is a void function which is passed an argument `processor` which is of type `vmiProcessorP`, a Boolean argument `simulateExceptions`, an argument `smpContext` of type `vmiSMPContextP` and an argument `parameterValues` of type `void*`. The `vmiProcessorP` type is an opaque type pointer representing the current processor state – we will see how this is used later in this section.

Always use the macros provided in the VMI header files to declare and define your functions: this protects any code you write from future changes to any of the Imperas function definitions.

4.2.2 Defining Processor Structure - `or1kStructure.h`

File `or1kStructure.h` defines a structure that will be used to hold the state of a single OR1K processor. Because this is a generic model, at this point the structure is empty except for a pointer describing the bus ports of the processor:

```
typedef struct or1kS {
    vmiBusPortP busPorts;   // bus port descriptions
} or1k, *or1kP;
```

4.2.3 Constructor, Destructor and Ports - `or1kMain.c`

File `or1kMain.c` implements two functions that must be present in every processor model: the *constructor* and *destructor*. The constructor function is called for each new instance of a processor. It should initialize the processor state (for example, by setting registers in the processor structure to a known state). The destructor is called at the end of simulation for each processor instance. It should perform any required processor-model-specific shutdown actions.

In this example, the constructor and destructor perform no action except to print that they have been called (using the message API defined in `vmiMessage.h`), and to allocate and free the model's *bus interface*:

```
VMI_CONSTRUCTOR_FN(or1kConstructor) {

    or1kP or1k = (or1kP)processor;

    vmiPrintf("%s called\n", FUNC_NAME);

    // create bus port specifications
    newBusPorts(or1k);
}

VMI_DESTRUCTOR_FN(or1kDestructor) {

    or1kP or1k = (or1kP)processor;

    vmiPrintf("%s called\n", FUNC_NAME);

    // free bus port specifications
    freeBusPorts(or1k);
}
```

> It is good practice to give each public model declaration a common, model specific prefix: this simplifies debugging the model in a simulation where several models of different types are in use. In this case, we have chosen the prefix `or1k`.

File `or1kMain.c` also implements a *bus port specification function* for the processor, which tells the simulator the number of bus ports that the OR1K has and their width. The OR1K, like many processors, has two bus ports. The first, called `INSTRUCTION` is a bus master port used to fetch instructions from memory. The second, called `DATA` is also a bus master port, used to read and write data to memory (on many systems, these two ports are connected to the same physical bus, so share the same address space). Objects representing these two ports are allocated and stored on the processor instance by function `newBusPorts`:

```
const static vmiBusPort busPorts[] = {
    {"INSTRUCTION", vmi_BP_MASTER, vmi_DOM_CODE, {32,32}, 1},
    {"DATA"       , vmi_BP_MASTER, vmi_DOM_DATA, {32,32}, 0},
};

static void newBusPorts(or1kP or1k) {

    Uns32 i;

    or1k->busPorts = STYPE_CALLOC_N(vmiBusPort, NUM_MEMBERS(busPorts));

    for(i=0; i<NUM_MEMBERS(busPorts); i++) {
        or1k->busPorts[i] = busPorts[i];
    }
}
```

The template structure busPorts describes the instruction and data bus ports. The vmiBusPort type is defined in vmiPorts.h as follows:

```
typedef enum vmiBusPortTypeE {
    vmi_BP_MASTER,
    vmi_BP_SLAVE,
    vmi_BP_MASTER_SLAVE
} vmiBusPortType;

typedef enum vmiDomainTypeE {
    vmi_DOM_CODE,      // code domain port
    vmi_DOM_DATA,      // data domain port
    vmi_DOM_OTHER      // other domain port
} vmiDomainType;

typedef struct vmiBusPortS {

    const char      *name;
    vmiBusPortType   type;
    vmiDomainType    domainType;
    struct           {Uns8 min; Uns8 max; Uns8 unset;} addrBits;
    Bool             mustBeConnected;

    // space for documentation
    const char      *description;
    void            *descriptionDom;

    // domain is non-NULL if port is connected
    memDomainP       domain;

} vmiBusPort;
```

The model fills the name, type, domainType, addrBits.min, addrBits.max and mustBeConnected fields; remaining fields are filled by the simulator as the model is instantiated. The fields have the following meanings:

1. name: the name of the port;
2. type: the port type (master or slave);
3. domainType: the port usage (code, data, or another purpose);
4. addrBits.min: the minimum width of a bus that can be connected;
5. addrBits.max: the maximum width of a bus that can be connected;
6. mustBeConnected: whether the port must be connected (if False, it may be left unconnected).

After the processor constructor has been called, the simulator obtains information about the model's bus ports by calling an iterator function here implemented by or1kGetBusPortSpec which returns a pointer to a vmiBusPort structure for each implemented port. Like most VMI iterators, it is called with zero to return the first object, with the previous object to return the next, and it returns zero when all objects have been iterated:

```
VMI_BUS_PORT_SPECS_FN(or1kGetBusPortSpec) {

    or1kP or1k = (or1kP)processor;

    if(!prev) {
```

```
        // first port
        return or1k->busPorts;

    } else {

        // port other than the first
        Uns32 prevIndex = (prev-or1k->busPorts);
        Uns32 thisIndex = prevIndex+1;

        return (thisIndex<NUM_MEMBERS(busPorts)) ? &or1k->busPorts[thisIndex]:0;
    }
}
```

When simulation ends, the destructor frees the allocated bus port list by calling function `freeBusPorts`:

```
static void freeBusPorts(or1kP or1k) {

    if(or1k->busPorts) {
        STYPE_FREE(or1k->busPorts);
        or1k->busPorts = 0;
    }
}
```

4.2.4 JIT Translations - `or1kMorph.c`

File `or1kMorph.c` implements the OR1K morpher function. The morpher function is responsible for defining how each processor instruction should be implemented; this is described in detail in section 7. The minimal example simply contains a call to function `vmimtExit`, which will cause the processor to terminate on the first instruction encountered.

4.2.5 Support Functions - `or1kUtils.c`

File `or1kUtils.c` implements two simulation support functions required in every model: the *processor endianness* function and the *next instruction* function.

4.2.5.1 The Endianness Function

This function must return the endianness of the processor when fetching code or performing a load or store. Currently supported options are `MEM_ENDIAN_BIG` and `MEM_ENDIAN_LITTLE`. This OR1K model is big endian, so the function is defined as:

```
VMI_ENDIAN_FN(or1kGetEndian) {
    return MEM_ENDIAN_BIG;
}
```

> Some models have endianness dependent upon the current processor state. For this reason, the endianness callback is passed the current processor as an argument so that its state can be accessed if required.
>
> This function can be called to request both the endianness of instruction fetches and the endianness of loads and stores: which is required is specified by the second

argument to the `VMI_ENDIAN_FN`, a Boolean called `isFetch`. This is `True` for an instruction fetch and `False` for a data access.

4.2.5.2 The Next Instruction Function

Given an instruction address, this function must return the *next* instruction address. This function is used by the simulator to step through the simulated code when generating (morphing) equivalent native code.

For processors with variable-length instructions (for example, x86 variants) the next instruction address function will be required to perform a full or partial instruction decode in order to determine the next instruction address. On RISC processors, the instruction size may be constant, so no decode is required. See section 5 for details of implementing an instruction decoder.

The minimal processor model assumes a constant instruction size of four bytes and is therefore implemented like this:

```
VMI_NEXT_PC_FN(or1kNextInstruction) {
    Uns32 nextAddress = (Uns32)(thisPC + 4);
    return nextAddress;
}
```

The next instruction function must correctly handle *instruction wraparound*. In the example above, it would be incorrect to implement the function as:

```
  return thisPC + 4;
```

(this would not wrap round as required after `0xfffffffc`, because the `Addr` type of `thisPC` is 64 bits, not 32).

4.2.6 Processor Information - `or1kInfo.c`

File `or1kInfo.c` implements the processor information function, which returns information about the model in several categories. Specifically, it defines a VLNV reference for the model (saying where it is located in a library), a VLNV reference for a default semihost library to use with the model, and information executable ELF codes and compatible debuggers:

```
#include "vmi/vmiAttrs.h"
#include "vmi/vmiModelInfo.h"

#include "or1kFunctions.h"

VMI_PROC_INFO_FN(or1kProcInfo) {
    static const vmiProcessorInfo info = {
        .vlnv.vendor      = "ovpworld.org"    ,
        .vlnv.library     = "processor"       ,
        .vlnv.name        = "or1k"            ,
        .vlnv.version     = "1.0"             ,

        .semihost.vendor  = "ovpworld.org"    ,
        .semihost.library = "semihosting"     ,
        .semihost.name    = "or1kNewlib"      ,
        .semihost.version = "1.0"             ,
```

```
        .elfCode           = 33906,
        .endianFixed       = True,
        .endian            = MEM_ENDIAN_BIG,
        .gdbPath           = "$IMPERAS_HOME/lib/$IMPERAS_ARCH/CrossCompiler/or32-
elf/bin/or32-elf-gdb" VMI_EXE_SUFFIX,
        .family            = "openCores",
    };
    return &info;
}
```

This example uses a static structure; in a more complex model the member values could depend on the current mode of the processor and therefore the structure might need to be dynamically allocated for a particular processor instance. The `vmiProcessorInfo` structure type is defined in header file `vmiModelInfo.h` like this:

```
typedef struct vmiProcessorInfoS {

    // Location of this model
    vmiVlnvInfo  vlnv;

    // semihost library used by default with this model.
    vmiVlnvInfo  semihost;

    // Path to the gdb used to debug this model.
    // Remember to use the VMI_EXE_SUFFIX so this works on Windows
    const char  *gdbPath;

    // Flags (if any) required by this GDB for this processor
    const char  *gdbFlags;

    // Startup commands (if any) sent to this GDB for this processor
    // Separate and terminate each command with newline (\n)
    // e.g. "command one\ncommand two\n"
    const char  *gdbInitCommands;

    // CPU Helper used by this model
    vmiVlnvInfo  helper;

    // Helper used by this model to replace GDB
    vmiVlnvInfo  debugHelper;

    // List of extension libraries to be unconditionally loaded
    // by this processor model.
    vmiVlnvInfoListCP mandatoryExtensions;

    // If endianFixed is true, this is it, otherwise ignore it.
    memEndian    endian;

    // This model supports one endian
    Bool         endianFixed;

    // By default, choose this model to execute a program with the following
    // elf code. Only one model in your library should have this set.
    Bool         defaultModel;

    // When reading executables for this model, the loader should use
    // physical addresses, not virtual.
    // (This is a legacy feature; unlikely to be required).
    Bool         loadPhysical;

    // True if this model can run in QuantumLeap mode
```

```
    Bool         QLQualified;

    // True if this model contains instances of different types
    Bool         AMP;

    // standard ELF code used by this processors
    Uns32        elfCode;

    // Alternative ELF codes used by this processor
    const Uns32 *alternativeElfCodes;

    // Deprecated field
    const char  *variant;

    // Processor family string
    const char  *family;

    // Processor group strings
    const char  *groupH;
    const char  *groupL;

    // If the gdb associated with this processor sets the lower address bits
    // to indicate the processor mode, use this field to clear the bits prior
    // to setting a breakpoint.
    //  =0 or 1 if no snap required
    //  =2 to snap to 2-byte short
    //  =4 to snap to 4-byte word
    //  =8 to snap to 8-byte long word

    Uns32        debugSnapAddress;

} vmiProcessorInfo;
```

Some of the important fields in this structure are explained below.

4.2.6.1 Model Location (`vlnv`)

The `vlnv` member specifies where the model will be stored, using the "**V**endor, **L**ibrary, **N**ame, **V**ersion" (VLNV) notation.

4.2.6.2 Default Semihost Library Location (`semihost`)

The `semihost` member specifies the location of the default semihost library (see Chapter 21) to be used with this model, using VLNV notation.

4.2.6.3 List Of Extension Libraries (mandatoryExtensions)

The `mandatoryExtensions` member specifies the start of a list of extension libraries used to enhance the instruction set of the model (see Chapter 26). The member can be `null` or pointer to a `vmiVlnvInfoList` structure which points to a `vmiVlnvInfo` structure and (optionally) another in the list. The `vmiVlnvInfo` should contain the VLNV reference of the library to be loaded.

4.2.6.4 ELF codes (`elfCode` and `alternativeElfCodes`)

The `elfCode` field specifies the primary ELF code that is expected for executables that can run on this processor. If non-NULL, field `alternativeElfCodes` specifies a zero-terminated list of other ELF code that are acceptable. These fields let the simulator check the compatibility of an application program before it is loaded for execution by this model.

4.2.6.5 Endian fields (`endianFixed` and `endian`)

Setting `endianFixed` to False indicates that the processor can be either endian. This is not to be confused with the endianness function defined with the `VMI_ENDIAN_FN` macro, which returns the *current endianness* of the processor. If `endianFixed` is True, the subsequent member `endian` specifies that fixed endianness (otherwise it is ignored).

4.2.6.6 gdbPath

This field specifies the path to the debugger to be used with the model. The `VMI_EXI_SUFFIX` macro can be used to conditionally add the `.exe` file suffix required for an executable on a Windows host.

4.2.6.7 gdbFlags

This field specifies any flags to be supplied on the debugger command line when it is invoked. If omitted (as in this case) then no special flags are supplied on the debugger command line.

4.2.6.8 gdbInitCommands

This field specifies any commands to be sent to the debugger after starting the executable, but before debugging begins. For example if the debugger supports several architectures, `set arch <specific_architecure>` can be used to choose one. If more than one command is required, separate them using the newline “\n” character. If omitted (as in this case), no special initialization commands are required.

4.2.6.9 helper

This field optionally specifies the VLNV reference of an intercept library that helps the VAP tools to understand the processor’s ABI.
Please refer to the Imperas_Binary_Intercept_Technology_User_Guide.

4.2.6.10 debugHelper

This field optionally specifies the VLNV reference of a library that helps the Imperas Multiprocessor debugger to understand the processor’s call stack, in the absence of a suitable gdb debugger.

4.2.6.11 QLQualified

This field specifies that the processor is able to run in parallel mode. See section 24.6.

4.2.6.12 Debugger snap address

Some processors do not use byte addressing; all instructions fall on 2-byte or 4-byte boundaries. A processor of this kind might use the least significant bits of its address to indicate special processor modes. If the gdb that is used with this processor requests breakpoints with the least significant bits set (also indicating the special processor modes), the simulator will not correctly detect the processor executing at a breakpoint address. The `debugSnapAddress` field can used to work around this problem; set this field to force breakpoint addresses from the debugger to the appropriate boundary:

debugSnapAddress	**meaning**
0 (default) or 1	Breakpoint lies on 8-bit boundary
2	Breakpoint lies on 16-bit boundary
4	Breakpoint lies on 32-bit boundary
8	Breakpoint lies on 16-bit boundary
(Other values are illegal)	

4.2.7 Function Registration - `or1kAttrs.c`

File `or1kAttrs.c` implements the *VMI instruction attributes* object for the OR1K processor. This is a C structure of type `vmiIASAttrs`, the type of which is defined in the VMI header file `vmiAttrs.h`. The structure encapsulates all required information about the processor in a form that is usable by the Imperas simulation products.

```
const vmiIASAttr modelAttrs = {

    ////////////////////////////////////////////////////////////////////////
    // VERSION & SIZE ATTRIBUTES
    ////////////////////////////////////////////////////////////////////////

    .versionString = VMI_VERSION,
    .modelType     = VMI_PROCESSOR_MODEL,
    .dictNames     = dictNames,
    .cpuSize       = sizeof(or1k),

    ////////////////////////////////////////////////////////////////////////
    // CREATE/DELETE ROUTINES
    ////////////////////////////////////////////////////////////////////////

    .constructorCB = or1kConstructor,
    .destructorCB  = or1kDestructor,

    ////////////////////////////////////////////////////////////////////////
    // MORPHER CORE ROUTINES
    ////////////////////////////////////////////////////////////////////////

    .morphCB = or1kMorphInstruction,

    ////////////////////////////////////////////////////////////////////////
    // SIMULATION SUPPORT ROUTINES
    ////////////////////////////////////////////////////////////////////////

    .getEndianCB = or1kGetEndian,
    .nextPCCB    = or1kNextInstruction,

    ////////////////////////////////////////////////////////////////////////
    // PORT ROUTINES
    ////////////////////////////////////////////////////////////////////////

    .busPortSpecsCB = or1kGetBusPortSpec,

    ////////////////////////////////////////////////////////////////////////
    // PROCESSOR INFO ROUTINE
    ////////////////////////////////////////////////////////////////////////
```

```
    .procInfoCB = or1kProcInfo,
};
```

Note that all fields in the structure are initialized *by name*. This is done so that source code changes are not required if new fields are added to the structure in future.

In the case of the minimal processor model, the structure contains:

1. A VMI version string, `VMI_VERSION` (defined in `vmiVersion.h`). This is used when the model is loaded by Imperas simulation products to ensure compatibility.
2. The type of model, `VMI_PROCESSOR_MODEL` (defined in `vmiTypes.h`). This is used by Imperas simulation products to ensure the correct kind of model is being loaded.
3. A list of *dictionary names* used by the model, `dictNames`. Dictionaries provide a mechanism to efficiently model *modal* processors and are discussed in chapter 14. Every processor must have at least one dictionary name, specified in a null-terminated array of constant strings. In the case of the minimal processor model, the dictionary names are specified as:
   `static const char *dictNames[] = {"NORMAL", 0};`
   so there is a single dictionary called *NORMAL* in this model.
4. An indication of the size required for the processor structure defined in `or1kStructure.h`, `sizeof(or1k)`.
5. References to all the callbacks required to implement the processor model. For this minimal model, there are references to seven functions – `or1kConstructor`, `or1kDestructor`, `or1kMorphInstruction`, `or1kGetEndian`, `or1kNextInstruction`, `or1kGetBusPortSpec` and `or1kProcInfo`.

4.3 Implementing a Test Platform using OVPsim

Section 4.2 described how an outline processor model was compiled using the `make` command. In order to validate and debug the outline model, it is useful to have a test harness to drive it. The easiest way to create a test harness is using OVPsim.

Within the `1.or1kOutline` directory, the subdirectory `platform` contains source files and a Makefile for the platform required in this case. There are in fact two forms of platform present:

1. File `platform/harness.c`

This file implements a test harness using the OP function API. This powerful API should be used for all future development. This test platform can be compiled to produce an executable, `harness.$IMPERAS_ARCH.exe`, by using this command in the `or1kOutline` directory:

```
make -C platform
```

2. File `platform/platform.c`

This file implements a test harness using the legacy ICM function API. This API is supported for legacy code only and should not be used for future development. This test platform can be compiled to produce an executable, `platform.$IMPERAS_ARCH.exe`, by using this command in the `or1kOutline` directory:

```
make -C platform BUILD_ICM_LEGACY=1
```

> Most examples described in this document have both an OP and legacy ICM harness associated with them, so that they may be compared when porting legacy code to the current OP interface. In this document, only the OP harnesses will be described.

File `harness.c` has a main function as follows:

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

    // initialize simulation session before calling any other OP functions
    opSessionInit(OP_VERSION);

    // check arguments
    if(!cmdParser(argc, argv)) {
       opMessage("E", "CLI", "Command Line parser error");
       return 1;
    }

    // create root module, enabling simulation interruption if Ctrl-C is pressed
    optModuleP mr = opRootModuleNew(
        0,
        MODULE_NAME,
        OP_PARAMS (
            OP_PARAM_BOOL_SET(OP_FP_STOPONCONTROLC, 1)
        )
    );

    // create a processor instance
    const char *modelFile = "model."IMPERAS_SHRSUF;
    optProcessorP processor = opProcessorNew(mr, modelFile, "cpu1", 0, 0);

    // create the processor bus
    optBusP bus = opBusNew(mr, "bus", 32, 0, 0);

    // connect processor instruction and data ports to the common bus
    opProcessorBusConnect(processor, bus, "INSTRUCTION");
    opProcessorBusConnect(processor, bus, "DATA");

    // create memory
    optMemoryP memory = opMemoryNew(mr, "local", OP_PRIV_RWX, 0xffffffff, 0, 0);

    // connect the memory onto the busses
    opMemoryBusConnect(memory, bus, "mp1", 0x00000000, 0xffffffff);

    // run processor, one instruction at a time
    while(simulate(processor, 1)) {
        // keep going while processor is still running
    }

    // terminate the simulation session
    opSessionTerminate();

    return 0;
}
```

This main function does the following:

1. It initializes the simulation interface by calling `opSessionInit`.
2. It creates a command line parser, allowing some standard arguments to be given on the command line (for example, the application executable to run, which can be defined by the `–program` argument).
3. It creates a new *root module instance* using function `opRootModuleNew`. All other components are specified to be children of this module.
4. It creates a single instance of the processor by calling `opProcessorNew`. The object file name of the processor shared object is specified as `model.so` or `model.dll` in the current directory (depending on whether simulating on a Linux or Windows host).
5. It creates a new bus using `opBusNew`, which is connected to both the instruction and data ports of the processor using `opProcessorBusConnect`.
6. It creates a memory using `opMemoryNew`, which is connected to the bus using `opMemoryConnect`.
7. It calls a routine `simulate` to simulate the processor one instruction at a time;
8. Finally, it calls `opSessionTerminate` to end the simulation.

The function `simulate` calls the `OP` routine `opProcessorSimulate` to simulate for a number of clocks, as follows:

```
static Bool simulate(optProcessorP processor, Uns64 clocks) {

    optStopReason stopReason = opProcessorSimulate(processor, clocks);

    switch(stopReason) {

       case OP_SR_SCHED:
           // hit the scheduler limit
           return True;

       case OP_SR_EXIT:
           // processor has exited
           return False;

       case OP_SR_FINISH:
           // simulation must end
           return False;

       default:
           opPrintf("unexpected stopReason %u\n", stopReason);
           return False;
    }
}
```

4.4 Creating an Application Test Case

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

Within the `or1kOutline` directory is a sample test case, `application/application.c`, which simply prints a message and exits. The application can be compiled by using this command in the `or1kOutline` directory:

```
make -C application
```

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

4.5 Running the Application Test Case with the Processor Model

Having compiled the outline processor model, test platform and application, you are now ready to run a simulation. Do this by running:

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

in the `1.or1kOutline` directory. You should see the following output:

```
or1kConstructor called
Warning (PC_NRI) No register information callback given for processor
'platform/cpu1'
or1kDestructor called
```

The output from the test case shows debug messages from the processor constructor and destructor. There is also a warning message about a missing callback in the model attributes structure (the *register information callback*, used to identify registers of particular interest to the simulator, such as the program counter); this can be ignored at this stage. As yet, the outline model has no functionality so it exits on execution of the first instruction. The steps you need to perform to make the model execute the application correctly are covered in the following chapters.

5 Implementing the Instruction Decoder

A key component of every processor model is the instruction decoder. The result of the decoder is used by several other model components, specifically:

1. The *morpher*, which generates equivalent native code for each simulated instruction.
2. The *disassembler*, which creates a text string representation of an instruction.
3. The *next address function*, which determines the address of the next instruction after a given address (for processors with variable-size instructions only).

5.1 The Template Decoder Model

A template model for the OR1K processor with a decoder can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/2.or1kDecoder
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/2.or1kDecoder .
```

Compile the model, harness and application using the `make` command:

```
cd 2.or1kDecoder
make
```

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is based on the previous outline model, with the changes listed below.

5.1.1 Defining Decoder Types - `or1kDecode.h`

File `or1kDecode.h` defines types used by the OR1K decoder and the decode function itself, as follows:

```
typedef enum or1kInstructionTypeE {

    // arithmetic instructions
    OR1K_IT_ADDI,
    OR1K_IT_ADDIC,
    OR1K_IT_ANDI,
    OR1K_IT_ORI,
    OR1K_IT_XORI,
    OR1K_IT_MULI,

    // KEEP LAST: for sizing the array
    OR1K_IT_LAST
```

```
} or1kInstructionType;
```

`or1kInstructionType` enumerates the instruction types that the decoder will find. The enumeration will grow to cover many instructions as the model develops. Currently, some simple binary operations are decoded.

```
typedef struct or1kInstructionInfoS {
    const char         *opcode;         // opcode name
    or1kInstructionType type;           // instruction type
    Uns32               thisPC;         // instruction address
    Uns32               instruction;    // raw instruction
    Uns32               r1;             // register 1
    Uns32               r2;             // register 2
    Uns32               c;              // constant value
} or1kInstructionInfo, *or1kInstructionInfoP;
```

`or1kInstructionInfo` is an intermediate structure that is filled by the decoder with information about the decoded instruction. At this stage, the following fields are present:

1. `opcode`: this string field is the instruction mnemonic;
2. `type`: this is a member of the `or1kInstructionType` enumeration, described above;
3. `thisPC`: this holds the address of the decoded instruction;
4. `r1` and `r2`: these are register indices extracted from the instruction;
5. `c`: this is a constant value extracted from the instruction.

```
void or1kDecode(or1kP or1k, Uns32 thisPC, or1kInstructionInfoP info);
```

`or1kDecode` is the decoder function entry point. It decodes the OR1K instruction at address `thisPC` and fills the passed `or1kInstructionInfo` structure with the results.

5.1.2 Decode Implementation - `or1kDecode.c`

File `or1kDecode.c` implements the OR1K instruction decoder using the decoder utility API defined in the VMI header file `vmi/vmiDecode.h`. The decoder API works as follows:

1. A new *decode table* is created using `vmidNewDecodeTable`.
2. A set of *decode entries* is added to the table using `vmidNewEntryFmtBin`. Each entry added identifies a single instruction type.
3. Instructions are decoded using a call to `vmidDecode`, passing the decode table and an instruction, which returns an identifier describing the instruction type found.

Refer to the *Imperas VMI Morph Time Function Reference* for more detailed information about the decoder API.

File `or1kDecode.c` first contains the following definitions:

```
#define WIDTH(_W, _ARG) ((_ARG) & ((1<<(_W))-1))

#define OP_D(_I)        WIDTH(5,(_I)>>21)
#define OP_A(_I)        WIDTH(5,(_I)>>16)
#define OP_U1(_I)       WIDTH(16,(_I)>>0)
#define OP_S1(_I)       ((Int16)OP_U1(_I))
```

These lines define C macros that extract fields from OR1K instructions, which have a fixed width of 32 bits. In this example, macros are defined only for some of the arithmetic and bitwise instructions of the OR1K: the list will be expanded in subsequent chapters.

Next, the file declares some structure types which are used to describe decoded instructions:

```
typedef enum regSpecE {
    R_NA,                       // no register
    R_D,                        // register at bits 25:21
    R_A,                        // register at bits 20:16
} regSpec;

//
// Define the location of constant in an instruction
//
typedef enum constSpecE {
    C_NA,                       // no register
    C_S1,                       // signed constant in bits 15:0
    C_U1,                       // unsigned constant in bits 15:0
} constSpec;

//
// Structure defining characteristics of each opcode type
//
typedef struct opAttrsS {
    const char *member;         // enumeration member name
    const char *opcode;         // opcode name
    const char *decode;         // decode string
    regSpec     r1      : 8;    // register 1 specification
    regSpec     r2      : 8;    // register 2 specification
    constSpec   c       : 8;    // constant value specification
} opAttrs, *opAttrsP;
```

A structure of this type describes each instruction recognized by the decoder. The particular bit pattern for the instruction is given by the `decode` member, described in detail below. The `r1`, `r2` and `c` members say how to extract register and constant descriptions from the instruction.

Next, a static decode table is defined for each recognized instruction, using macros that initialize members of the table:

```
#define ATTR_SET_ADDI(_NAME, _OPCODE, _DECODE) \
    [OR1K_IT_##_NAME] = {           \
        member : #_NAME,            \
        opcode : _OPCODE,           \
        decode : _DECODE,           \
        r1     : R_D,               \
        r2     : R_A,               \
        c      : C_S1               \
    }

#define ATTR_SET_ANDI(_NAME, _OPCODE, _DECODE) \
    [OR1K_IT_##_NAME] = {           \
        member : #_NAME,            \
        opcode : _OPCODE,           \
```

```
        decode : _DECODE,           \
        r1     : R_D,               \
        r2     : R_A,               \
        c      : C_U1               \
    }

const static opAttrs attrsArray[OR1K_IT_LAST+1] = {
    ATTR_SET_ADDI (ADDI,  "addi",  "|100111..........................|"),
    ATTR_SET_ADDI (ADDIC, "addic", "|101000..........................|"),
    ATTR_SET_ANDI (ANDI,  "andi",  "|101001..........................|"),
    ATTR_SET_ANDI (ORI,   "ori",   "|101010..........................|"),
    ATTR_SET_ADDI (XORI,  "xori",  "|101011..........................|"),
    ATTR_SET_ADDI (MULI,  "muli",  "|101100..........................|"),
};
```

The last argument to the `ATTR_SET_ADDI` and `ATTR_SET_ANDI` macros are patterns in suitable form for use by the VMI function `vmidNewEntryFmtBin`:

```
Bool vmidNewEntryFmtBin(
    vmidDecodeTableP table,
    const char      *name,
    Uns32            matchValue,
    const char      *format,
    Int32            priority
);
```

The arguments to this function are:

1. A *decode table* into which to add a new decode entry;
2. A *name* for the new entry;
3. A *value to return if the entry matches* (typically an enumeration member, in this example a member of the `or1kInstructionType` enumeration);
4. A *format string*, which specifies the bit pattern for a matching entry. Characters in this string may have the following meanings:
    a. **`0`**: the corresponding bit in the instruction must be 0;
    b. **`1`**: the corresponding bit in the instruction must be 1;
    c. **`|`** **`,`** **`/`** **<space>** **<tab>**: formatting character (ignored);
    d. **Any other character**: the corresponding bit can be either 1 or 0.
5. A *priority* for the entry. This allows instructions to be defined that are subsets of others. For example, a processor might have an instruction
    ```
    move r1, r2
    ```
    which actually decodes as
    ```
    ori r1, r2, 0
    ```
    (in other words, the `move` instruction is just a special case of the `ori` instruction). The above situation can be handled by adding two entries to the decode table, one for `ori` (with lower priority) and one for `move` (with higher priority).

As an example, the pattern `"|100111..........................|"` defined for the `addi` instruction specifies that the six most significant bits of an `ADDI` instruction are `'b100111`, and the remaining 26 bits can be any value (indicated by the '.' character in the format). The vertical bar characters are for formatting only and have no significance as part of the pattern.

The static array is used to create the decode table in function `createDecodeTable`:

```
static vmidDecodeTableP createDecodeTable(void) {

    vmidDecodeTableP    table = vmidNewDecodeTable(32, OR1K_IT_LAST);
    or1kInstructionType type;

    for(type=0; type<OR1K_IT_LAST; type++) {
        const opAttrs *attrs = &attrsArray[type];
        vmidNewEntryFmtBin(table, attrs->member, type, attrs->decode, 0);
    }

    return table;
}
```

The function creates a new decode table, specifying that the value `OR1K_IT_LAST` should be returned if there is no match for a particular instruction pattern. Then, it iterates over all members of the `attrsArray` table, creating a decode entry for each one. This function is called from `or1kDecode` (see below).

Next, there is a function that extracts information from an instruction word for a given `regSpec` value:

```
static Uns32 getReg(Uns32 instruction, regSpec rs) {

    Uns32 result = 0;

    switch(rs) {
        case R_NA:
            break;
        case R_D:
            result = OP_D(instruction);
            break;
        case R_A:
            result = OP_A(instruction);
            break;
        default:
            VMI_ABORT("unimplemented case"); // LCOV_EXCL_LINE
            break;
    }

    return result;
}
```

For example, a `regSpec` of `R_D` returns an index extracted from bits 25:21 of this instruction, i.e. the `rd` position defined by the OR1K instruction set. Then, there is a similar function for exacting constants from an instruction:

```
static Uns32 getConst(Uns32 instruction, constSpec cs) {

    Uns32 result = 0;

    switch(cs) {
        case C_NA:
            break;
        case C_S1:
            result = OP_S1(instruction);
            break;
```

```
        case C_U1:
            result = OP_U1(instruction);
            break;
        default:
            VMI_ABORT("unimplemented case"); // LCOV_EXCL_LINE
            break;
    }

    return result;
}
```

Note that the `constSpec` member `C_S1` returns the *sign-extended* value extracted from bits 15:0 of the instruction, and the `constSpec` member `C_U1` returns the *zero-extended* value from the same position.

Having the decoder be responsible for instruction field interpretation and sign/zero extensions means that downstream clients (the disassembler and JIT code morpher) are abstracted from details of instruction encoding which greatly improves modularity.

```
void or1kDecode(or1kP or1k, Uns32 thisPC, or1kInstructionInfoP info) {

    // get the instruction at the passed address - always 4 bytes on OR1K
    vmiProcessorP processor   = (vmiProcessorP)or1k;
    Uns32         instruction = vmicxtFetch4Byte(processor, thisPC);

    // get the OR1K decode table
    static vmidDecodeTableP decodeTable;
    if(!decodeTable) {
        decodeTable = createDecodeTable();
    }

    // decode the instruction to get the type and attributes
    or1kInstructionType type  = vmidDecode(decodeTable, instruction);
    const opAttrs      *attrs = &attrsArray[type];

    // fill structure fields
    info->opcode      = attrs->opcode;
    info->type        = type;
    info->thisPC      = thisPC;
    info->instruction = instruction;
    info->r1          = getReg(instruction, attrs->r1);
    info->r2          = getReg(instruction, attrs->r2);
    info->c           = getConst(instruction, attrs->c);
}
```

Finally, the routine `or1kDecode` implements the decoder entry point. It does the following:

1. It calls `vmicxtFetch4Byte` to get the four-byte instruction for the passed processor at the given address.
2. It calls `createDecodeTable` to create the OR1K decode table, if required.
3. It calls the VMI function `vmidDecode` to get the instruction type;
4. It fills the passed `or1kInstructionInfo` structure with data extracted from the instruction, given its type.

> In this example, the decode table is saved as a static variable, so it will be shared by all OR1K instances in a multiprocessor simulation. In more complex examples, where the table contents depend on model parameters, it could instead by saved as a field in the processor structure, so that each instance would have its own decode table.

5.1.3 JIT Translations - `or1kMorph.c`

This file implements the OR1K morpher function. The morpher function is responsible for defining how each processor instruction should be translated. This is described in detail in section 7; this example prepares the ground as follows:

```
typedef const struct or1kMorphAttrS *or1kMorphAttrCP;
typedef struct or1kMorphStateS *or1kMorphStateP;
```

These lines define pointers to a *morpher attributes* structure, `or1kMorphAttr`, and a *morpher state* structure, `or1kMorphState`, respectively. The morpher attributes structure gives information required to translate an instruction to native code. At this point, the only member of the structure is a morpher callback function:

```
#define OR1K_MORPH_FN(_NAME) void _NAME(or1kMorphStateP state)
typedef OR1K_MORPH_FN((*or1kMorphFn));

typedef struct or1kMorphAttrS {
    or1kMorphFn morphCB;    // function to translate one instruction
} or1kMorphAttr;
```

The morpher state structure is a scratchpad for useful information to provide to the morpher callback function. Currently, it is defined like this:

```
typedef struct or1kMorphStateS {
    or1kInstructionInfo info;        // instruction description (from decoder)
    or1kMorphAttrCP     attrs;       // instruction attributes
    or1kP               or1k;        // current processor
} or1kMorphState;
```

This file currently defines a single morpher callback function, which does nothing:

```
static OR1K_MORPH_FN(morphNOP) {
    // no action for a NOP
}
```

In this example, `morphNOP` is used for each of the arithmetic functions in `or1kDecode.h`. This implies that each arithmetic function is currently implemented as a NOP:

```
const or1kMorphAttr or1kMorphTable[OR1K_IT_LAST+1] = {

    // handle arithmetic instructions (second argument constant)
    [OR1K_IT_ADDI]  = {morphCB:morphNOP},
    [OR1K_IT_ADDIC] = {morphCB:morphNOP},
    [OR1K_IT_ANDI]  = {morphCB:morphNOP},
    [OR1K_IT_ORI]   = {morphCB:morphNOP},
    [OR1K_IT_XORI]  = {morphCB:morphNOP},
    [OR1K_IT_MULI]  = {morphCB:morphNOP},
};
```

This table associates function `morphNOP` with each of the instructions decoded by the decoder.

```
static void unimplemented(
    vmiProcessorP processor,
    Uns32         thisPC,
    Uns32         instruction
) {
    // report that unimplemented instruction was encountered
    vmiPrintf(
        "CPU '%s' 0x%08x:0x%08x *** undecoded instruction: exiting ***\n",
        vmirtProcessorName(processor),
        thisPC,
        instruction
    );

    // exit the CPU
    vmirtExit(processor);
}
```

Function `unimplemented` prints a message when an unimplemented instruction is encountered and halts the current processor by calling function `vmirtExit` (defined in `vmiRt.h`, the VMI *Run Time Function API*).

```
static OR1K_MORPH_FN(emitUnimplemented) {

    vmimtArgProcessor();
    vmimtArgUns32((Uns32)state->info.thisPC);
    vmimtArgUns32(state->info.instruction);
    vmimtCall((vmiCallFn)unimplemented);
}
```

Function `emitUnimplemented` is a dispatcher function that that is called for unimplemented instructions in `or1kDecode.c`. It creates native code to call the `undecoded` function previously defined. Code morphing is explained in detail in chapter 7.

```
VMI_MORPH_FN(or1kMorphInstruction) {

    or1kP          or1k  = (or1kP)processor;
    or1kMorphState state = {{0}};

    // decode instruction
    or1kDecode(or1k, thisPC, &state.info);

    // get morpher attributes for the decoded instruction and initialize other
    // state fields
    state.attrs = &or1kMorphTable[state.info.type];
    state.or1k  = or1k;

    if(state.attrs->morphCB) {
        // translate the instruction
        state.attrs->morphCB(&state);
    } else {
        // here if no translation callback specified
        emitUnimplemented(&state);
    }
}
```

Function `or1kMorphInstruction` is the morpher entry point (referenced in the `vmiIASAttrs` structure for this processor model, in `or1kAttrs.h`). It defines a local `or1kMorphState` structure, initialized to zero. It then calls the decoder interface to decode an instruction given the current PC, filling the `info` substructure. It then fills the supplementary `attrs` and `or1k` fields of the morpher state structure, based on the instruction type returned by the decoder. Finally, it calls the appropriate morpher callback for the instruction, or the unimplemented instruction callback if the decode failed.

> Note that the size of the `or1kMorphTable` array was defined to be `OR1K_IT_LAST+1`. This means that the table contains a final (all zero) entry that is found if instruction decode fails (returning type `OR1K_IT_LAST`). This entry has no defined `morphCB`, ensuring that the unimplemented instruction callback will be called for undecoded instructions.

5.1.4 Instruction Disassembler - `or1kDisassemble.c`

This file implements the OR1K disassembler function. The disassembler function is responsible for generating a string disassembly of a given instruction. This is described in detail in section 6; this example does the following:

```
static const char *disassembleInfo(
    or1kP                or1k,
    or1kInstructionInfoP info,
    vmiDisassAttrs       attrs
) {
    // static buffer to hold result
    static char result[256];

    // default disassembly just shows instruction pattern
    sprintf(result, "??? instruction:0x%08x", info->instruction);

    // return the result
    return result;
}

VMI_DISASSEMBLE_FN(or1kDisassemble) {

    // static buffer to hold disassembly result
    or1kP               or1k = (or1kP)processor;
    or1kInstructionInfo info;

    // decode instruction
    or1kDecode(or1k, thisPC, &info);

    // return disassembled instruction
    return disassembleInfo(or1k, &info, attrs);
}
```

Function `or1kDisassemble` disassembles one instruction. It calls the decoder and then utility function `disassembleInfo`, which fills a static string with the required disassembly, which is then returned. In this example, the disassembler simply echoes the instruction pattern to the disassembly string; later stages of the model implement a true disassembler.

Note that the disassembler is never called asynchronously or in a re-entrant manner by the simulator, and there is no requirement for the disassembly string to persist between calls. This means that it is acceptable to use a static array to hold the result, as above.

The prototype for `or1kDisassemble` is in file `or1kFunctions.h`, and is referenced in the `vmiIASAttr` structure defined in `or1kAttrs.c`. The reason for this will be seen when an application example is run using the new model.

5.2 Running the Application Test Case with the Processor Model

The platform is identical to the previous example. When you have compiled all components of the test, run:

```
platform/harness.$IMPERAS_ARCH.exe –-trace \
    --program application/application.OR1K.elf
```

in the `2.or1kDecoder` directory. The standard `–-trace` parameter enables a trace mode where the model disassembly routine is called just before each instruction is executed, which is why the disassembler routine was added to the model `vmiIASAttr` structure You should see the following output:

```
or1kConstructor called
Warning (PC_NRI) No register information callback given for processor
'platform/cpu1'
Info 'cpu1', 0x0000000000000100: ??? instruction:0x9c400000
Info 'cpu1', 0x0000000000000104: ??? instruction:0x9c600000
Info 'cpu1', 0x0000000000000108: ??? instruction:0x9c800000
Info 'cpu1', 0x000000000000010c: ??? instruction:0x9ca00000
Info 'cpu1', 0x0000000000000110: ??? instruction:0x9cc00000
Info 'cpu1', 0x0000000000000114: ??? instruction:0x9ce00000
Info 'cpu1', 0x0000000000000118: ??? instruction:0x9d000000
Info 'cpu1', 0x000000000000011c: ??? instruction:0x9d200000
Info 'cpu1', 0x0000000000000120: ??? instruction:0x9d400000
Info 'cpu1', 0x0000000000000124: ??? instruction:0x9d600000
Info 'cpu1', 0x0000000000000128: ??? instruction:0x9d800000
Info 'cpu1', 0x000000000000012c: ??? instruction:0x9da00000
Info 'cpu1', 0x0000000000000130: ??? instruction:0x9dc00000
Info 'cpu1', 0x0000000000000134: ??? instruction:0x9de00000
Info 'cpu1', 0x0000000000000138: ??? instruction:0x9e000000
Info 'cpu1', 0x000000000000013c: ??? instruction:0x9e200000
Info 'cpu1', 0x0000000000000140: ??? instruction:0x9e400000
Info 'cpu1', 0x0000000000000144: ??? instruction:0x9e600000
Info 'cpu1', 0x0000000000000148: ??? instruction:0x9e800000
Info 'cpu1', 0x000000000000014c: ??? instruction:0x9ea00000
Info 'cpu1', 0x0000000000000150: ??? instruction:0x9ec00000
Info 'cpu1', 0x0000000000000154: ??? instruction:0x9ee00000
Info 'cpu1', 0x0000000000000158: ??? instruction:0x9f000000
Info 'cpu1', 0x000000000000015c: ??? instruction:0x9f200000
Info 'cpu1', 0x0000000000000160: ??? instruction:0x9f400000
Info 'cpu1', 0x0000000000000164: ??? instruction:0x9f600000
Info 'cpu1', 0x0000000000000168: ??? instruction:0x9f800000
Info 'cpu1', 0x000000000000016c: ??? instruction:0x9fa00000
```

```
Info 'cpu1', 0x0000000000000170: ??? instruction:0x9fc00000
Info 'cpu1', 0x0000000000000174: ??? instruction:0x9fe00000
Info 'cpu1', 0x0000000000000178: ??? instruction:0x1820ffff
CPU 'cpu1' 0x00000178:0x1820ffff *** undecoded instruction: exiting ***
or1kDestructor called
```

After the constructor line, there is a line of trace output for every instruction that was successfully decoded by the decoder in this example. Each trace line gives the instruction address (starting with `0x100`, the start address specified in the ELF file) and the instruction disassembly, produced using the disassembler we defined. At address `0x178`, the processor encounters the first instruction not recognized by the decoder and is halted. In the next chapter, we will see how to terminate simulation more elegantly than this.

5.3 More Complex Decoders

The decoder in this OR1K example is quite simple because the OR1K instruction set has a small number of similar instructions of constant size (32 bits). CISC processors with variable-length instructions require a more sophisticated decoder. A good approach is to have multiple decode tables (a level 1 table is used to decode the first byte of the instruction, on the basis of this alternate level 2 tables are used, and so on). When decoders are complex, it is often useful to fill a data structure with information about the decoded instruction to use in later stages (instruction translation and disassembly); see the MIPS processor models on the ovpworld.com website for good examples.

All instruction fetches performed by a decoder should use `vmicxtFetch4Byte`, or related routines defined in `vmiCxt.h`. A single decode may perform several calls to fetch routines if required: in the example of the CISC processor, there may be an initial call to `vmicxtFetch1Byte` to fetch the first byte of an instruction, then a possible further call to `vmicxtFetch1Byte` to fetch the next byte, and so on.

6 Implementing the Instruction Disassembler

Having implemented an initial decoder framework, the next step is to start implementing the details of the instruction disassembler.

6.1 The Template Disassembler Model

A template model for the OR1K processor with a decoder and disassembler can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/3.or1kDisassembler
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/3.or1kDisassembler .
```

Compile the model, harness and application using the `make` command:

```
cd 3.or1kDisassembler
make
```

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is based on the previous decoder model, with the changes listed below.

6.1.1 Instruction Formats - `or1kDisassembleFormats.h`

The disassembler we are about to write is implemented by a state machine that consumes a *format string* which describes how to construct disassembled output arguments. The format strings are defined in file `or1kDisassembleFormats.h` as follows:

```
// These are placeholders in disassembly decoder
#define EMIT_R1             '\001'
#define EMIT_R2             '\002'
#define EMIT_XIMM           '\003'

// These are placeholders in disassembly format strings
#define EMIT_R1_S           "\001"
#define EMIT_R2_S           "\002"
#define EMIT_XIMM_S         "\003"

// These are disassembly format strings
#define FMT_R1_R2_XIMM      EMIT_R1_S "," EMIT_R2_S "," EMIT_XIMM_S
```

The format string uses primitive tokens, `EMIT_R1`, `EMIT_R2` and `EMIT_XIMM`, with the following meanings:

`EMIT_R1`: emit description of GPR in `r1` position
`EMIT_R2`: emit description of GPR in `r2` position
`EMIT_XIMM`: emit description of constant `c` in hexadecimal format.

Primitive tokens are specified using non-printing characters (\001, \002, \003 etc). A full format string is a concatenation of these tokens with other printable characters. For example, the format string:

```
EMIT_R1_S "," EMIT_R2_S "," EMIT_XIMM_S
```

Specifies that the disassembly arguments should be of the form *r1,r2,0xnnnnnnnn*, where commas are echoed literally.

6.1.2 Supporting Instruction Formats - `or1kDecode.h`

The `or1kInstructionInfoS` type now contains a new `format` field, which is a disassembly format string as described above:

```
typedef struct or1kInstructionInfoS {
    const char         *opcode;         // opcode name
    const char         *format;         // disassembly format string
    or1kInstructionType type;           // instruction type
    Uns32               thisPC;         // instruction address
    Uns32               instruction;    // raw instruction
    Uns32               r1;             // register 1
    Uns32               r2;             // register 2
    Uns32               c;              // constant value
} or1kInstructionInfo, *or1kInstructionInfoP;
```

6.1.3 Adding Instruction Formats - `or1kDecode.c`

The `opAttr` structure also contains a new `format` field:

```
typedef struct opAttrsS {
    const char *member;         // enumeration member name
    const char *opcode;         // opcode name
    const char *format;         // format string
    const char *decode;         // decode string
    regSpec     r1      : 8;    // register 1 specification
    regSpec     r2      : 8;    // register 2 specification
    constSpec   c       : 8;    // constant value specification
} opAttrs, *opAttrsP;
```

The `ATTR_SET_ADDI` and `ATTR_SET_ANDI` macros have been modified to initialize the new format entry using the **`FMT_R1_R2_XIMM`** value specified in `or1kDisassembleFormats.h`. For example:

```
#define ATTR_SET_ADDI(_NAME, _OPCODE, _DECODE) \
    [OR1K_IT_##_NAME] = {           \
        member : #_NAME,            \
        opcode : _OPCODE,           \
        format : FMT_R1_R2_XIMM,    \
        decode : _DECODE,           \
        r1     : R_D,               \
        r2     : R_A,               \
        c      : C_S1               \
```

```
    }
```

Function `or1kDecode` copies the new format value from the `opAttrs` to the `or1kInstructionInfo` structure:

```
void or1kDecode(or1kP or1k, Uns32 thisPC, or1kInstructionInfoP info) {

    . . . lines omitted . . .

    // fill structure fields
    info->opcode      = attrs->opcode;
    info->format      = attrs->format;
    info->type        = type;
    info->thisPC      = thisPC;
    info->instruction = instruction;
    info->r1          = getReg(instruction, attrs->r1);
    info->r2          = getReg(instruction, attrs->r2);
    info->c           = getConst(instruction, attrs->c);
}
```

6.1.4 Using Instruction Formats - `or1kDisassemble.c`

`or1kDisassemble.c` now implements a state machine disassembler, driven from the decoded instruction. The file first contains utility functions that append a single character and a string to a working buffer:

```
//
// Append the character to to the result
//
static void putChar(char **result, char ch) {

    // get the tail pointer
    char *tail = *result;

    // do the append
    *tail++ = ch;

    // add null terminator
    *tail = 0;

    // update the tail pointer
    *result = tail;
}

//
// Append the string to to the result
//
static void putString(char **result, const char *string) {

    // get the tail pointer
    char *tail = *result;
    char  ch;

    // do the append
    while((ch=*string++)) {
        *tail++ = ch;
    }

    // add null terminator
    *tail = 0;
```

```
    // update the tail pointer
    *result = tail;
}
```

Then there are two functions that will append a number in unsigned and hexadecimal format, respectively:

```
static void putU(char **result, Uns32 value) {

    char tmp[32];

    sprintf(tmp, "%u", value);

    putString(result, tmp);
}

static void putX(char **result, Uns32 value) {

    char tmp[32];

    sprintf(tmp, "0x%x", value);

    putString(result, tmp);
}
```

And also a function to write a GPR name, derived from an index:

```
static void putRegister(char **result, Uns32 r) {
    putChar(result, 'r');
    putU(result, r);
}
```

> Note that this example uses fixed-width types (`Uns32` etc). These types are defined in the include file:
>
> `$IMPERAS_HOME/ImpPublic/include/host/impTypes.h`.

Function `disassembleInfo` has been modified to call a new function, `disassembleFormat`[1]:

```
static const char *disassembleInfo(
    or1kP                or1k,
    or1kInstructionInfoP info,
    vmiDisassAttrs       attrs
) {
    // static buffer to hold result
    static char result[256];
    const char *format = info->format;
    char       *tail   = result;

    // disassemble using the format for the type
```

[1] This disassembler supports *uncooked disassembly*. The meaning and purpose of this are discussed at the end of this chapter. For the moment, assume that any reference to Boolean `uncooked` is `False`.

```
    if(format) {
        disassembleFormat(or1k, info, &tail, format, attrs==DSA_UNCOOKED);
    } else {
        sprintf(result, "??? instruction:0x%08x", info->instruction);
    }

    // return the result
    return result;
}
```

Function `disassembleFormat` operates in two parts. Firstly, the opcode is printed, using the `putString` utility function shown earlier:

```
static void disassembleFormat(
    or1kP                or1k,
    or1kInstructionInfoP info,
    char               **result,
    const char          *format,
    Bool                 uncooked
) {
    char *argStart = (*result)+9;
    char  ch;

    // emit opcode
    putString(result, "l.");
    putString(result, info->opcode);
```

Each opcode has a constant prefix `l.` followed by an opcode string extracted from the decode structure. If the instruction has arguments, the opcode is padded to 9 characters for alignment:

```
    if(*format) {

        // pad opcode to start of arguments
        if(!uncooked) {

            putChar(result, ' ');

            while(*result!=argStart) {
                putChar(result, ' ');
            }
        }
```

Secondly, there is a loop that consumes the format string:

```
        while((ch=*format++)) {

            switch(ch) {

                case EMIT_R1:
                    putUncookedKey(result, " R1", uncooked);
                    putRegister(result, info->r1);
                    break;

                case EMIT_R2:
                    putUncookedKey(result, " R2", uncooked);
                    putRegister(result, info->r2);
                    break;
```

```
                case EMIT_XIMM:
                    putUncookedKey(result, " CX", uncooked);
                    putX(result, info->c);
                    break;

                default:
                    if(!uncooked) {putChar(result, ch);}
                    break;
            }
        }
```

If the loop encounters one of the special tokens, it is handled appropriately. For example, token `EMIT_R1` causes the register name passed in the `info->r1` field to be emitted. Otherwise, if a non-token is encountered, it is emitted literally.

6.2 Running the Application Test Case with the Processor Model

Run:

```
platform/harness.$IMPERAS_ARCH.exe –-trace \
    --program application/application.OR1K.elf
```

in the `3.or1kDisassembler` directory. You should see the following output:

```
or1kConstructor called
Warning (PC_NRI) No register information callback given for processor ‘cpu1’
Info 'cpu1', 0x0000000000000100: l.addi   r2,r0,0x0
Info 'cpu1', 0x0000000000000104: l.addi   r3,r0,0x0
Info 'cpu1', 0x0000000000000108: l.addi   r4,r0,0x0
Info 'cpu1', 0x000000000000010c: l.addi   r5,r0,0x0
Info 'cpu1', 0x0000000000000110: l.addi   r6,r0,0x0
Info 'cpu1', 0x0000000000000114: l.addi   r7,r0,0x0
Info 'cpu1', 0x0000000000000118: l.addi   r8,r0,0x0
Info 'cpu1', 0x000000000000011c: l.addi   r9,r0,0x0
Info 'cpu1', 0x0000000000000120: l.addi   r10,r0,0x0
Info 'cpu1', 0x0000000000000124: l.addi   r11,r0,0x0
Info 'cpu1', 0x0000000000000128: l.addi   r12,r0,0x0
Info 'cpu1', 0x000000000000012c: l.addi   r13,r0,0x0
Info 'cpu1', 0x0000000000000130: l.addi   r14,r0,0x0
Info 'cpu1', 0x0000000000000134: l.addi   r15,r0,0x0
Info 'cpu1', 0x0000000000000138: l.addi   r16,r0,0x0
Info 'cpu1', 0x000000000000013c: l.addi   r17,r0,0x0
Info 'cpu1', 0x0000000000000140: l.addi   r18,r0,0x0
Info 'cpu1', 0x0000000000000144: l.addi   r19,r0,0x0
Info 'cpu1', 0x0000000000000148: l.addi   r20,r0,0x0
Info 'cpu1', 0x000000000000014c: l.addi   r21,r0,0x0
Info 'cpu1', 0x0000000000000150: l.addi   r22,r0,0x0
Info 'cpu1', 0x0000000000000154: l.addi   r23,r0,0x0
Info 'cpu1', 0x0000000000000158: l.addi   r24,r0,0x0
Info 'cpu1', 0x000000000000015c: l.addi   r25,r0,0x0
Info 'cpu1', 0x0000000000000160: l.addi   r26,r0,0x0
Info 'cpu1', 0x0000000000000164: l.addi   r27,r0,0x0
Info 'cpu1', 0x0000000000000168: l.addi   r28,r0,0x0
Info 'cpu1', 0x000000000000016c: l.addi   r29,r0,0x0
Info 'cpu1', 0x0000000000000170: l.addi   r30,r0,0x0
Info 'cpu1', 0x0000000000000174: l.addi   r31,r0,0x0
Info 'cpu1', 0x0000000000000178: ??? instruction:0x1820ffff
CPU 'cpu1' 0x00000178:0x1820ffff *** undecoded instruction: exiting ***
```

This reveals that the first instructions executed in the application are OR1K `addi` instructions, which clear the processor GPRs.

6.3 Creating Disassembler Point Tests

When a class of instructions has been added to a decoder (such as the arithmetic instructions above) it is good practice to fully test the disassembly behavior of the entire class before implementing any behavior for that instruction class. This is done most easily by assembler-level tests. File `asmtest.S` in directory `3.or1kDisassembler/application` is an OR1K assembler file that is a good starting point for a disassembler test:

```
.global _start
_start:      l.addi       r1,r2,0
             l.addi       r1,r2,1
             l.addi       r1,r2,-1
             l.addic      r1,r2,1
             l.addic      r1,r2,-1
             l.addic      r1,r2,0
             l.andi       r1,r2,1
             l.andi       r1,r2,-1
             l.andi       r1,r2,0
             l.ori        r1,r2,1
             l.ori        r1,r2,-1
             l.ori        r1,r2,0
             l.xori       r1,r2,1
             l.xori       r1,r2,-1
             l.xori       r1,r2,0
             l.muli       r1,r2,1
             l.muli       r1,r2,-1
             l.muli       r1,r2,0
.global exit
exit:
             l.add        r1,r2,0
```

Assemble this file using:

```
cd application
make asmtest.OR1K.elf
cd ..
```

Run the platform using the assembler executable file:

```
platform/harness.$IMPERAS_ARCH.exe --trace \
    --program application/asmtest.OR1K.elf
```

The output from this should be as follows:

```
or1kConstructor called
Warning (PC_NRI) No register information callback given for processor ‘cpu1’
Info 'cpu1', 0x0000000001000074: l.addi   r1,r2,0x0
Info 'cpu1', 0x0000000001000078: l.addi   r1,r2,0x1
Info 'cpu1', 0x000000000100007c: l.addi   r1,r2,0xffffffff
Info 'cpu1', 0x0000000001000080: l.addic  r1,r2,0x1
Info 'cpu1', 0x0000000001000084: l.addic  r1,r2,0xffffffff
```

```
Info 'cpu1', 0x0000000001000088: l.addic  r1,r2,0x0
Info 'cpu1', 0x000000000100008c: l.andi   r1,r2,0x1
Info 'cpu1', 0x0000000001000090: l.andi   r1,r2,0xffff
Info 'cpu1', 0x0000000001000094: l.andi   r1,r2,0x0
Info 'cpu1', 0x0000000001000098: l.ori    r1,r2,0x1
Info 'cpu1', 0x000000000100009c: l.ori    r1,r2,0xffff
Info 'cpu1', 0x00000000010000a0: l.ori    r1,r2,0x0
Info 'cpu1', 0x00000000010000a4: l.xori   r1,r2,0x1
Info 'cpu1', 0x00000000010000a8: l.xori   r1,r2,0xffffffff
Info 'cpu1', 0x00000000010000ac: l.xori   r1,r2,0x0
Info 'cpu1', 0x00000000010000b0: l.muli   r1,r2,0x1
Info 'cpu1', 0x00000000010000b4: l.muli   r1,r2,0xffffffff
Info 'cpu1', 0x00000000010000b8: l.muli   r1,r2,0x0
Info 'cpu1', 0x00000000010000bc: l.addi   r1,r2,0x0
Processor ‘cpu1’ terminated at ‘exit’, address 0x10000bc
or1kDestructor called
```

It is good practice to make the output from the disassembler conform as closely as possible to the output generated by existing tools (for example, the OR1K `objdump` executable). This simplifies verification because output generated by the disassembler can be automatically compared against a golden log generated by the existing tool.

6.3.1 Elegant Test Termination using Semihosting

Note that the assembler test terminated more elegantly than the previous run: instead of:

```
CPU 'cpu1' 0x00000178:0x0400037b *** undecoded instruction: exiting ***
```

We saw:

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

This was possible because the test platform used with this example was modified to use *semihosting*, which will be briefly introduced here and covered in detail in chapter 25.

Imperas semihosting allows the default behavior of specified functions or instructions to be modified using a semihosting shared object library that is loaded by the simulator in addition to the processor model. In this case, we defined a global label, `exit`, on the last instruction of the assembler test. This label can be used in conjunction with a standard Imperas semihosting shared object library, located at the following location under `$IMPERAS_HOME`:

```
$IMPERAS_VLNV/ovpworld.org/modelSupport/imperasExit/1.0/model.$SHRSUF
```

NOTE; $IMPERAS_VLNV is equivalent to $IMPERAS_HOME/lib/$IMPERAS_ARCH/ImperasLib

What this semihosting library does is terminate simulation immediately after any instruction labeled `exit`. To use the semihosting library, `platform/harness.c` has been modified as follows to select the `imperasExit` semihost library from the VLNV library and load it onto the instantiated processor:

```
    // get semihost library to exit simulation
    const char *semihostFile = opVLNVString(
```

```
        0, "ovpworld.org", "modelSupport", "imperasExit", "1.0", OP_EXTENSION,
        True
    );

    // attach imperasExit semihost library to processor
    opProcessorExtensionNew(processor, semihostFile, "imperasExit", 0);
```

You may use the `imperasExit` semihosting library with any processor model: it is not specific to the OR1K processor we are creating here.

6.4 Uncooked Disassembly

When implementing a disassembler, it is good practice to implement two different formats:

1. Normal (*cooked*) disassembly, as described above. In this mode, the output from the disassembler will be a string using the standard mnemonics and format for the model architecture.
2. *Uncooked* disassembly. In this mode, the output from the disassembler can be any format that is easy for downstream tools to consume and parse. When instruction sets are complex, implementing a good uncooked disassembly format can greatly simply tool construction, if those tools need to decode instructions.

Clients can request that a processor model disassemble an instruction in various ways. In the VMI interface, use the following function:

```
typedef enum vmiDisassAttrsE {
    DSA_NORMAL   = 0x00000000,  // normal disassembly
    DSA_UNCOOKED = 0x00000001,  // model-specific uncooked format
    DSA_BASE     = 0x00000002,  // use base model disassembly (not intercept)
    DSA_MODEL    = 0x80000000,  // model-specific mask
} vmiDisassAttrs;

const char *vmirtDisassemble(
    vmiProcessorP  processor,
    Addr           simPC,
    vmiDisassAttrs attrs
);
```

Here, passing `attrs` of `DSA_NORMAL` will cause cooked disassembly to be returned, and `attrs` of `DSA_UNCOOKED` will cause uncooked disassembly to be returned[2]. There is a very similar function available in the OP interface:

```
typedef enum optDisassAttrsE {
    OP_DSA_NORMAL   = 0x00000000, ///< normal disassembly
    OP_DSA_UNCOOKED = 0x00000001, ///< model-specific uncooked format
    OP_DSA_BASE     = 0x00000002, ///< use base model disassembly (not
                                  ///  intercept)
    OP_DSA_MODEL    = 0x80000000  ///< model-specific mask
} optDisassAttrs;

const char *opProcessorDisassemble (
```

[2] For completeness, `DSA_BASE` disables any disassembler specified in an intercept library, and the `DSA_MODEL` mask is passing model-specific flags to the disassembly callback.

```
    optProcessorP  processor,
    Addr           addr,
    optDisassAttrs attrs
);
```

To see how the disassembler behaves in uncooked mode, modify `platform/harness.c` as follows. Firstly, add a call to `opRootModulePreSimulate` before the first call to simulate in function `main` (required because disassembly is only possible in the *simulation* phase):

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

    . . . lines omitted . . .

    // connect the memory onto the busses
    opMemoryBusConnect(memory, bus, "mp1", 0x00000000, 0xffffffff);

    // complete elaboration
    opRootModulePreSimulate(mr);

    // run processor, one instruction at a time
    while(simulate(processor, 1)) {
    }

    . . . lines omitted . . .
}
```

Then, modify function `simulate` so that each instruction is disassembled in uncooked mode before it is executed:

```
static Bool simulate(optProcessorP processor, Uns64 clocks) {

    // validate uncooked disassembly
    Uns32 thisPC = opProcessorPC(processor);

    opPrintf(
        "UNCOOKED 0x%08x: %s\n",
        thisPC,
        opProcessorDisassemble(processor, thisPC, OP_DSA_UNCOOKED)
    );

    optStopReason stopReason = opProcessorSimulate(processor, clocks);

    switch(stopReason) {
        . . . lines omitted . . .
    }
}
```

Rebuild the harness using:

```
make –C platform clean
make –C platform
```

And rerun as before:

```
platform/harness.$IMPERAS_ARCH.exe –-trace \
    --program application/asmtest.OR1K.elf
```

The output from this should be as follows:

```
or1kConstructor called
Warning (PC_NRI) No register information callback given for processor
'platform/cpu1'
UNCOOKED 0x01000074: l.addi R1:r1 R2:r2 CX:0x0
Info 'platform/cpu1', 0x0000000001000074(_start): l.addi   r1,r2,0x0
UNCOOKED 0x01000078: l.addi R1:r1 R2:r2 CX:0x1
Info 'platform/cpu1', 0x0000000001000078(_start+4): l.addi   r1,r2,0x1
UNCOOKED 0x0100007c: l.addi R1:r1 R2:r2 CX:0xffffffff
Info 'platform/cpu1', 0x000000000100007c(_start+8): l.addi   r1,r2,0xffffffff
UNCOOKED 0x01000080: l.addic R1:r1 R2:r2 CX:0x1
Info 'platform/cpu1', 0x0000000001000080(_start+c): l.addic  r1,r2,0x1
UNCOOKED 0x01000084: l.addic R1:r1 R2:r2 CX:0xffffffff
Info 'platform/cpu1', 0x0000000001000084(_start+10): l.addic  r1,r2,0xffffffff
UNCOOKED 0x01000088: l.addic R1:r1 R2:r2 CX:0x0
Info 'platform/cpu1', 0x0000000001000088(_start+14): l.addic  r1,r2,0x0
UNCOOKED 0x0100008c: l.andi R1:r1 R2:r2 CX:0x1
Info 'platform/cpu1', 0x000000000100008c(_start+18): l.andi   r1,r2,0x1
UNCOOKED 0x01000090: l.andi R1:r1 R2:r2 CX:0xffff
Info 'platform/cpu1', 0x0000000001000090(_start+1c): l.andi   r1,r2,0xffff
UNCOOKED 0x01000094: l.andi R1:r1 R2:r2 CX:0x0
Info 'platform/cpu1', 0x0000000001000094(_start+20): l.andi   r1,r2,0x0
UNCOOKED 0x01000098: l.ori R1:r1 R2:r2 CX:0x1
Info 'platform/cpu1', 0x0000000001000098(_start+24): l.ori    r1,r2,0x1
UNCOOKED 0x0100009c: l.ori R1:r1 R2:r2 CX:0xffff
Info 'platform/cpu1', 0x000000000100009c(_start+28): l.ori    r1,r2,0xffff
UNCOOKED 0x010000a0: l.ori R1:r1 R2:r2 CX:0x0
Info 'platform/cpu1', 0x00000000010000a0(_start+2c): l.ori    r1,r2,0x0
UNCOOKED 0x010000a4: l.xori R1:r1 R2:r2 CX:0x1
Info 'platform/cpu1', 0x00000000010000a4(_start+30): l.xori   r1,r2,0x1
UNCOOKED 0x010000a8: l.xori R1:r1 R2:r2 CX:0xffffffff
Info 'platform/cpu1', 0x00000000010000a8(_start+34): l.xori   r1,r2,0xffffffff
UNCOOKED 0x010000ac: l.xori R1:r1 R2:r2 CX:0x0
Info 'platform/cpu1', 0x00000000010000ac(_start+38): l.xori   r1,r2,0x0
UNCOOKED 0x010000b0: l.muli R1:r1 R2:r2 CX:0x1
Info 'platform/cpu1', 0x00000000010000b0(_start+3c): l.muli   r1,r2,0x1
UNCOOKED 0x010000b4: l.muli R1:r1 R2:r2 CX:0xffffffff
Info 'platform/cpu1', 0x00000000010000b4(_start+40): l.muli   r1,r2,0xffffffff
UNCOOKED 0x010000b8: l.muli R1:r1 R2:r2 CX:0x0
Info 'platform/cpu1', 0x00000000010000b8(_start+44): l.muli   r1,r2,0x0
UNCOOKED 0x010000bc: l.addi R1:r1 R2:r2 CX:0x0
Info 'platform/cpu1', 0x00000000010000bc(exit): l.addi   r1,r2,0x0
Processor 'platform/cpu1' terminated at 'exit', address 0x10000bc
```

This clearly shows the uncooked format selected for this model: it consists of the opcode, followed by a space-separated list of key:value pairs. This format is good because it is easy to parse and extensible (it is easy to add new key:value pairs if required). Examine the calls to function `putUncookedKey` in `or1kDisassemble.c` to see how this is implemented.

7 Implementing Simple Behavior

When the processor decoder and disassembler are working correctly for a subset of processor instructions, you can start to implement behavior for those instructions. This chapter shows how this is done for simple instructions using the Imperas *code morphing* technology.

7.1 An Introduction to Code Morphing

Conventional processor models written in an HDL or similar modeling language might be implemented by a loop that is activated by a clock signal. On each activation of the clock, the model might fetch the next instruction, decode it, and call specific functions to perform the instruction (update model registers, read and write memory, and so on). If the model is cycle-accurate, there may be further complications of modeling pipelines, branch prediction and so on.

Although models written in this conventional style can be accurate and straightforward in structure, they are not fast: even a simple instruction accurate model written in C will probably run no faster than a few million instructions per second. Unfortunately, platform testing may require the execution of billions of instructions, which makes this style of model too slow.

Processor models designed for the Imperas tool set instead use just-in-time (JIT) *code morphing* technology. This works as follows:

1. As each new processor instruction is encountered during program execution, the instruction is translated (morphed) into equivalent native machine code. The exact translations to be made are specified by the processor modeler using the Imperas Virtual Machine Interface (VMI) API.
2. Contiguous sections of translated processor instructions are gathered into *code blocks*, which are held in a *dictionary* for the processor.
3. If a processor performs a jump to a simulated address that has already been translated to a code block held in the dictionary, there is no need to perform the translation again: the simulator simply re-executes the existing code block.

Imperas technology handles the generation of native machine code and the efficient management of code blocks and dictionaries to give extremely fast simulation. Depending on the complexity of the processor being simulated, speeds of billions of simulated instructions per second can be achieved. This is possible because, as simulation proceeds, *run time* (execution of translated code blocks) dominates *morph time* (JIT compilation).

To support the JIT compiler, you must implement the *morpher*, which is responsible for defining how each processor instruction should be executed.

7.2 The Template Simple Behavioral Model

A template model for the OR1K processor with a decoder, disassembler and behavior can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/4.or1kBehaviorSimple
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/4.or1kBehaviorSimple .
```

Compile the model, harness and application using the `make` command:

```
cd 4.or1kBehaviorSimple
make
```

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is based on the previous disassembler model, with the changes listed in following sections.

7.2.1 Defining GPRs - `or1kStructure.h`

The processor structure defined in file `or1kStructure.h` is where you define the registers and other state of the model. For this example, we need to model the 32 OR1K general-purpose registers. The structure is therefore declared like this:

```
#define OR1K_REGS 32            // basic OR1K registers
#define OR1K_BITS 32            // register size in bits

// processor structure
typedef struct or1kS {
    Uns32       regs[OR1K_REGS]; // basic registers
    vmiBusPortP busPorts;         // bus port descriptions
} or1k, *or1kP;
```

The OR1K general purpose registers are declared as a C array of `Uns32` values, `regs`.

As we will see in section 7.2.3, Imperas API routines for generating morphed code need to know about the *register byte offsets* of the register fields within the processor structure. In this case, the C structure offsets are as follows:

Register	**Byte Offset**
regs[0]	0
regs[1]	4
regs[2]	8
… etc …	

To simplify calculation of these offsets, `or1kStructure.h` defines the following macros for use in *variable* C expressions:

```
// macros to specify target registers in VARIABLE expressions
#define OR1K_CPU_REG(_F)        VMI_CPU_REG(or1kP, _F)
#define OR1K_REG(_R)            OR1K_CPU_REG(regs[_R])
```

For example:

```
OR1K_REG(3)
```

Is used to identify OR1K general purpose register `r3` in morpher VMI API calls (see section 7.2.3 for examples).

7.2.2 Initializing GPRs - `or1kMain.c`

Now that the OR1K structure has real state implemented, the constructor in file `or1kMain.c` should be upgraded to initialize that state.

The constructor is called by the simulator whenever a new instance of the OR1K processor model is created. It is passed a pointer to the new processor model instance using a generic parameter called `processor` of type `vmiProcessorP`. In order to initialize the processor, the generic `processor` pointer should be cast to a specific `or1kP` pointer so that fields in the structure can be set.

By default, the new processor model instance is entirely zeroed out. In this example, we initialize every general purpose register in the OR1K from `r2` to `r31` with the pattern `0xdeadbeef`:

```
VMI_CONSTRUCTOR_FN(or1kConstructor) {

    or1kP or1k = (or1kP)processor;
    Uns32 i;

    for(i=2; i<OR1K_REGS; i++) {
        or1k->regs[i] = 0xdeadbeef;
    }

    // create bus port specifications
    newBusPorts(or1k);
}
```

Register `r0` is left unmodified (zeroed out) because it is hardwired to zero in the OR1K. Register `r1` is also left zeroed out because this is the stack pointer register, implicitly initialized to zero.

(Note that the constructor and destructor no longer print that they have been called.)

7.2.3 Implementing Binops - `or1kMorph.c`

An extra field, `binop`, has been added to the `or1kMorphAttr` structure:

```
typedef struct or1kMorphAttrS {
    or1kMorphFn morphCB;    // function to translate one instruction
    vmiBinop    binop;      // if a simple binary operation
} or1kMorphAttr;
```

This new field will be used to control the precise action of the arithmetic instructions implemented here. The morpher table, `or1kMorphTable`, has been updated to initialize the new field as follows:

```
const or1kMorphAttr or1kMorphTable[OR1K_IT_LAST+1] = {

    // handle arithmetic instructions (second argument constant)
    [OR1K_IT_ADDI]  = {morphCB:morphBinopRRC, binop:vmi_ADD },
    [OR1K_IT_ADDIC] = {morphCB:morphBinopRRC, binop:vmi_ADC },
    [OR1K_IT_ANDI]  = {morphCB:morphBinopRRC, binop:vmi_AND },
    [OR1K_IT_ORI]   = {morphCB:morphBinopRRC, binop:vmi_OR  },
    [OR1K_IT_XORI]  = {morphCB:morphBinopRRC, binop:vmi_XOR },
    [OR1K_IT_MULI]  = {morphCB:morphBinopRRC, binop:vmi_IMUL},
};
```

The behavior of each of the instructions we will implement now is described by a common function, `morphBinopRRC`:

```
static OR1K_MORPH_FN(morphBinopRRC) {

    vmiBinop op = state->attrs->binop;
    vmiReg   rd = getGPR(state->info.r1);
    vmiReg   ra = getGPR(state->info.r2);
    Uns32    c  = state->info.c;

    vmimtBinopRRC(OR1K_BITS, op, rd, ra, c, 0);
}
```

This function uses a routine from the Imperas *morph time function* API (`vmiMt.h`) to *describe the behavior of the arithmetic instructions we are implementing in this example*. The destination register number (`rd`), argument register number (`ra`) and constant value (`c`) are extracted from the decoded instruction structure. It is very important to understand that `vmimt`-prefixed routines do not themselves perform any arithmetic operation on the processor registers: instead, *they describe the action to be performed*. The action descriptions are used as input to the Imperas JIT compiler to generate native code that, when executed, performs the required arithmetic operation.

To further clarify this example, we will consider lines from `morphBinopRRC` in detail. The first line gets the operation to implement:

```
    vmiBinop op = state->attrs->binop;
```

The operation `op` can be any of the operations specified in the `vmiBinop` enumeration, declared in file `vmiTypes.h`:

```
typedef enum {
                                        // ARITHMETIC OPERATIONS
    vmi_ADD        = OCL_BIN_ADD,       // d <- a + b
    vmi_ADC        = OCL_BIN_ADC,       // d <- a + b + C
```

```
    vmi_SUB         = OCL_BIN_SUB,        // d <- a - b
    vmi_SBB         = OCL_BIN_SBB,        // d <- a - b - C
    vmi_RSBB        = OCL_BIN_RSBB,       // d <- b - a - C
    vmi_RSUB        = OCL_BIN_RSUB,       // d <- b - a
    vmi_IMUL        = OCL_BIN_IMUL,       // d <- a * b (signed)
    vmi_MUL         = OCL_BIN_MUL,        // d <- a * b (unsigned)
    vmi_IDIV        = OCL_BIN_IDIV,       // d <- a / b (signed)
    vmi_DIV         = OCL_BIN_DIV,        // d <- a / b (unsigned)
    vmi_IREM        = OCL_BIN_IREM,       // d <- a % b (signed)
    vmi_REM         = OCL_BIN_REM,        // d <- a % b (unsigned)
    vmi_CMP         = OCL_BIN_CMP,        // a - b

                                          // SATURATED ARITHMETIC OPERATIONS
    vmi_ADDSQ       = OCL_BIN_ADDSQ,      // d <- saturate_signed(a + b)
    vmi_ADCSQ       = OCL_BIN_ADCSQ,      // d <- saturate_signed(a + b + C)
    vmi_SUBSQ       = OCL_BIN_SUBSQ,      // d <- saturate_signed(a - b)
    vmi_SBBSQ       = OCL_BIN_SBBSQ,      // d <- saturate_signed(a - b - C)
    vmi_RSUBSQ      = OCL_BIN_RSUBSQ,     // d <- saturate_signed(b - a)
    vmi_RSBBSQ      = OCL_BIN_RSBBSQ,     // d <- saturate_signed(b - a - C)
    vmi_ADDUQ       = OCL_BIN_ADDUQ,      // d <- saturate_unsigned(a + b)
    vmi_ADCUQ       = OCL_BIN_ADCUQ,      // d <- saturate_unsigned(a + b + C)
    vmi_SUBUQ       = OCL_BIN_SUBUQ,      // d <- saturate_unsigned(a - b)
    vmi_SBBUQ       = OCL_BIN_SBBUQ,      // d <- saturate_unsigned(a - b - C)
    vmi_RSUBUQ      = OCL_BIN_RSUBUQ,     // d <- saturate_unsigned(b - a)
    vmi_RSBBUQ      = OCL_BIN_RSBBUQ,     // d <- saturate_unsigned(b - a - C)

                                          // HALVING ARITHMETIC OPERATIONS
    vmi_ADDSH       = OCL_BIN_ADDSH,      // d <- ((signed)(a + b)) / 2
    vmi_SUBSH       = OCL_BIN_SUBSH,      // d <- ((signed)(a - b)) / 2
    vmi_RSUBSH      = OCL_BIN_RSUBSH,     // d <- ((signed)(b - a)) / 2
    vmi_ADDUH       = OCL_BIN_ADDUH,      // d <- ((unsigned)(a + b)) / 2
    vmi_SUBUH       = OCL_BIN_SUBUH,      // d <- ((unsigned)(a - b)) / 2
    vmi_RSUBUH      = OCL_BIN_RSUBUH,     // d <- ((unsigned)(b - a)) / 2
    vmi_ADDSHR      = OCL_BIN_ADDSHR,     // d <- round(((signed)(a + b)) / 2)
    vmi_SUBSHR      = OCL_BIN_SUBSHR,     // d <- round(((signed)(a - b)) / 2)
    vmi_RSUBSHR     = OCL_BIN_RSUBSHR,    // d <- round(((signed)(b - a)) / 2)
    vmi_ADDUHR      = OCL_BIN_ADDUHR,     // d <- round(((unsigned)(a + b)) / 2)
    vmi_SUBUHR      = OCL_BIN_SUBUHR,     // d <- round(((unsigned)(a - b)) / 2)
    vmi_RSUBUHR     = OCL_BIN_RSUBUHR,    // d <- round(((unsigned)(b - a)) / 2)

                                          // BITWISE OPERATIONS
    vmi_OR          = OCL_BIN_OR,         // d <- a | b
    vmi_AND         = OCL_BIN_AND,        // d <- a & b
    vmi_XOR         = OCL_BIN_XOR,        // d <- a ^ b
    vmi_ORN         = OCL_BIN_ORN,        // d <- a | ~b
    vmi_ANDN        = OCL_BIN_ANDN,       // d <- a & ~b
    vmi_XORN        = OCL_BIN_XORN,       // d <- a ^ ~b
    vmi_NOR         = OCL_BIN_NOR,        // d <- ~(a | b)
    vmi_NAND        = OCL_BIN_NAND,       // d <- ~(a & b)
    vmi_XNOR        = OCL_BIN_XNOR,       // d <- ~(a ^ b)

                                          // SHIFT/ROTATE OPERATIONS
    vmi_ROL         = OCL_BIN_ROL,        // d <- a << b | a >> <bits>-b
    vmi_ROR         = OCL_BIN_ROR,        // d <- a >> b | a << <bits>-b
    vmi_RCL         = OCL_BIN_RCL,        // (d,c) <- (a,c)<<b | (a,c) >> <bits>-b
    vmi_RCR         = OCL_BIN_RCR,        // (d,c) <- (a,c)>>b | (a,c) << <bits>-b
    vmi_SHL         = OCL_BIN_SHL,        // d <- a << b
    vmi_SHR         = OCL_BIN_SHR,        // d <- (unsigned)a >> b
    vmi_SAR         = OCL_BIN_SAR,        // d <- (signed)a >> b

                                          // SATURATED SHIFT OPERATIONS
    vmi_SHLSQ       = OCL_BIN_SHLSQ,      // d <- saturate_signed(a << b)
    vmi_SHLUQ       = OCL_BIN_SHLUQ,      // d <- saturate_unsigned(a << b)
```

```
                                    // ROUNDING SHIFT OPERATIONS
    vmi_SHRR       = OCL_BIN_SHRR,      // d <- round((unsigned)a >> b)
    vmi_SARR       = OCL_BIN_SARR,      // d <- round((signed)a >> b)

    vmi_BINOP_LAST = OCL_BIN_LAST       // KEEP LAST

} vmiBinop;
```

Next, destination register `rd` and argument register `ra` are obtained from the decoded instruction:

```
    vmiReg   rd = getGPR(state->info.r1);
    vmiReg   ra = getGPR(state->info.r2);
```

In the OR1K processor, register `r0` is hardwired to the constant value 0. Any attempt to write to this register should be discarded: this is indicated to the morph time API functions by using the special value `VMI_NOREG` for the register. If the target register is writable, the macro `OR1K_REG(rd)` from `or1kStructure.h` is used to specify it, as described in section 7.2.1. This complexity is encapsulated in function `getGPR`, defined as follows:

```
static vmiReg getGPR(Uns32 r) {
    return r ? OR1K_REG(r) : VMI_NOREG;
}
```

Next, a constant value is extracted from the decoded instruction:

```
    Uns32    c  = state->info.c;
```

Finally, the operation to perform is described by the morph time API function `vmimtBinopRRC`:

```
    vmimtBinopRRC(OR1K_BITS, op, rd, ra, c, 0);
```

Refer to the *Imperas VMI Morph Time Reference* manual for more detailed information on all of the morph-time functions available in this API.

7.3 Running the Application Test Case with the Processor Model

Run the application using this command:

```
platform/harness.$IMPERAS_ARCH.exe --trace --traceregs \
    --program application/application.OR1K.elf
```

in the `4.or1kBehaviorSimple` directory. The new `--traceregs` parameter, in combination with `--trace`, enables trace of *register values*. You should see the following output:

```
Warning (PC_NRI) No register information callback given for processor ‘cpu1’
Info 'cpu1', 0x0000000000000100: l.addi   r2,r0,0x0
Info 'cpu1' REGISTERS
CPU cpu1 (instruction 1):
       0: 00000000 00000000 00000000 deadbeef
      16: deadbeef deadbeef deadbeef deadbeef
      32: deadbeef deadbeef deadbeef deadbeef
      48: deadbeef deadbeef deadbeef deadbeef
      64: deadbeef deadbeef deadbeef deadbeef
      80: deadbeef deadbeef deadbeef deadbeef
      96: deadbeef deadbeef deadbeef deadbeef
     112: deadbeef deadbeef deadbeef deadbeef
     128: 08daa790
Info 'cpu1', 0x0000000000000104: l.addi   r3,r0,0x0
Info 'cpu1' REGISTERS
CPU cpu1 (instruction 2):
       0: 00000000 00000000 00000000 00000000
      16: deadbeef deadbeef deadbeef deadbeef
      32: deadbeef deadbeef deadbeef deadbeef
      48: deadbeef deadbeef deadbeef deadbeef
      64: deadbeef deadbeef deadbeef deadbeef
      80: deadbeef deadbeef deadbeef deadbeef
      96: deadbeef deadbeef deadbeef deadbeef
     112: deadbeef deadbeef deadbeef deadbeef
     128: 08daa790
Info 'cpu1', 0x0000000000000108: l.addi   r4,r0,0x0
Info 'cpu1' REGISTERS
CPU cpu1 (instruction 3):
       0: 00000000 00000000 00000000 00000000
      16: 00000000 deadbeef deadbeef deadbeef
      32: deadbeef deadbeef deadbeef deadbeef
      48: deadbeef deadbeef deadbeef deadbeef
      64: deadbeef deadbeef deadbeef deadbeef
      80: deadbeef deadbeef deadbeef deadbeef
      96: deadbeef deadbeef deadbeef deadbeef
     112: deadbeef deadbeef deadbeef deadbeef
     128: 08daa790

… (many lines cut) …

Info 'cpu1', 0x0000000000000178: ??? instruction:0x1820ffff
CPU 'cpu1' 0x00000178:0x1820ffff *** undecoded instruction: exiting ***
Info 'cpu1' REGISTERS
CPU cpu1 (instruction 31):
       0: 00000000 00000000 00000000 00000000
      16: 00000000 00000000 00000000 00000000
      32: 00000000 00000000 00000000 00000000
      48: 00000000 00000000 00000000 00000000
      64: 00000000 00000000 00000000 00000000
      80: 00000000 00000000 00000000 00000000
      96: 00000000 00000000 00000000 00000000
     112: 00000000 00000000 00000000 00000000
     128: 08daa790
```

We now see the processor model executing instructions for the first time. In detail, the sequence when generating output is:

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

> Because this ORK1 model has no register value print functionality specified at this point, the register values printed after each instruction are simply the raw contents of the OR1K structure. This includes the pointer value, busPorts, at offset 128. Because this is a pointer, its value will change from run to run. The next chapter describes how a model-specific register dump callback is written.

The initial instructions of the application zero out registers `r2-r31` of the OR1K processor using `l.addi` instructions. As this happens, we see each register value change from `0xdeadbeef` (set in the processor constructor) to `0x00000000`.

7.4 Instruction Temporaries

Some instructions cannot be implemented as a single VMI operation and instead require a sequence of operations and intermediate *temporaries* to generate the correct result. For example, suppose that there is a *signed halfword multiply* instruction, which works according to the following pseudo-code:

$$T1_{32..0} = \text{sign\_extend}(R1_{15..0});$$
$$T2_{32..0} = \text{sign\_extend}(R2_{15..0});$$
$$R3_{32..0} = T1_{32..0} * T2_{32..0};$$

In other words, the instruction sign-extends the lower half of the two arguments and then multiplies those sign-extended values to produce the result. Implementing this instruction requires the use of two temporaries that are not true processor registers but instead represent intermediate values that are required only within an instruction.

The way to implement this is to introduce two new pseudo-registers into the processor structure as follows:

```
#define OR1K_REGS 32            // basic OR1K registers
#define OR1K_BITS 32            // register size in bits
#define OR1K_TNUM  2            // number of temporaries

// processor structure
typedef struct or1kS {
    Uns32 regs[OR1K_REGS]; // basic registers
    Uns32 temp[OR1K_TNUM]; // temporary pseudo-registers
    vmiBusPortP busPorts;  // bus port descriptions
} or1k, *or1kP;
```

The temporaries are specified to the morpher as follows:

```
// macros to specify target registers in VARIABLE expressions
#define OR1K_CPU_REG(_F)        VMI_CPU_REG(or1kP, _F)
#define OR1K_REG(_R)            OR1K_CPU_REG(regs[_R])
#define OR1K_CPU_REG_TEMP(_F)   VMI_CPU_TEMP(or1kP, _F)
#define OR1K_TEMP(_I)           OR1K_CPU_REG_TEMP(temp[_I])
```

The macro `VMI_CPU_TEMP` identifies *temporaries* in exactly the same way that macro `VMI_CPU_REG` identifies *true registers*. Because the morpher knows that these values are temporaries and not true registers, it can generate more efficient code (the temporary values do not need to be written back to the processor structure at the end of the instruction).

These temporaries could then be used to implement the signed halfword multiply instruction as follows:

```
    . . .
    vmiReg target = (rd==0) ? VMI_NOREG : OR1K_REG(rd);

    // generate intermediates
    vmimtMoveExtendRR(OR1K_BITS, OR1K_TEMP(0), OR1K_BITS/2, OR1K_REG(ra), True);
    vmimtMoveExtendRR(OR1K_BITS, OR1K_TEMP(1), OR1K_BITS/2, OR1K_REG(rb), True);

    // generate result
    vmimtBinopRRR(OR1K_BITS, vmi_IMUL, target, OR1K_TEMP(0), OR1K_TEMP(1), 0);
```


8 Processor Flags and Register Dumping

In general, arithmetic operations can both take as input and generate as output *flag values*. For example an add-with-carry operation has a carry flag input, and might generate a carry flag output. This chapter enhances the previous simple behavioral model to handle flag values for arithmetic instructions, and shows how to implement a model specific *register dump* routine to simplify model validation.

8.1 The Template Flags Model

A template model for the OR1K processor implementing instruction flags can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/5.or1kBehaviorFlags
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/5.or1kBehaviorFlags .
```

Compile the model, harness and application using the `make` command:

```
cd 5.or1kBehaviorFlags
make
```

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is based on the previous model, with the changes listed in following sections.

8.1.1 Adding Flag Registers - `or1kStructure.h`

For this example, we need to model the 32 OR1K general-purpose registers and three boolean flags: *carry*, *overflow* and *branch*. The structure is therefore modified like this:

```
#define OR1K_REGS 32            // basic OR1K registers
#define OR1K_BITS 32            // register size in bits

// processor structure
typedef struct or1kS {

    Bool       carryFlag;       // carry flag
    Bool       overflowFlag;    // overflow flag
    Bool       branchFlag;      // branch flag

    Uns32      regs[OR1K_REGS]; // basic registers

    vmiBusPortP busPorts;       // bus port descriptions

} or1k, *or1kP;
```

The C structure byte offsets of the various fields are now as follows:

Register	**Byte Offset**
carryFlag	0
overflowFlag	1
branchFlag	2
regs[0]	4
regs[1]	8
… etc …	

To simplify calculation of these offsets, `or1kStructure.h` now has the following macros for use in *variable* C expressions:

```
// macros to specify target registers in VARIABLE expressions
#define OR1K_CPU_REG(_F)        VMI_CPU_REG(or1kP, _F)
#define OR1K_REG(_R)            OR1K_CPU_REG(regs[_R])
#define OR1K_CARRY              OR1K_CPU_REG(carryFlag)
#define OR1K_OVERFLOW           OR1K_CPU_REG(overflowFlag)
```

In constant expression contexts (for example static structure initializers) these variants should be used instead:

```
// macros to specify target registers in CONSTANT expressions
#define OR1K_CPU_REG_CONST(_F)  VMI_CPU_REG_CONST(or1kP, _F)
#define OR1K_REG_CONST(_R)      OR1K_CPU_REG_CONST(regs[_R])
#define OR1K_CARRY_CONST        OR1K_CPU_REG_CONST(carryFlag)
#define OR1K_OVERFLOW_CONST     OR1K_CPU_REG_CONST(overflowFlag)
```

8.1.2 Using Flags - `or1kMorph.c`

How flags should be handled in an arithmetic operation is indicated by a *flags* argument to the VMI morph-time API call describing the operation. This argument is a pointer to a structure type defined in `vmiTypes.h`:

```
typedef enum {
    vmi_CF=0,           // carry flag
    vmi_PF=1,           // parity flag
    vmi_ZF=2,           // zero flag
    vmi_SF=3,           // sign flag
    vmi_OF=4,           // overflow flag
    vmi_LF=5            // KEEP LAST
} vmiFlag;

typedef enum {
    vmi_FN_NONE  =0x00, // empty negate mask
    vmi_FN_CF_IN =0x01, // negate carry in flag
    vmi_FN_CF_OUT=0x02, // negate carry out flag
    vmi_FN_PF    =0x04, // negate parity flag
    vmi_FN_ZF    =0x08, // negate zero flag
    vmi_FN_SF    =0x10, // negate sign flag
    vmi_FN_OF    =0x20, // negate overflow flag
} vmiFlagNegate;

typedef struct vmiFlagsS {
```

```
    vmiReg        cin;          // register specifying carry in
    vmiReg        f[vmi_LF];    // registers to hold operation results
    vmiFlagNegate negate;       // bitmask of negated flags
} vmiFlags;
```

The `vmiFlag` enumeration lists all the flags that can be generated by an arithmetic operation: *carry*, *parity*, *zero*, *sign* and *overflow*. The `vmiFlagNegate` enumeration describes how flags are negated on input to and output from the operation.

The `vmiFlags` structure contains the following:

1. A field `cin` of type `vmiReg`. This field specifies *the register offset in a processor structure of a flag byte to use for the carry in value*.
2. An array of `vmiReg` values indexed by `vmiFlag` type. This field specifies *the register offsets in a processor structure of flag bytes into which generated flags should be written*.
3. A bitmask of type `vmiFlagNegate` specifying how flags should be negated on input to and output from the operation.

In other words, the `vmiFlags` structure allows you to specify boolean flag locations within your processor structure that can provide and accept flag values in arithmetic operations. These flags should always be declared in the processor structure as type `Bool`.

The `or1kMorphAttr` structure has now been enhanced to include an extra field of type `vmiFlagsCP`:

```
typedef struct or1kMorphAttrS {
    or1kMorphFn morphCB;    // function to translate one instruction
    vmiBinop    binop;      // if a simple binary operation
    vmiFlagsCP  flags;      // flags used by the instruction
} or1kMorphAttr;
```

Function `morphBinopRRC` now extracts this `flags` field and uses it in the call to `vmimtBinopRRC`:

```
static OR1K_MORPH_FN(morphBinopRRC) {

    vmiBinop   op    = state->attrs->binop;
    vmiReg     rd    = getGPR(state->info.r1);
    vmiReg     ra    = getGPR(state->info.r2);
    Uns32      c     = state->info.c;
    vmiFlagsCP flags = state->attrs->flags;

    vmimtBinopRRC(OR1K_BITS, op, rd, ra, c, flags);
}
```

In the specific example of the OR1K processor, the bitwise logical operations do not use or affect any processor flags. This is indicated by specifying a null pointer for the `flags` argument in the `or1kMorphTable` initialization:

```
    [OR1K_IT_ANDI]  = {morphCB:morphBinopRRC, binop:vmi_AND,  flags:0      },
    [OR1K_IT_ORI]   = {morphCB:morphBinopRRC, binop:vmi_OR,   flags:0      },
    [OR1K_IT_XORI]  = {morphCB:morphBinopRRC, binop:vmi_XOR,  flags:0      },
```

The remaining arithmetic operations can generate *carry* and *overflow* flags, and (in the case of instruction `l.adc`) take a carry flag as input. Other possible output flags do not exist on the OR1K. This is indicated using a `vmiFlags` structure `flagsCO`:

```
const vmiFlags flagsCO = {
    OR1K_CARRY_CONST,           // offset to carry in flag
    {
        OR1K_CARRY_CONST,       // offset to carry out flag
        VMI_NOFLAG_CONST,       // parity flag not used
        VMI_NOFLAG_CONST,       // zero flag not used
        VMI_NOFLAG_CONST,       // sign flag not used
        OR1K_OVERFLOW_CONST     // offset to overflow flag
    }
};
```

The new `flagsCO` structure is used in the `or1kMorphTable` initialization like this:

```
    [OR1K_IT_ADDI]  = {morphCB:morphBinopRRC, binop:vmi_ADD,  flags:&flagsCO},
    [OR1K_IT_ADDIC] = {morphCB:morphBinopRRC, binop:vmi_ADC,  flags:&flagsCO},
    [OR1K_IT_MULI]  = {morphCB:morphBinopRRC, binop:vmi_IMUL, flags:&flagsCO},
```

In detail, `flagsCO` specifies that:

1. Any input carry required by the arithmetic operation should be obtained from the processor structure at offset `OR1K_CARRY_CONST`, specified in `or1kStructure.h`. This corresponds to the `carry` Boolean field in the structure.
2. Any output carry generated by the arithmetic operation should be written to the processor structure at offset `OR1K_CARRY_CONST`.
3. Any output overflow generated by the arithmetic operation should be written to the processor structure at offset `OR1K_OVERFLOW_CONST`.
4. Any other output flags generated by the arithmetic operations should be discarded (indicated by using the special value `VMI_NOFLAG_CONST` in the appropriate `vmiFlags` structure field).
5. The carry flag should not be negated when used as an input and no flags should be negated on output. Therefore, the negate field of `flagsCO` is initialized to the default zero value (`vmi_FN_NONE`) by omitting it from the structure initializer.

> Note what happens in `morphBinopRRC` when the output register `rd` is `r0`. Recall that `r0` is hardwired to zero on the OR1K processor. What should happen to the processor flags for an instruction where the output register is `r0`? The result should be *discarded* but changes to the flag values *preserved*. This can be indicated to the VMI morph-time API by specifying the special value `VMI_NOREG` as the destination register to `vmimtBinopRRC`.

8.2 Validating Flag Behavior with Tests

For even apparently simple instructions like `l.addic`, it is clear that there are already a number of separate cases to be tested. An ideal test plan should cover the following options in various combinations:

1. target register `rd` of `r0-r31`
2. target register `rd` of `r0`
3. source register `ra` of `r1-r31`
4. source register `ra` of `r0`
5. validate carry output generated when required
6. validate overflow output generated when required
7. validate carry input used when required

File `asmtest.S` in directory `5.or1kBehaviorFlags/application` is an example of how this could be done.

```
.global _start
_start:                            // TEST PROLOGUE
        l.addi      r1,r0,0        // r1 = 0
        l.addi      r2,r0,1        // r2 = 1
        l.addi      r3,r0,-1       // r3 = -1
        l.addi      r4,r0,0x800    // r4 = 0x00000800
        l.muli      r4,r4,0x800    // r4 = 0x00400000
        l.muli      r4,r4,0x200    // r4 = 0x80000000
        l.addi      r5,r4,-1       // r5 = 0x7fffffff

        l.addic     r20,r0,-1
        l.addic     r20,r0,0
        l.addic     r20,r0,1
        l.addic     r20,r1,-1
        l.addic     r20,r1,0
        l.addic     r20,r1,1
        l.addic     r20,r2,-1
        l.addic     r20,r2,0
        l.addic     r20,r2,1
        l.addic     r20,r3,-1
        l.addic     r20,r3,0
        l.addic     r20,r3,1
        l.addic     r20,r4,-1
        l.addic     r20,r4,0
        l.addic     r20,r4,1
        l.addic     r20,r5,-1
        l.addic     r20,r5,0
        l.addic     r20,r5,1

        l.addic     r0,r0,-1
        l.addic     r0,r0,0
        l.addic     r0,r0,1
        l.addic     r0,r1,-1
        l.addic     r0,r1,0
        l.addic     r0,r1,1
        l.addic     r0,r2,-1
        l.addic     r0,r2,0
        l.addic     r0,r2,1
        l.addic     r0,r3,-1
        l.addic     r0,r3,0
        l.addic     r0,r3,1
        l.addic     r0,r4,-1
        l.addic     r0,r4,0
        l.addic     r0,r4,1
        l.addic     r0,r5,-1
        l.addic     r0,r5,0
        l.addic     r0,r5,1
```

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

Run the application using this command:

```
platform/harness.$IMPERAS_ARCH.exe –-trace --traceregs \
    --program application/asmtest.OR1K.elf
```

The output from this should be as follows:

```
Warning (PC_NRI) No register information callback given for processor ‘cpu1’
Info 'cpu1', 0x0000000001000074: l.addi   r1,r0,0x0
Info 'cpu1' REGISTERS
CPU cpu1 (instruction 1):
       0: 00000000 00000000 00000000 deadbeef
      16: deadbeef deadbeef deadbeef deadbeef
      32: deadbeef deadbeef deadbeef deadbeef
      48: deadbeef deadbeef deadbeef deadbeef
      64: deadbeef deadbeef deadbeef deadbeef
      80: deadbeef deadbeef deadbeef deadbeef
      96: deadbeef deadbeef deadbeef deadbeef
     112: deadbeef deadbeef deadbeef deadbeef
     128: deadbeef 08daa790

… (many lines cut) …

Info 'cpu1', 0x0000000001000118: l.addic  r0,r5,0x0
Info 'cpu1' REGISTERS
CPU cpu1 (instruction 42):
       0: 00000100 00000000 00000000 00000001
      16: ffffffff 80000000 7fffffff deadbeef
      32: deadbeef deadbeef deadbeef deadbeef
      48: deadbeef deadbeef deadbeef deadbeef
      64: deadbeef deadbeef deadbeef deadbeef
      80: deadbeef 80000000 deadbeef deadbeef
      96: deadbeef deadbeef deadbeef deadbeef
     112: deadbeef deadbeef deadbeef deadbeef
     128: deadbeef 08daa790
Info 'cpu1', 0x000000000100011c: l.addic  r0,r5,0x1
Info 'cpu1' REGISTERS
CPU cpu1 (instruction 43):
       0: 00000100 00000000 00000000 00000001
      16: ffffffff 80000000 7fffffff deadbeef
      32: deadbeef deadbeef deadbeef deadbeef
      48: deadbeef deadbeef deadbeef deadbeef
      64: deadbeef deadbeef deadbeef deadbeef
      80: deadbeef 80000000 deadbeef deadbeef
      96: deadbeef deadbeef deadbeef deadbeef
     112: deadbeef deadbeef deadbeef deadbeef
     128: deadbeef 08daa790
Info 'cpu1', 0x0000000001000120: l.addi   r1,r2,0x0
Processor 'cpu1' terminated at 'exit', address 0x1000120
Info 'cpu1' REGISTERS
CPU cpu1 (instruction 44):
       0: 00000000 00000000 00000001 00000001
      16: ffffffff 80000000 7fffffff deadbeef
      32: deadbeef deadbeef deadbeef deadbeef
      48: deadbeef deadbeef deadbeef deadbeef
      64: deadbeef deadbeef deadbeef deadbeef
      80: deadbeef 80000000 deadbeef deadbeef
```

```
      96: deadbeef deadbeef deadbeef deadbeef
     112: deadbeef deadbeef deadbeef deadbeef
     128: deadbeef 08daa790
```

8.3 Model-Specific Dump Format

Comparing the output from the above example with that from the simple behavioral model (section 7.3), there is a significant difference in format because each register dump now has 34 words (136 bytes) instead of 33 words (132 bytes). This is because adding the flags to the processor structure has increased its size. It also isn't clear what the dump is showing: which values represent general purpose registers, which represent flags, and which are supplemental values (for example the `busPorts` pointer) which do not represent true processor state at all? To address this problem, we need to add a model-specific register dump routine. A template model for the OR1K with this routine added can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/6.or1kBehaviorDump
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/6.or1kBehaviorDump .
```

Compile the model, harness and application using the `make` command:

```
cd 6.or1kBehaviorDump
make
```

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is based on the previous model, with the changes listed in following sections.

8.3.1 Defining Status Register - `or1kStructure.h`

While implementing the OR1K register dump routine, we will update the processor model to partially implement the OR1K status register (`sr`). This is a 32-bit register which must be added to the processor definition in `or1kStructure.h` as follows:

```
#define OR1K_REGS 32            // basic OR1K registers
#define OR1K_BITS 32            // register size in bits

// processor structure
typedef struct or1kS {

    Bool       carryFlag;       // carry flag
    Bool       overflowFlag;    // overflow flag
    Bool       branchFlag;      // branch flag

    Uns32      regs[OR1K_REGS]; // basic registers
```

```
    Uns32      SR;              // status register

    vmiBusPortP busPorts;       // bus port descriptions

} or1k, *or1kP;
```

The status register bits are conveniently accessed using these macros:

```
// Bit definitions for the SR register
#define SPR_SR_CID      0xf0000000  // Context ID
#define SPR_SR_SUMRA    0x00010000  // Supervisor SPR read access
#define SPR_SR_FO       0x00008000  // Fixed one
#define SPR_SR_EPH      0x00004000  // Exception Prefix High
#define SPR_SR_DSX      0x00002000  // Delay Slot Exception
#define SPR_SR_OVE      0x00001000  // Overflow flag Exception
#define SPR_SR_OV       0x00000800  // Overflow flag
#define SPR_SR_CY       0x00000400  // Carry flag
#define SPR_SR_F        0x00000200  // Condition Flag
#define SPR_SR_CE       0x00000100  // CID Enable
#define SPR_SR_LEE      0x00000080  // Little Endian Enable
#define SPR_SR_IME      0x00000040  // Instruction MMU Enable
#define SPR_SR_DME      0x00000020  // Data MMU Enable
#define SPR_SR_ICE      0x00000010  // Instruction Cache Enable
#define SPR_SR_DCE      0x00000008  // Data Cache Enable
#define SPR_SR_IEE      0x00000004  // Interrupt Exception Enable
#define SPR_SR_TEE      0x00000002  // Tick timer Exception Enable
#define SPR_SR_SM       0x00000001  // Supervisor Mode
```

8.3.2 Adding Register Dump - `or1kUtils.c`

This file implements the OR1K register dump function using the `VMI_DEBUG_FN` macro, defined in `vmiDbg.h`. The function is as follows:

```
VMI_DEBUG_FN(or1kDumpRegisters) {

    or1kP or1k = (or1kP)processor;
    Uns32 i    = 0;

    vmiPrintf("--------------- -------------- -------------- ---------------
\n");

    // print general-purpose registers
    while(i<OR1K_REGS) {
        vmiPrintf(" R%-2u: %08x", i, or1k->regs[i]);
        i++;
        if(!(i&3)) {
            vmiPrintf("\n");
        } else {
            vmiPrintf("  ");
        }
    }

    // newline if required before derived registers
    if(i&3) {
        vmiPrintf("\n");
    }

    // flags
    vmiPrintf(
        " BF:%u CF:%u OF:%u ",
```

```
        or1k->branchFlag,
        or1k->carryFlag,
        or1k->overflowFlag
    );

    // program counter
    vmiPrintf(" PC : %08x  ", (Uns32)vmirtGetPC(processor));

    // status register
    vmiPrintf(" SR : %08x  ", or1kGetSR(or1k));

    vmiPrintf("\n-------------- -------------- -------------- --------------
\n\n");
}
```

The register dump function is passed a single `vmiProcessorP` argument, indicating the processor for which to dump registers. The first step is to cast this to an `or1kP` type:

```
    or1kP or1k = (or1kP)processor;
```

Next, the function prints out the values of the OR1K general purpose registers, naming them `r0`, `r1`, `r2` and so on. All output is generated using the VMI routine `vmiPrintf`, defined in `vmiMessage.h`:

```
    while(i<OR1K_REGS) {
        vmiPrintf(" R%-2u: %08x", i, or1k->regs[i]);
        i++;
        if(!(i&3)) {
            vmiPrintf("\n");
        } else {
            vmiPrintf("  ");
        }
    }
```

Next, the function prints the current settings of the *branch*, *carry* and *overflow* flags:

```
    // flags
    vmiPrintf(
        " BF:%u CF:%u OF:%u ",
        or1k->branchFlag,
        or1k->carryFlag,
        or1k->overflowFlag
    );
```

8.3.2.1 Printing the Program Counter (PC)

The OR1K processor has a program counter register, PC, which we would like to print in the dump routine. Until this point, we have not modeled the processor program counter at all; how should it be done?

One solution would be to introduce an extra `pc` field into the processor structure, which we could update at the start of every instruction using a morph-time operation. For example:

```
// processor structure
typedef struct or1kS {
```

```
    Bool        carryFlag;       // carry flag
    Bool        overflowFlag;    // overflow flag
    Bool        branchFlag;      // branch flag

    Uns32       regs[OR1K_REGS]; // basic registers

    Uns32       SR;              // status register
    Uns32       PC;              // program counter

    vmiBusPortP busPorts;        // bus port descriptions

} or1k, *or1kP;

#define OR1K_PC OR1K_REG(PC)

VMI_MORPH_FN(or1kMorphInstruction) {
    vmimtMoveRC(OR1K_BITS, OR1K_PC, (Uns32)thisPC);
    or1kDecode((or1kP)processor, thisPC, OR1K_MORPH, 0);
}
```

However, this is unnecessarily inefficient: we have already seen from the instruction trace in previous examples that the simulator always knows the address of the current instruction. Instead of maintaining the program counter value in the model, it would be much better just to ask the simulator for the current program counter value when we need it. A routine to give exactly what is required is available in the VMI *run-time* interface (defined in file `vmiRt.h`):

```
//
// Return the current program counter for a processor
//
Addr vmirtGetPC(vmiProcessorP processor);
```

The OR1K register dump function uses this as follows:

```
    vmiPrintf(" PC : %08x  ", (Uns32)vmirtGetPC(processor));
```

This highlights a *very important point*: when writing a processor model, do not explicitly model register values that are *infrequently referenced* and can *easily be created on demand*. This is *always* the case for the program counter and very often the case for processor status registers. Failure to do this will result in processor models which are much slower than they need to be.

8.3.2.2 Printing the Status Register (`sr`)

As a second example of creating register values on demand, the OR1K also contains a status register, `sr`. This register encodes the values of the three OR1K flags (carry, overflow and branch) in addition to other status information (whether the processor is in supervisor mode, for example). The OR1K register dump function prints the current value of the status register like this:

```
    vmiPrintf(" SR : %08x  ", or1kGetSR(or1k));
```

The routine `or1kGetSR` is implemented in `or1kUtils.c` like this:

```
Uns32 or1kGetSR(or1kP or1k) {
    fillSR(or1k);
    return or1k->SR;
}
```

The routine `fillSR` updates the current value of the `sr` register field in the processor structure so that it includes the three boolean flags:

```
#define SET_BIT(_R, _C, _M) \
    if(_C) {                    \
        (_R) = (_R) | (_M);     \
    } else {                    \
        (_R) = (_R) & ~(_M);    \
    }

inline static void fillSR(or1kP or1k) {
    SET_BIT(or1k->SR, or1k->branchFlag,   SPR_SR_F);
    SET_BIT(or1k->SR, or1k->carryFlag,    SPR_SR_CY);
    SET_BIT(or1k->SR, or1k->overflowFlag, SPR_SR_OV);
}
```

In other words, when the model requires the current value of the OR1K status register `sr`, it should call the routine `or1kGetSR`, which assembles the value by combining some bits stored in the processor structure `SR` field with the current values of the three flag registers. This is much more efficient than regenerating the full value of `sr` after each instruction that could possibly modify flag values.

For completeness, `or1kUtils.c` also implements a public function to set the `sr` register, `or1kSetSR`. This isn't used in this example, but will be required in the full model.

```
#define GET_BIT(_R, _M) \
    (((_R) & (_M)) ? 1 : 0)

void or1kSetSR(or1kP or1k, Uns32 value) {

    // it is never possible to clear the fixed-one (FO) bit
    value |= SPR_SR_FO;

    // set the SR
    or1k->SR = value;

    // set the current branch flag, carry flag and overflow flag from the SR
    or1k->branchFlag   = GET_BIT(value, SPR_SR_F);
    or1k->carryFlag    = GET_BIT(value, SPR_SR_CY);
    or1k->overflowFlag = GET_BIT(value, SPR_SR_OV);
}
```

The function `or1kSetSR` extracts the flag bits from the new value of the status register `sr` and copies them into the flag fields in the processor model structure so that consistency is maintained.

8.3.3 Initializing Status Register - `or1kMain.c`

The constructor has been changed to initialize the new status register `sr`:

```
VMI_CONSTRUCTOR_FN(or1kConstructor) {

    or1kP or1k = (or1kP)processor;
    Uns32 i;

    // initialize general purpose registers
    for(i=2; i<OR1K_REGS; i++) {
        or1k->regs[i] = 0xdeadbeef;
    }

    // initialize status register SR
    or1k->SR = SPR_SR_FO | SPR_SR_SM;

    // create bus port specifications
    newBusPorts(or1k);
}
```

8.3.4 Dump Function Registration - `or1kAttrs.c`

The register dump routine has been added to the `vmiIASAttr` structure for the OR1K:

```
const vmiIASAttr modelAttrs = {

    ////////////////////////////////////////////////////////////////////////
    // VERSION & SIZE ATTRIBUTES
    ////////////////////////////////////////////////////////////////////////

    .versionString = VMI_VERSION,
    .modelType     = VMI_PROCESSOR_MODEL,
    .dictNames     = dictNames,
    .cpuSize       = sizeof(or1k),

    ////////////////////////////////////////////////////////////////////////
    // CREATE/DELETE ROUTINES
    ////////////////////////////////////////////////////////////////////////

    .constructorCB = or1kConstructor,
    .destructorCB  = or1kDestructor,

    ////////////////////////////////////////////////////////////////////////
    // MORPHER CORE ROUTINES
    ////////////////////////////////////////////////////////////////////////

    .morphCB = or1kMorphInstruction,

    ////////////////////////////////////////////////////////////////////////
    // SIMULATION SUPPORT ROUTINES
    ////////////////////////////////////////////////////////////////////////

    .getEndianCB = or1kGetEndian,
    .nextPCCB    = or1kNextInstruction,
    .disCB       = or1kDisassemble,

    ////////////////////////////////////////////////////////////////////////
    // REGISTER ACCESS SUPPORT ROUTINES (DEBUGGER & SEMIHOSTING)
    ////////////////////////////////////////////////////////////////////////

    .debugCB = or1kDumpRegisters,

    ////////////////////////////////////////////////////////////////////////
    // PORT ROUTINES
```

```
    ////////////////////////////////////////////////////////////////////////

    .busPortSpecsCB = or1kGetBusPortSpec,

    ////////////////////////////////////////////////////////////////////////
    // PROCESSOR INFO ROUTINE
    ////////////////////////////////////////////////////////////////////////

    .procInfoCB = or1kProcInfo
};
```

8.4 Validating Register Dumping with Point Tests

Run the platform using the assembler executable file:

```
platform/harness.$IMPERAS_ARCH.exe –-trace –-traceregs \
    --program application/asmtest.OR1K.elf
```

The output from this should be as follows:

```
Warning (PC_NRI) No register information callback given for processor 'cpu1'
Info 'cpu1', 0x0000000001000074: l.addi   r1,r0,0x0
Info 'cpu1' REGISTERS
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000000   R2 : deadbeef   R3 : deadbeef
 R4 : deadbeef   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : deadbeef   R10: deadbeef   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: deadbeef   R31: deadbeef
 BF:0 CF:0 OF:0  PC : 01000078   SR : 00008001
--------------- --------------- --------------- ---------------

Info 'cpu1', 0x0000000001000078: l.addi   r2,r0,0x1
Info 'cpu1' REGISTERS
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000000   R2 : 00000001   R3 : deadbeef
 R4 : deadbeef   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : deadbeef   R10: deadbeef   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: deadbeef   R31: deadbeef
 BF:0 CF:0 OF:0  PC : 0100007c   SR : 00008001
--------------- --------------- --------------- ---------------

… etc …
```

Now the trace output is much easier to understand because registers are printed with meaningful names.

8.5 Derived Flags

We have seen that the VMI API allows any of the *sign*, *carry*, *overflow*, *zero* or *parity* flags to be generated by an operation. It is often required to derive more complex flags from these: for example, it may be required to implement an unsigned below-or-equal condition flag, which is true if either the carry flag is set or the zero flag is set. The best approach is as follows:

1. generate the sign, carry, overflow, zero or parity flags as required as true processor registers;
2. use binary operations with width 8 to generate the derived flag using the basic flags as arguments as described below.

As an example, suppose that the OR1K model has been modified to implement sign and zero flags and a new temporary flag as follows:

```
#define OR1K_REGS 32            // basic OR1K registers
#define OR1K_BITS 32            // register size in bits

// processor structure
typedef struct or1kS {

    Bool       carryFlag;       // carry flag
    Bool       overflowFlag;    // overflow flag
    Bool       zeroFlag;        // carry flag
    Bool       signFlag;        // overflow flag
    Bool       branchFlag;      // branch flag
    Bool       tempFlag;        // temporary flag

    Uns32      regs[OR1K_REGS]; // basic registers

    vmiBusPortP busPorts;       // bus port descriptions

} or1k, *or1kP;
```

and that new accessor macros for these flags have been added:

```
// macros to specify target registers in VARIABLE expressions
#define OR1K_CPU_REG(_F)        VMI_CPU_REG(or1kP, _F)
#define OR1K_CPU_TEMP(_F)       VMI_CPU_TEMP(or1kP, _F)
#define OR1K_REG(_R)            OR1K_CPU_REG(regs[_R])
#define OR1K_CARRY              OR1K_CPU_REG(carryFlag)
#define OR1K_OVERFLOW           OR1K_CPU_REG(overflowFlag)
#define OR1K_ZERO               OR1K_CPU_REG(zeroFlag)
#define OR1K_SIGN               OR1K_CPU_REG(signFlag)
#define OR1K_TF                 OR1K_CPU_TEMP(tempFlag)
```

Given these changes, use the following sequences to generate a derived flag in `tempFlag`:

Unsigned below-or-equal `(CF==1) || (ZF==1)`:

```
vmimtBinopRRR(8, vmi_OR, OR1K_TF, OR1K_CARRY, OR1K_ZERO, 0);
```

Signed less-than `(SF!=OF)`:

```
vmimtBinopRRR(8, vmi_XOR, OR1K_TF, OR1K_SIGN, OR1K_OVERFLOW, 0);
```

Signed less-than-or-equal (`(ZF==1) || (SF!=OF)`):

```
vmimtBinopRRR(8, vmi_XOR, OR1K_TF, OR1K_SIGN, OR1K_OVERFLOW, 0);
vmimtBinopRR(8, vmi_OR, OR1K_TF, OR1K_ZERO, 0);
```

Complement of any flag:

```
vmimtBinopRRC(8, vmi_XOR, OR1K_TF, <flag_reg>, 1, 0);
```

Note that the recommended way to complement a flag is to exclusive-or it with 1.

9 Implementing Unconditional Jump Instructions

Up to this point, the OR1K examples have executed straight line code only. We will now implement unconditional jump instructions to allow simple non-linear programs to be run.

9.1 The Template Unconditional Jump Model

A template model for the OR1K processor implementing unconditional jump instructions can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/7.or1kBehaviorUncondJump
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/7.or1kBehaviorUncondJump .
```

Compile the model, harness and application using the `make` command:

```
cd 7.or1kBehaviorUncondJump
make
```

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is based on the previous model, with the changes listed in following sections.

9.1.1 Defining Link Register - `or1kStructure.h`

The OR1K has a *link register*, `r9`, which is set to required return address in a *jump-and-link* (call) instruction:

```
#define OR1K_LINK   9            // link register index (R9)
#define OR1K_LINKREG             OR1K_REG(OR1K_LINK)
```

9.1.2 Defining Jump Instruction Types - `or1kDecode.h`

The OR1K unconditional jump instructions have been added to the `or1kInstructionType` enumeration:

```
typedef enum or1kInstructionTypeE {

    // arithmetic instructions
    OR1K_IT_ADDI,
    OR1K_IT_ADDIC,
    OR1K_IT_ANDI,
    OR1K_IT_ORI,
    OR1K_IT_XORI,
    OR1K_IT_MULI,
```

```
    // branch instructions
    OR1K_IT_J,
    OR1K_IT_JR,
    OR1K_IT_JAL,
    OR1K_IT_JALR,

    // KEEP LAST: for sizing the array
    OR1K_IT_LAST

} or1kInstructionType;
```

9.1.3 Decoding Jump Instructions - `or1kDecode.c`

New macros have been added for extraction of a register index defined in bits 15:11 of an instruction and a signed offset from bits 25:0 of an instruction (the signed offset is multiplied by 4):

```
#define OP_B(_I)        WIDTH(5,(_I)>>11)
#define OP_T(_I)        (((Int32)(WIDTH(26,(_I)>>0)<<6))>>4)
```

The enumerations describing register and constant types have been enhanced:

```
typedef enum regSpecE {
    R_NA,                       // no register
    R_D,                        // register at bits 25:21
    R_A,                        // register at bits 20:16
    R_B,                        // register at bits 15:11
} regSpec;

//
// Define the location of constant in an instruction
//
typedef enum constSpecE {
    C_NA,                       // no register
    C_S1,                       // signed constant in bits 15:0
    C_U1,                       // unsigned constant in bits 15:0
    C_T,                        // signed target offset in bits 25:0
} constSpec;
```

The OR1K unconditional jump instructions have been added to the `attrsArray` table in a similar manner as for previous instructions:

```
const static opAttrs attrsArray[OR1K_IT_LAST+1] = {

    // handle arithmetic instructions (second argument constant)
    ATTR_SET_ADDI (ADDI,  "addi",  "|100111..........................|"),
    ATTR_SET_ADDI (ADDIC, "addic", "|101000..........................|"),
    ATTR_SET_ANDI (ANDI,  "andi",  "|101001..........................|"),
    ATTR_SET_ANDI (ORI,   "ori",   "|101010..........................|"),
    ATTR_SET_ADDI (XORI,  "xori",  "|101011..........................|"),
    ATTR_SET_ADDI (MULI,  "muli",  "|101100..........................|"),

    // handle branch instructions
    ATTR_SET_BF   (J,     "j",     "|000000..........................|"),
    ATTR_SET_JR   (JR,    "jr",    "|010001..........................|"),
    ATTR_SET_BF   (JAL,   "jal",   "|000001..........................|"),
    ATTR_SET_JR   (JALR,  "jalr",  "|010010..........................|"),
};
```

The new macros `ATTR_SET_BF` and `ATTR_SET_JR` describe jump instructions that take a relative offset and register argument, respectively:

```
#define ATTR_SET_BF(_NAME, _OPCODE, _DECODE) \
    [OR1K_IT_##_NAME] = {           \
        member : #_NAME,            \
        opcode : _OPCODE,           \
        format : FMT_TARGET,        \
        decode : _DECODE,           \
        c      : C_T                \
    }

#define ATTR_SET_JR(_NAME, _OPCODE, _DECODE) \
    [OR1K_IT_##_NAME] = {           \
        member : #_NAME,            \
        opcode : _OPCODE,           \
        format : FMT_R1,            \
        decode : _DECODE,           \
        r1     : R_B                \
    }
```

Function `getReg` has a new case to handle a register in the `rb` position in an instruction:

```
static Uns32 getReg(Uns32 instruction, regSpec rs) {

    Uns32 result = 0;

    switch(rs) {
        case R_NA:
            break;
        case R_D:
            result = OP_D(instruction);
            break;
        case R_A:
            result = OP_A(instruction);
            break;
        case R_B:
            result = OP_B(instruction);
            break;
        default:
            VMI_ABORT("unimplemented case"); // LCOV_EXCL_LINE
            break;
    }

    return result;
}
```

And function getConst now handles a constant target address (note that the offset from the instruction is added to the current PC to get the result in this case):

```
static Uns32 getConst(Uns32 instruction, constSpec cs, Uns32 thisPC) {

    Uns32 result = 0;

    switch(cs) {
        case C_NA:
            break;
        case C_S1:
```

```
            result = OP_S1(instruction);
            break;
        case C_U1:
            result = OP_U1(instruction);
            break;
        case C_T:
            // PC-relative address
            result = thisPC + OP_T(instruction);
            break;
        default:
            VMI_ABORT("unimplemented case"); // LCOV_EXCL_LINE
            break;
    }

    return result;
}
```

9.1.4 Jump Instruction Formats - `or1kDisassembleFormats.h`

A new token has been added for a target address, and new formats for instructions with a single register argument and an address argument:

```
//
// These are placeholders in disassembly decoder
//
#define EMIT_R1             '\001'
#define EMIT_R2             '\002'
#define EMIT_XIMM           '\003'
#define EMIT_TARGET         '\004'

//
// These are placeholders in disassembly format strings
//
#define EMIT_R1_S           "\001"
#define EMIT_R2_S           "\002"
#define EMIT_XIMM_S         "\003"
#define EMIT_TARGET_S       "\004"

//
// These are disassembly format strings
//
#define FMT_TARGET          EMIT_TARGET_S
#define FMT_R1              EMIT_R1_S
#define FMT_R1_R2_XIMM      EMIT_R1_S "," EMIT_R2_S "," EMIT_XIMM_S
```

9.1.5 Jump Instruction Disassembly - `or1kDisassemble.c`

A new function `putTarget` has been added to print a target address:

```
static void putTarget(char **result, Uns32 value) {

    char tmp[32];

    sprintf(tmp, "0x%08x", value);

    putString(result, tmp);
}
```

And function `disassembleFormat` has been enhanced to handle the new target address token:

```
static void disassembleFormat(
    or1kP                or1k,
    or1kInstructionInfoP info,
    char               **result,
    const char          *format,
    Bool                 uncooked
) {
    . . . lines omitted . . .

        // generate arguments in appropriate format
        while((ch=*format++)) {

            switch(ch) {

                case EMIT_R1:
                    putUncookedKey(result, " R1", uncooked);
                    putRegister(result, info->r1);
                    break;

                case EMIT_R2:
                    putUncookedKey(result, " R2", uncooked);
                    putRegister(result, info->r2);
                    break;

                case EMIT_XIMM:
                    putUncookedKey(result, " CX", uncooked);
                    putX(result, info->c);
                    break;

                case EMIT_TARGET:
                    putUncookedKey(result, " T", uncooked);
                    putTarget(result, info->c);
                    break;

                default:
                    if(!uncooked) {putChar(result, ch);}
                    break;
            }
        }
    }
}
```

9.1.6 Implementing Jump Instructions - `or1kMorph.c`

This file has been upgraded to implement morph callback functions for the jump instructions, as described below.

The main morpher entry point function, `or1kMorphInstruction`, has been modified to indicate whether the current instruction is in a delay slot:

```
typedef struct or1kMorphStateS {
    or1kInstructionInfo info;        // instruction description (from decoder)
    or1kMorphAttrCP     attrs;       // instruction attributes
    or1kP               or1k;        // current processor
    Bool                inDelaySlot; // true if instruction is in delay slot
} or1kMorphState;

VMI_MORPH_FN(or1kMorphInstruction) {

    or1kP          or1k  = (or1kP)processor;
```

```
    or1kMorphState state = {{0}};

    // decode instruction
    or1kDecode(or1k, thisPC, &state.info);

    // get morpher attributes for the decoded instruction and initialize other
    // state fields
    state.attrs       = &or1kMorphTable[state.info.type];
    state.or1k        = or1k;
    state.inDelaySlot = inDelaySlot;

    if(state.attrs->morphCB) {
        // translate the instruction
        state.attrs->morphCB(&state);
    } else {
        // here if no translation callback specified
        emitUnimplemented(&state);
    }
}
```

When performing just-in-time compilation using the model morph callback, the simulator always knows whether the current instruction is a delay slot instruction. It provides this information to the model morpher entry point function as an argument, `inDelaySlot`, of the call to the `VMI_MORPH_FN` of the model.

There are new entries in the or1kMorphTable array for the unconditional jumps:

```
const or1kMorphAttr or1kMorphTable[OR1K_IT_LAST+1] = {

    // handle arithmetic instructions (second argument constant)
    [OR1K_IT_ADDI]  = {morphCB:morphBinopRRC, binop:vmi_ADD,  flags:&flagsCO},
    [OR1K_IT_ADDIC] = {morphCB:morphBinopRRC, binop:vmi_ADC,  flags:&flagsCO},
    [OR1K_IT_ANDI]  = {morphCB:morphBinopRRC, binop:vmi_AND,  flags:0       },
    [OR1K_IT_ORI]   = {morphCB:morphBinopRRC, binop:vmi_OR,   flags:0       },
    [OR1K_IT_XORI]  = {morphCB:morphBinopRRC, binop:vmi_XOR,  flags:0       },
    [OR1K_IT_MULI]  = {morphCB:morphBinopRRC, binop:vmi_IMUL, flags:&flagsCO},

    // handle branch instructions
    [OR1K_IT_J]     = {morphCB:morphJump,    link:      False},
    [OR1K_IT_JR]    = {morphCB:morphJumpReg, link:      False},
    [OR1K_IT_JAL]   = {morphCB:morphJump,    link:      True },
    [OR1K_IT_JALR]  = {morphCB:morphJumpReg, link:      True },
};
```

A new link field in the `or1kMorphAttr` structure indicates whether this is *a jump-and-link* instruction. It is `True` for `l.jr` and `l.jalr` instructions:

```
typedef struct or1kMorphStateS {
    or1kInstructionInfo info;        // instruction description (from decoder)
    or1kMorphAttrCP     attrs;       // instruction attributes
    or1kP               or1k;        // current processor
    Bool                inDelaySlot; // true if instruction is in delay slot
} or1kMorphState;
```

9.1.6.1 Direct Unconditional Jump Instructions (`l.j` and `l.jal`)

The OR1K supports two direct unconditional jump instructions that we will implement now. Instruction `l.j` is a simple jump to a target address. Instruction `l.jal` is a jump-and-link instruction: there is a jump to a target address and a return address is saved in the link register (r9). Both these instructions are implemented with a single function: morphJump:

```
static OR1K_MORPH_FN(morphJump) {

    Uns32       toAddress   = state->info.c;
    Uns32       nextAddress = state->info.thisPC + 8;
    Bool        inDelaySlot = state->inDelaySlot;
    Bool        link        = state->attrs->link;
    vmiReg      linkReg     = link ? OR1K_LINKREG : VMI_NOREG;
    vmiJumpHint hint;

    // select an appropriate jump hint
    if(link) {
        hint = vmi_JH_CALL;
    } else {
        hint = vmi_JH_NONE;
    }

    if(inDelaySlot) {
        // jump in the delay slot does nothing
    } else {
        vmimtUncondJumpDelaySlot(
            1,                  // slotOps
            nextAddress,        // linkPC
            toAddress,          // toAddress
            linkReg,            // linkReg
            hint,               // hint
            0                   // slotCB
        );
    }
}
```

Whether the required instruction is a jump or a jump-and-link is specified by the link field of the or1kMorphAttr structure described previously. For these direct jumps, the jump target is calculated from the current instruction address plus a signed offset encoded in a field in the instruction, but this complexity is implemented in the decoder: the morpher callback is presented with a value indicating the full target address:

```
    Uns32       toAddress   = state->info.c;
```

The main work of doJump is in these lines:

```
    if(inDelaySlot) {
        // jump in the delay slot does nothing
    } else {
        vmimtUncondJumpDelaySlot(
            1,                  // slotOps
            nextAddress,        // linkPC
            toAddress,          // toAddress
            linkReg,            // linkReg
            hint,               // hint
            0                   // slotCB
        );
    }
```

If the current instruction is a delay slot instruction, *both `l.j` and `l.jal` have no effect*. It is therefore important that we know whether the current instruction is a delay slot so that appropriate action can be taken.

The morph-time function `vmimtUncondJumpDelaySlot` is used to describe the jump to the simulator. This function has six arguments:

1. `slotOps` is the number of instructions in the delay slot of this jump instruction. These OR1K instructions have one delay slot instruction. A value of 0 for `slotOps` specifies a jump with no delay slot instructions.
2. `linkPC` is used only if the jump is a jump-and-link, in which case it specifies the address that should be placed in the link register. For the OR1K, this is the address of the instruction after the delay slot instruction, i.e. `thisPC+8`.
3. `toAddress` is the jump target address.
4. `linkReg` is used to specify the link register for the jump, if this is a jump-and-link. If there is no link register (this is a simple jump), the value `VMI_NOREG` should be passed.
5. `hint` is used to help the simulator understand what kind of jump this is. In this chapter, we will see three values used:
    a. `vmi_JH_CALL`: the jump is a *call* to a function;
    b. `vmi_JH_RETURN`: the jump is a *return* from a function;
    c. `vmi_JH_NONE`: the jump is neither a call nor a return.

    Jump hints do not affect the behavior of a simulation but do improve performance (the example in later section 11.2.3 demonstrates this).
    In this function, the instruction `l.jal` has a call hint of `vmi_JH_CALL`, and instruction `l.j` has a call hint of `vmi_JH_NONE`.
6. `slotCB`, if non-`NULL`, specifies a *post-delay-slot callback* function, taking the current processor as its only argument. The function is called just before the delayed branch is taken. If the branch is *not* taken for any reason (for example, if there is a simulated exception in the delay slot instruction), then the callback is *not* called.
    The post-delay-slot callback is typically used to update processor state that should only be changed if the branch is taken. For example, if the instruction implements a switch to kernel mode then the state change reflecting this should typically be done in the post-delay-slot callback.

9.1.6.2 Indirect unconditional Jump Instructions (`l.jr` and `l.jalr`)

The OR1K also has two indirect conditional jump instructions. Instruction `l.jr` is a jump to a target address specified in a register. Instruction `l.jalr` is a jump-and-link instruction: there is a jump to a target address specified in a register, and a return address is saved in the link register (`r9`). Both these instructions are implemented with a single function: `morphJumpReg`:

```
static OR1K_MORPH_FN(morphJumpReg) {

    Uns32       r1          = state->info.r1;
    vmiReg      toReg       = getGPR(r1);
    Uns32       nextAddress = state->info.thisPC + 8;
```

```
    Bool        inDelaySlot = state->inDelaySlot;
    Bool        link        = state->attrs->link;
    vmiReg      linkReg     = link ? OR1K_LINKREG : VMI_NOREG;
    vmiJumpHint hint;

    // select an appropriate jump hint
    if(link) {
        hint = vmi_JH_CALL;
    } else if(r1==OR1K_LINK) {
        hint = vmi_JH_RETURN;
    } else {
        hint = vmi_JH_NONE;
    }

    if(inDelaySlot) {
        // jump in the delay slot does nothing
    } else {
        vmimtUncondJumpRegDelaySlot(
            1,                  // slotOps
            nextAddress,        // linkPC
            toReg,              // toReg
            linkReg,            // linkReg
            hint,               // hint
            0                   // slotCB
        );
    }
}
```

Whether the required instruction is a jump or a jump-and-link is again specified by the `link` field of the `or1kMorphAttr` structure described previously. For these indirect jumps, the jump target address is in a register encoded within the instruction:

```
    Uns32       r1          = state->info.r1;
    vmiReg      toReg       = getGPR(r1);
```

The main work of `doJumpReg` is in these lines:

```
    if(inDelaySlot) {
        // jump in the delay slot does nothing
    } else {
        vmimtUncondJumpRegDelaySlot(
            1,                  // slotOps
            nextAddress,        // linkPC
            toReg,              // toReg
            linkReg,            // linkReg
            hint,               // hint
            0                   // slotCB
        );
    }
```

Just as for direct jumps, indirect jumps have no effect in the delay slot of another jump.

The morph-time function `vmimtUncondJumpRegDelaySlot` is used to describe the jump to the simulator. This function has six arguments; all except the third argument are exactly the same as for `vmimtUncondJumpDelaySlot` (described in section 9.1.6.1). The third argument is used to specify the register containing the jump target address.

The jump hint to use with the indirect jump is determined as follows:

```
    // select an appropriate jump hint
    if(link) {
        hint = vmi_JH_CALL;
    } else if(r1==OR1K_LINK) {
        hint = vmi_JH_RETURN;
    } else {
        hint = vmi_JH_NONE;
    }
```

In other words, the jump hint indicates the type of the jump using the following rules:

1. if the this is a jump-and-link, then assume the jump is a *function call*;
2. otherwise, if this is an indirect jump using the OR1K link register (`r9`), then assume the jump is a *function return*;
3. otherwise, assume the jump is neither a call nor a return.

9.2 Validating Unconditional Jumps with Point Tests

Run the platform using the assembler executable file:

```
platform/harness.$IMPERAS_ARCH.exe –-trace \
    --program application/asmtest.OR1K.elf
```

The output from this should be as follows:

```
Warning (PC_NRI) No register information callback given for processor ‘cpu1’
Info 'cpu1', 0x0000000001000074: l.addi   r1,r0,0x0
Info 'cpu1', 0x0000000001000078: l.addi   r2,r0,0x0
Info 'cpu1', 0x000000000100007c: l.jal    0x01000090
Info 'cpu1', 0x0000000001000080: l.addi   r1,r1,0x1
Info 'cpu1', 0x0000000001000090: l.j      0x0100009c
Info 'cpu1', 0x0000000001000094: l.addi   r1,r1,0x1
Info 'cpu1', 0x000000000100009c: l.addi   r8,r9,0x0
Info 'cpu1', 0x00000000010000a0: l.jal    0x010000b8
Info 'cpu1', 0x00000000010000a4: l.addi   r1,r1,0x1
Info 'cpu1', 0x00000000010000b8: l.addi   r10,r9,0x4
Info 'cpu1', 0x00000000010000bc: l.jr     r9
Info 'cpu1', 0x00000000010000c0: l.addi   r1,r1,0x1
Info 'cpu1', 0x00000000010000a8: l.addi   r9,r8,0x0
Info 'cpu1', 0x00000000010000ac: l.jr     r9
Info 'cpu1', 0x00000000010000b0: l.addi   r1,r1,0x1
Info 'cpu1', 0x0000000001000084: l.jalr   r10
Info 'cpu1', 0x0000000001000088: l.addi   r1,r1,0x1
Info 'cpu1', 0x00000000010000ac: l.jr     r9
Info 'cpu1', 0x00000000010000b0: l.addi   r1,r1,0x1
Info 'cpu1', 0x000000000100008c: l.addi   r1,r1,0x0
Processor 'cpu1' terminated at 'exit', address 0x100008c
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000007   R2 : 00000000   R3 : deadbeef
 R4 : deadbeef   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : 01000084   R9 : 0100008c   R10: 010000ac   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: deadbeef   R31: deadbeef
 BF:0 CF:0 OF:0  PC : 01000090   SR : 00008001
```

```
-------------- -------------- -------------- --------------
```

The file `platform/harness.c` has been slightly modified from the previous example: at the end of the simulation, there is a call to `opProcessorRegDump` to display the final processor register state.

The test case `application/asmtest.S` is as follows:

```
        ////////////////////////////////////////////////////////////////////////////////
        // MAIN ROUTINE
        ////////////////////////////////////////////////////////////////////////////////
.global _start
_start:
        l.addi      r1,r0,0         // r1 = 0 (counts taken jumps)
        l.addi      r2,r0,0         // r2 = 0 (counts untaken jumps)
        l.jal       func1           // call func1
        l.addi      r1,r1,1         // increment r1 (delay slot instruction)
        l.jalr      r10             // call indirect using r10 (i.e. r10_addr)
        l.addi      r1,r1,1         // increment r1 (delay slot instruction)

        ////////////////////////////////////////////////////////////////////////////////
        // EXIT FROM POINT TEST
        ////////////////////////////////////////////////////////////////////////////////
.global exit
exit:
        l.addi      r1,r2,0

        ////////////////////////////////////////////////////////////////////////////////
        // FUNCTION func1
        ////////////////////////////////////////////////////////////////////////////////
func1:
        l.j         forward         // jump forward in this function
        l.addi      r1,r1,1         // increment r1 (delay slot instruction)
        l.addi      r2,r2,1         // ** not executed **
forward:
        l.addi      r8,r9,0         // save return address in r8
        l.jal       func2           // call func2
        l.addi      r1,r1,1         // increment r1 (delay slot instruction)
        l.addi      r9,r8,0         // restore return address from r8
r10_addr:
        l.jr        r9              // return from function
        l.addi      r1,r1,1         // increment r1 (delay slot instruction)
        l.addi      r2,r2,1         // ** not executed **

        ////////////////////////////////////////////////////////////////////////////////
        // FUNCTION func2
        ////////////////////////////////////////////////////////////////////////////////
func2:
        l.addi      r10,r9,4        // save return address+4 in r10
        l.jr        r9              // return from function
        l.addi      r1,r1,1         // increment r1 (delay slot instruction)
        l.addi      r2,r2,1         // ** not executed **
```

The test case has been designed to execute each of the jump instructions at least once. To exercise the `l.jalr` instruction, `func2` stores the address of label `r10_addr` in register `r10` to provide an appropriate target for the subsequent `l.jalr`.

10 Implementing Conditional Jump Instructions

In this chapter, we will implement comparison operations and conditional jumps for the OR1K. On this processor, conditional jumps are performed using two sets of instructions:

1. two registers (or a register and a constant) are compared using an instruction with the `l.sf` prefix (for example, `l.sfeq` compares two registers for equality). An internal *branch flag* is set based on the comparison result.
2. instructions `l.bf` and `l.bnf` then conditionally branch if the flag is true or false, respectively.

10.1 The Template Conditional Jump Model

A template model for the OR1K processor implementing conditional jump instructions can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/8.or1kBehaviorCondJump
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/8.or1kBehaviorCondJump .
```

Compile the model, harness and application using the `make` command:

```
cd 8.or1kBehaviorCondJump
make
```

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is based on the previous model, with the changes listed in following sections.

> Decoder, disassembler and morpher files have been updated in this example to implement register-register arithmetic/bitwise and `l.nop` instructions (previously, only register-constant arithmetic/bitwise instructions were implemented). These instructions require no significant new knowledge or techniques and are not directly relevant to this chapter, so no further detail has been given here.

10.1.1 Defining Branch Flag - `or1kStructure.h`

The OR1K branch flag is set by the comparison operations and used by the conditional branch instructions. The flag is represented by a boolean field in the `or1kS` structure, accessed by the `OR1K_BRANCH` macro:

```
#define OR1K_BRANCH            OR1K_CPU_REG(branchFlag)
```

10.1.2 Decoding Instructions - `or1kDecode.[ch]`

The OR1K branch and comparison instructions have been added to the `attrsArray` table in a very similar manner as for previous instructions.

The comparison operation decode differs from previous instructions: the decode for `OR1K_IT_SF` and `OR1K_IT_SFI` instructions match 16 distinct instruction types with a different `CMPOP` field, but only 10 of these comparison operations are valid. Therefore, to decode a comparison operation, both the decode pattern *and* an opcode validity test are applied. The comparison condition is extracted from the instruction by function `getCondition`:

```
or1kCondition getCondition(Uns32 instruction, Bool isCond) {

    or1kCondition result = OR1K_COND_NA;

    if(isCond) {

        const static or1kCondition map[16] = {
            OR1K_COND_EQ,   // 0x0: a == b
            OR1K_COND_NE,   // 0x1: a != b
            OR1K_COND_GTU,  // 0x2: a >  b (unsigned)
            OR1K_COND_GEU,  // 0x3: a >= b (unsigned)
            OR1K_COND_LTU,  // 0x4: a <  b (unsigned)
            OR1K_COND_LEU,  // 0x5: a <= b (unsigned)
            OR1K_COND_BAD,  // 0x6: invalid
            OR1K_COND_BAD,  // 0x7: invalid
            OR1K_COND_BAD,  // 0x8: invalid
            OR1K_COND_BAD,  // 0x9: invalid
            OR1K_COND_GTS,  // 0xa: a >  b (signed)
            OR1K_COND_GES,  // 0xb: a >= b (signed)
            OR1K_COND_LTS,  // 0xc: a <  b (signed)
            OR1K_COND_LES,  // 0xd: a <= b (signed)
            OR1K_COND_BAD,  // 0xe: invalid
            OR1K_COND_BAD   // 0xf: invalid
        };

        result = map[OP_CMPOP(instruction)];
    }

    return result;
};
```

Function `or1kDecode` validates the condition and sets the instruction type to `OR1K_IT_LAST` if it is bad (indicating that decode failed):

```
void or1kDecode(or1kP or1k, Uns32 thisPC, or1kInstructionInfoP info) {

    . . . lines omitted . . .

    info->cond        = getCondition(instruction, attrs->isCond);
    info->iSuffix     = attrs->iSuffix;

    // detect invalid condition and indicate that decode failed if found
    if(info->cond==OR1K_COND_BAD) {
        info->type = OR1K_IT_LAST;
    }
}
```

10.1.3 Disassembling Conditions - `or1kDisassemble.c`

This file has been upgraded to implement disassembly callback functions for the comparison and branch instructions. There is a new function `putCondition` which adds a condition description to the disassembly string:

```
static void putCondition(char **result, or1kCondition cond) {

    static const char *map[] = {
        [OR1K_COND_NA]  = "",
        [OR1K_COND_EQ]  = "eq",
        [OR1K_COND_NE]  = "ne",
        [OR1K_COND_GTU] = "gtu",
        [OR1K_COND_GEU] = "geu",
        [OR1K_COND_LTU] = "ltu",
        [OR1K_COND_LEU] = "leu",
        [OR1K_COND_GTS] = "gts",
        [OR1K_COND_GES] = "ges",
        [OR1K_COND_LTS] = "lts",
        [OR1K_COND_LES] = "les",
        [OR1K_COND_BAD] = "???",
    };

    putString(result, map[cond]);
}
```

The condition string is actually embedded within the opcode instead of being a parameter to it, and may be followed by an `i` character (for example, `l.sfnei`). This is handled as follows:

```
//
// Emit "i" suffix if required
//
static void putISuffix(char **result, Bool iSuffix) {

    if(iSuffix) {
        putChar(result, 'i');
    }
}

static void disassembleFormat(
    or1kP                or1k,
    or1kInstructionInfoP info,
    char               **result,
    const char          *format,
    Bool                 uncooked
) {
    char *argStart = (*result)+9;
    char  ch;

    // emit opcode
    putString(result, "l.");
    putString(result, info->opcode);
    putCondition(result, info->cond);
    putISuffix(result, info->iSuffix);

    . . . lines omitted . . .
}
```

10.1.4 Implementing Conditional Jumps - `or1kMorph.c`

This file has been upgraded to implement morph callback functions for the comparison and branch instructions, as described below.

10.1.4.1 Conditional Branch Instructions (`l.bf` and `l.bnf`)

The two branch instructions are implemented with a single function: `morphBranch`:

```
static OR1K_MORPH_FN(morphBranch) {

    Uns32 toAddress   = state->info.c;
    Bool  jumpIfTrue  = state->attrs->jumpIfTrue;
    Bool  inDelaySlot = state->inDelaySlot;

    if(inDelaySlot) {
        // jump in the delay slot does nothing
    } else {
        vmimtCondJumpDelaySlot(
            1,                  // slotOps
            OR1K_BRANCH,        // flagReg
            jumpIfTrue,         // jumpIfTrue
            0,                  // linkPC
            toAddress,          // toAddress
            VMI_NOREG,          // linkReg
            vmi_JH_NONE,        // hint
            0                   // slotCB
        );
    }
}
```

Whether the branch should be taken when the flag is set or cleared is specified by the new `jumpIfTrue` field in the `or1kMorphAttr` structure:

```
typedef struct or1kMorphAttrS {
    or1kMorphFn morphCB;    // function to translate one instruction
    vmiBinop    binop;      // if a simple binary operation
    vmiFlagsCP  flags;      // flags used by the instruction
    Bool        jumpIfTrue; // whether to branch if condition flag is True
    Bool        link;       // whether to set link register
} or1kMorphAttr;

const or1kMorphAttr or1kMorphTable[OR1K_IT_LAST+1] = {

    . . . lines omitted . . .

    // handle branch instructions
    [OR1K_IT_BF]    = {morphCB:morphBranch,  jumpIfTrue:True },
    [OR1K_IT_BNF]   = {morphCB:morphBranch,  jumpIfTrue:False},

    . . . lines omitted . . .
};
```

The main work of `morphBranch` is in these lines:

```
    if(inDelaySlot) {
        // jump in the delay slot does nothing
    } else {
        vmimtCondJumpDelaySlot(
            1,                  // slotOps
```

```
            OR1K_BRANCH,          // flagReg
            jumpIfTrue,           // jumpIfTrue
            0,                    // linkPC
            toAddress,            // toAddress
            VMI_NOREG,            // linkReg
            vmi_JH_NONE,          // hint
            0                     // slotCB
        );
    }
```

If the current instruction is a delay slot instruction, both `l.bf` and `l.bnf` have no effect. Otherwise, the morph-time function `vmimtCondJumpDelaySlot` is used to describe the jump to the simulator. This function has eight arguments:

1. `slotOps` is the number of instructions in the delay slot of this jump instruction.
2. `flagReg` specifies a register in the processor model that is used to determine whether the branch is taken. Here, we use the OR1K branch flag.
3. `jumpIfTrue` indicates how the branch register is used. If `jumpIfTrue` is non-zero, the jump will be taken if the branch register is non zero. Otherwise, the jump will be taken if the branch register is zero.
4. `linkPC` is used only if the jump is a jump-and-link, in which case it specifies the address that should be placed in the link register. This does not apply for OR1K conditional branches.
5. `toAddress` is the jump target address.
6. `linkReg` is used to specify the link register for the jump, if this is a jump-and-link. If there is no link register (as in this case), the value `VMI_NOREG` should be passed.
7. `hint` is used to help the simulator understand what kind of jump this is – see chapter 9 for more details.
8. `slotCB`, if non-`NULL`, specifies a *post-delay-slot callback* function, taking the current processor as its only argument. The function is called just before the delayed branch is taken. If the branch is *not* taken, then the callback is *not* called. The post-delay-slot callback is typically used to update processor state that should only be changed if the branch is taken. For example, if the instruction implements a switch to kernel mode then the state change reflecting this should typically be done in the post-delay-slot callback.

10.1.4.2 Comparison Instructions

The comparison instructions are implemented with `morphCompareRR` (for register-register comparisons) and `morphCompareRC` (for register-constant comparisons):

```
static OR1K_MORPH_FN(morphCompareRC) {

    vmiReg       ra   = getGPR(state->info.r1);
    Uns32        c    = state->info.c;
    vmiCondition cond = mapCondition(state->info.cond);

    vmimtCompareRC(OR1K_BITS, cond, ra, c, OR1K_BRANCH);
}

static OR1K_MORPH_FN(morphCompareRR) {

    vmiReg       ra   = getGPR(state->info.r1);
    vmiReg       rb   = getGPR(state->info.r2);
```

```
    vmiCondition cond = mapCondition(state->info.cond);

    vmimtCompareRR(OR1K_BITS, cond, ra, rb, OR1K_BRANCH);
}
```

The equivalent vmiCondition for an or1kCondition is produced by function mapCondition:

```
static vmiCondition mapCondition(or1kCondition cond) {

    static const vmiCondition map[] = {
        [OR1K_COND_EQ]  = vmi_COND_Z,   // 0x0: a == b
        [OR1K_COND_NE]  = vmi_COND_NZ,  // 0x1: a != b
        [OR1K_COND_GTU] = vmi_COND_NBE, // 0x2: a >  b (unsigned)
        [OR1K_COND_GEU] = vmi_COND_NB,  // 0x3: a >= b (unsigned)
        [OR1K_COND_LTU] = vmi_COND_B,   // 0x4: a <  b (unsigned)
        [OR1K_COND_LEU] = vmi_COND_BE,  // 0x5: a <= b (unsigned)
        [OR1K_COND_GTS] = vmi_COND_NLE, // 0xa: a >  b (signed)
        [OR1K_COND_GES] = vmi_COND_NL,  // 0xb: a >= b (signed)
        [OR1K_COND_LTS] = vmi_COND_L,   // 0xc: a <  b (signed)
        [OR1K_COND_LES] = vmi_COND_LE,  // 0xd: a <= b (signed)
        [OR1K_COND_BAD] = -1,           // invalid condition
    };

    return map[cond];
}
```

Function `morphCompareRR` uses the morph-time function `vmimtCompareRC` to describe the jump to the simulator. This function has five arguments:

1. `bits` is the bit width of the registers to be compared. All OR1K registers are `OR1K_BITS` bits wide (32 in this model).
2. `cond` describes the comparison that should be made. The members of the `vmiCondition` enumeration are specified in `vmiTypes.h`.
3. `ra` specifies the first register argument of the comparison
4. `c` specifies the second constant argument of the comparison
5. `flag` specifies the `Uns8` register that should be written with 1 if the condition is true and 0 if it is false. In this model, the `OR1K_BRANCH` register is written.

Function `morphCompareRR` is similar, except that it uses the morph-time function `vmimtCompareRR` to describe the jump to the simulator. This function takes identical arguments to `vmimtCompareRC` except that argument 4 is a `vmiReg` register description instead of a constant.

10.2 Validating Conditional Jumps with Point Tests

Run the platform using the assembler executable file:

```
platform/harness.$IMPERAS_ARCH.exe –-trace \
    --program application/asmtest.OR1K.elf
```

The output from this should be as follows:

```
Warning (PC_NRI) No register information callback given for processor ‘cpu1’
```

```
Info 'cpu1', 0x0000000001000074: l.addi   r1,r0,0x3
Info 'cpu1', 0x0000000001000078: l.addi   r2,r0,0x2
Info 'cpu1', 0x000000000100007c: l.jal    0x01000108
Info 'cpu1', 0x0000000001000080: l.nop    0x0
Info 'cpu1', 0x0000000001000108: l.addi   r29,r0,0x0
Info 'cpu1', 0x000000000100010c: l.addi   r30,r0,0x1
Info 'cpu1', 0x0000000001000110: l.addi   r31,r0,0x1
Info 'cpu1', 0x0000000001000114: l.sfeq   r1,r2
Info 'cpu1', 0x0000000001000118: l.bf     0x01000124
Info 'cpu1', 0x000000000100011c: l.xor    r29,r29,r31
Info 'cpu1', 0x0000000001000120: l.xor    r29,r29,r31
Info 'cpu1', 0x0000000001000124: l.bnf    0x01000130
Info 'cpu1', 0x0000000001000128: l.xor    r30,r30,r31
Info 'cpu1', 0x0000000001000130: l.add    r31,r31,r31
Info 'cpu1', 0x0000000001000134: l.sfne   r1,r2
Info 'cpu1', 0x0000000001000138: l.bf     0x01000144
Info 'cpu1', 0x000000000100013c: l.xor    r29,r29,r31
Info 'cpu1', 0x0000000001000144: l.bnf    0x01000150
Info 'cpu1', 0x0000000001000148: l.xor    r30,r30,r31
Info 'cpu1', 0x000000000100014c: l.xor    r30,r30,r31
Info 'cpu1', 0x0000000001000150: l.add    r31,r31,r31
. . . etc . . .
Info 'cpu1', 0x0000000001000234: l.sfles  r1,r2
Info 'cpu1', 0x0000000001000238: l.bf     0x01000244
Info 'cpu1', 0x000000000100023c: l.xor    r29,r29,r31
Info 'cpu1', 0x0000000001000240: l.xor    r29,r29,r31
Info 'cpu1', 0x0000000001000244: l.bnf    0x01000250
Info 'cpu1', 0x0000000001000248: l.xor    r30,r30,r31
Info 'cpu1', 0x0000000001000250: l.add    r31,r31,r31
Info 'cpu1', 0x0000000001000254: l.jr     r9
Info 'cpu1', 0x0000000001000258: l.nop    0x0
Info 'cpu1', 0x00000000010000fc: l.addi   r14,r29,0x0
Info 'cpu1', 0x0000000001000100: l.addi   r15,r30,0x0
Info 'cpu1', 0x0000000001000104: l.nop    0x0
Processor 'cpu1' terminated at 'exit', address 0x1000104
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : fffffffd   R2 : fffffffc   R3 : 000000ce
 R4 : 00000330   R5 : 000002a9   R6 : 00000157   R7 : 00000332
 R8 : 000000cc   R9 : 010000fc   R10: 00000332   R11: 000000cc
 R12: 000002a9   R13: 00000157   R14: 000000ce   R15: 00000330
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: 000000ce   R30: 00000330   R31: 00000400
 BF:0 CF:0 OF:0  PC : 01000108   SR : 00008001
--------------- --------------- --------------- ---------------
```

The test case `application/asmtest.S` is as follows:

```
        ////////////////////////////////////////////////////////////////////////////
        // MAIN ROUTINE
        ////////////////////////////////////////////////////////////////////////////
.global _start
_start:
        // test1: r1=3, r2=2
        l.addi      r1,r0,3         // r1=3
        l.addi      r2,r0,2         // r2=2
        l.jal       test            // call test
        l.nop                       // (delay slot instruction)
        l.addi      r3,r29,0        // move bf taken mask to r3
        l.addi      r4,r30,0        // move bnf taken mask to r4
```

```
        . . . etc . . .

        // test6: r1=-3, r2=-4
        l.addi      r1,r0,-3        // r1=-3
        l.addi      r2,r0,-4        // r2=-4
        l.jal       test            // call test
        l.nop                       // (delay slot instruction)
        l.addi      r14,r29,0       // move bf taken mask to r14
        l.addi      r15,r30,0       // move bnf taken mask to r15

        ////////////////////////////////////////////////////////////////////////
        // EXIT FROM POINT TEST
        ////////////////////////////////////////////////////////////////////////
.global exit
exit:
        l.nop

        ////////////////////////////////////////////////////////////////////////
        // FUNCTION test
        ////////////////////////////////////////////////////////////////////////
test:   l.addi      r29,r0,0        // clear output mask r29 (bf taken)
        l.addi      r30,r0,1        // clear output mask r30 (bnf taken)
        l.addi      r31,r0,1        // initialize bitmask

        // test for sfeq
        l.sfeq      r1,r2           // r1==r2?
        l.bf        sfeqF           // go if true
        l.xor       r29,r29,r31     // add mask (delay slot instruction)
        l.xor       r29,r29,r31     // remove mask
sfeqF:  l.bnf       sfeqNF          // go if false
        l.xor       r30,r30,r31     // add mask (delay slot instruction)
        l.xor       r30,r30,r31     // remove mask
sfeqNF: l.add       r31,r31,r31     // shift mask

        . . . etc . . .

        // test for sfles
        l.sfles     r1,r2           // r1<=r2? (signed)
        l.bf        sflesF          // go if true
        l.xor       r29,r29,r31     // add mask (delay slot instruction)
        l.xor       r29,r29,r31     // remove mask
sflesF: l.bnf       sflesNF         // go if false
        l.xor       r30,r30,r31     // add mask (delay slot instruction)
        l.xor       r30,r30,r31     // remove mask
sflesNF:l.add       r31,r31,r31     // shift mask

        l.jr        r9              // return, results in r29 and r30
        l.nop                       // (delay slot instruction)
```

The test case has been design to exercise all register-register comparison instructions, with a variety of input operands, and build up masks indicating how the comparison results are treated by both the `l.bf` and `l.bnf` instructions. For example, this is an instruction sequence that is executed when function `test` is called for the first time, when `r1`=3 and `r2`=2:

```
Info 'cpu1', 0x0000000001000114: l.sfeq   r1,r2        // test for equality
Info 'cpu1', 0x0000000001000118: l.bf     0x01000124   // branch if equal
Info 'cpu1', 0x000000000100011c: l.xor    r29,r29,r31  // (delay slot insn)
Info 'cpu1', 0x0000000001000120: l.xor    r29,r29,r31  // *** branch NOT taken
```

```
Info 'cpu1', 0x0000000001000124: l.bnf    0x01000130   // branch if not equal
Info 'cpu1', 0x0000000001000128: l.xor    r30,r30,r31  // (delay slot insn)
Info 'cpu1', 0x0000000001000130: l.add    r31,r31,r31  // *** branch TAKEN
```

In this example, the delay slot instruction is executed whether the branch is taken or not. It is also possible to describe branches that *annul* the delay slot instruction if the branch is not taken – refer to these routines in the *Imperas VMI Morph Time Reference* manual for more information:

```
vmimtCondJumpDelaySlotAnnul
vmimtCondJumpRegDelaySlotAnnul
vmimtSkipIfAnnul
```

11 Implementing Memory Access Instructions

In this chapter, we will implement memory load and store instructions for the OR1K. The processor supports six load instructions:

1. `l.lwz`: load 4 bytes; zero extend to 32 bits;
2. `l.lws`: load 4 bytes, sign extend to 32 bits (same as `l.lwz` on 32-bit core);
3. `l.lhz`: load 2 bytes, zero extend to 32 bits;
4. `l.lhs`: load 2 bytes, sign extend to 32 bits;
5. `l.lbz`: load 1 byte, zero extend to 32 bits;
6. `l.lbs`: load 1 byte, sign extend to 32 bits.

There are three store instructions:

1. `l.sw`: store 4 bytes;
2. `l.sh`: store 2 bytes;
3. `l.sb`: store 1 byte.

Accesses longer than one byte must be aligned with memory, otherwise the access generates an alignment exception – chapter 12 shows how this requirement can be modeled efficiently.

11.1 The Template Memory Access Model

A template model for the OR1K processor implementing memory access instructions can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/9.or1kBehaviorLoadStore
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/9.or1kBehaviorLoadStore.
```

Compile the model, harness and application using the `make` command:

```
cd 9.or1kBehaviorLoadStore
make
```

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is based on the previous model, with the changes listed in following sections.

> Decoder, disassembler and morpher files have been updated in this example to implement shift/rotate and `l.movhi` instructions. The shift/rotate instructions are very similar to the arithmetic/bitwise instructions discussed previously. Instruction `l.movhi` implements a form of constant load to the high part of a register, common on RISC processors. These instructions require no significant new knowledge or

techniques and are not directly relevant to this chapter, so no further detail has been given here.

11.1.1 Decoding Loads and Stores - `or1kDecode.c`

The `opAttrs` type now contains fields giving the number of bytes for a load or store and (for a load) whether zero or sign extension is required:

```
typedef enum or1kExtendE {
    OR1K_EXTEND_NA,             // no extension
    OR1K_EXTEND_S,              // sign extension
    OR1K_EXTEND_Z,              // zero extension
} or1kExtend;

typedef struct opAttrsS {
    const char *member;         // enumeration member name
    const char *opcode;         // opcode name
    const char *format;         // format string
    const char *decode;         // decode string
    regSpec     r1      : 8;    // register 1 specification
    regSpec     r2      : 8;    // register 2 specification
    regSpec     r3      : 8;    // register 3 specification
    constSpec   c       : 8;    // constant value specification
    Uns32       bytes   : 8;    // load/store size specification
    or1kExtend  extend  : 8;    // load sign extension
    Bool        isCond  : 1;    // is instruction conditional?
    Bool        iSuffix : 1;    // does opcode require 'i' suffix?
} opAttrs, *opAttrsP;
```

The variants of load and store instruction are specified using the `ATTR_SET_LWZ` and `ATTR_SET_SW` macros, respectively:

```
#define ATTR_SET_LWZ(_NAME, _OPCODE, _DECODE, _BYTES, _EXT) \
    [OR1K_IT_##_NAME] = {           \
        member : #_NAME,            \
        opcode : _OPCODE,           \
        format : FMT_R1_OFFSET_R2,  \
        decode : _DECODE,           \
        r1     : R_D,               \
        r2     : R_A,               \
        c      : C_S1,              \
        bytes  : _BYTES,            \
        extend : OR1K_EXTEND_##_EXT \
    }

#define ATTR_SET_SW(_NAME, _OPCODE, _DECODE, _BYTES) \
    [OR1K_IT_##_NAME] = {           \
        member : #_NAME,            \
        opcode : _OPCODE,           \
        format : FMT_OFFSET_R1_R2,  \
        decode : _DECODE,           \
        r1     : R_A,               \
        r2     : R_B,               \
        c      : C_S2,              \
        bytes  : _BYTES             \
    }

const static opAttrs attrsArray[OR1K_IT_LAST+1] = {
```

```
    . . . lines omitted . . .

    // handle load instructions
    ATTR_SET_LWZ  (LWZ,   "l",     "|100001..........................|", 4, Z),
    ATTR_SET_LWZ  (LWS,   "l",     "|100010..........................|", 4, S),
    ATTR_SET_LWZ  (LBZ,   "l",     "|100011..........................|", 1, Z),
    ATTR_SET_LWZ  (LBS,   "l",     "|100100..........................|", 1, S),
    ATTR_SET_LWZ  (LHZ,   "l",     "|100101..........................|", 2, Z),
    ATTR_SET_LWZ  (LHS,   "l",     "|100110..........................|", 2, S),

    // handle store instructions
    ATTR_SET_SW   (SW,    "s",     "|110101..........................|", 4),
    ATTR_SET_SW   (SB,    "s",     "|110110..........................|", 1),
    ATTR_SET_SW   (SH,    "s",     "|110111..........................|", 2),

    . . . lines omitted . . .
};
```

The OR1K has six load instructions, which load 1, 2 and 4 byte values either with or without sign extension. Similarly, there are three store instructions that store 1, 2 and 4 byte values. In the decode table, these are specified using opcodes "l" and "s", respectively; the full opcode is constructed in the disassembler taking into account the bytes and extend values specified by the decoder (see the next subsection).

The `or1kInstructionInfo` structure also has new `bytes` and `extend` fields, which get initialized from the instruction attributes in function `or1kDecode`:

```
void or1kDecode(or1kP or1k, Uns32 thisPC, or1kInstructionInfoP info) {

    . . . lines omitted . . .

    info->bytes       = attrs->bytes;
    info->extend      = attrs->extend;

    . . . lines omitted . . .

}
```

11.1.2 Load/Store Disassembly - `or1kDisassemble.c`

New function `putBytes` writes an opcode character `b`, `h` and `w` (indicating load store size):

```
static void putBytes(char **result, Uns32 bytes) {

    static const char map[] = {
        [1] = 'b',
        [2] = 'h',
        [4] = 'w'
    };

    if(map[bytes]) {
        putChar(result, map[bytes]);
    }
}
```

Similarly, new function `putExtend` writes an opcode character `s` or `z` (indicating sign or zero extension):

```
static void putExtend(char **result, or1kExtend extend) {

    static const char map[] = {
        [OR1K_EXTEND_S] = 's',
        [OR1K_EXTEND_Z] = 'z',
    };

    if(map[extend]) {
        putChar(result, map[extend]);
    }
}
```

These functions are used to modify the generated opcode as follows:

```
static void disassembleFormat(
    or1kP                or1k,
    or1kInstructionInfoP info,
    char               **result,
    const char          *format,
    Bool                 uncooked
) {
    char *argStart = (*result)+9;
    char  ch;

    // emit opcode
    putString(result, "l.");
    putString(result, info->opcode);
    putCondition(result, info->cond);
    putISuffix(result, info->iSuffix);
    putBytes(result, info->bytes);
    putExtend(result, info->extend);

    . . . lines omitted . . .
}
```

By generating the decoded output from the instruction attributes in this way, model robustness is improved: any error in size or extension decode will be evident because *instruction disassembly will be incorrect*.

11.1.3 Implementing Loads and Stores - `or1kMorph.c`

This file has been upgraded as described below.

11.1.3.1 Load Instructions

The six load instructions are implemented with a single function: `morphLoad`:

```
static OR1K_MORPH_FN(morphLoad) {

    vmiReg    rd         = getGPR(state->info.r1);
    vmiReg    ra         = getGPR(state->info.r2);
    Int32     offset     = state->info.c;
    Uns32     bytes      = state->info.bytes;
    Bool      signExtend = (state->info.extend==OR1K_EXTEND_S);
    memEndian endian     = getEndian();

    vmimtLoadRRO(
        OR1K_BITS,              // destBits
```

```
        bytes*8,                // memBits
        offset,                 // offset
        rd,                     // rd
        ra,                     // ra
        endian,                 // endian
        signExtend,             // signExtend
        MEM_CONSTRAINT_ALIGNED  // constraint
    );
}
```

The size of the load in bytes and whether sign extension is required are extracted from the decoded instruction attributes:

```
    Int32     offset     = state->info.c;
    Uns32     bytes      = state->info.bytes;
```

Each load is specified by a call to the function `vmimtLoadRRO` from the Imperas Morph Time Function API. This takes eight arguments, as follows:

1. `destBits`: the size in bits of the destination register for the load;
2. `memBits`: the size in bits of the value in memory;
3. `offset`: a constant offset to be added to the address register `ra` to give the full memory address;
4. `rd`: the destination register for the load (if `rd` is `VMI_NOREG`, the load is performed but the fetched value discarded);
5. `ra`: a register holding the address from which to load (or `VMI_NOREG` if the load is from an address specified by `offset` only);
6. `endian`: the endianness of the load. This can be either `MEM_ENDIAN_BIG` or `MEM_ENDIAN_LITTLE`.
7. `signExtend`: whether the memory value should be assign extended if smaller than the register – if `False`, then the value is zero extended.
8. `constraint`: what *constraints* should be placed on the memory access. In this case, the value `MEM_CONSTRAINT_ALIGNED` is used to specify that the memory access must be aligned to the data size, and any unaligned access will either cause simulation to terminate or a simulated exception to be taken: this is described in chapter 12.

For the OR1K processor, the address from which to load is calculated by adding address register `ra` to the constant value `c` from the instruction.

The endianness of the load is specified by function `getEndian`. This model supports big-endian only:

```
static memEndian getEndian(void) {
    return MEM_ENDIAN_BIG;
}
```

11.1.3.2 Store Instructions

The three store instructions are implemented with a single function: `morphStore`:

```
static OR1K_MORPH_FN(morphStore) {
```

```
    vmiReg    ra     = getGPR(state->info.r1);
    vmiReg    rb     = getGPR(state->info.r2);
    Int32     offset = state->info.c;
    Uns32     bytes  = state->info.bytes;
    memEndian endian = getEndian();

    vmimtStoreRRO(
        bytes*8,                // bits
        offset,                 // offset
        ra,                     // ra
        rb,                     // rb
        endian,                 // endian
        MEM_CONSTRAINT_ALIGNED  // constraint
    );
}
```

The size of the store in bytes is extracted from the decoded instruction attributes in a similar way as for function `morphLoad`.

A store of any register except `r0` is specified using `vmimtStoreRRO` from the Imperas Morph Time Function API. This takes six arguments, as follows:

1. `bits`: the size in bits of the destination register to be stored;
2. `offset`: a constant offset to be added to the address register `ra` to give the full memory address;
3. `ra`: a register holding the address to which to store (or `VMI_NOREG` if the store is to an address specified by `offset` only);
4. `rb`: the register to be stored;
5. `endian`: the endianness of the store. This can be either `MEM_ENDIAN_BIG` or `MEM_ENDIAN_LITTLE`.
6. `constraint`: what *constraints* should be placed on the memory access. In this case, the value `MEM_CONSTRAINT_ALIGNED` is used to specify that the memory access must be aligned to the data size, and any unaligned access will either cause simulation to terminate or a simulated exception to be taken: this is described in chapter 12.

11.1.4 Load/Store Test Harness - `platform/harness.c`

The test platform for this example, `platform/harness.c`, has been changed as follows:

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

    // initialize simulation session before calling any other OP functions
    opSessionInit(OP_VERSION);

    // check arguments
    if(!cmdParser(argc, argv)) {
       opMessage("E", "CLI", "Command Line parser error");
       return 1;
    }

    // create root module, enabling simulation interruption if Ctrl-C is pressed
```

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

    // create a processor instance
    const char *modelFile = "model."IMPERAS_SHRSUF;
    optProcessorP processor = opProcessorNew(mr, modelFile, "cpu1", 0, 0);

    // get semihost library to exit simulation
    const char *semihostFile = opVLNVString(
        0,
        "ovpworld.org",
        "modelSupport",
        "imperasExit",
        "1.0",
        OP_EXTENSION,
        True
    );

    // attach imperasExit semihost library to processor
    opProcessorExtensionNew(processor, semihostFile, "imperasExit", 0);

    // create the processor bus
    optBusP bus = opBusNew(mr, "bus", 32, 0, 0);

    // connect processor instruction and data ports to the common bus
    opProcessorBusConnect(processor, bus, "INSTRUCTION");
    opProcessorBusConnect(processor, bus, "DATA");

    // create memory
    optMemoryP memory = opMemoryNew(mr, "local", OP_PRIV_RWX, 0xffffffff, 0, 0);

    // connect the memory onto the busses
    opMemoryBusConnect(memory, bus, "mp1", 0x00000000, 0xffffffff);

    // run processor, one instruction at a time
    while(simulate(processor, -1)) {
        // keep going while processor is still running
    }

    // dump the final register contents
    opProcessorRegDump(processor);

    // report the total number of instructions executed
    opPrintf(
        "processor has executed " FMT_64u " instructions\n",
        opProcessorICount(processor)
    );

    // terminate the simulation session
    opSessionTerminate();

    return 0;
}
```

The significant change is in the call to function `simulate`:

```
    while(simulate(processor, -1)) {
```

Previously, each call to `simulate` requested a single instruction to be executed. In this example, we use the value -1 instead, which indicates that the simulator can execute an unlimited number of instructions before returning[3]. In this case, this means that the call will only return when the program has completed.

11.2 Fibonacci Example

To demonstrate the load and store functions, we will use an assembler program that calculates Fibonacci numbers[4] using a naive recursive algorithm (normally, instruction point tests should be created and tested first, or course). Once the basic example is working, we will use it to demonstrate simulator performance and the effect of *jump hints* (first encountered in chapter 9).

11.2.1 Basic Example

Run the platform using the assembler executable file:

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

The output from this should be as follows:

```
Warning (PC_NRI) No register information callback given for processor ‘cpu1’
Processor 'cpu1' terminated at 'exit', address 0x1000080
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000262   R2 : 00000179   R3 : deadbeef
 R4 : deadbeef   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : 01000080   R10: deadbeef   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: deadbeef   R31: 00000000
 BF:1 CF:1 OF:0  PC : 01000084   SR : 00008601
--------------- --------------- --------------- ---------------

processor has executed 22687 instructions
```

The test case `application/asmtest.S` is as follows:

```
        ////////////////////////////////////////////////////////////////////
        // MAIN ROUTINE
        ////////////////////////////////////////////////////////////////////
.global _start
_start:
        l.addi      r31,r0,0        // initialize stack pointer to 0
        l.jal       fib             // calculate fib(15)
        l.addi      r1,r0,15        // r1 = 15 (delay slot)
```

[3] To be precise, the second argument to `opProcessorSimulate` (and `simulate`) is an `Uns64`, so a value of -1 (sign-extended to 64 bits) in fact specifies that 2^{64}-1 instructions should be executed – a very large number, but not quite unlimited.

[4] Fibonacci numbers are defined as follows:

1. for N <= 1: fib(N) = N;
2. for N > 1: fib(N) = fib(N-1) + fib(N-2)

```
        ////////////////////////////////////////////////////////////////////////
        // EXIT FROM POINT TEST
        ////////////////////////////////////////////////////////////////////////
.global exit
exit:
        l.nop

        ////////////////////////////////////////////////////////////////////////
        // FUNCTION fib - calculate Fibonacci number of N, passed in r1.
        //                result is returned in r1, r2 is destroyed
        ////////////////////////////////////////////////////////////////////////
fib:    l.sflesi    r1,1            // r1<=1? (signed)
        l.bf        done            // done if so, result is r1
        l.nop                       // (delay slot)

        l.addi      r31,r31,-12     // create stack frame
        l.sw        0(r31),r9       // save link register
        l.sw        4(r31),r1       // save input r1

        l.jal       fib             // calculate fib(N-1)
        l.addi      r1,r1,-1        // r1 = N-1 (delay slot)
        l.sw        8(r31),r1       // save fib(N-1)

        l.lwz       r1,4(r31)       // restore initial N
        l.jal       fib             // calculate fib(N-2)
        l.addi      r1,r1,-2        // r1 = N-2 (delay slot)

        l.lwz       r2,8(r31)       // restore fib(N-1)
        l.add       r1,r1,r2        // r1 = fib(N-2) + fib(N-1)

        l.lwz       r9,0(r31)       // restore link register
        l.addi      r31,r31,12      // destroy stack frame

done:   l.jr        r9              // return, result in r1
        l.nop                       // (delay slot instruction)
```

The testcase calculates the value of `fib(15)`, returning the value in register `r1` (0x262, or 610 decimal). Register `r2` is used as an intermediate and is destroyed; register `r31` is used as a stack pointer. Because this is a naive recursive implementation, each call to `fib` creates up to two further recursive calls, and the current link register value (`r9`) and input value (`r1`) need to be preserved in a stack frame at each level using the load and store instructions we have just implemented.

11.2.2 Validating Simulation Performance

This Fibonacci implementation rapidly becomes computationally complex. Even when calculating a relatively small Fibonacci number, such as `fib(15)`, 22,687 instructions are performed. We can therefore use the example to test the basic simulation speed of the processor model.

Modify lines 25 and 26 of the test case `application/asmtest.S` as follows:

```
        l.jal       fib             // calculate fib(40)
        l.addi      r1,r0,40        // r1 = 40 (delay slot)
```

Regenerate the assembler test case and run it like this:

```
make -C application
time platform/harness.$IMPERAS_ARCH.exe --program application/asmtest.OR1K.elf
```

The output from this (after a few seconds) should be as follows:

```
Warning (PC_NRI) No register information callback given for processor ‘cpu1’
Processor 'cpu1' terminated at 'exit', address 0x1000080
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 06197ecb   R2 : 03c50ea2   R3 : deadbeef
 R4 : deadbeef   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : 01000080   R10: deadbeef   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: deadbeef   R31: 00000000
 BF:1 CF:1 OF:0  PC : 01000084   SR : 00008601
--------------- --------------- --------------- ---------------

processor has executed 3808343229 instructions
```

The program has calculated `fib(40)` as 0x6197ecb (102,334,155 decimal) using 3,808,343,229 simulated instructions. On a 2.8GHz Intel Core2 processor, `time` shows this takes about 7 seconds, giving a simulation speed for this example of about 544 simulated MIPS.

11.2.3 Demonstrating Jump Hint Effectiveness

Chapter 9 showed how *jump hints* should be used to tell the simulator what kind of jump is being performed (a call, a return or a simple jump that is neither a call nor a return). Now we have a test case that executes many calls and returns, we can demonstrate how effective these jump hints are when correctly applied. To do this, we will temporarily remove the jump hints from the processor model and then rerun `fib(40)` to see the effect.

Modify functions `morphJump` and `morphJumpReg` in file `or1kMorph.c` to remove the jump hints like this:

```
static OR1K_MORPH_FN(morphJump) {

    Uns32       toAddress   = state->info.c;
    Uns32       nextAddress = state->info.thisPC + 8;
    Bool        inDelaySlot = state->inDelaySlot;
    Bool        link        = state->attrs->link;
    vmiReg      linkReg     = link ? OR1K_LINKREG : VMI_NOREG;
    vmiJumpHint hint;

    // provide no jump hint!
    hint = vmi_JH_NONE;

    if(inDelaySlot) {
        // jump in the delay slot does nothing
    } else {
        vmimtUncondJumpDelaySlot(
            1,                  // slotOps
```

```
            nextAddress,        // linkPC
            toAddress,          // toAddress
            linkReg,            // linkReg
            hint,               // hint
            0                   // slotCB
        );
    }
}

static OR1K_MORPH_FN(morphJumpReg) {

    Uns32       r1          = state->info.r1;
    vmiReg      toReg       = getGPR(r1);
    Uns32       nextAddress = state->info.thisPC + 8;
    Bool        inDelaySlot = state->inDelaySlot;
    Bool        link        = state->attrs->link;
    vmiReg      linkReg     = link ? OR1K_LINKREG : VMI_NOREG;
    vmiJumpHint hint;

    // provide no jump hint!
    hint = vmi_JH_NONE;

    if(inDelaySlot) {
        // jump in the delay slot does nothing
    } else {
        vmimtUncondJumpRegDelaySlot(
            1,                  // slotOps
            nextAddress,        // linkPC
            toReg,              // toReg
            linkReg,            // linkReg
            hint,               // hint
            0                   // slotCB
        );
    }
}
```

Rebuild the processor model and rerun `fib(40)` as follows:

```
make
time platform/harness.$IMPERAS_ARCH.exe --program application/asmtest.OR1K.elf
```

The program output is identical to before (jump hints do not affect behavior), but simulation speed is slower: on a 2.8 GHz Intel Core2 processor, `time` shows this takes about 9 seconds, giving a simulation speed for this example of about 423 simulated MIPS, *120 simulated MIPS slower than previously*.

> When creating a new processor model, use a Fibonacci test case to validate that jump hints are working correctly. If performance is unchanged (or slower!) with jump hints present then they are not being used correctly.

12 Modeling Exceptions

In chapter 11, we implemented load and store instructions and noted that on the OR1K processor all loads and stores should be aligned to the load/store size. In this chapter, we will model the processor *exception behavior* that happens when an unaligned load or store is encountered.

This chapter also shows how to write exception handlers for arithmetic exceptions such as a divide by zero.

12.1 Basic Example

Firstly, we will examine the simulator behavior when no special action is taken to handle exceptions. Directory `10.or1kBehaviorExceptions/application` contains the following example in file `asmtest.S`:

```
.org 0x200
        ////////////////////////////////////////////////////////////////////////////////
        // ALIGNMENT EXCEPTION HANDLER (AT 0x200)
        ////////////////////////////////////////////////////////////////////////////////
        l.addi      r30,r30,1       // increment count of alignment exceptions
        l.addi      r1,r1,1         // increment store address
        l.rfe                       // return from exception

.org 0x10000
        ////////////////////////////////////////////////////////////////////////////////
        // APPLICATION CODE (AT 0x10000)
        ////////////////////////////////////////////////////////////////////////////////
.global _start
_start:
        l.ori       r30,r0,0        // r30 = 0 (counts alignment exceptions)
        l.movhi     r1,0x8000       // r1 = 0x80000000
        l.movhi     r2,0x1234       // r2 = 0x12340000
        l.ori       r2,r2,0x5678    // r2 = 0x12345678
        l.ori       r3,r0,0         // r3 = 0 (loop count)

loop:
        l.sb        0(r1),r2         // do one-byte store
        l.sh        0(r1),r2         // do two-byte store
        l.sw        0(r1),r2        // do four-byte store

        l.addi      r3,r3,1         // increment loop count
        l.sfeqi     r3,10           // r3==10?
        l.bnf       loop            // go if not
        l.addi      r1,r1,1         // increment store address (delay slot)

        l.div       r30,r30,r0      // divide by zero

.global exit
exit:
        l.nop
```

This example uses r1 to hold a write pointer, initially at address `0x80000000`. It then executes a loop ten times, trying to do one-byte, two-byte and four-byte writes to the pointer. Each time round the loop the pointer `r1` is incremented. Finally, it performs an arithmetic divide-by-zero.

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/10.or1kBehaviorExceptions .
```

Compile the model, harness and application using the `make` command:

```
cd 10.or1kBehaviorExceptions
make
```

Note that the harness and application can be compiled individually if required using these commands:

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

Run the platform using the assembler executable file:

```
platform/harness.$IMPERAS_ARCH.exe –-trace \
    --program application/asmtest.OR1K.elf
```

The output from this should be as follows:

```
Warning (PC_NRI) No register information callback given for processor ‘cpu1’
Info 'cpu1', 0x0000000000010000: l.ori    r30,r0,0x0
Info 'cpu1', 0x0000000000010004: l.movhi  r1,0x8000
Info 'cpu1', 0x0000000000010008: l.movhi  r2,0x1234
Info 'cpu1', 0x000000000001000c: l.ori    r2,r2,0x5678
Info 'cpu1', 0x0000000000010010: l.ori    r3,r0,0x0
Info 'cpu1', 0x0000000000010014: l.sb     0x0(r1),r2
Info 'cpu1', 0x0000000000010018: l.sh     0x0(r1),r2
Info 'cpu1', 0x000000000001001c: l.sw     0x0(r1),r2
Info 'cpu1', 0x0000000000010020: l.addi   r3,r3,0x1
Info 'cpu1', 0x0000000000010024: l.sfeqi  r3,0xa
Info 'cpu1', 0x0000000000010028: l.bnf    0x00010014
Info 'cpu1', 0x000000000001002c: l.addi   r1,r1,0x1
Info 'cpu1', 0x0000000000010014: l.sb     0x0(r1),r2
Info 'cpu1', 0x0000000000010018: l.sh     0x0(r1),r2
Processor Exception (PC_PRX) Processor 'cpu1' 0x10018: l.sh     0x0(r1),r2
Processor Exception (PC_WAX) Misaligned 2-byte write to 0x80000001
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 80000001   R2 : 12345678   R3 : 00000001
 R4 : deadbeef   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : deadbeef   R10: deadbeef   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: 00000000   R31: deadbeef
 PC : 00010018   SR : 00008001   ESR: deadbeef   EPC: deadbeef
 BF:0 CF:0 OF:0
--------------- --------------- --------------- ---------------

processor has executed 14 instructions
```

The test case runs successfully until an attempt is made to perform a 2-byte store at a 1-byte-aligned address. Then the processor terminates with a Processor Exception error, which is signaled to the platform by returning the value `OP_SR_WR_ALIGN` from `opProcessorSimulate`. File `platform/harness.c` in directory `10.or1kBehaviorExceptions` has been updated like this to handle the possible exception case:

```
static Bool simulate(optProcessorP processor, Uns64 clocks) {

    optStopReason stopReason = opProcessorSimulate(processor, clocks);

    switch(stopReason) {

       case OP_SR_SCHED:
           // hit the scheduler limit
           return True;

       case OP_SR_EXIT:
           // processor has exited
           return False;

       case OP_SR_FINISH:
           // simulation must end
           return False;

       case OP_SR_RD_PRIV:
       case OP_SR_WR_PRIV:
       case OP_SR_RD_ALIGN:
       case OP_SR_WR_ALIGN:
       case OP_SR_ARITH:
          // unhandled processor exception: simulation must end
          return False;

       default:
           opPrintf("unexpected stopReason %u\n", stopReason);
           return False;
    }
}
```

Note that the only action taken in the model to enforce alignment checking was to pass `True` to the `checkAlign` parameter to each of the load/store morph-time functions: the simulator did the rest automatically.

It is possible instead to envisage using `vmimt` calls to explicitly construct an alignment check (for example, the address to which to store could be created in a temporary register using `vmimtBinopRRC`, the low-order bits of the temporary register value could be extracted using `vmimtBinopRRC` with a `vmiBinop` of `vmi_AND` and a constant value 3, this could be checked for zero and the simulation terminated by `vmimtFinish` if not zero). This is not however a good approach: the simulator's built in checks are much more efficient than any model code that tries to do the same thing. *Always use simulator alignment checking capabilities in preference to coding your own.*

A real processor does not of course "exit" when an exception is encountered: it typically enters a privileged mode and jumps to an exception vector instead. The model files in directory `10.or1kBehaviorExceptions` have been enhanced to support this as follows.

12.1.1 Adding Exception Registers - `or1kStructure.h`

To support exceptions, the OR1K has three new registers we need to model now:

1. `esr` (exception status register): this saves the value of the `sr` register on exception entry;
2. `epc` (exception program counter register): this saves the current program counter on exception entry.
3. `eear` (exception effective address register): for an exception caused by an invalid memory operation, this records the address that caused the exception.

These have been added to the processor structure:

```
typedef struct or1kS {

    Bool       carryFlag;        // carry flag
    Bool       overflowFlag;     // overflow flag
    Bool       branchFlag;       // branch flag

    Uns32      regs[OR1K_REGS]; // basic registers

    Uns32      SR;               // status register
    Uns32      ESR;              // exception status register
    Uns32      EPC;              // exception program counter register
    Uns32      EEAR;             // exception effective address register

    vmiBusPortP busPorts;        // bus port descriptions

} or1k, *or1kP;
```

There are also new macros to access the registers when morphing code:

```
#define OR1K_ESR   OR1K_CPU_REG(ESR)
#define OR1K_EPC   OR1K_CPU_REG(EPC)
#define OR1K_EEAR  OR1K_CPU_REG(EEAR)
```

Initialization routines (in `or1kMain.c`) and register dump routines (in `or1kUtils.c`) have also been modified to handle the new registers.

12.1.2 Declaring Exception Handlers - `or1kFunctions.h`

Prototypes for model routines that are called when the simulator detects a memory exception have been added:

```
VMI_RD_PRIV_EXCEPT_FN(or1kRdPrivExceptionCB);
VMI_WR_PRIV_EXCEPT_FN(or1kWrPrivExceptionCB);
VMI_RD_ALIGN_EXCEPT_FN(or1kRdAlignExceptionCB);
VMI_WR_ALIGN_EXCEPT_FN(or1kWrAlignExceptionCB);
```

The memory access handlers are called for load/store *privilege* exceptions and load/store *alignment* exceptions. In this example, we are only interested in alignment exceptions but privilege exceptions will be implemented as well for completeness.

```
VMI_ARITH_EXCEPT_FN(or1kArithExceptionCB);
```

This handler is called when the simulator detects an arithmetic exception at run time (for example, a divide by zero).

12.1.3 Defining Exception Types - `or1kExceptionTypes.h`

This is a new file giving information about the exceptions on the OR1K. There is an enumeration of the possible exception types:

```
typedef enum or1kExceptionE {
    OR1K_EXCPT_RST,         // reset
    OR1K_EXCPT_BUS,         // alignment
    OR1K_EXCPT_DPF,         // data privilege
    OR1K_EXCPT_IPF,         // instruction privilege
    OR1K_EXCPT_TTI,         // tick timer
    OR1K_EXCPT_ILL,         // illegal instruction
    OR1K_EXCPT_EXI,         // external interrupt
    OR1K_EXCPT_SYS,         // system call
    OR1K_EXCPT_LAST         // KEEP LAST: for sizing
} or1kException;
```

Vector addresses for each exception are also defined:

```
#define RST_ADDRESS 0x100   // reset exception vector
#define BUS_ADDRESS 0x200   // alignment exception vector
#define DPF_ADDRESS 0x300   // data privilege exception vector
#define IPF_ADDRESS 0x400   // instruction privilege exception vector
#define TTI_ADDRESS 0x500   // tick timer exception vector
#define ILL_ADDRESS 0x700   // illegal instruction exception vector
#define EXI_ADDRESS 0x800   // external interrupt exception vector
#define SYS_ADDRESS 0xc00   // sys exception vector
```

12.1.4 Implementing Exceptions - `or1kExceptions.c`

This file implements the exception handler callbacks, which are called at run time when a potential simulated exception occurs. The purpose of the callbacks is to put the processor into the state that it would enter if the same exception was encountered on the real hardware: typically, this means entering a privileged mode, saving some exception context state and jumping to an exception vector address. This is exactly what we will implement now for the OR1K.

> It is possible for *multiple exception conditions to be encountered in a single simulated instruction*: for example, a store may be attempted to an address that is both misaligned and read-only. To handle this situation, the memory exception handlers work as follows:
>
> 1. The *alignment* handler is called first. This returns an unsigned result indicating whether the privilege exception handler should be called subsequently;

2. If the alignment handler returns *non-zero*, and there is also a privilege exception condition, then the *privilege* exception handler will be called. If the alignment handler returns *zero*, the privilege exception handler will *not* be called.

A non-zero result from the alignment handler may also indicate that the load/store address requires *snapping*, or that the value to load or store requires *rotation*. This is discussed in detail later in this section.

The memory exception handler callbacks for the OR1K are as follows:

```
VMI_RD_PRIV_EXCEPT_FN(or1kRdPrivExceptionCB) {
    if(MEM_AA_IS_TRUE_ACCESS(attrs)) {
        or1kP or1k = (or1kP)processor;
        or1k->EEAR = (Uns32)address;
        or1kTakeException(or1k, OR1K_EXCPT_DPF, 0);
    }
}

VMI_WR_PRIV_EXCEPT_FN(or1kWrPrivExceptionCB) {
    if(MEM_AA_IS_TRUE_ACCESS(attrs)) {
        or1kP or1k = (or1kP)processor;
        or1k->EEAR = (Uns32)address;
        or1kTakeException(or1k, OR1K_EXCPT_DPF, 0);
    }
}

VMI_RD_ALIGN_EXCEPT_FN(or1kRdAlignExceptionCB) {
    or1kP or1k = (or1kP)processor;
    or1k->EEAR = (Uns32)address;
    or1kTakeException(or1k, OR1K_EXCPT_BUS, 0);
    return 0;
}

VMI_WR_ALIGN_EXCEPT_FN(or1kWrAlignExceptionCB) {
    or1kP or1k = (or1kP)processor;
    or1k->EEAR = (Uns32)address;
    or1kTakeException(or1k, OR1K_EXCPT_BUS, 0);
    return 0;
}
```

In this case, both alignment handlers return zero, which means that *alignment exceptions have priority over privilege exceptions* (in other words, a store to an address that is both misaligned and read-only will cause an alignment exception only). Each exception is implemented by a call to `or1kTakeException` (implemented in `or1kUtils.c`), passing the appropriate exception type (`OR1K_EXCPT_DPF` for privilege exceptions, `OR1K_EXCPT_BUS` for alignment exceptions) and a zero offset (explained in the description of `or1kTakeException`). The faulting address is saved in the `eear` register in the processor structure.

The read and write privilege handlers are both passed an argument, `attrs`, of type `memAccessAttrs`, defined in `vmiTypes.h` as follows:

```
typedef enum memAccessAttrsE {
    MEM_AA_FALSE = 0x0, // this is an artifact access
```

```
    MEM_AA_TRUE  = 0x1, // this is a true processor access
    MEM_AA_FETCH = 0x2, // this access is a fetch
} memAccessAttrs;
```

The `memAccessAttrs` type tells the processor model what kind of access is being performed. There are four possible values:

1. `MEM_AA_TRUE`: this indicates that the exception handler is being called because of a true processor read or write, and that the model should take any action needed to model the exception.
2. `MEM_AA_FALSE`: this indicates that this access is not a true processor read or write, but is instead some kind of *artifact* access. For example, it might be an access being made by the simulator itself, or by an attached debugger reading memory. In this case, the processor model should not update its state to reflect an exception, but might need to take some other action to make the memory readable or writable. As an example, the OVP ARM processor model implements a TLB model that maps memory pages on demand based on the contents of a page table stored in memory, and these mappings need to be made even for an artifact access (so that a debugger can query virtual memory address locations even if that virtual address is not currently mapped, for example).
3. `MEM_AA_TRUE|MEM_AA_FETCH`: this indicates that the exception handler is being called because of a true processor fetch. Processor state should be updated to model the exception.
4. `MEM_AA_FALSE|MEM_AA_FETCH`: this indicates that the exception handler is being called because of an artifact fetch (usually caused by the JIT code generation engine). In this case, the processor model should not update its state to reflect an exception, but might need to take some other action to make the memory readable or writable.

For the OR1K, the read and write exception handlers both validate that the access is a non-artifact access before updating any processor state:

```
VMI_RD_PRIV_EXCEPT_FN(or1kRdPrivExceptionCB) {
    if(MEM_AA_IS_TRUE_ACCESS(attrs)) {
        or1kP or1k = (or1kP)processor;
        or1k->EEAR = (Uns32)address;
        or1kTakeException(or1k, OR1K_EXCPT_DPF, 0);
    }
}

VMI_WR_PRIV_EXCEPT_FN(or1kWrPrivExceptionCB) {
    if(MEM_AA_IS_TRUE_ACCESS(attrs)) {
        or1kP or1k = (or1kP)processor;
        or1k->EEAR = (Uns32)address;
        or1kTakeException(or1k, OR1K_EXCPT_DPF, 0);
    }
}
```

This OR1K model does not implement any structure such as a demand-mapped TLB, so no action is taken for artifact accesses.

File `or1kExceptions.c` also implements an *arithmetic* exception handler:

```
VMI_ARITH_EXCEPT_FN(or1kArithExceptionCB) {

    or1kP or1k = (or1kP)processor;

    switch(exceptionType) {

        // integer divide-by-zero and overflow should not generate exceptions
        // but instead set the carry flag
        case VMI_INTEGER_DIVIDE_BY_ZERO:
        case VMI_INTEGER_OVERFLOW:
            or1k->carryFlag = 1;
            return VMI_INTEGER_ABORT;

        // not expecting any other arithmetic exception types
        default:
            return VMI_INTEGER_UNHANDLED;
    }
}
```

When an integer divide or overflow is encountered, the OR1K does not jump to an exception vector: instead, it indicates the error by setting the processor carry flag. Other processor types that jump to exception vectors can be simulated in a similar manner to the memory exception handlers (i.e. save the current program counter and other state, and then jump to an exception vector – see the discussion of `or1kTakeException` in section 12.1.5).

The return value from the arithmetic exception callback is an enumerated value defined in `vmiTypes.h`:

```
typedef enum vmiIntegerExceptionResultE {
    VMI_INTEGER_UNHANDLED,      // not handled
    VMI_INTEGER_ABORT,          // handled, abort current instruction
    VMI_INTEGER_CONTINUE,       // handled, continue current instruction
} vmiIntegerExceptionResult;
```

A return value of `VMI_INTEGER_UNHANDLED` indicates that the numeric exception was not expected by this model and simulation should terminate.

A return value of `VMI_INTEGER_ABORT` indicates that the handler accepted the exception, and simulation should abort the remainder of this simulated instruction and resume execution with the *next* simulated instruction (or at an exception vector address, if `vmirtSetPC` or `vmirtSetPCException` are used in the handler: see section 12.1.5).

A return value of `VMI_INTEGER_CONTINUE` indicates that the handler accepted the exception, and simulation should resume at the next *native* instruction address after the offending instruction.

When writing code that could cause simulated exceptions, or which makes an embedded call that could update the current program counter using `vmirtSetPC` or `vmirtSetPCException`, always remember that the part of the instruction *after* the embedded call or simulated exception *will not be executed* if the program counter has been modified by `vmirtSetPC` or `vmirtSetPCException`, or if there is an arithmetic exception for which the handler returns `VMI_INTEGER_ABORT`. *Care must be taken to leave the processor model in a consistent state in this case*.

As a contrived example, suppose that a processor is being modeled that has a single instruction that implements a pair of loads into registers from different addresses:

```
0x0020000: r1=(ra1), r2=(ra2)
```

The obvious way to implement this would be with two `vmimtLoadRRO` calls, for example:

```
vmimtLoadRRO(
    32, 32, 0, CPU_REG(r1), CPU_REG(ra1), endian, False,
    MEM_CONSTRAINT_ALIGNED
);
vmimtLoadRRO(
    32, 32, 0, CPU_REG(r2), CPU_REG(ra2), endian, False,
    MEM_CONSTRAINT_ALIGNED
);
```

However, suppose that there is a memory access violation on the access using `ra2` (but not `ra1`) that caused control to be transferred to a simulated exception handler. In this case, the processor would be left in a state with the instruction half-executed, because the load to `r1` would already have been done.

To get correct model behavior in this case, the first load should save its result in a temporary, which is written to the target register only if the second load succeeds:

```
vmimtLoadRRO(
    32, 32, 0, CPU_TEMP1, CPU_REG(ra1), endian, False,
    MEM_CONSTRAINT_ALIGNED
);
vmimtLoadRRO(
    32, 32, 0, CPU_REG(r2), CPU_REG(ra2), endian, False,
    MEM_CONSTRAINT_ALIGNED
);
vmimtMoveRR(32, CPU_REG(r1), CPU_TEMP1);
```

12.1.5 Taking Exceptions - `or1kUtils.[ch]`

The new routine `or1kTakeException` is implemented as:

```
void or1kTakeException(or1kP or1k, or1kException exception, Uns32 pcOffset) {

    Uns8  simD;
    Uns32 simPC = (Uns32)vmirtGetPCDS((vmiProcessorP)or1k, &simD);

    or1kEnterSupervisorMode(or1k);
    or1k->EPC = simPC + pcOffset;

    // set sr[DSX] for exception in a delay slot
    if(simD) {
        or1k->SR |= SPR_SR_DSX;
```

```
    }

    // jump to the vector
    vmirtSetPCException((vmiProcessorP)or1k, exceptions[exception].code);
}
```

Because this routine is called at run time (as opposed to morph time) it uses functions form the *Imperas Run Time Function API* to update the processor model state. In detail, it works as follows:

```
    Uns8  simD;
    Uns32 simPC = (Uns32)vmirtGetPCDS((vmiProcessorP)or1k, &simD);
```

The function `vmirtGetPCDS` returns the currently-executing instruction address with any delay-slot byte offset. For non-delay-slot instructions, `simPC` will be set to the current instruction address and `simD` set to zero. For delay-slot instructions, `simPC` will be set to *the address of the preceding jump or branch instruction* and `simD` will be *the byte offset of the current instruction from the preceding jump or branch*. Since all OR1K instructions are four bytes long, `simD` will therefore be 4 for a delay-slot instruction.

```
    or1kEnterSupervisorMode(or1k);
```

This is a routine that puts the simulated processor into supervisor mode, described below.

```
    or1k->EPC = simPC + pcOffset;
```

This line saves the current program counter in register `epc` (or the jump/branch instruction address for delay slot instructions). A call-specific offset is added to the value saved (this value is zero for the memory exceptions).

```
    if(simD) {
        or1k->SR |= SPR_SR_DSX;
    }
```

These lines set a special bit in the status register `sr` if the exception occurred in a delay slot instruction. Recovery from delay slot instruction exceptions requires special processing in application exception handlers, so they need some way to find out whether the original exception was in a delay slot instruction or not.

```
    vmirtSetPCException((vmiProcessorP)or1k, exceptions[exception].code);
```

This line uses `vmirtSetPCException` to force the processor to jump to the exception vector address associated with the exception type. A table maps exception types to vector addresses:

```
#define OR1K_EXCEPTION_INFO(_D,_DESC) [OR1K_EXCPT_##_D] = { \
    name        : #_D,          \
    code        : _D##_ADDRESS, \
    description : _DESC         \
}

static const vmiExceptionInfo exceptions[OR1K_EXCPT_LAST] = {
```

```
    OR1K_EXCEPTION_INFO(RST, "Reset"),
    OR1K_EXCEPTION_INFO(BUS, "Bus error"),
    OR1K_EXCEPTION_INFO(DPF, "Data privilege"),
    OR1K_EXCEPTION_INFO(IPF, "Instruction privilege"),
    OR1K_EXCEPTION_INFO(TTI, "Tick timer"),
    OR1K_EXCEPTION_INFO(ILL, "Illegal instruction"),
    OR1K_EXCEPTION_INFO(EXI, "External interrupt"),
    OR1K_EXCEPTION_INFO(SYS, "System call"),
};
```

The exceptions are described using an array of `vmiExceptionInfo` type structures (defined in `vmiTypes.h`). This structure type will be required when adding debugger integration support routines to the model (see chapter 17).

The new routine `or1kEnterSupervisorMode` saves `sr` in `esr`, updates `sr` to mask out various exceptions that must be disabled in supervisor mode, and indicates that we are in supervisor mode by setting the `SM` bit in `sr`, like this:

```
void or1kEnterSupervisorMode(or1kP or1k) {

    const Uns32 clearBits = (
        SPR_SR_IEE |    // interrupt enable
        SPR_SR_TEE |    // tick timer enable
        SPR_SR_DME |    // data MMU enable
        SPR_SR_IME |    // instruction MMU enable
        SPR_SR_OVE      // overflow exception enable
    );

    // save the current status register in esr
    or1k->ESR = or1kGetSR(or1k);

    // mask out the 'clear' bits and mask in supervisor mode
    or1k->SR = (or1k->ESR & ~clearBits) | SPR_SR_SM;
}
```

Note that `or1kGetSR` is used to get the value of the `sr` register. This routine ensures that the flag bits are present in the returned value.

12.1.6 Exception Function Registration - `or1kAttrs.c`

File `or1kAttrs.c` has been updated to include references to the five new exception handler callbacks in the `modelAttrs` structure:

```
const vmiIASAttr modelAttrs = {

     . . . skipped lines . . .

    ////////////////////////////////////////////////////////////////////////
    // EXCEPTION ROUTINES
    ////////////////////////////////////////////////////////////////////////

    .rdPrivExceptCB  = or1kRdPrivExceptionCB,
    .wrPrivExceptCB  = or1kWrPrivExceptionCB,
    .rdAlignExceptCB = or1kRdAlignExceptionCB,
    .wrAlignExceptCB = or1kWrAlignExceptionCB,
    .arithExceptCB   = or1kArithExceptionCB,

    . . . skipped lines . . .
```

```
};
```

12.1.7 `l.rfe` and `l.sys` Instructions

The model has been enhanced to implement the `l.rfe` instruction. This OR1K instruction performs a return from an exception handler: it copies register `esr` to register `sr` and performs an unconditional jump to the address stored in `epc`. Decode and disassembly for this instruction are very similar to previous instructions so no further details will be given here. The functionality of the `l.rfe` instruction is implemented in `or1kMorph.c` like this:

```
static OR1K_MORPH_FN(morphRFE) {

    // set sr from esr (must call or1kSetSR to do this)
    vmimtArgProcessor();
    vmimtArgReg(OR1K_BITS, OR1K_ESR);
    vmimtCall((vmiCallFn)or1kSetSR);

    // return to exception program counter
    vmimtUncondJumpReg(0, OR1K_EPC, VMI_NOREG, vmi_JH_RETURNINT);
}
```

The first part of `morphRFE` constructs an *embedded call* to function `or1kSetSR`, passing the current processor as the first argument and the value of register `esr` from the processor model structure as the second argument. The effect of this is to assign the current value of `esr` to register `sr`. Remember that whenever `sr` is set, we must use `or1kSetSR` to do it because assigning to this register implicitly sets the flag fields we maintain separately in the model.

The second part of `morphRFE` uses `vmimtUncondJumpReg` to perform a jump to the address in the `epc` register. Note that the `l.rfe` instruction is not followed by a delay slot instruction. As this instruction implements a return from an exception handler, a new jump hint type is used - `vmi_JH_RETURNINT`.

> In general, it is possible to emit code to call *any* function from morphed code by using a sequence of `vmimtArg`-prefixed functions followed by a call to `vmimtCall`. This means that for many instructions there is an important implementation choice to be made: is it best to implement the instruction directly using `vmimtBinopRRR`, `vmimtBinopRRC` etc, or is it best to use `vmimtCall` to call a C function to do the work instead?
>
> In general, the rule is that *if the behavior of the instruction requires more than a few `vmimt`-prefixed calls to implement, or is difficult to encode using `vmimt` operations, then use `vmimtCall` and a C function to implement the instruction behavior*.
>
> One important exception is that, wherever possible, jumps should be implemented using `vmimt` jump primitives instead of using `vmirtSetPC`, which is significantly slower. Note that a single instruction can be implemented using a mixture or `vmimt` primitive operations and `vmirtCall` calls, as in this example.

The `l.sys` instruction is also now implemented (though not required for this example). This instruction enters supervisor mode and jumps to an exception vector at address `0xc00`, saving the *next instruction* address in `epc`. It is implemented in `or1kMorph.c` as:

```
static OR1K_MORPH_FN(morphSYS) {

    vmimtArgProcessor();
    vmimtArgUns32(OR1K_EXCPT_SYS);
    vmimtArgUns32(4);
    vmimtCallAttrs((vmiCallFn)or1kTakeException, VMCA_EXCEPTION);
}
```

Note that the value 4 is passed as the third argument to `or1kTakeException` to ensure that the saved address in epc is the *next instruction* address.

The call to `or1kTakeException` in `morphSYS` is implemented using a more generic embedded call function, `vmimtCallAttrs`. This function is similar to `vmimtCall` except that it takes an additional argument of type `vmiCallAttrs` which gives the JIT compiler further information about the purpose and nature of the call:

```
typedef enum vmiCallAttrsE {
    VMCA_NA            = 0x0,   // no attributes
    VMCA_PURE          = 0x1,   // this call is to a pure function
    VMCA_EXCEPTION     = 0x2,   // this call causes a simulated exception
    VMCA_NO_INVALIDATE = 0x4,   // this call cannot invalidate this block
    VMCA_FP_RESTORE    = 0x8,   // restore floating point state before call
    VMCA_FLT32_RESULT  = 0x10,  // function result is of type Flt32
    VMCA_FLT64_RESULT  = 0x20,  // function result is of type Flt64
} vmiCallAttrs;
```

The value `VMCA_EXCEPTION` indicates that this call is to an exception entry routine and that therefore the current code block should be terminated after this instruction (because the next simulated instruction is never executed sequentially). It also indicates that the called function might cause a processor mode switch (processor modes are discussed in chapter 14). Refer to the *Imperas VMI Morph Time Function Reference* for more detailed information about when to use `vmimtCallAttrs`.

If the test case already contained code to implement simulated exceptions, why did the original run at the start of this chapter exit from `opProcessorSimulate` with an unhandled processor exception? The reason is that whether or not simulated exceptions should be enabled is specified by a model flag in the platform (most application code should not generate exceptions in the normal case, and it is usually desired that any exception is an error that should stop simulation).

To enable simulated exception modeling, modify the processor instantiation in `10.or1kBehaviorExceptions/platform/harness.c` as follows:

```
    const char *modelFile = "model."IMPERAS_SHRSUF;
    optProcessorP processor = opProcessorNew(
        mr, modelFile, "cpu1", 0,
        OP_PARAMS(5
            OP_PARAM_BOOL_SET(OP_FP_SIMULATEEXCEPTIONS, 1)
        )
    );
```

Then rebuild the platform and resimulate:

```
make -C platform
platform/harness.$IMPERAS_ARCH.exe –-trace --program
application/asmtest.OR1K.elf
```

The output from this should now be as follows:

```
Warning (PC_NRI) No register information callback given for processor ‘cpu1’
Info 'cpu1', 0x0000000000010000: l.ori    r30,r0,0x0
Info 'cpu1', 0x0000000000010004: l.movhi  r1,0x8000
Info 'cpu1', 0x0000000000010008: l.movhi  r2,0x1234
Info 'cpu1', 0x000000000001000c: l.ori    r2,r2,0x5678
Info 'cpu1', 0x0000000000010010: l.ori    r3,r0,0x0
Info 'cpu1', 0x0000000000010014: l.sb     0x0(r1),r2
Info 'cpu1', 0x0000000000010018: l.sh     0x0(r1),r2
Info 'cpu1', 0x000000000001001c: l.sw     0x0(r1),r2
Info 'cpu1', 0x0000000000010020: l.addi   r3,r3,0x1
Info 'cpu1', 0x0000000000010024: l.sfeqi  r3,0xa
Info 'cpu1', 0x0000000000010028: l.bnf    0x00010014
Info 'cpu1', 0x000000000001002c: l.addi   r1,r1,0x1
Info 'cpu1', 0x0000000000010014: l.sb     0x0(r1),r2
Info 'cpu1', 0x0000000000010018: l.sh     0x0(r1),r2
Info 'cpu1', 0x0000000000000200: l.addi   r30,r30,0x1
Info 'cpu1', 0x0000000000000204: l.addi   r1,r1,0x1
Info 'cpu1', 0x0000000000000208: l.rfe
Info 'cpu1', 0x0000000000010018: l.sh     0x0(r1),r2
Info 'cpu1', 0x000000000001001c: l.sw     0x0(r1),r2
Info 'cpu1', 0x0000000000000200: l.addi   r30,r30,0x1
Info 'cpu1', 0x0000000000000204: l.addi   r1,r1,0x1
Info 'cpu1', 0x0000000000000208: l.rfe
Info 'cpu1', 0x000000000001001c: l.sw     0x0(r1),r2
Info 'cpu1', 0x0000000000000200: l.addi   r30,r30,0x1
Info 'cpu1', 0x0000000000000204: l.addi   r1,r1,0x1
Info 'cpu1', 0x0000000000000208: l.rfe
Info 'cpu1', 0x000000000001001c: l.sw     0x0(r1),r2
Info 'cpu1', 0x0000000000010020: l.addi   r3,r3,0x1
Info 'cpu1', 0x0000000000010024: l.sfeqi  r3,0xa
```

[5] This shows how processor instantiations can be parameterized. In general, such parameters can be specified in two ways:

1. Using the `OP_PARAMS` list (as here); or
2. Using *command line parser overrides*.

As an example, instead of modifying the instantiation, we could have specified the additional command line argument `--override platform/cpu1/simulateexceptions=T` instead.

If the parameter should *always* be applied, use the first method. If it is only *sometimes* required, use the second.

```
. . . etc . . .
Info 'cpu1', 0x0000000000010020: l.addi   r3,r3,0x1
Info 'cpu1', 0x0000000000010024: l.sfeqi  r3,0xa
Info 'cpu1', 0x0000000000010028: l.bnf    0x00010014
Info 'cpu1', 0x000000000001002c: l.addi   r1,r1,0x1
Info 'cpu1', 0x000000000001002c: l.div    r30,r30,r0
Info 'cpu1', 0x0000000000010030: l.nop    0x0
Processor 'cpu1' terminated at 'exit', address 0x10034
-------------- -------------- -------------- --------------
 R0 : 00000000   R1 : 80000025   R2 : 12345678   R3 : 0000000a
 R4 : deadbeef   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : deadbeef   R10: deadbeef   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: 0000001b   R31: deadbeef
 PC : 00010034   SR : 00008201   ESR: 00008001   EPC: 0001001c
 BF:1 CF:1 OF:0
-------------- -------------- -------------- --------------

processor has executed 185 instructions
```

Now, instead of terminating after 14 instructions, the processor jumps to the exception handler at address `0x200` instead (execution within the exception handler is highlighted in bold in the trace above for clarity). The exception handler increments the address in `r1` and returns using `l.rfe`, which re-executes the faulting instruction (obviously, in a real application the exception handler would do something more sensible than this, but the example is sufficient for demonstration purposes). As a side effect, the exception handler also increments `r30` so that on termination we have a count of the number of times it was called (0x1b, i.e. 27 times).

The divide-by-zero in the penultimate instruction now does not cause the simulation to exit, but instead sets the carry flag.

12.2 Misaligned Load/Store Address Snapping and Value Rotation

On some processors, loads and stores to misaligned addresses do not cause exceptions but are instead *snapped* to the correct alignment for the data size (so two-byte load addresses are rounded down to a two-byte boundary, four-byte load addresses to a four-byte boundary, and so on). In addition, some processors (e.g. old ARM processors) *rotate* values read from misaligned addresses, with the rotate amount based on the misaligned byte offset.

One way to implement address snapping would be to copy each load/store address to a temporary, mask it to the appropriate size using `vmimtBinopRC` with a `vmiBinop` of `vmi_AND`, and then use this resulting address as an argument to the load/store operation. Unfortunately, this approach adds significant run-time overhead to each memory access. Implementing value rotation is even more complex.

A better solution is possible using *read and write address snap handlers* defined in the processor model. These are defined using a macro in `vmiAttrs.h`:

```
#define VMI_RD_WR_SNAP_FN(_NAME) Uns32 _NAME( \
    vmiProcessorP processor,    \
    memDomainP    domain,       \
    Addr          address,      \
    Uns32         bytes         \
)
typedef VMI_RD_WR_SNAP_FN((*vmiRdWrSnapFn));
```

The read and write handlers are specified using the `rdSnapCB` and `wrSnapCB` fields in the processor attributes structure, respectively:

```
    vmiRdWrSnapFn  rdSnapCB;            // read alignment snap function
    vmiRdWrSnapFn  wrSnapCB;            // write alignment snap function
```

The return value from each handler is an integer which indicates what address snapping or value rotation is required. The return value is constructed using macro `MEM_SNAP` in `vmiTypes.h`:

```
#define MEM_SNAP(_SNAP, _ROTATE) (((Uns8)(_SNAP)) | ((_ROTATE)<<8))
```

In this macro, `_SNAP` specifies an *address rounding granularity* in bytes (typically 1, 2, 4 or 8), and `_ROTATE` specifies a value rotation in bits. As an example:

```
MEM_SNAP(4, 24)
```

indicates that a read/write address should be snapped to 4-byte alignment. In addition, a value being written should be rotated left by 24 bits before it is written, and a value being read should be rotated left by 24 bits before being assigned to a processor register.

In detail, the read and write snap handlers are used by the simulator as follows:

1. If an access is made to a misaligned address, any defined *address snap handler* is called first. If the handler is defined and returns *non-zero*, then the read or written value is modified using the granularity and rotation specified by the result.
2. Otherwise (if there is no address snap handler, or the address snap handler returns zero) any defined *align exception handler* is called for a misaligned address access. This should either return 0 (if the read or write should be terminated, possibly because an exception is taken) or 1 (if the read or write should proceed, possibly with a modified value)[6].
3. If the read/write address has insufficient privileges, and either the address was aligned, or the snap handler or align exception handler returns non-zero, then the *privilege exception handler* is called.

[6] In fact, the align exception handler returns a granularity/rotate value in the same format as for the snap handler. A return value of 1 therefore indicates 1-byte alignment with zero rotation.

12.2.1 ARM Model Load/Store Address Snap Callback

This is the read snap address callback from the OVP ARM processor model:

```
VMI_RD_WR_SNAP_FN(armRdSnapCB) {

    armP               arm = (armP)processor;
    armUnalignedAction ua;

    if(ALIGN_ENABLED(arm) || ((ua=getUnalignedAction(arm))==ARM_UA_DABORT)) {

        // take exception
        return 0;

    } else if((ua==ARM_UA_ROTATE) && arm->configInfo.rotateUnaligned) {

        // read snaps address and loads rotated value
        Uns32 rotate = address&(bytes-1);

        if(getEndian(arm)==MEM_ENDIAN_LITTLE) {
            rotate = bytes - rotate;
        }

        return MEM_SNAP(bytes, rotate*8);

    } else {

        // read snaps address
        return MEM_SNAP(bytes, 0);
    }
}
```

This callback does three things:

1. If the current instruction should cause an alignment exception, it returns 0 (so that the read alignment exception handler will be called);
2. Otherwise, if this is an ARM variant in which unaligned reads cause rotation of the read value, it calculates the required rotation based on the address are returns a result aligned to the item byte size with that rotation;
3. Otherwise, it returns a result with an aligned address but no rotation.

12.3 Memory Aborts

In addition to alignment and privilege exceptions, there is one other type of exception that can be handled in a processor model: a *memory abort*. Memory aborts are generated by the memory subsystem, typically when there is no implemented memory at a particular address. Read and write abort handlers are specified using the `rdAbortExceptCB` and `wrAbortExceptCB` fields in the processor attributes structure, respectively:

```
    vmiRdAbortExceptFn rdSnapCB;  // read abort exception
    vmiRdAbortExceptFn wrSnapCB;  // write abort exception
```

The read and write abort handlers are called in one of two circumstances:

1. When a read or write privilege exception handler indicates the access should be retried, but the simulator determines that there is no accessible memory at the faulting address.

2. When an externally-implemented memory model indicates that a memory access has not succeeded (for example, by calling `opProcessorReadAbort` or `opProcessorWriteAbort`).

In the first case, read and write privilege exception handlers can indicate that a read or write should be retried on completion using a by-ref argument, `action`, which should be set to the value `VMI_LOAD_STORE_CONTINE`. As an example, here is the read privilege exception handler from the OVP ARM model:

```
VMI_RD_PRIV_EXCEPT_FN(armRdPrivExceptionCB) {

    armP arm = (armP)processor;

    if(!armVMMiss(arm, domain, MEM_PRIV_R, address, bytes, attrs)) {
        *action = VMI_LOAD_STORE_CONTINUE;
    }
}
```

The function `armVMMiss` attempts to map the faulting address using either a TLB or MPU entry, returning `True` if the address could not be mapped (indicating a miss). If there is no miss, the function uses the `action` argument to indicate that the load should be retried.

The read abort handler in the OVP ARM model triggers an external memory abort:

```
VMI_RD_ABORT_EXCEPT_FN(armRdAbortExceptionCB) {
    armP arm = (armP)processor;
    armExternalMemoryAbort(arm, address, isFetch ? MEM_PRIV_X : MEM_PRIV_R);
}
```

The full load/store exception escalation process, including address snapping, alignment, privilege and abort handlers, is shown in the following figure.

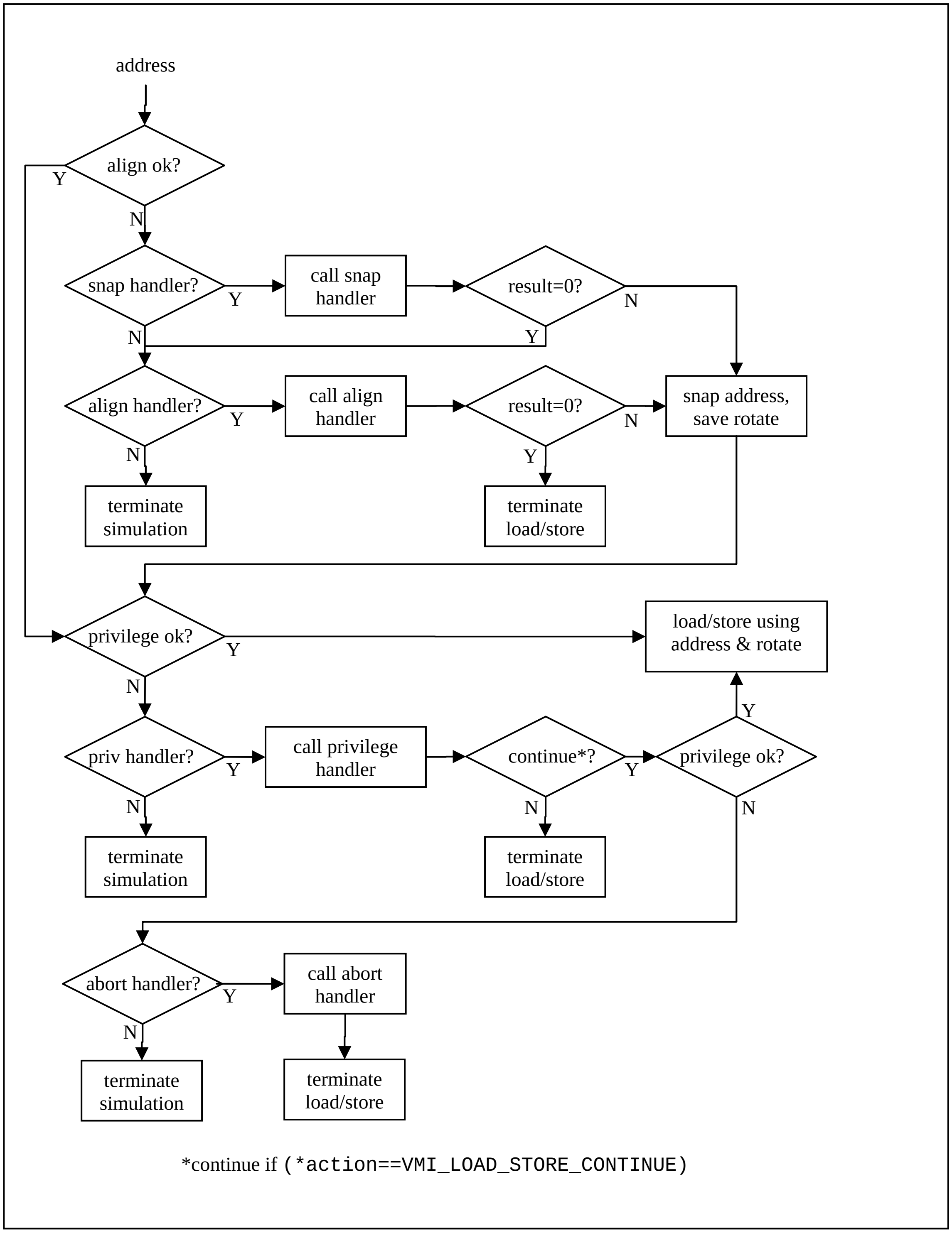


Figure 1: Load/Store Address Snapping and Exception Flow

12.4 Misaligned Fetch Address Snapping

Some processors snap misaligned fetch addresses to even boundaries. For example, the ARC 600/700 series processors snap all fetch addresses to a 2-byte aligned boundary.

There is a specific callback, `fetchSnapCB`, in the `vmiIASAttr` structure for fetch address snapping. The *address snapping callback* should be defined using the macro `VMI_FETCH_SNAP_FN`, defined in `vmiAttrs.h` as:

```
#define VMI_FETCH_SNAP_FN(_NAME) Addr _NAME( \
    vmiProcessorP processor,    \
    Addr          thisPC        \
)
typedef VMI_FETCH_SNAP_FN((*vmiFetchSnapFn));
```

The address snapping callback takes a processor and an address argument and should return that address, appropriately snapped. For example, to snap addresses to a 2-byte boundary:

```
VMI_FETCH_SNAP_FN(or1kFetchSnap) {
    return thisPC & ~1;
}
```

The `vmiIASAttr` structure should reference the fetch address snapping callback:

```
const vmiIASAttr modelAttrs = {

    . . . fields omitted . . .

    ////////////////////////////////////////////////////////////////////////
    // MORPHER CORE ROUTINES
    ////////////////////////////////////////////////////////////////////////

    .morphCB     = or1kMorphInstruction,
    .fetchSnapCB = or1kFetchSnap,


    . . . fields omitted . . .
};
```

13 Modeling Mode-Dependent Behavior (Part 1)

Up to now, all processor instructions have been modeled in a mode-independent way: the actions performed by each instruction have been independent of the current processor state. In real processors, there are usually instructions for which this is not the case. For example, some instructions may be intended for use only in a kernel or supervisor mode, and any attempt to use those instructions in user mode will generate a privileged instruction exception. For the OR1K, one such instruction has already been encountered: `l.rfe`, which should in fact only allow a return from exception in supervisor mode (so the implementation in chapter 12 was incorrect as it takes no account of this).

In this chapter, we will correct the functionality of `l.rfe` so that it takes account of the processor mode and also implement two mode modal instructions, `l.mfspr` and `l.mtspr`. Chapter 14 shows how modal instructions can be modeled differently to give higher performance.

13.1 The Template Modal Model

A template model for the OR1K processor implementing modal instructions can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/11.or1kBehaviorSPR
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/11.or1kBehaviorSPR .
```

Compile the model, harness and application using the `make` command:

```
cd 11.or1kBehaviorSPR
make OPT=1
```

Note that the processor model has been built with compiler optimizations enabled (`OPT=1`) for this example, to get the fastest possible model. This is because we will use the model for performance testing at the end of this chapter.

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is based on the previous model, with the changes listed in following sections.

13.2 Correcting `l.rfe` Behavior

The `l.rfe` instruction should return from an exception only in the case that the processor is in supervisor mode. In user mode, the processor should take an illegal instruction exception (at vector address `0x700`). To implement this behavior, function `morphRFE` in `or1kMorph.c` has been updated as follows:

```
static OR1K_MORPH_FN(morphRFE) {

    vmiLabelP inUserMode = vmimtNewLabel();

    // test the SPR_SR_SM bit in OR1K_SR, setting OR1K_TEMPFLAG
    vmimtBinopRRC(OR1K_BITS, vmi_AND, VMI_NOREG, OR1K_SR, SPR_SR_SM, &flagsTZ);

    // go to label inUserMode if tempFlag set (SPR_SR_SM bit is zero)
    vmimtCondJumpLabel(OR1K_TEMPFLAG, True, inUserMode);

    ////////////////////////////////////////////////////////////////////////////
    // HERE IN SUPERVISOR MODE
    ////////////////////////////////////////////////////////////////////////////

    // set sr from esr (must call or1kSetSR to do this)
    vmimtArgProcessor();
    vmimtArgReg(OR1K_BITS, OR1K_ESR);
    vmimtCall((vmiCallFn)or1kSetSR);

    // return to exception program counter
    vmimtUncondJumpReg(0, OR1K_EPC, VMI_NOREG, vmi_JH_RETURNINT);

    ////////////////////////////////////////////////////////////////////////////
    // HERE IN USER MODE
    ////////////////////////////////////////////////////////////////////////////

    // insert the label targeted by vmimtCondJumpLabel above
    vmimtInsertLabel(inUserMode);

    // take illegal instruction exception
    vmimtArgProcessor();
    vmimtArgUns32(OR1K_EXCPT_ILL);
    vmimtArgUns32(0);
    vmimtCallAttrs((vmiCallFn)or1kTakeException, VMCA_EXCEPTION);
}
```

This function uses an *intra-instruction jump* to execute one of two morphed-code subsequences depending on the setting of the `SPR_SR_SM` bit in the `SR` register. In detail, it works as follows:

```
    vmiLabelP inUserMode = vmimtNewLabel();
```

This allocates a *label* that is used as a target of an intra-instruction jump.

```
    // test the SPR_SR_SM bit in OR1K_SR, setting OR1K_TEMPFLAG
    vmimtBinopRRC(OR1K_BITS, vmi_AND, VMI_NOREG, OR1K_SR, SPR_SR_SM, &flagsTZ);
```

This morphs code to perform a bitwise-and of the `SR` register (identified using `OR1K_SR`) and the constant `SPR_SR_SM` (defined in `or1kStructure.h`). Because argument 3 of `vmimtBinopRRC` is `VMI_NOREG`, the result is discarded. The last (`flags`) argument of `vmimtBinopRRC` is passed this `vmiFlags` structure:

```
const vmiFlags flagsTZ = {
    VMI_NOFLAG_CONST,            // carry in flag not used
    {
        VMI_NOFLAG_CONST,        // carry out flag not used
        VMI_NOFLAG_CONST,        // parity flag not used
        OR1K_TEMPFLAG_CONST,     // offset to zero flag
        VMI_NOFLAG_CONST,        // sign flag not used
        VMI_NOFLAG_CONST         // overflow flag not used
    }
};
```

The `vmiFlags` structure specifies that all flags generated by the bitwise-and should be discarded, except for the *zero* flag, which should be stored in a new temporary flag register in the OR1K processor structure (in `or1kStructure.h`):

```
typedef struct or1kS {

    Bool       carryFlag;       // carry flag
    Bool       overflowFlag;    // overflow flag
    Bool       branchFlag;      // branch flag
    Bool       tempFlag;        // temporary flag

    Uns32      regs[OR1K_REGS]; // basic registers

    Uns32      SR;              // status register
    Uns32      ESR;             // exception status register
    Uns32      EPC;             // exception program counter register
    Uns32      EEAR;            // exception effective address register

    vmiBusPortP busPorts;       // bus port descriptions

} or1k, *or1kP;

// macros to specify target registers in VARIABLE expressions
#define OR1K_CPU_REG(_F)        VMI_CPU_REG(or1kP, _F)
#define OR1K_CPU_TEMP(_F)       VMI_CPU_TEMP(or1kP, _F)
#define OR1K_REG(_R)            OR1K_CPU_REG(regs[_R])
#define OR1K_CARRY              OR1K_CPU_REG(carryFlag)
#define OR1K_OVERFLOW           OR1K_CPU_REG(overflowFlag)
#define OR1K_BRANCH             OR1K_CPU_REG(branchFlag)
#define OR1K_TEMPFLAG           OR1K_CPU_TEMP(tempFlag)
#define OR1K_LINKREG            OR1K_REG(OR1K_LINK)
#define OR1K_SR                 OR1K_CPU_REG(SR)
#define OR1K_ESR                OR1K_CPU_REG(ESR)
#define OR1K_EPC                OR1K_CPU_REG(EPC)
#define OR1K_EEAR               OR1K_CPU_REG(EEAR)

// macros to specify target registers in CONSTANT expressions
#define OR1K_CPU_REG_CONST(_F)  VMI_CPU_REG_CONST(or1kP, _F)
#define OR1K_CPU_TEMP_CONST(_F) VMI_CPU_TEMP_CONST(or1kP, _F)
#define OR1K_REG_CONST(_R)      OR1K_CPU_REG_CONST(regs[_R])
#define OR1K_CARRY_CONST        OR1K_CPU_REG_CONST(carryFlag)
#define OR1K_OVERFLOW_CONST     OR1K_CPU_REG_CONST(overflowFlag)
#define OR1K_TEMPFLAG_CONST     OR1K_CPU_TEMP_CONST(tempFlag)
```

Note that the new `tempFlag` field does not represent a true processor register: it is simply a temporary required for modeling purposes. The flag is special because its value is used as a temporary within one instruction and *need not be saved when the instruction completes*: for such temporaries, the JIT compiler is able to generate more efficient code. To identify a temporary, use the `VMI_CPU_TEMP` and `VMI_CPU_TEMP_CONST` macros as in the above example.

> For best performance, always ensure that all instruction-local temporaries are correctly identified with the `VMI_CPU_TEMP` and `VMI_CPU_TEMP_CONST` macros.

Having generated code that sets the value of the new `tempFlag` field if the processor is not in supervisor mode, `doRFE` then emits code to perform an intra-instruction jump if the flag is set:

```
    // go to label inUserMode if tempFlag set (SPR_SR_SM bit is zero)
    vmimtCondJumpLabel(OR1K_TEMPFLAG, True, inUserMode);
```

Next, code is generated to perform a return from exception in supervisor mode, just as in the previous example:

```
    // set sr from esr (must call or1kSetSR to do this)
    vmimtArgProcessor();
    vmimtArgReg(OR1K_BITS, OR1K_ESR);
    vmimtCall((vmiCallFn)or1kSetSR);

    // return to exception program counter
    vmimtUncondJumpReg(0, OR1K_EPC, VMI_NOREG, vmi_JH_RETURNINT);
```

Now the label is inserted at the location where user mode code starts:

```
    // insert the label targeted by vmimtCondJumpLabel above
    vmimtInsertLabel(inUserMode);
```

And in user mode, `l.rfe` should cause an illegal instruction exception, implemented by a run-time call to `or1kTakeException` (in `or1kUtils.c`):

```
    // take illegal instruction exception
    vmimtArgProcessor();
    vmimtArgUns32(OR1K_EXCPT_ILL);
    vmimtArgUns32(0);
    vmimtCallAttrs((vmiCallFn)or1kTakeException, VMCA_EXCEPTION);
```

As for the `l.sys` instruction, function `vmimtCallAttrs` is used to indicate that this call is an exception which terminates the current code block.

The example at the end of this section shows the new code in action.

13.3 Implementing `l.mtspr`

The `l.mtspr` instruction implements a *move to special purpose register*. It allows a value in an OR1K general purpose register (`r0`, `r1` etc) to be written to a special purpose register (e.g. `sr`, `epc`, `esr`, and many other special purpose registers that are currently unimplemented). The target special purpose register is identified by a unique index number: for example, register `sr` has index `0x11`, register `epc` has index `0x20` and register `esr` has index `0x40`.

The index number of the special purpose register is calculated by adding an index register `ra` and a constant index `k`. This means that if the index register is anything other than `r0`, the special purpose register to update must be identified at *run time* (since there is no way to know the future value of `ra` when morphing code). However, if `ra` is `r0` (which is always zero) we know at *morph time* the SPR index (k) and therefore more efficient code can be created, as we will see below.

`l.mtspr` has been added to the decoder and disassembler in a similar way as for previous instructions. In `or1kMorph.c`, the instruction is implemented by function `morphMTSPR`:

```
static OR1K_MORPH_FN(morphMTSPR) {

    vmiReg ra = getGPR(state->info.r1);
    vmiReg rb = getGPR(state->info.r2);
    Uns32  k  = state->info.c;

    if(VMI_ISNOREG(ra)) {
        // faster variant when ra is r0
        morphMTSPR_ra_0(rb, k);
    } else {
        // slower variant when ra is not r0
        vmimtArgProcessor();
        vmimtArgReg(OR1K_BITS, ra);
        vmimtArgReg(OR1K_BITS, rb);
        vmimtArgUns32(k);
        vmimtCall((vmiCallFn)vmic_MTSPR);
    }
}
```

In the case that `ra` (the index register) is `r0`, it calls `morphMTSPR_ra_0` to emit code that targets a specific special purpose register. Otherwise a run-time call is created to function, `vmic_MTSPR`, which handles writing any special purpose register, like this:

```
static void vmic_MTSPR(or1kP or1k, Uns32 ra, Uns32 rb, Uns32 k) {

    Uns32 sprNum = ra | k;

    switch(sprNum) {

        case SPR_OFF(SPR_SYS,SYS_SR):
            vmic_MTSPR_SYS_SR(or1k, rb);
            break;

        case SPR_OFF(SPR_SYS,SYS_EPC):
            vmic_MTSPR_SYS_EPC(or1k, rb);
            break;

        case SPR_OFF(SPR_SYS,SYS_EEAR):
            vmic_MTSPR_SYS_EEAR(or1k, rb);
            break;
```

```
        case SPR_OFF(SPR_SYS,SYS_ESR):
            vmic_MTSPR_SYS_ESR(or1k, rb);
            break;

        default:
            ignoreMTSPR(or1k, sprNum);
            break;
    }
}
```

To avoid confusion between functions that should be called at *morph time* and those which are only applicable at *run time*, it can be helpful to use function prefixes: for example, use the prefix `vmic_` for a function that can only be called at run time.

The function constructs the special purpose register index and uses a case statement to determine the register to update. It then calls a specific update function for that register. As an example, the specific function to modify the `sr` register is:

```
static void vmic_MTSPR_SYS_SR(or1kP or1k, Uns32 value) {
    if(!IN_SUPERVISOR_MODE(or1k)) {
        or1kTakeException(or1k, OR1K_EXCPT_ILL, 0);
    } else {
        or1kSetSR(or1k, value);
    }
}
```

This function calls `or1kTakeException` if the processor is in user mode, which will cause the processor to take an illegal instruction exception. In supervisor mode, it calls `or1kSetSR` (from `or1kUtils.c`) to update the value of supervisor register `sr`.

In section 13.2, `l.rfe` was implemented using an intra-instruction conditional jump. It could just as well (and more clearly) have been implemented by a call to a run time function that performed the supervisor mode check, as above.

When the index register is `r0`, code to implement the assignment of the special purpose register is created by *morph time* function `morphMTSPR_ra_0`:

```
static void morphMTSPR_ra_0(vmiReg rb, Uns32 sprNum) {

    switch(sprNum) {

        case SPR_OFF(SPR_SYS,SYS_SR):
            vmimtArgProcessor();
            vmimtArgReg(OR1K_BITS, rb);
            vmimtCall((vmiCallFn)vmic_MTSPR_SYS_SR);
            break;

        case SPR_OFF(SPR_SYS,SYS_EPC):
            vmimtArgProcessor();
            vmimtArgReg(OR1K_BITS, rb);
            vmimtCall((vmiCallFn)vmic_MTSPR_SYS_EPC);
            break;

        case SPR_OFF(SPR_SYS,SYS_EEAR):
            vmimtArgProcessor();
```

```
            vmimtArgReg(OR1K_BITS, rb);
            vmimtCall((vmiCallFn)vmic_MTSPR_SYS_EEAR);
            break;

        case SPR_OFF(SPR_SYS,SYS_ESR):
            vmimtArgProcessor();
            vmimtArgReg(OR1K_BITS, rb);
            vmimtCall((vmiCallFn)vmic_MTSPR_SYS_ESR);
            break;

        default:
            vmimtArgProcessor();
            vmimtArgUns32(sprNum);
            vmimtCall((vmiCallFn)ignoreMTSPR);
            break;
    }
}
```

Note that `morphMTSPR_ra_0` determines the register to be written at *morph time*, and emits code that targets the specific register. `vmic_MTSPR` has to perform the equivalent check at *run time*, which will be slower. We will see this in the examples that follow.

13.4 Implementing `l.mfspr`

The `l.mfspr` instruction implements a *move from special purpose register*. It allows a value in an OR1K special purpose register to be assigned to a general purpose register. It is implemented in an analogous way to `l.mfspr` by function `morphMFSPR` in `or1kMorph.c`.

13.5 Root Module Simulation - `platform/harness.c`

A new OP function is used to run simulation in this example:

```
    // run platform
    opRootModuleSimulate(mr);
```

This function simulates all processors in the root module until completion using a built-in scheduling algorithm.

13.6 Testing Illegal Instruction Exceptions

Directory `11.or1kBehaviorSPR/application` contains the following example in file `asmtest.S`:

```
.org 0x700
        ////////////////////////////////////////////////////////////////////////
        // ILLEGAL INSTRUCTION EXCEPTION HANDLER (AT 0x700)
        ////////////////////////////////////////////////////////////////////////
        l.addi      r30,r30,1       // increment count of illegal instructions
        l.sw        -4(r31),r1      // save value in r1
        l.mfspr     r1,r0,0x20      // get epc in r1
        l.addi      r1,r1,4         // move epc past faulting instruction
        l.mtspr     r0,r1,0x20      // set epc from r1
        l.lwz       r1,-4(r31)      // restore original r1
        l.rfe                       // return from exception
```

```
.org 0x10000
        ////////////////////////////////////////////////////////////////////////
        // APPLICATION CODE (AT 0x10000)
        ////////////////////////////////////////////////////////////////////////
.global _start
_start:
        l.ori       r30,r0,0        // r30 = 0 (counts illegal instructions)
        l.ori       r31,r0,0        // r31 = 0 (stack pointer)
        l.mtspr     r0,r0,0x20      // clear epc

        ////////////////////////////////////////////////////////////////////////
        // SUPERVISOR MODE LOOP TEST
        ////////////////////////////////////////////////////////////////////////
        l.ori       r1,r0,2         // r1 = 2 (loop count)
loop1:
        l.mfspr     r2,r0,0x20      // get epc in r2
        l.addi      r2,r2,1         // increment r2
        l.mtspr     r0,r2,0x20      // set epc from r2
        l.addi      r1,r1,-1        // decrement r1
        l.sfeqi     r1,0            // r1==0?
        l.bnf       loop1           // go if not
        l.nop                       // (delay slot)

        ////////////////////////////////////////////////////////////////////////
        // SUPERVISOR MODE FUNCTION CALL TEST
        ////////////////////////////////////////////////////////////////////////
        l.jal       incEPC          // incEPC (in supervisor mode)
        l.nop                       // (delay slot)
        l.mtspr     r0,r0,0x11      // clear supervisor mode

        ////////////////////////////////////////////////////////////////////////
        // USER MODE FUNCTION CALL TEST
        ////////////////////////////////////////////////////////////////////////
        l.jal       incEPC          // incEPC (in user mode)
        l.nop                       // (delay slot)
        l.rfe                       // *ILLEGAL* return from exception

.global exit
exit:
        l.nop

        ////////////////////////////////////////////////////////////////////////
        // FUNCTION CALLED IN BOTH USER AND SUPERVISOR MODE
        ////////////////////////////////////////////////////////////////////////
incEPC:
        l.mfspr     r2,r0,0x20      // get epc in r2
        l.addi      r2,r2,1         // increment r2
        l.mtspr     r0,r2,0x20      // set epc from r2
        l.jr        r9              // return
        l.nop                       // (delay slot)
```

This example begins execution at `_start` in supervisor mode. It then goes twice round `loop1`, incrementing the value of register `epc` (SPR index `0x20`) each time. These instructions are legal because the processor is in supervisor mode.

After the second loop iteration, the processor calls function `incEPC`, which also increments register `epc`.

It then clears supervisor mode with the instruction:

```
        l.mtspr     r0,r0,0x11      // clear supervisor mode
```

Then, in user mode, the processor attempts to execute `l.rfe`. This fails, because it is now in user mode, and the handler at address `0x700` is executed. The handler updates the saved `epc` to skip the faulting instruction and returns (of course, a real handler would do something more useful than this). Finally, it calls `incEPC` again. The attempts to read and write `epc` in this function also fail, calling the handler.

Run the platform using the assembler executable file:

```
platform/harness.$IMPERAS_ARCH.exe --trace \
    --program application/asmtest.OR1K.elf
```

The output from this should be as follows (execution in the illegal instruction exception handler is highlighted in bold):

```
Warning (PC_NRI) No register information callback given for processor 'cpu1'
Info 'cpu1', 0x0000000000010000: l.ori    r30,r0,0x0
Info 'cpu1', 0x0000000000010004: l.ori    r31,r0,0x0
Info 'cpu1', 0x0000000000010008: l.mtspr  r0,r0,32
Info 'cpu1', 0x000000000001000c: l.ori    r1,r0,0x2
Info 'cpu1', 0x0000000000010010: l.mfspr  r2,r0,32
Info 'cpu1', 0x0000000000010014: l.addi   r2,r2,0x1
Info 'cpu1', 0x0000000000010018: l.mtspr  r0,r2,32
Info 'cpu1', 0x000000000001001c: l.addi   r1,r1,0xffffffff
Info 'cpu1', 0x0000000000010020: l.sfeqi  r1,0x0
Info 'cpu1', 0x0000000000010024: l.bnf    0x00010010
Info 'cpu1', 0x0000000000010028: l.nop    0x0
Info 'cpu1', 0x0000000000010010: l.mfspr  r2,r0,32
Info 'cpu1', 0x0000000000010014: l.addi   r2,r2,0x1
Info 'cpu1', 0x0000000000010018: l.mtspr  r0,r2,32
Info 'cpu1', 0x000000000001001c: l.addi   r1,r1,0xffffffff
Info 'cpu1', 0x0000000000010020: l.sfeqi  r1,0x0
Info 'cpu1', 0x0000000000010024: l.bnf    0x00010010
Info 'cpu1', 0x0000000000010028: l.nop    0x0
Info 'cpu1', 0x000000000001002c: l.jal    0x00010048
Info 'cpu1', 0x0000000000010030: l.nop    0x0
Info 'cpu1', 0x0000000000010048: l.mfspr  r2,r0,32
Info 'cpu1', 0x000000000001004c: l.addi   r2,r2,0x1
Info 'cpu1', 0x0000000000010050: l.mtspr  r0,r2,32
Info 'cpu1', 0x0000000000010054: l.jr     r9
Info 'cpu1', 0x0000000000010058: l.nop    0x0
Info 'cpu1', 0x0000000000010034: l.mtspr  r0,r0,17
Info 'cpu1', 0x0000000000010038: l.jal    0x00010048
Info 'cpu1', 0x000000000001003c: l.nop    0x0
Info 'cpu1', 0x0000000000010048: l.mfspr  r2,r0,32
Info 'cpu1', 0x0000000000000700: l.addi   r30,r30,0x1
Info 'cpu1', 0x0000000000000704: l.sw     0xfffffffc(r31),r1
Info 'cpu1', 0x0000000000000708: l.mfspr  r1,r0,32
Info 'cpu1', 0x000000000000070c: l.addi   r1,r1,0x4
Info 'cpu1', 0x0000000000000710: l.mtspr  r0,r1,32
Info 'cpu1', 0x0000000000000714: l.lwz    r1,0xfffffffc(r31)
Info 'cpu1', 0x0000000000000718: l.rfe
Info 'cpu1', 0x000000000001004c: l.addi   r2,r2,0x1
Info 'cpu1', 0x0000000000010050: l.mtspr  r0,r2,32
Info 'cpu1', 0x0000000000000700: l.addi   r30,r30,0x1
Info 'cpu1', 0x0000000000000704: l.sw     0xfffffffc(r31),r1
```

```
Info 'cpu1', 0x0000000000000708: l.mfspr  r1,r0,32
Info 'cpu1', 0x000000000000070c: l.addi   r1,r1,0x4
Info 'cpu1', 0x0000000000000710: l.mtspr  r0,r1,32
Info 'cpu1', 0x0000000000000714: l.lwz    r1,0xfffffffc(r31)
Info 'cpu1', 0x0000000000000718: l.rfe
Info 'cpu1', 0x0000000000010054: l.jr     r9
Info 'cpu1', 0x0000000000010058: l.nop    0x0
Info 'cpu1', 0x0000000000010040: l.rfe
Info 'cpu1', 0x0000000000000700: l.addi   r30,r30,0x1
Info 'cpu1', 0x0000000000000704: l.sw     0xfffffffc(r31),r1
Info 'cpu1', 0x0000000000000708: l.mfspr  r1,r0,32
Info 'cpu1', 0x000000000000070c: l.addi   r1,r1,0x4
Info 'cpu1', 0x0000000000000710: l.mtspr  r0,r1,32
Info 'cpu1', 0x0000000000000714: l.lwz    r1,0xfffffffc(r31)
Info 'cpu1', 0x0000000000000718: l.rfe
Info 'cpu1', 0x0000000000010044: l.nop    0x0
Processor 'cpu1' terminated at 'exit', address 0x10044
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000000   R2 : 00000004   R3 : deadbeef
 R4 : deadbeef   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : 00010040   R10: deadbeef   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: 00000003   R31: 00000000
 PC : 00010048   SR : 00008000   ESR: 00008000   EPC: 00010044
 BF:0 CF:0 OF:0
--------------- --------------- --------------- ---------------

processor has executed 56 instructions
```

13.7 Testing `l.mtspr` and `l.mfspr` Performance

When implementing `l.mtspr` and `l.mfspr`, we optimized the case where the index register is `r0` to improve performance. We can test the effect of this as follows.

13.7.1 Increase `application/asmtest.S` to Loop Count

Modify line 44 of `application/asmtest.S` to greatly increase the number of iterations of loop1 as follows:

```
        l.movhi     r1,0x1000       // r1 = 0x10000000 (loop count)
```

This will cause the loop to be executed over 268 million times, which should take long enough to get meaningful performance numbers.

Then rebuild the test case and rerun (*without* tracing enabled):

```
make -C application
time platform/harness.$IMPERAS_ARCH.exe --program application/asmtest.OR1K.elf
```

The output from this should be:

```
Warning (PC_NRI) No register information callback given for processor 'cpu1'
Processor 'cpu1' terminated at 'exit', address 0x10040
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000000   R2 : 10000002   R3 : deadbeef
```

```
 R4 : deadbeef   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : 00010040   R10: deadbeef   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: 00000003   R31: 00000000
 PC : 00010048   SR : 00008000   ESR: 00008000   EPC: 00010044
 BF:0 CF:0 OF:0
--------------- --------------- --------------- ---------------

processor has executed 1879048234 instructions
```

On a 3.5GHz Intel i7-4770K processor, `time` shows this takes about 1.87 seconds to execute 1,879,048,234 OR1K instructions, giving a simulation speed for this example of about 1,000 simulated MIPS.

13.7.2 Use Index Register r31

Now modify the inner loop in `application/asmtest.S` to use `r31` as the index register instead of `r0` (`r31` happens to hold the value zero in this test case, but isn't *hard wired* to zero like `r0`):

```
        l.mfspr     r2,r31,0x20     // get epc in r2
        l.addi      r2,r2,1         // increment r2
        l.mtspr     r31,r2,0x20     // set epc from r2
        l.addi      r1,r1,-1        // decrement r1
        l.sfeqi     r1,0            // r1==0?
        l.bnf       loop1           // go if not
        l.nop                       // (delay slot)
```

This will cause the identification of the special purpose register to update to be deferred to run time, using calls to `vmic_MTSPR` and `vmic_MTSPR`, instead of calling the specific register SPR register access routines directly (e.g. `vmic_MTSPR_SYS_SR`).

Then rebuild the test case and rerun (again, *without* tracing enabled):

```
make –C application
time platform/harness.$IMPERAS_ARCH.exe --program application/asmtest.OR1K.elf
```

Output is identical to before, but the run time is now 4.0 seconds, giving a simulation speed for this example of about 470 simulated MIPS. Although the change we made caused only a small amount of extra C code to be executed at run time (an extra function call and case statement) simulation performance is *530 MIPS slower than before*.

> It is *very important* when creating a high-performance processor model to do as much work as possible at *morph time* and as little as possible at *run time*. The difference in simulation speed can be dramatic.

13.8 Passing Register Arguments to Embedded Calls

The implementation of `l.mtspr` and `l.mfspr` in this chapter made extensive use of embedded calls, defined by morph-time calls to `vmimtCall` and `vmimtCallAttrs`. When creating such functions there is often an implementation choice: when passing a GPR argument or assigning a GPR result, should the *value* of the register be used, or its *index*?

Recall that the embedded call to get the value of the sr register (when the SPR index is known at morph-time) was implemented like this:

```
static void morphMFSPR_ra_0(vmiReg rd, Uns32 sprNum) {

    switch(sprNum) {

        case SPR_OFF(SPR_SYS,SYS_SR):
            vmimtArgProcessor();
            vmimtCallResult((vmiCallFn)vmic_MFSPR_SYS_SR, OR1K_BITS, rd);
            break;

        . . . lines omitted . . .
    }
}
```

And that the implementation of vmic_MFSPR_SYS_SR was like this:

```
static Uns32 vmic_MFSPR_SYS_SR(or1kP or1k) {
    if(!IN_SUPERVISOR_MODE(or1k)) {
        or1kTakeException(or1k, OR1K_EXCPT_ILL, 0);
        return 0;
    } else {
        return or1kGetSR(or1k);
    }
}
```

In other words, we chose to return the value of register SR from the embedded function and do the final result assignment by means of the vmimtCallResult target. We could instead have structured the code to pass the *index* of the result register like this:

```
static void morphMFSPR_ra_0(Uns32 rd, Uns32 sprNum) {

    switch(sprNum) {

        case SPR_OFF(SPR_SYS,SYS_SR):
            vmimtArgProcessor();
            vmimtArgUns32(rd);
            vmimtCall((vmiCallFn)vmic_MFSPR_SYS_SR);
            break;

        . . . lines omitted . . .
    }
}
```

And perform the update of the result register in vmic_MFSPR_SYS_SR, like this:

```
static void vmic_MFSPR_SYS_SR(or1kP or1k, Uns32 rd) {
    if(!IN_SUPERVISOR_MODE(or1k)) {
        or1kTakeException(or1k, OR1K_EXCPT_ILL, 0);
        return 0;
    } else if(rd) {
        or1k->regs[rd] = or1kGetSR(or1k);
    }
```



```
}
```

As a general rule, it is much better *not* to use index-based code like this, for several reasons:

1. The JIT code generator can create better code if it is aware of the arguments and results of each embedded call (especially if calls are defined to be *pure*, i.e. to have `VMCA_PURE` call attributes – pure functions return a result dependent only on their arguments).
2. The code is often simpler. In the above example, the callback function had to take special action to ensure that `r0` is never updated (since this register is hard-wired to zero). This happens automatically in the `vmimtCallResult`-based version.
3. When the model is enhanced to support instruction attributes (see a later chapter), much more information is automatically available.

14 Modeling Mode-Dependent Behavior (Part 2)

In chapter 12.4, we saw how to model mode-dependent processor instructions with an example running at up to 1,000 simulated MIPS. This chapter shows how to get even faster performance on the same test case.

One significant problem with the implementation of the `l.rfe`, `l.mtspr` and `l.mfspr` instructions in chapter 12.4 is that they are coded to implement both kernel and user mode, and they select which behavior to perform at run time. The timing experiments at the end of the last chapter showed that it is possible to get dramatically faster performance if work can be moved from run time to morph time. Is it somehow possible to perform the supervisor-mode check at morph time to improve performance of these instructions?

Recall that the instruction morpher callback function defined using the `VMI_MORPH_FN` macro is passed the current processor as one of its arguments, which is then saved in the `or1kMorphState` structure:

```
VMI_MORPH_FN(or1kMorphInstruction) {

    or1kP          or1k  = (or1kP)processor;
    or1kMorphState state = {{0}};

    // decode instruction
    or1kDecode(or1k, thisPC, &state.info);

    // get morpher attributes for the decoded instruction and initialize other
    // state fields
    state.attrs       = &or1kMorphTable[state.info.type];
    state.or1k        = or1k;
    state.inDelaySlot = inDelaySlot;

    . . . lines omitted . . .
}
```

We know that we can determine whether an or1k is in supervisor mode using the `IN_SUPERVISOR_MODE` macro. Therefore, we can tell when morphing code whether user or supervisor mode code must be generated, which is exactly what we require. We will now see how this can be used to generate a faster model.

14.1 The Template Fast Modal Model

A template fast model for the OR1K processor implementing modal instructions can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/12.or1kBehaviorModeDict
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/12.or1kBehaviorModeDict .
```


Compile the model, harness and application using the `make` command:

```
cd 12.or1kBehaviorModeDict
make OPT=1
```

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is based on the previous model, with the changes listed in following sections.

14.2 Remove Temporary Flag - `or1kStructure.h`

The temporary flag `tempFlag` introduced in the previous chapter is no longer required and has been deleted.

14.3 Mode-Dependent JIT Functions - `or1kMorph.c`

Function `morphRFE` has been recoded to optimize for the current processor supervisor mode as follows:

```
static OR1K_MORPH_FN(morphRFE) {

    if(IN_SUPERVISOR_MODE(state->or1k)) {

        // set sr from esr (must call or1kSetSR to do this)
        vmimtArgProcessor();
        vmimtArgReg(OR1K_BITS, OR1K_ESR);
        vmimtCall((vmiCallFn)or1kSetSR);

        // return to exception program counter
        vmimtUncondJumpReg(0, OR1K_EPC, VMI_NOREG, vmi_JH_RETURNINT);

    } else {

        // take illegal instruction exception
        vmimtArgProcessor();
        vmimtArgUns32(OR1K_EXCPT_ILL);
        vmimtArgUns32(0);
        vmimtCallAttrs((vmiCallFn)or1kTakeException, VMCA_EXCEPTION);
    }
}
```

What this does is as follows:

1. If the processor is currently in supervisor mode, it emits code to update register `sr` from register `esr` and return to the address in register `epc`, exactly as in the previous example.
2. Otherwise (the processor is in user mode), it emits code to jump to the illegal instruction exception vector.

Functions `morphMTSPR` and `morphMFSPR` have also been changed in an analogous way. Here is the new implementation of `morphMTSPR`:

```
static OR1K_MORPH_FN(morphMTSPR) {

    or1kP  or1k = state->or1k;
    vmiReg ra   = getGPR(state->info.r1);
    vmiReg rb   = getGPR(state->info.r2);
    Uns32  k    = state->info.c;

    if(VMI_ISNOREG(ra)) {

        // faster variant when ra is r0 - select either supervisor mode or user
        // mode function, based on current mode setting in sr
        if(IN_SUPERVISOR_MODE(or1k)) {
            morphMTSPR_ra_0_SM(or1k, rb, k);
        } else {
            morphMTSPR_ra_0_UM(or1k, rb, k);
        }

    } else {

        // slower variant when ra is not r0
        vmimtArgProcessor();
        vmimtArgReg(OR1K_BITS, ra);
        vmimtArgReg(OR1K_BITS, rb);
        vmimtArgUns32(k);

        // select either supervisor mode or user mode callback, based on
        // current mode setting in sr
        if(IN_SUPERVISOR_MODE(or1k)) {
            vmimtCall((vmiCallFn)vmic_MTSPR_SM);
        } else {
            vmimtCall((vmiCallFn)vmic_MTSPR_UM);
        }
    }
}
```

So user and supervisor mode behaviors are implemented separately. Here is the supervisor-mode function used when the SPR register to which to assign is known at morph time:

```
static void morphMTSPR_ra_0_SM(or1kP or1k, vmiReg rb, Uns32 sprNum) {

    switch(sprNum) {

        case SPR_OFF(SPR_SYS,SYS_SR):
            vmimtArgProcessor();
            vmimtArgReg(OR1K_BITS, rb);
            vmimtCall((vmiCallFn)or1kSetSR);
            break;

        case SPR_OFF(SPR_SYS,SYS_EPC):
            vmimtMoveRR(OR1K_BITS, OR1K_EPC, rb);
            break;

        case SPR_OFF(SPR_SYS,SYS_EEAR):
            vmimtMoveRR(OR1K_BITS, OR1K_EEAR, rb);
            break;
```

```
        case SPR_OFF(SPR_SYS,SYS_ESR):
            vmimtMoveRR(OR1K_BITS, OR1K_ESR, rb);
            break;

        default:
            morphMTSPR_ra_0_Default(sprNum);
            break;
    }
}
```

The morph-time function `vmimtMoveRR` is used here to specify a direct register-to-register assignment in three cases (no embedded call is required). The user-mode function simply calls the exception handler as follows:

```
static void morphMTSPR_ra_0_UM(or1kP or1k, vmiReg rb, Uns32 sprNum) {

    switch(sprNum) {

        case SPR_OFF(SPR_SYS,SYS_SR):
        case SPR_OFF(SPR_SYS,SYS_EPC):
        case SPR_OFF(SPR_SYS,SYS_EEAR):
        case SPR_OFF(SPR_SYS,SYS_ESR):
            vmimtArgProcessor();
            vmimtArgUns32(OR1K_EXCPT_ILL);
            vmimtArgUns32(0);
            vmimtCallAttrs((vmiCallFn)or1kTakeException, VMCA_EXCEPTION);
            break;

        default:
            morphMTSPR_ra_0_Default(sprNum);
            break;
    }
}
```

14.4 Testing Optimized Illegal Instruction Exceptions

Run the platform using the assembler executable file:

```
platform/harness.$IMPERAS_ARCH.exe –-trace \
    --program application/asmtest.OR1K.elf
```

The output from this should be as follows (execution in the illegal instruction exception handler is highlighted in bold):

```
Warning (PC_NRI) No register information callback given for processor ‘cpu1’
Info 'cpu1', 0x0000000000010000: l.ori    r30,r0,0x0
Info 'cpu1', 0x0000000000010004: l.ori    r31,r0,0x0
Info 'cpu1', 0x0000000000010008: l.mtspr  r0,r0,32
Info 'cpu1', 0x000000000001000c: l.ori    r1,r0,0x2
Info 'cpu1', 0x0000000000010010: l.mfspr  r2,r0,32
Info 'cpu1', 0x0000000000010014: l.addi   r2,r2,0x1
Info 'cpu1', 0x0000000000010018: l.mtspr  r0,r2,32
Info 'cpu1', 0x000000000001001c: l.addi   r1,r1,0xffffffff
Info 'cpu1', 0x0000000000010020: l.sfeqi  r1,0x0
Info 'cpu1', 0x0000000000010024: l.bnf    0x00010010
Info 'cpu1', 0x0000000000010028: l.nop    0x0
Info 'cpu1', 0x0000000000010010: l.mfspr  r2,r0,32
Info 'cpu1', 0x0000000000010014: l.addi   r2,r2,0x1
Info 'cpu1', 0x0000000000010018: l.mtspr  r0,r2,32
```

```
Info 'cpu1', 0x000000000001001c: l.addi   r1,r1,0xffffffff
Info 'cpu1', 0x0000000000010020: l.sfeqi  r1,0x0
Info 'cpu1', 0x0000000000010024: l.bnf    0x00010010
Info 'cpu1', 0x0000000000010028: l.nop    0x0
Info 'cpu1', 0x000000000001002c: l.jal    0x00010048
Info 'cpu1', 0x0000000000010030: l.nop    0x0
Info 'cpu1', 0x0000000000010048: l.mfspr  r2,r0,32
Info 'cpu1', 0x000000000001004c: l.addi   r2,r2,0x1
Info 'cpu1', 0x0000000000010050: l.mtspr  r0,r2,32
Info 'cpu1', 0x0000000000010054: l.jr     r9
Info 'cpu1', 0x0000000000010058: l.nop    0x0
Info 'cpu1', 0x0000000000010034: l.mtspr  r0,r0,17
Info 'cpu1', 0x0000000000010038: l.jal    0x00010048
Info 'cpu1', 0x000000000001003c: l.nop    0x0
Info 'cpu1', 0x0000000000010048: l.mfspr  r2,r0,32
Info 'cpu1', 0x000000000001004c: l.addi   r2,r2,0x1
Info 'cpu1', 0x0000000000010050: l.mtspr  r0,r2,32
Info 'cpu1', 0x0000000000010054: l.jr     r9
Info 'cpu1', 0x0000000000010058: l.nop    0x0
Info 'cpu1', 0x0000000000010040: l.rfe
Info 'cpu1', 0x0000000000000700: l.addi   r30,r30,0x1
Info 'cpu1', 0x0000000000000704: l.sw     0xfffffffc(r31),r1
Info 'cpu1', 0x0000000000000708: l.mfspr  r1,r0,32
Info 'cpu1', 0x000000000000070c: l.addi   r1,r1,0x4
Info 'cpu1', 0x0000000000000710: l.mtspr  r0,r1,32
Info 'cpu1', 0x0000000000000714: l.lwz    r1,0xfffffffc(r31)
Info 'cpu1', 0x0000000000000718: l.rfe
Info 'cpu1', 0x0000000000010044: l.nop    0x0
Processor 'cpu1' terminated at 'exit', address 0x10044
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000000   R2 : 00000004   R3 : deadbeef
 R4 : deadbeef   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : 00010040   R10: deadbeef   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: 00000001   R31: 00000000
 PC : 00010048   SR : 00008000   ESR: 00008000   EPC: 00010044
 BF:0 CF:0 OF:0
--------------- --------------- --------------- ---------------

processor has executed 42 instructions
```

Comparing with the log output from chapter 12.4, it is obvious that something went wrong: only 42 instructions have been executed instead of 56, and the exception vector was executed only once instead of three times. The problem is that the application function `incEPC` at address `0x00010048` *is called both in supervisor and user mode*. When it is called first, the processor is in supervisor mode: at this point, a fragment of translated native code is created for the function body and saved in the processor code dictionary. On the second call (in user mode) the same fragment is re-executed. This is wrong because the code fragment was *specific to supervisor mode*.

14.5 Using Multiple Code Dictionaries

One way to solve the problem we have seen in this example is to use *two* code dictionaries: one for supervisor mode code fragments and one for user mode code fragments. To specify that we need multiple dictionaries, the list of dictionary names in `or1kAttrs.c` has been changed to:

```
static const char *dictNames[] = {"SUPERVISOR", "USER", 0};
```

This definition says that we want this processor to have two dictionaries, the first (index 0) called `SUPERVISOR` and the second (index 1) called `USER`[7].

When a processor starts executing it initially uses the *first* dictionary in this list (in this case `SUPERVISOR`). However, the current dictionary can be switched at any time using the VMI Run Time API function `vmirtSetMode`:

```
void vmirtSetMode(vmiProcessorP processor, Uns32 mode);
```

The `mode` argument in this prototype is the zero-based offset into the `dictNames` list of the new dictionary. So to switch to the supervisor mode dictionary, we would use:

```
    vmirtSetMode((vmiProcessorP)or1k, 0);
```

And to switch to the user mode dictionary, we would use:

```
    vmirtSetMode((vmiProcessorP)or1k, 1);
```

To use the new supervisor and user mode dictionaries, `or1kUtils.c` has been updated as follows:

```
static void setSRSwitchMode(or1kP or1k, Uns32 value) {

    Uns32 oldSM = or1k->SR & SPR_SR_SM;
    Uns32 newSM = value    & SPR_SR_SM;

    // set the SR field
    or1k->SR = value;

    // switch mode if required
    if(MODAL && (oldSM != newSM)) {
        or1kMode newMode = newSM ? OR1K_MODE_SUPERVISOR : OR1K_MODE_USER;
        vmirtSetMode((vmiProcessorP)or1k, newMode);
    }
}

void or1kSetSR(or1kP or1k, Uns32 value) {

    // it is never possible to clear the fixed-one (FO) bit
    value |= SPR_SR_FO;

    // set the SR
    setSRSwitchMode(or1k, value);

    // set the current branch flag, carry flag and overflow flag from the SR
    or1k->branchFlag   = GET_BIT(value, SPR_SR_F);
```

[7] The processor model can have as many dictionaries as required (as long as there is at least one). The number of dictionaries is the number of strings in this zero-terminated array.

```
    or1k->carryFlag    = GET_BIT(value, SPR_SR_CY);
    or1k->overflowFlag = GET_BIT(value, SPR_SR_OV);
}

void or1kEnterSupervisorMode(or1kP or1k) {

    const Uns32 clearBits = (
        SPR_SR_IEE |    // interrupt enable
        SPR_SR_TEE |    // tick timer enable
        SPR_SR_DME |    // data MMU enable
        SPR_SR_IME |    // instruction MMU enable
        SPR_SR_OVE      // overflow exception enable
    );

    // save the current status register in esr
    or1k->ESR = or1kGetSR(or1k);

    // mask out the 'clear' bits and mask in supervisor mode
    setSRSwitchMode(or1k, (or1k->ESR & ~clearBits) | SPR_SR_SM);
}
```

The existing functions `or1kSetSR` and `or1kEnterSupervisorMode` have been modified so that they no longer update the `SR` field directly, but instead call a new function `setSRSwitchMode`. This new function determines whether the `SM` (supervisor mode) bit in the `sr` register has changed; if it has, it calls `vmirtSetMode` to switch code dictionaries. The index numbers for each mode are specified in `or1kStructure.h` as:

```
typedef enum or1kModeE {
    OR1K_MODE_SUPERVISOR,
    OR1K_MODE_USER,
    OR1K_MODE_LAST
} or1kMode;
```

To use the new modal code, we need to recompile the processor model with `MODAL` defined:

```
make clean
make OPT=1 MODAL=1
```

And then rerun the example:

```
platform/harness.$IMPERAS_ARCH.exe –-trace \
    --program application/asmtest.OR1K.elf
```

The output from this should now be correct: 56 instructions executed and three calls to the exception handler.

Now that the model is functionally correct, we should determine what effect the changes have had on its performance. To do this, redo exactly the steps described in section 13.7. On a 3.5GHz Intel i7-4770K processor the results are as follows:

1. Using `r0` for the index register: 1,879,048,234 instructions in 0.57 seconds (3,300 simulated MIPS);

2. Using `r31` for the index register: 1,879,048,234 instructions in 4.3 seconds (437 simulated MIPS).

The first example (where the SPR is known at morph time) is vastly quicker – over three billion simulated instructions per second! Performance is similar for the second example - 437 simulated MIPS, as compared to 470 previously.

14.6 Cautionary Notes about Code Dictionaries

Although the use of multiple code dictionaries has significantly improved performance in these examples, this technique is not a panacea and should be used with discretion. This section describes the issues related to code dictionaries in more detail.

14.6.1 `vmirtSetMode` is Slow

Although we vastly accelerated `l.mtspr` and `l.mfspr`, that acceleration has a cost: we have added a call to `vmirtSetMode`, which slows down any instruction that causes a mode switch (i.e. which updates register `sr`). This can be seen using a small OR1K test case as follows:

```
.org 0xc00
        ////////////////////////////////////////////////////////////////////////
        // SYS VECTOR (AT 0xc00)
        ////////////////////////////////////////////////////////////////////////
        l.rfe                        // return from exception (SWITCHES MODE)

.org 0x10000
        ////////////////////////////////////////////////////////////////////////
        // APPLICATION CODE (AT 0x10000)
        ////////////////////////////////////////////////////////////////////////
.global _start
_start:
        l.mtspr     r0,r0,0x11      // clear supervisor mode
        l.movhi     r1,0x0100       // r1 = 0x01000000 (loop count)
loop1:
        l.sys       0               // call sys (SWITCHES MODE)
        l.addi      r1,r1,-1        // decrement r1
        l.sfeqi     r1,0            // r1==0?
        l.bnf       loop1           // go if not
        l.nop                       // (delay slot)

.global exit
exit:
        l.nop
```

This example performs a tight loop of six instructions of which two cause a mode switch.

On a 3.5GHz Intel i7-4770K processor, this executes 100,663,299 simulated instructions in 1.58 seconds using the current model (a simulated speed of only 64 MIPS). On the processor model used in chapter 12.4, the run time is 1.22 seconds (a simulated speed of 83 MIPS).

Whether performance is better with multiple dictionaries therefore depends on the frequency of mode switching instructions compared with the frequency of instructions that can be optimized when there are multiple dictionaries: in some cases, application code may actually run slower when multiple dictionaries are used.

14.6.2 Model Code is More Complicated

The model needs to be carefully designed to ensure that dictionary code is always consistent with the simulated processor state. This is generally fairly easy as long as any code that could affect the mode is channeled through a single routine (`setSRSwitchMode` in this case). It is easy to get difficult-to-find bugs in poorly-structured models where calls to `vmirtSetMode` are not carefully controlled. As a general rule, there should only be one call to `vmirtSetMode` in a model, and this should be right next to code that updates the processor model register that affects the mode.

15 Implementing a Tick Timer

We have already seen in chapter 12 how synchronous exceptions (for example, alignment exceptions) can be efficiently modeled. We will now see how to model *asynchronous exceptions*, or *interrupts*. The VMI modeling API allows generic external exception behavior to be specified, as we will see in chapter 16. Additionally, it allows *tick timer* exceptions to be modeled very efficiently: the subject of this chapter.

15.1 OR1K Tick Timer Overview

The OR1K tick timer is controlled by two processor registers, the *tick timer mode register* (`TTMR`) and the *tick timer count register* (`TTCR`). These two SPR registers may be read and written using the `l.mfspr` and `l.mtspr` instructions we have seen previously (`TTMR` has SPR index `0x5000`, `TTCR` has SPR index `0x5100`).

The `TTCR` register is a 32-bit register that is incremented on each cycle when enabled by the `TTMR` register, as described below.

The `TTMR` register is subdivided into fields as follows:

Bit	31:30	29	28	27:0
Identifier	M	IE	IP	TP
R/W	R/W	R/W	R	R/W

The fields have the following meanings:

TP	**Time Period**
	`0x0000000`: shortest comparison time period
	`0xfffffff`: longest comparison time period
IP	**Interrupt Pending**
	`0`: tick timer interrupt is not pending
	`1`: tick timer interrupt is pending
	(`IP` can be *cleared* by writing 0 with `l.mtspr`, but may not be *set*)
IE	**Interrupt Enable**
	`0`: tick timer does not generate interrupt
	`1`: tick timer generates interrupt when `TTMR[TP]` matches `TTCR[27:0]`
M	**Mode**
	`00`: timer is disabled
	`01`: timer is restarted when `TTMR[TP]` matches `TTCR[27:0]`
	`10`: timer stops when `TTMR[TP]` matches `TTCR[27:0]`
	`11`: timer does not stop when `TTMR[TP]` matches `TTCR[27:0]`
	(if the timer is stopped in mode `10`, writing to `TTCR` restarts it).

In our model, both `TTCR` and `TTMR` will be set to zero at reset, so the tick timer will initially be disabled.

15.2 Tick Timer Modeling Considerations

In true hardware, tick timers usually count processor *cycles*. In architectural models that are not cycle accurate, a common approximation is instead to count processor *instructions*. We will make this approximation in this OR1K model.

The tick timer could be modeled directly using the VMI API. At the start of every instruction, we could, for example, emit a call to a function that does the following:

1. Determine whether the counter is enabled by `TTMR[M]`.
2. If so, increment `TTCR` and compare `TTCR[27:0]` against `TTMR[TP]`.
3. If `TTCR[27:0]` and `TTMR[TP]` match, update state to stop the counter (if `TTMR[M]` is `10`) and set `TTMR[IP]` (if `TTMR[IE]` is set).
4. If `TTMR[IP]`and `SR[TEE]` are set, make a call to the exception vector at `0x500`.

This would work perfectly well, but would be very slow. A much more efficient model can be made by using a combination of three routines from the VMI Run Time Function API:

```
Uns64 vmirtGetICount(vmiProcessorP processor);
```

`vmirtGetICount` returns a 64-bit count giving the total number of instructions that the processor has executed since simulation started.

```
void vmirtSetICountInterrupt(vmiProcessorP processor, Uns64 iDelta);
```

`vmirtSetICountInterrupt` causes a model callback function to be executed after `iDelta` more processor instructions have been simulated. The callback function is used to indicate whether the counter expiry alters the processor's behavior (whether an exception handler should be called, for example).

```
void vmirtClearICountInterrupt(vmiProcessorP processor);
```

`vmirtClearICountInterrupt` disables any instruction count interrupt previously enables using `vmirtSetICountInterrupt`.

A much more efficient model can be built using these functions as follows:

1. When `TTCR` or `TTMR` are written, determine the *implied* timer expiry count – in other words, after what count would the timer expire given the current SPR settings?
2. Use `vmirtSetICountInterrupt` to schedule a model callback after that count, or `vmirtClearICountInterrupt` to deschedule the callback if required.
3. When the callback is activated, schedule a call to the tick timer exception vector if the exception is enabled.
4. Do not model the `TTCR` register directly by incrementing it each instruction. Instead, derive the value if `TTCR` when requested using the processor instruction count returned by `vmirtGetICount` (in a similar manner as previously used for the status register `SR`).

Following sections describe the OR1K tick timer modeled using this approach.

15.3 The Template Tick Timer Model

A template model for the OR1K processor implementing a tick timer can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/13.or1kBehaviorTickTimer
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/13.or1kBehaviorTickTimer .
```

Compile the model, harness and application using the `make` command:

```
cd 13.or1kBehaviorTickTimer
make OPT=1
```

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is based on the previous model, with the changes listed in following sections.

15.4 Adding Timer Registers - `or1kStructure.h`

The OR1K processor structure has been updated as follows:

```
typedef struct or1kS {

    Bool        carryFlag;        // carry flag
    Bool        overflowFlag;     // overflow flag
    Bool        branchFlag;       // branch flag

    Uns32       regs[OR1K_REGS]; // basic registers

    Uns32       SR;               // status register
    Uns32       ESR;              // exception status register
    Uns32       EPC;              // exception program counter
    Uns32       EEAR;             // exception effective address register
    Uns32       TTCR;             // tick timer count register
    Uns32       TTCRSetCount;     // cycle count when TTCR set
    Bool        timerRunning;     // whether the timer is running

    union {                       // tick timer mode register
        Uns32 TTMR;
        struct {
            Uns32 TTMR_TP: 28;    // timeout count
            Uns32 TTMR_IP: 1;     // interrupt pending
            Uns32 TTMR_IE: 1;     // interrupt enable
            Uns32 TTMR_M : 2;     // timer mode
        };
    };
```

```
    vmiBusPortP busPorts;       // bus port descriptions

} or1k, *or1kP;
```

We have added fields TTCR and TTMR that will be used to model the tick timer SPR registers. There are also two fields TTCRSetCount and timerRunning which do not correspond to processor registers but which are modeling artifacts: TTCRSetCount records the processor instruction count when TTCR is written (required to derive the value of TTCR in later instructions); timerRunning is a boolean that indicates whether or not TTCR should be incremented each instruction.

15.5 Timer Register Read and Write - or1kMorph.c

This file has been modified to enhance l.mtspr and l.mfspr to allow reading and writing of TTCR and TTMR registers. TTCR is accessed by calling two new functions, or1kGetTTCR and or1kSetTTCR, implemented in or1kExceptions.c as described below. TTMR is written by or1kSetTTMR, also implemented in or1kExceptions.c.

15.6 Adding Timer Exceptions - or1kExceptions.c

This file implements most of the new functionality to implement tick timer exceptions. The changes are as below.

```
typedef enum TTMRmodeE {
    TTMR_DISABLED = 0,  //  TTCR does not run
    TTMR_RESTART  = 1,  //  TTCR counts up until TTMR_TP, then restarts at 0
    TTMR_ONCE     = 2,  //  TTCR counts up until TTMR_TP, then stops
    TTMR_FREE     = 3   //  TTCR counts up, overflowing at max
} TTMRmode;
```

This enumeration gives names for the four timer modes.

```
inline static Uns32 getThisICount(or1kP or1k) {
    return (Uns32)vmirtGetICount((vmiProcessorP)or1k);
}

inline static Uns32 getTTCR(or1kP or1k) {
    if(or1k->timerRunning) {
        return or1k->TTCR - or1k->TTCRSetCount + getThisICount(or1k);
    } else {
        return or1k->TTCR;
    }
}
```

getTTCR is an internal routine that returns the current effective value of the TTCR register. If the timer is running, TTCR is derived as follows:

1. Get the TTCR value recorded with the model.
2. Subtract the processor instruction count *when TTCR was written.*
3. Add the *current processor instruction count.*

If the timer is not running, the current TTCR value stored in the model is used.

```
static void setTTCR(or1kP or1k, Uns32 TTCR) {
```

```
    // update fields dependent on TTCR
    or1k->TTCR         = TTCR;
    or1k->TTCRSetCount = getThisICount(or1k);

    // if the timer is running, calculate the cycle delay to any interrupt
    // (28 bits maximum) and schedule timer interrupt
    if(or1k->timerRunning) {
        Uns32 iCount = (or1k->TTMR_TP-TTCR-1) & 0xfffffff;
        vmirtSetICountInterrupt((vmiProcessorP)or1k, iCount);
    } else {
        vmirtClearICountInterrupt((vmiProcessorP)or1k);
    }
 }
```

`setTTCR` is an internal routine that is called when the `TTCR` register value is updated. It first saves the new `TTCR` value in the processor model and saves the current processor instruction count in `TTCRSetCount` (this is required so that the correct implied value of `TTCR` can be derived later). Next, if the timer is running, it calculates the implied timeout to counter expiry: the delta to the expiry instruction is the difference between `TTMR_TP` and `TTCR[27:0]` (masked to 28 bits), so it calls `vmirtSetICountInterrupt` to schedule a model callback after this number of instructions. If the timer is not running, it calls `vmirtClearICountInterrupt` to deschedule the callback.

```
Uns32 or1kGetTTCR(or1kP or1k) {
    return getTTCR(or1k);
}

void or1kSetTTCR(or1kP or1k, Uns32 TTCR) {

    // restart the timer if mode is TTMR_ONCE
    if(or1k->TTMR_M==TTMR_ONCE) {
        or1k->timerRunning = True;
    }

    setTTCR(or1k, TTCR);
}
```

These two routines implement the public interface to read and write the `TTCR` register. Note that writing `TTCR` when the timer mode is `TTMR_ONCE` causes the timer to be restarted if it is stopped.

```
void or1kSetTTMR(or1kP or1k, Uns32 TTMR) {

    Uns32 TTCR = getTTCR(or1k);

    // update TTMR, recording old and new values of TTMR_IP
    Bool oldIP = or1k->TTMR_IP;
    or1k->TTMR = TTMR;
    Bool newIP = or1k->TTMR_IP;

    // TTMR_IP must not be set by l.mtspr!
    if(!oldIP && newIP) {
        or1k->TTMR_IP = 0;
    }

    // start the timer if mode is TTMR_RESTART or TTMR_FREE
    // (for TTMR_ONCE, timer is restarted by write to TTCR)
```

```
    if((or1k->TTMR_M==TTMR_RESTART) || (or1k->TTMR_M==TTMR_FREE)) {
        or1k->timerRunning = True;
    }

    setTTCR(or1k, TTCR);
}
```

`or1kSetTTMR` implements the public interface to write `TTMR`. It first gets the current derived value of `TTCR`. It then sets the `TTMR` field in the processor structure, ensuring that the `TTMR_IP` bit does not change from 0 to 1 (`l.mtspr` cannot be used to set the interrupt pending bit, only to clear it). If the new mode is either `TTMR_RESTART` or `TTMR_FREE`, the timer is then restarted by setting `timerRunning`. Finally, `setTTCR` is called to reset the implied `TTCR` to the original value.

`TTCR` must be read using `getTTCR` and restored using `setTTCR` around the body of this routine for two reasons:

1. The way in which the derived value of `TTCR` is generated depends on the current setting of `timerRunning`. If `setTTCR` is not called, the next call to `getTTCR` will return a bogus value.
2. `setTTCR` is responsible for scheduling the instruction count callback, using `vmirtSetICountInterrupt`. If `setTTCR` isn't called, the instruction count callback will occur at the wrong time because changes to `TTMR` that affect the timeout (for example, and update of `TTMR[TP]`) won't be taken into account.

```
VMI_ICOUNT_FN(or1kICountPendingCB) {

    or1kP or1k = (or1kP)processor;

    switch(or1k->TTMR_M) {

        case TTMR_RESTART:
            // restart the timer from 0 on the NEXT instruction
            setTTCR(or1k, -1);
            break;

        case TTMR_FREE:
            // schedule the next interrupt event
            setTTCR(or1k, getTTCR(or1k));
            break;

        case TTMR_ONCE:
            // stop the timer on the NEXT instruction count
            or1k->TTCR         = (getTTCR(or1k)+1) & 0xfffffff;
            or1k->timerRunning = False;
            break;

        case TTMR_DISABLED:
            // how did we get here?
            VMI_ABORT("timer interrupt, but timer was disabled");
            break;
    }

    // if interrupt generation is enabled, set TTMR_IP
    if(or1k->TTMR_IE) {
        or1k->TTMR_IP = 1;
    }
```

```
    // handle exception if required
    if(takeTEE(or1k)) {
        vmirtDoSynchronousInterrupt(processor);
    }
}
```

Function `or1kICountPendingCB` is the callback that is called when the instruction count timeout specified by `vmirtSetICountInterrupt` has elapsed. The function prototype is specified in the VMI header file `vmiTypes.h` as follows:

```
#define VMI_ICOUNT_FN(_NAME) void _NAME( \
    vmiProcessorP  processor,    \
    vmiModelTimerP timer,        \
    Uns64          iCount,       \
    void          *userData      \
)
```

The arguments to this function are as follows:

1. The processor on which the timer has expired;
2. An argument `timer` of type `vmiModelTimerP`. This is an opaque type representing the implicit processor timer which is managed by the functions `vmirtSetICountInterrupt` and `vmirtClearICountInterrupt`;
3. An argument `iCount`, giving the current processor instruction count when the callback is activated;
4. A `userData` argument, which is always `NULL` for the implicit processor timer.

The function should update the processor state to reflect any changes caused by the timer expiry (for example, setting a pending-timer-interrupt bit). If necessary, it should signal that the processor needs to stop what it is doing and handle an exception by calling `vmirtDoSynchronousInterrupt`, as described below.

Based on the current timer mode setting when the timer expires, the processor state is updated in one of several ways:

```
        case TTMR_RESTART:
            // restart the timer from 0 on the NEXT instruction
            setTTCR(or1k, -1);
            break;
```

If the mode is `TTMR_RESTART`, the timer needs to restart from 0 at the *next* instruction. To do this, the callback sets `TTCR` to `-1` now; when the timer is incremented before the next instruction is executed, it will have the value 0.

```
        case TTMR_FREE:
            // schedule the next interrupt event
            setTTCR(or1k, getTTCR(or1k));
            break;
```

With the timer free-running (mode is `TTMR_FREE`), `TTCR` is reset to its current value. This idiom ensures that another timeout is scheduled after `0x10000000` instructions.

```
        case TTMR_ONCE:
```

```
            // stop the timer on the NEXT instruction count
            or1k->TTCR         = (getTTCR(or1k)+1) & 0xfffffff;
            or1k->timerRunning = False;
            break;
```

With the timer in mode `TTMR_ONCE`, `TTCR` should be set to the value that it should hold from the next instruction onwards. Because the callback is invoked before execution of the faulting instruction, we need to increment the current value of `TTCR`.

Whether the timer expiry should cause a processor state change is determined by calling `takeTEE`, which is defined earlier in `or1kExceptions.c` as follows:

```
inline static Uns32 isTEEPending(or1kP or1k) {
    return (or1k->TTMR_IP && or1k->TTMR_IE);
}

inline static Bool isTEEEnabled(or1kP or1k) {
    return (or1k->SR & SPR_SR_TEE);
}

inline static Uns32 takeTEE(or1kP or1k) {
    return isTEEPending(or1k) && isTEEEnabled(or1k);
}
```

In other words, state change is required if `TTMR[IP]` and `SR[TEE]` are both set (the timer interrupt is both pending and enabled.

The instruction count timeout callback function `or1kICountPendingCB` **must not itself try to handle the interrupt** (for example, by calling `or1kTakeException`, which we first saw in chapter 12). Instead, *it must call `vmirtDoSynchronousInterrupt` to indicate that a timer exception is pending*. The timer interrupt must be handled by the *instruction fetch exception handler* function, specified by the `VMI_IFETCH_FN` macro in `vmiAttrs.h`:

```
#define VMI_IFETCH_FN(_NAME) vmiFetchAction _NAME( \
    vmiProcessorP processor,     \
    memDomainP    domain,        \
    Addr          address,       \
    Bool          complete,      \
    Bool          annulled       \
)
```

Argument `domain` specifies the memory domain in which the fetch is being performed. The value of the domain can be used to control mode-specific fetch features (for example, how TLB mappings are performed). Argument `annulled` specifies whether the fetch is being made for an annulled delay slot instruction. Annulled instructions are sometimes treated differently (for example, they sometimes do not cause TLB misses). These two arguments are required to model some advanced features, but are not discussed further here.

Type `vmiFetchAction` is defined in `vmiTypes.h` as follows:

```
typedef enum vmiFetchActionE {
    VMI_FETCH_NONE              = 0,
```

```
    VMI_FETCH_EXCEPTION_COMPLETE = 1,
    VMI_FETCH_EXCEPTION_PENDING  = 2
} vmiFetchAction;
```

The instruction fetch exception handler is called in two phases. In the first phase (indicated by `complete` argument `False`), the function should determine *whether there is a pending exception on the processor that should prevent execution at the passed address and instead cause control to be transferred to an exception handler*. If there is such an exception pending, the function should return `VMI_FETCH_EXCEPTION_PENDING`; otherwise, it should return `VMI_FETCH_NONE`. In this phase, the instruction fetch handler *should not update the processor state*.

If the instruction fetch exception handler returns `VMI_FETCH_EXCEPTION_PENDING`, then it will subsequently be called again in a *second* phase (indicated by `complete` argument `True`). At this point, it should make any changes to the processor state required to handle the pending exception and return `VMI_FETCH_EXCEPTION_COMPLETE` to indicate that exception state has been updated.

Typically, the instruction fetch handler is required to handle a variety of exceptions: tick timer exceptions (as in this example), other external interrupts or synchronous exceptions such as invalid execute permission or alignment. In other words, the instruction count timeout callback is *specific to timer exceptions*, whereas the instruction fetch handler covers *all possible fetch exceptions*.

The initial implementation of the instruction fetch handler is as follows:

```
VMI_IFETCH_FN(or1kIFetchExceptionCB) {

    or1kP ork1 = (or1kP)processor;

    if(takeTEE(or1k)) {

        // tick timer interrupt must be taken
        if(complete) {
            or1kTakeException(or1k, OR1K_EXCPT_TTI, 0);
            return VMI_FETCH_EXCEPTION_COMPLETE;
        } else {
            return VMI_FETCH_EXCEPTION_PENDING;
        }

    } else if(address & 3) {

        // handle misaligned fetch exception
        if(complete) {
            or1k->EEAR = (Uns32)address;
            or1kTakeException(or1k, OR1K_EXCPT_BUS, 0);
            return VMI_FETCH_EXCEPTION_COMPLETE;
        } else {
            return VMI_FETCH_EXCEPTION_PENDING;
        }

    } else if(!vmirtIsExecutable(processor, address)) {

        // handle execute privilege exception
        if(complete) {
```

```
            or1k->EEAR = (Uns32)address;
            or1kTakeException(or1k, OR1K_EXCPT_IPF, 0);
            return VMI_FETCH_EXCEPTION_COMPLETE;
        } else {
            return VMI_FETCH_EXCEPTION_PENDING;

    } else {

        // no fetch exception
        return VMI_FETCH_NONE;
    }
}
```

For a tick timer exception, the fetch exception handler causes control to be transferred immediately to the exception vector at `TTI_ADDRESS` without further execution of the instruction at the current address.

We have also implemented the instruction fetch alignment exception, which transfers control immediately to the exception vector at `BUS_ADDRESS` unless the fetch address is aligned to a 4-byte boundary, and the execute privilege exception, which transfers control immediately to the exception vector at `IPF_ADDRESS` if the fetch address does not have execute privileges[8].

There is often also a requirement to transfer control to an exception handler vector *after the completion of the current instruction*. For example, the tick timer interrupt in the OR1K is enabled by a mask bit in the status register, `SR[TEE]`. What happens if `TTMR[IP]` is set and `SR[TEE]` is changed from 0 to 1 by execution of an `l.mtspr` instruction? In this case, the `l.mtspr` instruction should complete and the tick timer exception should occur before the *next* instruction is executed. To allow this behavior, there is one other useful public function defined in `or1kExceptions.c`:

```
void or1kInterruptNext(or1kP or1k) {
    if(takeTEE(or1k)) {
        vmirtDoSynchronousInterrupt((vmiProcessorP)or1k);
    }
}
```

`vmirtDoSynchronousInterrupt` causes the fetch exception handler to be invoked just before the next processor instruction is executed.

15.7 Status Register Update - `or1kUtils.c`

Function `or1kSetSR` has been modified as follows to handle the case described in the previous section where `TTMR[IP]` is set and `SR[TEE]` is changed from 0 to 1:

```
void or1kSetSR(or1kP or1k, Uns32 value) {

    // it is never possible to clear the fixed-one (FO) bit
    value |= SPR_SR_FO;

    // set the SR
```

[8] See section 18 for an example that exercises the execute privilege exception handler

```
    setSRSwitchMode(or1k, value);

    // set the current branch flag, carry flag and overflow flag from the SR
    or1k->branchFlag   = GET_BIT(value, SPR_SR_F);
    or1k->carryFlag    = GET_BIT(value, SPR_SR_CY);
    or1k->overflowFlag = GET_BIT(value, SPR_SR_OV);

    // ensure any pending interrupt is taken before the next instruction
    if(value & SPR_SR_TEE) {
        or1kInterruptNext(or1k);
    }
}
```

Function `or1kDumpRegisters` has also been updated to write the `TTCR` and `TTMR` register values.

15.8 Fetch/Timer Callback Registration - `or1kAttrs.c`

The `vmiIASAttr` structure for the processor model has been modified to add both the instruction count timeout callback and the instruction fetch handler, as follows:

```
const vmiIASAttr modelAttrs = {

    . . . etc . . .

    ////////////////////////////////////////////////////////////////////////
    // EXCEPTION ROUTINES
    ////////////////////////////////////////////////////////////////////////

    .rdPrivExceptCB  = or1kRdPrivExceptionCB,
    .wrPrivExceptCB  = or1kWrPrivExceptionCB,
    .rdAlignExceptCB = or1kRdAlignExceptionCB,
    .wrAlignExceptCB = or1kWrAlignExceptionCB,
    .ifetchExceptCB  = or1kIFetchExceptionCB,
    .arithExceptCB   = or1kArithExceptionCB,
    .icountExceptCB  = or1kICountPendingCB,

    . . . etc . . .
};
```

15.9 Testing Tick Timer Exceptions

Run the platform using the assembler executable file:

```
platform/harness.$IMPERAS_ARCH.exe –-trace --traceregs –-traceshowicount \
    --program application/asmtest.OR1K.elf
```

The new `–traceshowicount` argument enables printing of an *instruction count* at the start of reach trace line. The output from this should be as follows (much irrelevant output has been cut for conciseness):

```
Warning (PC_NRI) No register information callback given for processor ‘cpu1’
Info 1: 'cpu1', 0x0000000000010000: l.ori    r30,r0,0x0
Info 'cpu1' REGISTERS
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000000   R2 : deadbeef   R3 : deadbeef
 R4 : deadbeef   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : deadbeef   R10: deadbeef   R11: deadbeef
```

```
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: 00000000   R31: deadbeef
 PC : 00010004   SR : 00008001   ESR: deadbeef   EPC: deadbeef
 TCR: 00000000   TMR: 00000000   BF:0 CF:0 OF:0
--------------- --------------- --------------- ---------------

. . .

Info 8: 'cpu1', 0x0000000000000c00: l.mtspr  r0,r1,20480
 TCR: 00000000   TMR: a0000008   BF:0 CF:0 OF:0
Info 9: 'cpu1', 0x0000000000000c04: l.rfe
 TCR: 00000000   TMR: a0000008   BF:0 CF:0 OF:0
Info 10: 'cpu1', 0x000000000001001c: l.mtspr  r0,r0,20736
 TCR: 00000000   TMR: a0000008   BF:0 CF:0 OF:0
Info 11: 'cpu1', 0x0000000000010020: l.ori    r1,r0,0x8
 TCR: 00000001   TMR: a0000008   BF:0 CF:0 OF:0
Info 12: 'cpu1', 0x0000000000010024: l.addi   r1,r1,0xffffffff
 TCR: 00000002   TMR: a0000008   BF:0 CF:1 OF:0
Info 13: 'cpu1', 0x0000000000010028: l.sfeqi  r1,0x0
 TCR: 00000003   TMR: a0000008   BF:0 CF:1 OF:0
Info 14: 'cpu1', 0x000000000001002c: l.bnf    0x00010024
 TCR: 00000004   TMR: a0000008   BF:0 CF:1 OF:0
Info 15: 'cpu1', 0x0000000000010030: l.nop    0x0
 TCR: 00000005   TMR: a0000008   BF:0 CF:1 OF:0
Info 16: 'cpu1', 0x0000000000010024: l.addi   r1,r1,0xffffffff
 TCR: 00000006   TMR: a0000008   BF:0 CF:1 OF:0
Info 17: 'cpu1', 0x0000000000010028: l.sfeqi  r1,0x0
 TCR: 00000007   TMR: a0000008   BF:0 CF:1 OF:0
Info 18: 'cpu1', 0x000000000001002c: *** FETCH EXCEPTION ***
 TCR: 00000008   TMR: b0000008   BF:0 CF:1 OF:0
Info 19: 'cpu1', 0x0000000000000500: l.addi   r30,r30,0x1
Info 'cpu1' REGISTERS
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000006   R2 : deadbeef   R3 : deadbeef
 R4 : deadbeef   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : deadbeef   R10: deadbeef   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: 00000001   R31: 00000000
 PC : 00000504   SR : 00008001   ESR: 00008407   EPC: 0001002c
 TCR: 00000008   TMR: b0000008   BF:0 CF:0 OF:0
--------------- --------------- --------------- ---------------

. . . etc . . .
```

The source code for this example is as follows:

```
.org 0x500
        ///////////////////////////////////////////////////////////////////////
        // TICK TIMER EXCEPTION HANDLER (AT 0x500)
        ///////////////////////////////////////////////////////////////////////
        l.addi      r30,r30,1       // increment count of timer exceptions
        l.sw        -4(r31),r1      // save value in r1
        l.sw        -8(r31),r2      // save value in r2
        l.mfspr     r1,r0,0x5000    // get ttmr in r1
        l.movhi     r2,0xefff       // r2 hi = ~TTMR_IP mask
        l.ori       r2,r2,0xffff    // r1 lo = ~TTMR_IP mask
```

```
        l.and       r1,r1,r2        // clear TTMR_IP
        l.mtspr     r0,r1,0x5000    // set ttmr from r1
        l.lwz       r1,-4(r31)      // restore original r1
        l.lwz       r2,-8(r31)      // restore original r2
        l.sfeqi     r30,3           // r30==3?
        l.bf        noReset         // go if so
        l.nop                       // (delay slot)
        l.mtspr     r0,r0,0x5100    // clear ttcr (restarts counter)
noReset:l.rfe                       // return from exception

.org 0xc00
        ////////////////////////////////////////////////////////////////////
        // SYSCALL VECTOR (AT 0xc00)
        ////////////////////////////////////////////////////////////////////
        l.mtspr     r0,r1,0x5000    // set ttmr from r1
        l.rfe                       // return from exception

.org 0x10000
        ////////////////////////////////////////////////////////////////////
        // APPLICATION CODE (AT 0x10000)
        ////////////////////////////////////////////////////////////////////
.global _start
_start:
        l.ori       r30,r0,0        // r30 = 0 (counts timer exceptions)
        l.ori       r31,r0,0        // r31 = 0 (stack pointer)

        l.ori       r1,r0,7         // r1 = 7 (SR_IEE | SR_TEE | SR_SM)
        l.mtspr     r0,r1,0x11      // set sr from r1 (enables interrupts)
        l.movhi     r1,0xa000       // M=ONCE,IE=True
        l.ori       r1,r1,8         // TP=8 (count to match)
        l.sys       0
        l.mtspr     r0,r0,0x5100    // clear ttcr (starts counter)

        ////////////////////////////////////////////////////////////////////
        // TEST INTERRUPT WITH MODE ONCE
        ////////////////////////////////////////////////////////////////////
        l.ori       r1,r0,8         // r1 = 8 (loop count)
loop1:
        l.addi      r1,r1,-1        // decrement r1
        l.sfeqi     r1,0            // r1==0?
        l.bnf       loop1           // go if not
        l.nop                       // (delay slot)

.global exit
exit:
        l.nop
```

Execution starts at label `_start`. At instruction 8, the `TTMR` register is set with `TTMR[M]=ONCE`, `TTMR[IE]=1` and `TTMR[TP]=8`. The counter does not start at this point because in mode `ONCE` it is activated only on a write to `TTCR`. At instruction 10, TTCR is written with 0, which starts the timer counting. At instruction 18, the timer expires generating the `FETCH EXCEPTION` message; instead of executing the instruction at address `0x1002c`, control is transferred to the exception handler.

The exception handler counts the number of exceptions. If fewer than 3 have occurred, it resets `TTCR`, which restarts the counter. In the full test case log, there are therefore three tick timer exceptions in total.

15.10 Explicit Processor Timers

This example uses the *implicit processor timer*, managed by functions `vmirtSetICountInterrupt` and `vmirtClearICountInterrupt`. It is also possible to create any number of additional *explicit* processor timers using functions described below. Each timer runs independently of the others.

Function `vmirtCreateModelTimer` creates a new timer for a processor. It is defined in `vmiRt.h` as follows:

```
vmiModelTimerP vmirtCreateModelTimer(
    vmiProcessorP processor,
    vmiICountFn   icountCB,
    Uns32         scale,
    void         *userData
);
```

The argument `processor` is the processor to which the timer is to be attached. Argument `icountCB` is the timer expiry callback function. Argument `scale` is a scale factor by which the timer runs slower than the processor with which it is associated: for example, a scale value of 3 implies that the timer will appear to increment every three processor instructions. Argument `userData` is passed as the `userData` argument of the expiry function when it is called. The function returns an opaque type `vmiModelTimerP` which can be used to update the timer later; typically, this value will be saved in a field in the processor structure.

A previously-created timer can be cleared and deleted by `vmirtDeleteModelTimer`:

```
void vmirtDeleteModelTimer(vmiModelTimerP modelTimer);
```

The delay after which a timer will expire can be modified using `vmirtSetModelTimer`:

```
void vmirtSetModelTimer(vmiModelTimerP modelTimer, Uns64 iDelta);
```

This function sets the passed timer in exactly the same way that `vmirtSetICountInterrupt` sets the implicit timer. An explicit timer can be cleared using `vmirtClearModelTimer`:

```
void vmirtClearModelTimer(vmiModelTimerP modelTimer);
```

This function clears the passed timer in exactly the same way that `vmirtClearICountInterrupt` clears the implicit timer.

There are also three functions enabling the timer state to be queried. Function `vmirtIsModelTimerEnabled` returns a Boolean indicating if the timer is enabled (i.e. whether it has been activated using `vmirtSetModelTimer`):

```
Bool vmirtIsModelTimerEnabled(vmiModelTimerP modelTimer);
```

Function `vmirtGetModelTimerCurrentCount` returns the current timer value, either in terms of instructions or ticks (instructions scaled by the `scale` value when the timer was created):

```
Uns64 vmirtGetModelTimerCurrentCount(vmiModelTimerP modelTimer);
```

Function `vmirtGetModelTimerExpiryCount` returns the timer value at which the timer is scheduled to expire, either in terms of instructions or ticks (instructions scaled by the `scale` value when the timer was created):

```
Uns64 vmirtGetModelTimerExpiryCount(vmiModelTimerP modelTimer);
```

A basic model timer created using `vmirtCreateModelTimer` counts *the exact number of instructions executed by a processor*. In a quantized multiprocessor simulation, this might give the impression that time is running backwards, in the following scenario:

- CPU A reads a timer value *towards the end of a quantum* and uses it to calculate the current time.
- CPU A reaches the quantum end. CPU B starts simulating the same quantum.
- CPU B reads a timer value *towards the beginning of a quantum* and uses it to calculate the current time.

In this case, the time calculated by CPU A will appear to be later that that calculated by CPU B, even though it was obtained earlier. This is simply an artifact of multiprocessor quantized simulation, but this can cause problems for applications that rely upon a monotonically increasing view of time to work correctly. If the overall view of simulation time must increase monotonically, use function `vmirtCreateMonotonicModelTimer` to create the timer instead. It is defined in `vmiRt.h` as follows:

```
vmiModelTimerP vmirtCreateMonotonicModelTimer(
    vmiProcessorP processor,
    vmiICountFn   icountCB,
    Uns32         scale,
    void         *userData
);
```

Monotonic timers are exactly like normal timers, except that the implied time seen when reading the timer is guaranteed to increase monotonically for all monotonic timers in the platform. See the *VMI Run Time Function Reference* manual for more information about the algorithm used.

16 Modeling External Interrupts

The previous chapter showed how to model tick timer exceptions. The OR1K also supports generic external interrupts and a reset signal which we will model now.

16.1 OR1K PIC Overview

The OR1K programmable interrupt controller (PIC) is controlled by two processor registers, the *PIC mask register* (`PICMR`) and the *PIC status register* (`PICSR`). These two SPR registers may be read and written using the `l.mfspr` and `l.mtspr` instructions we have seen previously (`PICMR` has SPR index `0x4800`, `PICSR` has SPR index `0x4802`).

The `PICMR` register is used to mask or unmask up to 32 programmable interrupt sources.

The `PICSR` register is used by external interrupt sources to signal up to 32 interrupts. The OR1K is defined to allow either level-triggered or edge-triggered interrupts, or a mixture of both. In this implementation, we will support only level-triggered interrupts (so the external device will be responsible for all changes to bits in the `PICSR` register).

16.2 The Template External Interrupt Model

A template model for the OR1K processor implementing external interrupts can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/14.or1kBehaviorExternalInterrupt
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/14.or1kBehaviorExternalInterrupt .
```

Compile the model, harness and application using the `make` command:

```
cd 14.or1kBehaviorExternalInterrupt
make OPT=1
```

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is based on the previous model, with the changes listed in following sections.

16.3 Defining PIC Registers - `or1kStructure.h`

The OR1K processor structure has been updated to add fields for the `PICMR` and `PICSR` registers, and also macros to access the registers in code morphing routines. There is also a new member `netPorts` of type `vmiNetPortP` which holds an array of net port descriptors for the processor. To allow modeling of edge-triggered reset, there are two new Boolean fields: `resetInput`, which holds the state of the result input net, and `reset`, which indicates whether the processor should perform a reset on the next instruction.

16.4 Adding PIC Register Read and Write - `or1kMorph.c`

This file has been modified to enhance `l.mtspr` and `l.mfspr` to allow reading and writing of `PICSR` and `PICMR` registers. `PICSR` can be read but not written (only external devices can modify this register). `PICMR` is written by calling `or1kSetPICMR`, implemented in `or1kExceptions.c`.

16.5 Adding PIC Exceptions - `or1kExceptions.c`

This file implements most of the new functionality to implement external interrupts and reset. The changes are as below.

The existing routines `or1kIFetchExceptionCB` and `or1kInterruptNext` have been modified to react to external interrupts and reset in addition to timer interrupts. `or1kIFetchExceptionCB` implements external interrupts as higher priority than timer interrupts, and a processor reset as highest priority of all:

```
VMI_IFETCH_FN(or1kIFetchExceptionCB) {

    or1kP ork1 = (or1kP)processor;

    if(or1k->reset) {

        // reset pending - update registers to complete exception if required
        if(complete) {
            or1k->reset = False;
            or1kTakeException(or1k, OR1K_EXCPT_RST, 0);
            return VMI_FETCH_EXCEPTION_COMPLETE;
        } else {
            return VMI_FETCH_EXCEPTION_PENDING;
        }

    } else if(takeIEE(or1k)) {

        // external interrupt must be taken
        if(complete) {
            or1kTakeException(or1k, OR1K_EXCPT_EXI, 0);
            return VMI_FETCH_EXCEPTION_COMPLETE;
        } else {
            return VMI_FETCH_EXCEPTION_PENDING;
        }

    } else if(takeTEE(or1k)) {

        // tick timer interrupt must be taken
        if(complete) {
            or1kTakeException(or1k, OR1K_EXCPT_TTI, 0);
            return VMI_FETCH_EXCEPTION_COMPLETE;
        } else {
            return VMI_FETCH_EXCEPTION_PENDING;
        }

    } else if(address & 3) {
```

```
        // handle misaligned fetch exception
        if(complete) {
            or1k->EEAR = (Uns32)address;
            or1kTakeException(or1k, OR1K_EXCPT_BUS, 0);
            return VMI_FETCH_EXCEPTION_COMPLETE;
        } else {
            return VMI_FETCH_EXCEPTION_PENDING;
        }

    } else if(!vmirtIsExecutable(processor, address)) {

        // handle execute privilege exception
        if(complete) {
            or1k->EEAR = (Uns32)address;
            or1kTakeException(or1k, OR1K_EXCPT_IPF, 0);
            return VMI_FETCH_EXCEPTION_COMPLETE;
        } else {
            return VMI_FETCH_EXCEPTION_PENDING;
        }

    } else {

        return VMI_FETCH_NONE;
    }
}

void or1kInterruptNext(or1kP or1k) {

    if(ork1->reset || takeIEE(or1k) || takeTEE(or1k)) {
        vmirtDoSynchronousInterrupt((vmiProcessorP)or1k);
    }
}
```

Function `takeIEE` is implemented as:

```
inline static Uns32 isIEEPending(or1kP or1k) {
    return (or1k->PICMR & or1k->PICSR);
}

inline static Bool isIEEEnabled(or1kP or1k) {
    return (or1k->SR & SPR_SR_IEE);
}

inline static Uns32 takeIEE(or1kP or1k) {
    return isIEEPending(or1k) && isIEEEnabled(or1k);
}
```

Function `or1kSetPICMR` is called whenever register `PICMR` is written by instruction `l.mtspr`. Writing the programmable interrupt controller mask register could enable an interrupt that was previously disabled – function `or1kInterruptNext` is used to schedule an interrupt before the next instruction in this case:

```
void or1kSetPICMR(or1kP or1k, Uns32 PICMR) {

    or1k->PICMR = PICMR;

    // take any pending interrupt before the next instruction
    or1kInterruptNext(or1k);
}
```

In order to allow interrupts to be raised externally to the model, it is necessary to register *net change functions* that are activated on external events. Each net change function should be defined using the `VMI_NET_CHANGE_FN` macro from `vmiTypes.h`:

```
#define VMI_NET_CHANGE_FN(_NAME) void _NAME( \
    vmiProcessorP processor,    \
    void         *userData,     \
    Uns64         newValue      \
)
```

The net change function is passed three arguments:

1. The processor that is being interrupted;
2. A processor-specific data pointer;
3. A new value for the net, the meaning of which is processor-specific[9].

For the OR1K model, a single net change function is currently used for external interrupts, defined as follows:

```
VMI_NET_CHANGE_FN(or1kExternalInterrupt) {

    or1kP or1k     = (or1kP)processor;
    Uns32 deviceId = (UnsPS)userData;

    if(newValue) {
        or1k->PICSR |= deviceId;
    } else {
        or1k->PICSR &= ~deviceId;
    }

    // take any pending interrupt before the next instruction
    or1kInterruptNext(or1k);
}
```

For this model, the processor specific `userData` pointer is used to hold a bit mask representing the interrupting device, and `newValue` is a boolean indicating whether that mask has been enabled or disabled. The new mask value is calculated and applied to the processor `PICSR` register. Finally, function `or1kInterruptNext` is called to interrupt the processor on the next instruction if required. Note that these external interrupts are *level-sensitive*; There is also a new net change function for the *edge-sensitive* reset signal, defined as follows:

```
VMI_NET_CHANGE_FN(or1kExternalReset) {

    or1kP or1k     = (or1kP)processor;
    Bool  oldReset = or1k->resetInput;

    // save new value of reset signal
    or1k->resetInput = (newValue!=0);

    if(!oldReset && or1k->resetInput) {
```

[9] Note that from VMI version 7.29.0, the type of a net value has changed from `Uns32` to `Uns64`, effectively widening the maximum width of a net.

```
        // reset signal raised: halt processor
        vmirtHalt(processor);

    } else if(oldReset && !or1k->resetInput) {

        // reset signal lowered: restart processor
        vmirtRestartNext(processor);

        // indicate that processor reset is required
        or1k->reset = True;

        // take any pending interrupt before the next instruction
        or1kInterruptNext(or1k);
    }
}
```

In this function, the `newValue` parameter is 1 if reset is being asserted and 0 if it is being deasserted. The reset procedure is as follows:

1. When the reset is *asserted*, the processor halts, and remains halted while the reset remains applied. Halting is implemented by calling function `vmirtHalt`.
2. Then, when the reset is *deasserted*, the processor is restarted by calling `vmirtRestartNext`, and the Boolean `reset` on the processor structure is set to `True`. Finally, `or1kInterruptNext` is called, which will cause function `or1kIFetchExceptionCB` to be executed at the start of the next instruction. In the fetch exception handler, the `reset` field is used to trigger a reset exception if required and then cleared to `False`.

Prototypes for functions `or1kExternalInterrupt` and `or1kExternalReset` have been added to file `or1kExceptions.h`, for use in file `or1kMain.c`.

16.6 Adding Net Ports - `or1kMain.c`

To notify the simulator of the existence of net ports, the model must provide an iterator function which returns the first or subsequent net port specifications, or 0 at the end of the list (in a similar manner to the existing bus ports). The function must be registered in the model attributes table using the `busPortSpecsCB` member:

```
typedef struct vmiIASAttrS {

    ... members omitted ...

    vmiBusPortSpecsFn   busPortSpecsCB;     // callback for next bus port

    ... members omitted ...

} vmiIASAttr;
```

It should be defined using this macro from `vmiPorts.h`:

```
#define VMI_NET_PORT_SPECS_FN(_NAME) vmiNetPortP _NAME ( \
    vmiProcessorP processor,    \
    vmiNetPortP   prev          \
)
```

Note that the iterator is also supplied with the processor pointer, so can adjust its behavior according to the configuration of the current module instance.

Each specification includes:

- Net port name.
- Net port type (*input*, *output* or *inout*).
- A callback function and user-data field (for an input).
- The address offset of a handle (used for output).
- Optional description.

There are also fields used by the simulator in order to make connections to the net port. The net port specification structure is defined in `vmiPorts.h` as follows:

```
typedef struct vmiNetPortS {

    const char      *name;
    vmiNetPortType   type;
    void            *userData;

    // callback for input or I/O net change
    vmiNetChangeFn   netChangeCB;
    Uns32           *handle;

    // space for documentation
    const char      *description;
    void            *descriptionDom;

    // domain is non-NULL if port is connected
    memDomainP       domain;

} vmiNetPort;
```

A template of the implemented net ports is defined in `or1kMain.c` like this:

```
const static vmiNetPort netPorts[] = {
    {"reset", vmi_NP_INPUT, (void*)0, or1kExternalReset    },
    {"intr0", vmi_NP_INPUT, (void*)1, or1kExternalInterrupt},
    {"intr1", vmi_NP_INPUT, (void*)2, or1kExternalInterrupt},
    {"intr2", vmi_NP_INPUT, (void*)4, or1kExternalInterrupt},
    {"intr3", vmi_NP_INPUT, (void*)8, or1kExternalInterrupt}
};
```

New functions have been added to allocate and free the net port lists, in a similar fashion to the existing bus ports:

```
static void newNetPorts(or1kP or1k) {

    Uns32 i;

    or1k->netPorts = STYPE_CALLOC_N(vmiNetPort, NUM_MEMBERS(netPorts));

    for(i=0; i<NUM_MEMBERS(netPorts); i++) {
        or1k->netPorts[i] = netPorts[i];
    }
}
```

```
static void freeNetPorts(or1kP or1k) {

    if(or1k->netPorts) {
        STYPE_FREE(or1k->netPorts);
        or1k->netPorts = 0;
    }
}
```

These new functions are called in the processor constructor and destructor, respectively. Finally, there is the net port iterator definition, which returns each allocated net port descriptor in turn:

```
VMI_NET_PORT_SPECS_FN(or1kGetNetPortSpec) {

    or1kP or1k = (or1kP)processor;

    if(!prev) {

        // first port
        return or1k->netPorts;

    } else {

        // port other than the first
        Uns32 prevIndex = (prev-or1k->netPorts);
        Uns32 thisIndex = prevIndex+1;

        return (thisIndex<NUM_MEMBERS(netPorts))?&or1k->netPorts[thisIndex]:0;
    }
}
```

16.7 Enhancing Register Dump - `or1kUtils.c`

Function `or1kDumpRegisters` now also writes the new `PICMR` and `PICSR` registers.

16.8 PIC Test Harness - `platform/harness.c`

To stimulate the external interrupt signals, `platform/harness.c`, has been changed as follows. Net objects have been created in the platform and connected to the `intr0`, `intr1` and `reset` input ports:

```
    optNetP intr0Net = opNetNew(mr, "intr0Net", 0, 0);
    optNetP intr1Net = opNetNew(mr, "intr1Net", 0, 0);
    optNetP resetNet = opNetNew(mr, "resetNet", 0, 0);
    opObjectNetConnect(processor, intr0Net, "intr0");
    opObjectNetConnect(processor, intr1Net, "intr1");
    opObjectNetConnect(processor, resetNet, "reset");
```

The simulation is controlled by this sequence:

```
    // run processor for 9 instructions
    simulate(processor, 9);

    // raise reset for five instructions
    opNetWrite(resetNet, 1);
    simulate(processor, 5);
    opNetWrite(resetNet, 0);
```

```
    // run processor for 9 instructions
    simulate(processor, 9);

    // raise intr0 for one instruction
    opNetWrite(intr0Net, 1);
    simulate(processor, 1);
    opNetWrite(intr0Net, 0);

    // run processor for 9 instructions
    simulate(processor, 9);

    // raise intr1 for one instruction
    opNetWrite(intr1Net, 1);
    simulate(processor, 1);
    opNetWrite(intr1Net, 0);

    // run processor until done (no instruction limit)
    while(simulate(processor, -1)) {
        // keep going while processor is still running
    }
```

The processor is first run for nine instructions. Then, `reset` is raised (by `opNetWrite`) and the processor run for five more instructions before `reset` is lowered. We then run for nine more instructions before raising `intr0` for a single instruction. After running for nine more instructions, `intr1` is raised for a single instruction. After that, the simulation is run to completion.

16.9 Testing External Exceptions

Run the platform using the assembler executable file:

```
platform/harness.$IMPERAS_ARCH.exe --trace --traceshowicount \
    --program application/asmtest.OR1K.elf
```

The output from this should be as follows:

```
Warning (PC_NRI) No register information callback given for processor
'platform/cpu1'
Info 1: 'platform/cpu1', 0x0000000000010000: l.ori    r30,r0,0x0
Info 2: 'platform/cpu1', 0x0000000000010004: l.addi   r1,r0,0xffffffff
Info 3: 'platform/cpu1', 0x0000000000010008: l.mtspr  r0,r1,18432
Info 4: 'platform/cpu1', 0x000000000001000c: l.ori    r1,r0,0x6
Info 5: 'platform/cpu1', 0x0000000000010010: l.mtspr  r0,r1,17
Info 6: 'platform/cpu1', 0x0000000000010014: l.ori    r1,r0,0x4
Info 7: 'platform/cpu1', 0x0000000000010018: l.addi   r1,r1,0xffffffff
Info 8: 'platform/cpu1', 0x000000000001001c: l.sfeqi  r1,0x0
Info 9: 'platform/cpu1', 0x0000000000010020: l.bnf    0x00010018
Info 15: 'platform/cpu1', 0x0000000000010024: *** FETCH EXCEPTION ***
Info 16: 'platform/cpu1', 0x0000000000000100: l.j      0x00010000
Info 17: 'platform/cpu1', 0x0000000000000104: l.addi   r30,r30,0x1
Info 18: 'platform/cpu1', 0x0000000000010000: l.ori    r30,r0,0x0
Info 19: 'platform/cpu1', 0x0000000000010004: l.addi   r1,r0,0xffffffff
Info 20: 'platform/cpu1', 0x0000000000010008: l.mtspr  r0,r1,18432
Info 21: 'platform/cpu1', 0x000000000001000c: l.ori    r1,r0,0x6
Info 22: 'platform/cpu1', 0x0000000000010010: l.mtspr  r0,r1,17
Info 23: 'platform/cpu1', 0x0000000000010014: l.ori    r1,r0,0x4
Info 24: 'platform/cpu1', 0x0000000000010018: *** FETCH EXCEPTION ***
Info 25: 'platform/cpu1', 0x0000000000000800: l.addi   r30,r30,0x1
Info 26: 'platform/cpu1', 0x0000000000000804: l.rfe
```

```
Info 27: 'platform/cpu1', 0x0000000000010018: l.addi   r1,r1,0xffffffff
Info 28: 'platform/cpu1', 0x000000000001001c: l.sfeqi  r1,0x0
Info 29: 'platform/cpu1', 0x0000000000010020: l.bnf    0x00010018
Info 30: 'platform/cpu1', 0x0000000000010024: l.nop    0x0
Info 31: 'platform/cpu1', 0x0000000000010018: l.addi   r1,r1,0xffffffff
Info 32: 'platform/cpu1', 0x000000000001001c: l.sfeqi  r1,0x0
Info 33: 'platform/cpu1', 0x0000000000010020: l.bnf    0x00010018
Info 34: 'platform/cpu1', 0x0000000000010024: *** FETCH EXCEPTION ***
Info 35: 'platform/cpu1', 0x0000000000000800: l.addi   r30,r30,0x1
Info 36: 'platform/cpu1', 0x0000000000000804: l.rfe
Info 37: 'platform/cpu1', 0x0000000000010020: l.bnf    0x00010018
Info 38: 'platform/cpu1', 0x0000000000010024: l.nop    0x0
Info 39: 'platform/cpu1', 0x0000000000010018: l.addi   r1,r1,0xffffffff
Info 40: 'platform/cpu1', 0x000000000001001c: l.sfeqi  r1,0x0
Info 41: 'platform/cpu1', 0x0000000000010020: l.bnf    0x00010018
Info 42: 'platform/cpu1', 0x0000000000010024: l.nop    0x0
Info 43: 'platform/cpu1', 0x0000000000010018: l.addi   r1,r1,0xffffffff
Info 44: 'platform/cpu1', 0x000000000001001c: l.sfeqi  r1,0x0
Info 45: 'platform/cpu1', 0x0000000000010020: l.bnf    0x00010018
Info 46: 'platform/cpu1', 0x0000000000010024: l.nop    0x0
Info 47: 'platform/cpu1', 0x0000000000010028: l.nop    0x0
Processor 'platform/cpu1' terminated at 'exit', address 0x10028
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000000   R2 : deadbeef   R3 : deadbeef
 R4 : deadbeef   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : deadbeef   R10: deadbeef   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: 00000002   R31: deadbeef
 PC : 0001002c   SR : 00008606   ESR: 00008406   EPC: 00010020
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: ffffffff
 BF:1 CF:1 OF:0
--------------- --------------- --------------- ---------------
processor has executed 42 instructions
```

The source code for this example is as follows:

```
.org 0x100

        ////////////////////////////////////////////////////////////////////////
        // RESET HANDLER (AT 0x100)
        ////////////////////////////////////////////////////////////////////////
        l.j         _start          // jump to start address
        l.addi      r30,r30,1       // increment count of external exceptions

.org 0x800

        ////////////////////////////////////////////////////////////////////////
        // EXTERNAL INTERRUPT HANDLER (AT 0x800)
        ////////////////////////////////////////////////////////////////////////
        l.addi      r30,r30,1       // increment count of external exceptions
        l.rfe                       // return from exception

.org 0x10000

        ////////////////////////////////////////////////////////////////////////
        // APPLICATION CODE (AT 0x10000)
        ////////////////////////////////////////////////////////////////////////

.global _start
```

```
_start:
        l.ori       r30,r0,0        // r30 = 0 (counts timer exceptions)
        l.addi      r1,r0,-1        // r1 = -1
        l.mtspr     r0,r1,0x4800    // set picmr from r1 (enables interrupts)
        l.ori       r1,r0,6         // r1 = SR_IEE|SR_TEE, user mode
        l.mtspr     r0,r1,0x11      // set sr from r1 (enables interrupts)

        l.ori       r1,r0,4         // r1 = 4 (loop count)
loop1:
        l.addi      r1,r1,-1        // decrement r1
        l.sfeqi     r1,0            // r1==0?
        l.bnf       loop1           // go if not
        l.nop                       // (delay slot)

.global exit
exit:
        l.nop
```

Execution starts at label `_start`. The application enables external interrupts and starts executing a simple loop, `loop1`. When the reset signal is applied after nine instructions, the processor halts for five instructions, before resuming at the restart address (in supervisor mode):

```
Info 9: 'platform/cpu1', 0x0000000000010020: l.bnf    0x00010018
Info 15: 'platform/cpu1', 0x0000000000010024: *** FETCH EXCEPTION ***
Info 16: 'platform/cpu1', 0x0000000000000100: l.j      0x00010000
```

The external interrupt exception handler at `0x800` counts the number of external interrupts in `r30`. The count is cleared to zero on a reset.

Note that external interrupts interrupt the flow of execution at instructions 24 and 34.

17 Implementing the Debug Interface

This section describes the implementation of a *debug interface* for the OR1K processor. This has several purposes:

1. It enables debuggers that support the *gdb remote serial protocol* (RSP) to be connected to the processor model;
2. It enables query functions in the Imperas OP interface (for example `opProcessorRegNext` and `opProcessorRegByName`), which can in turn be used to implement custom debugger integrations or advanced test harnesses that are able to query and modify processor state.
3. It enables use of enhanced trace functionality based on *changed registers*.
4. It is a prerequisite for advanced model features such as save/restore and instruction attribute support.

17.1 The Template Debug Interface Model

A template model for the OR1K processor implementing a debugger interface can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/15.or1kDebugSupport
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/15.or1kDebugSupport .
```

Compile the model, harness and application using the `make` command:

```
cd 15.or1kDebugSupport
make OPT=1
```

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is based on the previous model, with the changes listed in following sections.

17.2 Adding Query Functions - `or1kUtils.c`

File `or1kUtils.c` has been modified to implement processor *mode* and *exception* query callbacks. For each, there are two functions:

1. An iterator that lists each mode or exception type supported by the processor;
2. A function that returns the *currently active* mode or exception.

17.2.1 Processor Mode Iterator Function

The processor *mode iterator* is defined using the `VMI_MODE_INFO_FN` macro, defined in `vmiDbg.h` as follows:

```
#define VMI_MODE_INFO_FN(_NAME) vmiModeInfoCP _NAME( \
    vmiProcessorP processor,    \
    vmiModeInfoCP prev          \
)
typedef VMI_MODE_INFO_FN((*vmiModeInfoFn));
```

When called with a `NULL` value of `prev`, the function should return a description of the first mode supported by the processor. When called with a non-`NULL` value of `prev`, the function should return a description of the *next* mode supported by the processor. When all modes have been returned, the function should return `NULL`. Each mode is described by returning a pointer to an object of type `vmiModeInfo`:

```
typedef struct vmiModeInfoS {
    const char *name;           // exception name
    Uns32       code;           // model-specific mode code
    const char *description;    // description string
} vmiModeInfo;
```

The `name` field gives a test name for the mode. The `code` field is model-specific, and typically will correspond to an enumeration in the model. The `description` field provides extra information about the mode (typically used in documentation). For the OR1K, the mode iterator function is defined like this:

```
VMI_MODE_INFO_FN(or1kModeInfo) {

    vmiModeInfoCP end = modes+OR1K_MODE_LAST;

    // on the first call, start with the first member of the table
    if(!prev) {
        prev = modes-1;
    }

    // get the next member
    vmiModeInfoCP this = prev+1;

    // return the next member, or NULL if at the end of the list
    return (this==end) ? 0 : this;
}
```

A prototype of this function has been added to `or1kFunctions.h`.

This function refers to a constant static list of modes in `or1kUtils.c`:

```
#define OR1K_MODE_INFO(_D) [OR1K_MODE_##_D] = {name:#_D, code:OR1K_MODE_##_D}

static const vmiModeInfo modes[OR1K_MODE_LAST] = {
    OR1K_MODE_INFO(SUPERVISOR),
    OR1K_MODE_INFO(USER)
};
```

The modes array defines two modes, `SUPERVISOR` (with code `OR1K_MODE_SUPERVISOR`) and `USER` (with code `OR1K_MODE_USER`). Entry `OR1K_MODE_LAST` in the enumeration does not define a real mode but is instead used as a terminator for sizing of the `modes` array.

17.2.2 Processor Current Mode Query Function

The processor *current mode query function* is defined using the `VMI_GET_MODE_FN` macro, defined in `vmiDbg.h` as follows:

```
#define VMI_GET_MODE_FN(_NAME) vmiModeInfoCP _NAME(vmiProcessorP processor)
typedef VMI_GET_MODE_FN((*vmiGetModeFn));
```

This function should return a `vmiModeInfoCP` description for the current mode. In the OR1K mode, the function is implemented like this:

```
VMI_GET_MODE_FN(or1kGetMode) {

    or1kP    or1k    = (or1kP)processor;
    Uns32    SM      = or1k->SR & SPR_SR_SM;
    or1kMode newMode = SM ? OR1K_MODE_SUPERVISOR : OR1K_MODE_USER;

    return modes+newMode;
}
```

So depending on the current value of the `SM` bit in the status register, an appropriate entry from the `modes` table is selected.

A prototype of this function has been added to `or1kFunctions.h`.

17.2.3 Processor Exception Iterator Function

The processor *exception iterator* is defined using the `VMI_EXCEPTION_INFO_FN` macro, defined in `vmiDbg.h` as follows:

```
#define VMI_EXCEPTION_INFO_FN(_NAME) vmiExceptionInfoCP _NAME( \
    vmiProcessorP      processor,   \
    vmiExceptionInfoCP prev         \
)
typedef VMI_EXCEPTION_INFO_FN((*vmiExceptionInfoFn));
```

The works in an analogous fashion to the mode iterator, described previously. Each exception is described by returning a pointer to an object of type `vmiExceptionInfo`:

```
typedef struct vmiExceptionInfoS {
    const char *name;           // exception name
    Uns32       code;           // model-specific exception code
    const char *description;    // description string
} vmiExceptionInfo;
```

Once again, the description contains a string description of the exception, a model-specific code and an optional description. For the OR1K, the exception iterator function is defined like this:

```
VMI_EXCEPTION_INFO_FN(or1kExceptionInfo) {

    vmiExceptionInfoCP end = exceptions+OR1K_EXCPT_LAST;

    // on the first call, start with the first member of the table
```

```
    if(!prev) {
        prev = exceptions-1;
    }

    // get the next member
    vmiExceptionInfoCP this = prev+1;

    // return the next member, or NULL if at the end of the list
    return (this==end) ? 0 : this;
}
```

This function works in exactly the same way as the mode iterator, returning members of the `exceptions` array, previously described in section 12.1.5. A prototype of this function has been added to `or1kFunctions.h`.

17.2.4 Processor Current Exception Query Function

The processor *current exception query function* is defined using the `VMI_GET_EXCEPTION_FN` macro, defined in `vmiDbg.h` as follows:

```
#define VMI_GET_EXCEPTION_FN(_NAME) vmiExceptionInfoCP _NAME( \
    vmiProcessorP processor \
)
typedef VMI_GET_EXCEPTION_FN((*vmiGetExceptionFn));
```

This function should return a `vmiExceptionInfoCP` description for the current exception. In the OR1K mode, the function is implemented like this:

```
VMI_GET_EXCEPTION_FN(or1kGetException) {
    or1kP or1k = (or1kP)processor;
    return &exceptions[or1k->exception];
}
```

A prototype of this function has been added to `or1kFunctions.h`.

To implement this function, a new pseudo-register called `exception` has been added to the OR1K structure:

```
#include "or1kExceptionTypes.h"

// processor structure
typedef struct or1kS {

    . . . fields omitted . . .

    Uns32          SR;                // status register
    Uns32          ESR;               // exception status register
    Uns32          EPC;               // exception program counter register
    Uns32          EEAR;              // exception effective address register
    Uns32          PICMR;             // PIC mask register
    Uns32          PICSR;             // PIC status register
    Uns32          TTCR;              // tick timer count register
    Uns32          TTCRSetCount;      // cycle count when TTCR set
    Bool           timerRunning;      // whether the timer is running
    Bool           reset;             // whether the processor is being reset
    Bool           resetInput;        // external value of reset signal
    Bool           artifactAccess;    // whether artifact register update
```

```
    or1kException exception;        // current exception

    . . . fields omitted . . .

} or1k, *or1kP;
```

Finally, function `or1kTakeException` has been modified to update the new pseudo-register when an exception occurs:

```
void or1kTakeException(or1kP or1k, or1kException exception, Uns32 pcOffset) {

    Uns8  simD;
    Uns32 simPC = (Uns32)vmirtGetPCDS((vmiProcessorP)or1k, &simD);

    or1kEnterSupervisorMode(or1k);
    or1k->EPC = simPC + pcOffset;

    // set sr[DSX] for exception in a delay slot
    if(simD) {
        or1k->SR |= SPR_SR_DSX;
    }

    // jump to the vector
    or1k->exception = exception;
    vmirtSetPCException((vmiProcessorP)or1k, exceptions[exception].code);
}
```

17.3 Register Access Functions - `or1kRegisters.c`

In order to implement a register interface, functions need to be added to allow the debugger to *read*, *write* and *query* registers in the processor model. This is done be specifying an array of `vmiRegInfo` structures, one for each register in the processor that should be accessible externally. The definition of this structure is as follows (in file `vmiDbg.h`):

```
typedef struct vmiRegInfoS {
    const char   *name;                  // register identification name
    const char   *description;           // description string
    vmiRegGroupCP group;                 // group for this register
    Uns32         gdbIndex;              // gdb ordinal index number
    vmiRegUsage   usage          :  3; // any special usage for this register
    vmiRegAccess  access         :  2; // allowed access
    Bool          noSaveRestore  :  1; // does not participate in save/restore
    Bool          noTraceChange  :  1; // does not participate in tracing
    Bool          instrAttrIgnore :  1; // ignore for instruction attributes
    Bool          isAlias        :  1; // register an alias of another?
    Bool          extension      :  1; // defined in an extension library?
    Uns32         unused         : 22; // unused fields
    Uns32         bits;                  // size of register in bits
    vmiRegReadFn  readCB;                // read callback function
    vmiRegWriteFn writeCB;               // write callback function
    vmiReg        raw;                   // raw register value (if no callback)
    void         *userData;              // model-specific data pointer
    vmiosContextP interceptContext;      // intercept library context
} vmiRegInfo;
```

The fields in the structure are as follows:

1. `name`: the name to use to refer to the register.

2. `description`: this is an optional constant description string (generally used in documentation).
3. `group`: this is a pointer to a register group description (see below).
4. `gdbIndex`: this should be an index number unique to the registers in the processor model. If you intend to use a *gdb* debugger, the number should match the index expected by gdb: you will need to examine the *gdb* processor-specific source code for your processor to find the value to enter here. Otherwise, choose any indexing strategy that makes sense for the processor model. For example, it is often a good strategy to subdivide the range of indices so that high-order bits are used to specify register class (e.g. GPR or system register).
5. `usage`: any special usage for the register should be given using this field, which is a member of the `vmiRegUsage` enumeration in `vmiDbg.h`. Special usages are the program counter (`vmi_REG_PC`), stack pointer (`vmi_REG_SP`) , frame pointer (`vmi_REG_FP`) and link register (`vmi_REG_LR`).
6. `access`: this defines the register accessibility, a member of the `vmiRegAccess` enumeration in `vmiDbg.h`. Valid values are `vmi_RA_NONE` (no access), `vmi_RA_R` (read-only), `vmi_RA_W` (write-only) and `vmi_RA_RW` (read/write).
7. `noSaveRestore`: this Boolean value should be set to `True` if the register should not be automatically saved and restored. Processor save and restore is discussed later in this manual.
8. `noTraceChange`: this Boolean value should be set to `True` if value changes for this register should not be shown when tracing of changed values is enabled (discussed later in this chapter).
9. `instAttrIgnore`: this Boolean value should be set to `True` if usage of this register should not be reported by the instruction attributes API. The instruction attributes API is discussed later in this manual.
10. `isAlias`: this Boolean value should be set to `True` if this register shares a `gdbIndex` value with another register and the other register should be accessed by preference when registers are iterated by index (for example, when using `opProcessorRegByIndex`).
11. `extension`: this Boolean value should be set to `True` if this register is implemented by a processor *extension library* (see section 26). This is informative only and does not affect model behavior.
12. `bits`: this is the size of the register in bits.
13. `readCB`: this is a read callback function that returns the current value of the register, if non-`NULL`.
14. `writeCB`: this is a write callback function that is used to set the current value of the register, if non-`NULL`.
15. `raw`: for a plain register that requires no special behavior for read or write, set `readCB` and/or `writeCB` to `NULL` and use this `vmiReg` field to specify the register location in the processor structure instead.
16. `userData`: this is a pointer to model-specific data that can be used if required in read and write callback functions.
17. `interceptContext`: this field is used internally by the simulator when iterating intercept library registers (see chapter 26); It should always be `NULL` when registers are defined.

The core data structure in `or1kRegisters.c` is an null-terminated array of these structures with one entry for each OR1K register that is made visible externally:

```
static const vmiRegInfo registers[] = {

    // registers visible in gdb
    OR1K_REG_INFO("R0",     0,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_R (regs[0],
OR1K_GROUP(GPR))   ,      "constant zero"),
    OR1K_REG_INFO("R1",     1,  vmi_REG_SP,   OR1K_BITS, OR1K_RAW_REG_RW(regs[1],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R2",     2,  vmi_REG_FP,   OR1K_BITS, OR1K_RAW_REG_RW(regs[2],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R3",     3,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[3],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R4",     4,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[4],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R5",     5,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[5],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R6",     6,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[6],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R7",     7,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[7],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R8",     8,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[8],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R9",     9,  vmi_REG_LR,   OR1K_BITS, OR1K_RAW_REG_RW(regs[9],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R10",   10,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[10],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R11",   11,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[11],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R12",   12,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[12],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R13",   13,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[13],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R14",   14,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[14],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R15",   15,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[15],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R16",   16,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[16],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R17",   17,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[17],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R18",   18,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[18],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R19",   19,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[19],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R20",   20,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[20],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R21",   21,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[21],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R22",   22,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[22],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R23",   23,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[23],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R24",   24,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[24],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R25",   25,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[25],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R26",   26,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[26],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R27",   27,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[27],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R28",   28,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[28],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R29",   29,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[29],
OR1K_GROUP(GPR))   ,      0),
    OR1K_REG_INFO("R30",   30,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[30],
OR1K_GROUP(GPR))   ,      0),
```

```
    OR1K_REG_INFO("R31",   31,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(regs[31],
OR1K_GROUP(GPR))    ,      0),
    OR1K_REG_INFO("PC",    64,  vmi_REG_PC,   OR1K_BITS, OR1K_CB_REG_RW (PC,
OR1K_GROUP(SYSTEM)),       0),
    OR1K_REG_INFO("SR",    65,  vmi_REG_NONE, OR1K_BITS, OR1K_CB_REG_RW (SR,
OR1K_GROUP(SYSTEM)),       "status register"),
    OR1K_REG_INFO("EPCR",  66,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(EPC,
OR1K_GROUP(SYSTEM)),       "exception PC"),
    OR1K_REG_INFO("EEAR",  67,  vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(EEAR,
OR1K_GROUP(SYSTEM)),       "exception effective address "),

    // registers not visible in gdb
    OR1K_REG_INFO("ESR",   100, vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(ESR,
OR1K_GROUP(SYSTEM)),       "exception status register"),
    OR1K_REG_INFO("PICMR", 101, vmi_REG_NONE, OR1K_BITS, OR1K_CB_REG_W  (PICMR,
OR1K_GROUP(SYSTEM)),       "PIC mask register"),
    OR1K_REG_INFO("PICSR", 102, vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(PICSR,
OR1K_GROUP(SYSTEM)),       "PIC status register"),
    OR1K_REG_INFO("TTCR",  103, vmi_REG_NONE, OR1K_BITS, OR1K_CB_REG_RW (TTCR,
OR1K_GROUP(SYSTEM)),       "tick timer count register"),
    OR1K_REG_INFO("TTMR",  104, vmi_REG_NONE, OR1K_BITS, OR1K_CB_REG_W  (TTMR,
OR1K_GROUP(SYSTEM)),       "tick timer mode register"),
    OR1K_REG_INFO("EXCPT", 200, vmi_REG_NONE, OR1K_BITS, OR1K_RAW_REG_RW(exception,
OR1K_GROUP(INTEGRATION)), "current exception"),

    {0},
};
```

The table contains two kinds of registers: those visible to gdb and those that are not. In the latter category, there are the architectural `ESR`, `PICMR`, `PICSR`, `TTCR` and `TTMR` registers and an artifact register, `EXCPT`, which is the new `exception` pseudo-register that records any active exception. Which group a particular register is in is determined by macro `IS_GDB_HIDDEN_REG`, which is True if the register is invisible to gdb:

```
#define OR1K_GDB_HIDDEN_INDEX 100
#define IS_GDB_HIDDEN_REG(_I) ((_I)>=OR1K_GDB_HIDDEN_INDEX)
```

Each register is assigned to a register group using the `group` field in the `vmiRegInfo` structure. Groups have no simulation purpose, but can be used by debuggers to gather registers into sets for display purposes. Each group is defined as follows:

```
typedef struct vmiRegGroupS {
    const char    *name;             // register group name
    vmiosContextP interceptContext; // intercept library context
} vmiRegGroup;
```

When defining a group, models should assign a value to the `name` field, but ensure the `interceptContext` field is `NULL` (this is used internally by the simulator when iterating registers defined in intercept libraries – see chapter 26).

In the OR1K, there are three groups, `GPR`, `SYSTEM` and `INTEGRATION`, defined in `or1kRegisters.c` like this:

```
typedef enum or1kRegGroupIdE {
    OR1K_RG_GPR,            // GPR group
    OR1K_RG_SYSTEM,         // System register group
    OR1K_RG_INTEGRATION,    // System register group
    OR1K_RG_LAST            // KEEP LAST: for sizing
} or1kRegGroupId;
```

```
static const vmiRegGroup groups[OR1K_RG_LAST+1] = {
    [OR1K_RG_GPR]         = {name: "GPR"                },
    [OR1K_RG_SYSTEM]      = {name: "System"             },
    [OR1K_RG_INTEGRATION] = {name: "Integration_Support"}
};
```

A *register group iterator* is defined which returns all the register groups implemented by the model:

```
VMI_REG_GROUP_FN(or1kRegGroup) {

    if(!prev) {
        return groups;
    } else if((prev+1)->name) {
        return prev+1;
    } else {
        return 0;
    }
}
```

Given an argument of type `vmiRegGroupCP` (i.e. a pointer to a member of the `groups` array) this function should return the *next* register group description in the array, or `NULL` if there are no more register group descriptions. If called with a `NULL` argument, it should return the first register group description in the array.

There is a macro used to select the appropriate group for a register in the table of register descriptions:

```
#define OR1K_GROUP(_G) &groups[OR1K_RG_##_G]
```

The contents of the register description array are made available to the simulator by implementing a *register structure iterator* function using the `VMI_REG_INFO_FN` macro, defined in `vmiDbg.h` as follows:

```
#define VMI_REG_INFO_FN(_NAME) vmiRegInfoCP _NAME( \
    vmiProcessorP  processor,   \
    vmiRegInfoCP   prev,        \
    vmiRegInfoType gdbFrame     \
)
typedef VMI_REG_INFO_FN((*vmiRegInfoFn));
```

Given an argument of type `vmiRegInfoCP` (i.e. a pointer to a member of the `registers` array) this function should return the *next* register description in the array, or `NULL` if there are no more register descriptions. If called with a `NULL` argument, it should return the first register description in the array. The iterator is used both to access registers in *RSP/gdb* contexts and to provide more general register access; the exact access required is indicated by the `gdbFrame` argument, of type `vmiRegInfoType`, defined as follows:

```
typedef enum vmiRegInfoTypeE {
    VMIRIT_NORMAL,      // normal register request (not RSP)
    VMIRIT_GPACKET,     // RSP 'g' packet register request
    VMIRIT_PPACKET      // RSP 'p' packet register request
} vmiRegInfoType;
```

When the gdbFrame argument has value VMIRIT_NORMAL, this indicates that a general register access is being requested. Normally, the register structure iterator should return the full set of implemented registers in this case.

When the gdbFrame argument has value VMIRIT_GPACKET, this indicates that a register access is being requested for a register in an RSP g packet. The registers returned in this case *must comply with the fixed frame format expected by the gdb*.

When the gdbFrame argument has value VMIRIT_PPACKET, this indicates that a register access is being requested for a register in an RSP p packet. The registers returned in this case are normally the same as for the g packet, but may include extra registers as well.

The OR1K register structure iterator has this definition:

```
VMI_REG_INFO_FN(or1kRegInfo) {
    return getNextRegister((or1kP)processor, prev, gdbFrame);
}
```

A prototype of this function has been added to or1kFunctions.h. Function getNextRegister is defined as follows:

```
static vmiRegInfoCP getNextRegister(or1kP or1k, vmiRegInfoCP reg, Bool gdbFrame)
{

    do {
        if(!reg) {
            reg = registers;
        } else if((reg+1)->name) {
            reg = reg+1;
        } else {
            reg = 0;
        }
    } while(reg && !isRegSupported(or1k, reg, gdbFrame));

    return reg;
}
```

The function returns each member of registers in turn, skipping those that are defined to be unsupported by isRegSupported:

```
static Bool isRegSupported(or1kP or1k, vmiRegInfoCP reg, Bool gdbFrame) {

    if(gdbFrame && IS_GDB_HIDDEN_REG(reg->gdbIndex)) {

        // if this is a GDB frame request then registers that should be hidden
        // from GDB should be ignored
        return False;

    } else {

        // other registers are always supported
        return True;
    }
}
```

In this case, `isRegSupported` hides registers that should not be visible when this is a *gdb* request, implying that registers reported to *gdb* are always a subset of the full register list. A more general implementation would be to return entirely distinct sets of registers in normal mode and `p`/`g` packet modes: there is no requirement for *gdb* registers to be a subset of true registers, or even for the index numbers to match. Note that in this case, no distinction is made between `g` and `p` packet requests: as a consequence, only registers visible in the `g` packet will be accessible using a `p` packet request.

17.3.1 Register Read Callback Functions

Register read callback functions are defined using the `VMI_REG_READ_FN` macro, defined in `vmiDbg.h`:

```
#define VMI_REG_READ_FN(_NAME) Bool _NAME( \
    vmiProcessorP processor,    \
    vmiRegInfoCP  reg,          \
    void         *buffer        \
)
```

Given a processor and a `vmiRegInfoCP` structure representing a processor register, the function should fill the passed buffer with the current value of the register in the passed processor. As an example, there is a special register read callback function for the program counter register that uses `vmirtGetPC` as follows:

```
static VMI_REG_READ_FN(readPC) {
    or1kP or1k = (or1kP)processor;
    OR1K_ARTIFACT_ACCESS(or1k, *(Uns32*)buffer = (Uns32)vmirtGetPC(processor));
    return True;
}
```

The macro `OR1K_ARTIFACT_ACCESS` is defined like this:

```
#define OR1K_ARTIFACT_ACCESS(_OR1K, _B) \
    (_OR1K)->artifactAccess = True;     \
    _B;                                 \
    (_OR1K)->artifactAccess = False
```

This definition ensures that the expression passed as the second argument is evaluated while a new processor Boolean, `artifactAccess`, is set to `True`. This is important because sometimes register reads or writes should behave differently when executed externally (through a debugger, for example) than when executed by the processor: see section 17.5 for an example.

Similarly, registers `SR` and `TTCR` are handled specially:

```
static VMI_REG_READ_FN(readSR) {
    or1kP or1k = (or1kP)processor;
    OR1K_ARTIFACT_ACCESS(or1k, *(Uns32*)buffer = or1kGetSR(or1k));
    return True;
}

static VMI_REG_READ_FN(readTTCR) {
```

```
    or1kP or1k = (or1kP)processor;
    OR1K_ARTIFACT_ACCESS(or1k, *(Uns32*)buffer = or1kGetTTCR(or1k));
    return True;
}
```

17.3.2 Register Write Callback Functions

Register write callback functions are defined using the `VMI_REG_WRITE_FN` macro, defined in `vmiDbg.h`:

```
#define VMI_REG_WRITE_FN(_NAME) Bool _NAME( \
    vmiProcessorP processor,    \
    vmiRegInfoCP  reg,          \
    const void   *buffer        \
)
```

Given a processor and a `vmiRegInfoCP` structure representing a processor register, the function should set the current value of the register in the passed processor from the buffer. As an example, there is a special register write callback function for the program counter register that uses `vmirtSetPC` as follows:

```
static VMI_REG_WRITE_FN(writePC) {
    or1kP or1k = (or1kP)processor;
    OR1K_ARTIFACT_ACCESS(or1k, vmirtSetPC(processor, *(Uns32*)buffer));
    return True;
}
```

Once again, the macro `OR1K_ARTIFACT_ACCESS` is used to ensure that new processor Boolean `artifactAccess` is set to `True` in the context of the write.

17.4 Raw and Callback Register Access

For registers whose values are held directly in the processor structure and which have no special behavior on access, there is no need to define read or write callback functions: instead, it is possible to define that these registers are accessible in their *raw* state. In this case, all this is required is to specify a `vmiReg` for the register in the `raw` field of the register description.

File `or1kRegisters.c` contains a helper macro used in the register table to specify a read/write register that can be accessed raw for both read and write:

```
#define OR1K_RAW_REG_RW(_R, _G) \
    access    : vmi_RA_RW,          \
    raw       : OR1K_CPU_REG(_R),    \
    group     : _G
```

This macro is used to define access to registers `R1` to `R31`.

Another macro defines read-only registers that can be accessed raw:

```
#define OR1K_RAW_REG_R(_R, _G) \
    access    : vmi_RA_R,           \
    raw       : OR1K_CPU_REG(_R),    \
    group     : _G
```

This macro is used to define access to registers `R0`, `EPC` and `EEAR`.

Some read/write registers must be accessed using callbacks for both read and write (for example, `PC`). For such registers, macro `OR1K_CB_REG_RW` is used:

```
#define OR1K_CB_REG_RW(_R, _G) \
    access   : vmi_RA_RW,           \
    readCB   : read##_R,            \
    writeCB  : write##_R,           \
    group    : _G
```

Note that this uses the ## preprocessor directive to manufacture read and write callback names, given the register name. For example, the callback function names for register `PC` are therefore `readPC` and `writePC`, described previously.

Finally, there are some registers that can be accessed raw when read but required a callback when written. For such registers, macro `OR1K_CB_REG_W` is used:

```
#define OR1K_CB_REG_W(_R, _G) \
    access   : vmi_RA_RW,           \
    raw      : OR1K_CPU_REG(_R),    \
    writeCB  : write##_R,           \
    group    : _G
```

These register declarations specify both the `raw` field and the `writeCB` field, but not the `readCB` field.

17.5 Handling Artifact Accesses - `or1kExceptions.c`

Previously, we mentioned that a new Boolean, `artifactAccess`, has been added to the processor structure, and that this new field is set to `True` when a register is being read or written using the register interface. The new field is required to ensure correct behavior when the `TTMR` register is written: if this register is written by the processor executing an `l.mtspr` instruction, *the `IP` bit cannot be set to 1*. However, if the register is written by an artifact access through the register interface (for example, by the debugger or during processor state restore) *it must be possible to set the `IP` bit to 1*. These cases are distinguished in `or1kSetTTMR` using the new `artifactAccess` field, as follows:

```
void or1kSetTTMR(or1kP or1k, Uns32 TTMR) {

    Uns32 TTCR = getTTCR(or1k);

    // update TTMR, recording old and new values of TTMR_IP
    Bool oldIP = or1k->TTMR_IP;
    or1k->TTMR = TTMR;
    Bool newIP = or1k->TTMR_IP;

    // TTMR_IP must not be set by l.mtspr!
    if(!oldIP && newIP && !or1k->artifactAccess) {
        or1k->TTMR_IP = 0;
    }
```

```
    // start the timer if mode is TTMR_RESTART or TTMR_FREE
    // (for TTMR_ONCE, timer is restarted by write to TTCR)
    if((or1k->TTMR_M==TTMR_RESTART) || (or1k->TTMR_M==TTMR_FREE)) {
        or1k->timerRunning = True;
    }

    setTTCR(or1k, TTCR);
}
```

17.6 Debug Function Registration - `or1kAttrs.c`

The `modelAttrs` structure in `or1kAttrs.c` has been changed to include references to all the debug interface functions, as follows:

```
const vmiIASAttr modelAttrs = {

    . . . lines omitted . . .

    ////////////////////////////////////////////////////////////////////////
    // DEBUGGER INTEGRATION SUPPORT ROUTINES
    ////////////////////////////////////////////////////////////////////////

    .regGroupCB       = or1kRegGroup,
    .regInfoCB        = or1kRegInfo,
    .exceptionInfoCB = or1kExceptionInfo,
    .modeInfoCB       = or1kModeInfo,
    .getExceptionCB  = or1kGetException,
    .getModeCB        = or1kGetMode,
    .debugCB          = or1kDumpRegisters,

    . . . lines omitted . . .

};
```

17.7 Debug Function Test Harness - platform/harness.c

The test harness for this example, `platform/harness.c`, has been changed to include code to test the register, exception and mode query functions, as follows:

```
    queryRegisters(processor);
    queryExceptions(processor);
    queryModes(processor);
```

Function `queryRegisters` iterates over each register in each register group, printing their names:

```
static void queryRegisters(optProcessorP processor) {

    opPrintf("%s REGISTERS\n", opObjectHierName(processor));

    optRegGroupP group = 0;

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

        opPrintf("  GROUP %s\n", opRegGroupName(group));

        optRegP reg = 0;

        while((reg=opRegGroupRegNext(processor, group, reg))) {
```

```
            opPrintf("    REGISTER %s\n", opRegName(reg));
        }
    }

    opPrintf("\n");
}
```

Function `queryExceptions` uses the exception iterator and current exception query functions:

```
static void queryExceptions(optProcessorP processor) {

    const char *name = opObjectHierName(processor);

    if(!opProcessorExceptionNext(processor, 0)) {

        opPrintf("%s HAS NO EXCEPTION INFORMATION\n", name);

    } else {

        opPrintf("%s EXCEPTIONS\n", name);

        optExceptionP info = 0;

        while((info=opProcessorExceptionNext(processor, info))) {
            opPrintf(
                "  %s (code %u)\n",
                opExceptionName(info),
                opExceptionCode(info)
            );
        }

        if((info=opProcessorExceptionCurrent(processor))) {
            opPrintf(
                "current: %s (code %u)\n",
                opExceptionName(info),
                opExceptionCode(info)
            );
        }
    }

    opPrintf("\n");
}
```

Function `queryModes` uses the mode iterator and current mode query functions:

```
static void queryModes(optProcessorP processor) {

    const char *name = opObjectHierName(processor);

    if(!opProcessorModeNext(processor, 0)) {

        opPrintf("%s HAS NO MODE INFORMATION\n", name);

    } else {

        opPrintf("%s MODES\n", name);

        optModeP info = 0;

        while((info=opProcessorModeNext(processor, info))) {
```

```
            opPrintf(
                "  %s (code %u)\n",
                opModeName(info),
                opModeCode(info)
            );
        }

        if((info=opProcessorModeCurrent(processor))) {
            opPrintf(
                "current: %s (code %u)\n",
                opModeName(info),
                opModeCode(info)
            );
        }
    }

    opPrintf("\n");
}
```

17.8 Testing the Debugger Interface

Run the platform using the assembler executable file:

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

The new `--gdbconsole` argument causes the simulator to launch a *gdb* in a console window and attach it to the platform when it starts to execute[10]. The output will start with:

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

Then, the query functions defined in the harness will be called, producing the following output:

```
platform/cpu1 REGISTERS
  GROUP GPR
    REGISTER R0
    REGISTER R1
    REGISTER R2
    REGISTER R3
    REGISTER R4
    REGISTER R5
    REGISTER R6
    REGISTER R7
    REGISTER R8
    REGISTER R9
    REGISTER R10
    REGISTER R11
    REGISTER R12
    REGISTER R13
    REGISTER R14
    REGISTER R15
    REGISTER R16
```

[10] The details of which *gdb* to start and how to start it are given by the `vmiProcessorInfo` structure returned by the `vmiProcessorInfoFn` callback for the processor: see section 4.2.6 for more details.

```
    REGISTER R17
    REGISTER R18
    REGISTER R19
    REGISTER R20
    REGISTER R21
    REGISTER R22
    REGISTER R23
    REGISTER R24
    REGISTER R25
    REGISTER R26
    REGISTER R27
    REGISTER R28
    REGISTER R29
    REGISTER R30
    REGISTER R31
  GROUP System
    REGISTER PC
    REGISTER SR
    REGISTER EPCR
    REGISTER EEAR
    REGISTER ESR
    REGISTER PICMR
    REGISTER PICSR
    REGISTER TTCR
    REGISTER TTMR
  GROUP Integration_Support
    REGISTER EXCPT

platform/cpu1 EXCEPTIONS
  RST (code 256)
  BUS (code 512)
  DPF (code 768)
  IPF (code 1024)
  TTI (code 1280)
  ILL (code 1792)
  EXI (code 2048)
  SYS (code 3072)
current: RST (code 256)

platform/cpu1 MODES
  SUPERVISOR (code 0)
  USER (code 1)
current: SUPERVISOR (code 0)
```

The output first lists all of the *registers* in the processor model (by group), then the *exceptions*, then the *modes*. At this point, the debugger should connect, printing this message:

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

We can carry out normal debugging commands supported by *gdb* in the console window. For example, try this sequence (typed commands are shown in bold):

```
(gdb) b fib
Breakpoint 1 at 0x10
(gdb) cont
Continuing.

Breakpoint 1, 0x00000010 in fib ()
(gdb) disass $pc $pc+12
```

```
Dump of assembler code from 0x10 to 0x1c:
0x10 <fib>:     l.sflesi r1,0x1
0x14 <fib+4>:   l.bf 0xf
0x18 <fib+8>:   l.nop 0x0
End of assembler dump.
(gdb) info registers
            R0       R1       R2       R3       R4       R5       R6       R7
      00000000 0000000f deadbeef deadbeef deadbeef deadbeef deadbeef deadbeef
            R8       R9      R10      R11      R12      R13      R14      R15
      deadbeef 0000000c deadbeef deadbeef deadbeef deadbeef deadbeef deadbeef
           R16      R17      R18      R19      R20      R21      R22      R23
      deadbeef deadbeef deadbeef deadbeef deadbeef deadbeef deadbeef deadbeef
           R24      R25      R26      R27      R28      R29      R30      R31
      deadbeef deadbeef deadbeef deadbeef deadbeef deadbeef deadbeef 00000000
            PC       SR     EPCR
      00000010 00008001 deadbeef
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

When you quit from the debugger, simulation will terminate.

17.9 Testing Register Change Tracing

Previously, to show register values during trace we have used the command line arguments `--trace` and `--traceregisters`. When the debug register interface is implemented, an alternative trace format is possible: *register change tracing*.

To view the effect of register change tracing, run the platform as follows:

```
platform/harness.$IMPERAS_ARCH.exe --trace --tracechange \
    --program application/asmtest.OR1K.elf
```

You should see the following output:

```
platform/cpu1 REGISTERS
    . . . lines omitted . . .
platform/cpu1 EXCEPTIONS
    . . . lines omitted . . .
platform/cpu1 MODES
    . . . lines omitted . . .

Info 'platform/cpu1', 0x0000000000000000(_start): l.addi   r31,r0,0x0
Info   R31 deadbeef -> 00000000
Info 'platform/cpu1', 0x0000000000000004(_start+4): l.jal    0x00000010
Info   R9 deadbeef -> 0000000c
Info 'platform/cpu1', 0x0000000000000008(_start+8): l.addi   r1,r0,0xf
Info   R1 00000000 -> 0000000f
Info 'platform/cpu1', 0x0000000000000010(fib): l.sflesi r1,0x1
Info 'platform/cpu1', 0x0000000000000014(fib+4): l.bf     0x00000050
Info 'platform/cpu1', 0x0000000000000018(fib+8): l.nop    0x0
Info 'platform/cpu1', 0x000000000000001c(fib+c): l.addi   r31,r31,0xfffffff4
Info   R31 00000000 -> fffffff4
Info 'platform/cpu1', 0x0000000000000020(fib+10): l.sw     0x0(r31),r9
Info 'platform/cpu1', 0x0000000000000024(fib+14): l.sw     0x4(r31),r1
Info 'platform/cpu1', 0x0000000000000028(fib+18): l.jal    0x00000010
Info   R9 0000000c -> 00000030
Info 'platform/cpu1', 0x000000000000002c(fib+1c): l.addi   r1,r1,0xffffffff
Info   R1 0000000f -> 0000000e
Info   SR 00008001 -> 00008401
```

```
. . . lines omitted . . .

Info 'platform/cpu1', 0x0000000000000044(fib+34): l.add    r1,r1,r2
Info   R1 000000e9 -> 00000262
Info 'platform/cpu1', 0x0000000000000048(fib+38): l.lwz    r9,0x0(r31)
Info   R9 00000040 -> 0000000c
Info 'platform/cpu1', 0x000000000000004c(fib+3c): l.addi   r31,r31,0xc
Info   R31 fffffff4 -> 00000000
Info   SR 00008201 -> 00008601
Info 'platform/cpu1', 0x0000000000000050(done): l.jr     r9
Info 'platform/cpu1', 0x0000000000000054(done+4): l.nop    0x0
Info 'platform/cpu1', 0x000000000000000c(exit): l.nop    0x0
Processor 'platform/cpu1' terminated at 'exit', address 0xc
processor has executed 22687 instructions
```

Note that in this trace format, any register that has *changed value* is shown after the instruction line that caused the change. Trace change works by reading the value of every *readable* register defined by the model[11] after each instruction completes, and comparing with the previous value, printing the old and new values on a change. The format has several advantages over the model-specific register trace enabled by `--traceregisters`:

1. The mode is automatically available once the debug register interface is implemented.
2. Both old and new register values are shown, so it is easy to see what changed.
3. The output is generally much more concise.

[11] Except registers defined with `noTraceChange=True`, which are ignored.

18 Adding an Extended Programmers View

This section provides details of extending a programmers view on the processor model from the standard view provided by the debug interface. The extended programmer's view is supported by the Imperas Professional Tools such as the MP Debugger and interception plugins; it is not supported by OVPsim.

The extended programmers view is implemented using functions from the VMI Run Time API.

18.1 An Example Programmers View

A model for the OR1K processor with additional programmer's view can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/16.or1kProgrammersView
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/16.or1kProgrammersView .
```

Compile the model, harness and application using the `make` command:

```
cd 16.or1kProgrammersView
make
```

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is based on the previous example, with the changes listed in following sections.

18.2 Adding View Object and Event - `or1kStructure.h`

The or1k structure has new fields, `viewObject` and `addrExEvent`, used to hold the created object parent and an address exception event respectively:

```
typedef struct or1kS {

    . . . lines omitted . . .

    vmiViewObjectP exObject;        // view object for address exception info
    vmiViewEventP  addrExEvent;     // event generated on an address exception

} or1k, *or1kP;
```

18.3 Implementing Programmer's View - `or1kView.c`

This is a new file in which the extended programmers view is defined for the processor. In this example we have added a new event and some associated objects.

The event `addrExEvent` is triggered if an address exception occurs.
The view of the address exception has been created as an individual object with other objects associated with it. An object allows events and other objects to be grouped.

```
void or1kCreateView(or1kP or1k) {

    // get the base processor view object
    vmiProcessorP  processor       = (vmiProcessorP)or1k;
    vmiViewObjectP processorObject = vmirtGetProcessorViewObject(processor);

    // add new view object
    or1k->exObject  = vmirtAddViewObject(
        processorObject, "addressException", "Address exception"
    );

    // Create an event to be generated on an address exception
    or1k->addrExEvent = vmirtAddViewEvent(
        or1k->exObject, "address", "Address exception event trigger"
    );

    // Create an object to access the EEAR
    vmiViewObjectP earObject = vmirtAddViewObject(or1k->exObject, "eear", "");
    vmirtSetViewObjectRefValue(eearObject, VMI_VVT_UNS32, &or1k->EEAR);

    // Create an object to access the EPC
    vmiViewObjectP epcObject = vmirtAddViewObject(or1k->exObject, "epc", "");
    vmirtSetViewObjectRefValue(epcObject, VMI_VVT_UNS32, &or1k->EPC);

    // Create an object to access the ESR
    vmiViewObjectP esrObject = vmirtAddViewObject(or1k->exObject, "esr", "");
    vmirtSetViewObjectRefValue(esrObject, VMI_VVT_UNS32, &or1k->ESR);
}
```

18.4 Triggering View Events - `or1kExceptions.c`

The address exception is generated in the exception functions.

```
//
// Read privilege exception handler callback function
//
VMI_RD_PRIV_EXCEPT_FN(or1kRdPrivExceptionCB) {

    if(MEM_AA_IS_TRUE_ACCESS(attrs)) {

        or1kP or1k = (or1kP)processor;
        or1k->EEAR = (Uns32)address;
        or1kTakeException(or1k, DPF_ADDRESS);

        if(or1k->addrExEvent) {
            vmirtTriggerViewEvent(or1k->addrExEvent);
        }
    }
}
```

```
//
// Write privilege exception handler callback function
//
VMI_WR_PRIV_EXCEPT_FN(or1kWrPrivExceptionCB) {

    if(MEM_AA_IS_TRUE_ACCESS(attrs)) {

        or1kP or1k = (or1kP)processor;
        or1k->EEAR = (Uns32)address;
        or1kTakeException(or1k, DPF_ADDRESS);

        if(or1k->addrExEvent) {
            vmirtTriggerViewEvent(or1k->addrExEvent);
        }
    }
}

//
// Read alignment exception handler callback function
//
VMI_RD_ALIGN_EXCEPT_FN(or1kRdAlignExceptionCB) {

    or1kP or1k = (or1kP)processor;
    or1k->EEAR = (Uns32)address;
    or1kTakeException(or1k, BUS_ADDRESS);

    if(or1k->addrExEvent) {
        vmirtTriggerViewEvent(or1k->addrExEvent);
    }

    return 0;
}

//
// Write alignment exception handler callback function
//
VMI_WR_ALIGN_EXCEPT_FN(or1kWrAlignExceptionCB) {

    or1kP or1k = (or1kP)processor;
    or1k->EEAR = (Uns32)address;
    or1kTakeException(or1k, BUS_ADDRESS);

    if(or1k->addrExEvent) {
        vmirtTriggerViewEvent(or1k->addrExEvent);
    }

    return 0;
}
```

18.5 Testing the Extended Programmers View

The extended programmers view is only accessible from the Imperas Professional Tools.

18.5.1 Running in OVP

Run the platform using the assembler executable file:

```
platform/harness.$IMPERAS_ARCH.exe –-trace \
    --program application/asmtest.OR1K.elf
```

The output from this should be as follows:

```
. . . lines omitted . . .

Info 'cpu1', 0x0000000000010000: l.ori    r30,r0,0x0
Info 'cpu1', 0x0000000000010004: l.movhi  r1,0x8000
Info 'cpu1', 0x0000000000010008: l.movhi  r2,0x1234
Info 'cpu1', 0x000000000001000c: l.ori    r2,r2,0x5678
Info 'cpu1', 0x0000000000010010: l.ori    r3,r0,0x0
Info 'cpu1', 0x0000000000010014: l.sb     0x0(r1),r2
Info 'cpu1', 0x0000000000010018: l.sh     0x0(r1),r2
Info 'cpu1', 0x000000000001001c: l.sw     0x0(r1),r2
Info 'cpu1', 0x0000000000010020: l.addi   r3,r3,0x1
Info 'cpu1', 0x0000000000010024: l.sfeqi  r3,0xa
Info 'cpu1', 0x0000000000010028: l.bnf    0x00010014
Info 'cpu1', 0x000000000001002c: l.addi   r1,r1,0x1
Info 'cpu1', 0x0000000000010014: l.sb     0x0(r1),r2
Info 'cpu1', 0x0000000000010018: l.sh     0x0(r1),r2
Info 'cpu1', 0x0000000000000200: l.addi   r30,r30,0x1
Info 'cpu1', 0x0000000000000204: l.addi   r1,r1,0x1
Info 'cpu1', 0x0000000000000208: l.rfe

. . . lines omitted . . .

Info 'cpu1', 0x000000000001001c: l.sw     0x0(r1),r2
Info 'cpu1', 0x0000000000010020: l.addi   r3,r3,0x1
Info 'cpu1', 0x0000000000010024: l.sfeqi  r3,0xa
Info 'cpu1', 0x0000000000010028: l.bnf    0x00010014
Info 'cpu1', 0x000000000001002c: l.addi   r1,r1,0x1
Info 'cpu1', 0x0000000000010030: l.div    r30,r30,r0
Info 'cpu1', 0x0000000000010034: l.nop    0x0
Processor 'cpu1' terminated at 'exit', address 0x10034
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 80000025   R2 : 12345678   R3 : 0000000a
 R4 : deadbeef   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : deadbeef   R10: deadbeef   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: 0000001b   R31: deadbeef
 PC : 00010038   SR : 00008601   ESR: 00008001   EPC: 0001001c
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:1 CF:1 OF:0
--------------- --------------- --------------- ---------------

processor has executed 185 instructions
```

18.5.2 Operation in Imperas MP Debugger

The same platform can be executed with the Imperas professional simulator for debugging with the Imperas MP Debugger

```
IMPERAS_RUNTIME=CpuManager \
    platform/harness.$IMPERAS_ARCH.exe --trace --idebug \
        --program application/asmtest.OR1K.elf \
```

The debugger allows access to the extended programmers view using the *view* command.

The platform is created as a shared object to load into the Imperas MP Debugger as part of the normal platform build process

```
make -C platform
```

If the same platform is executed with the address exception event point enabled the debugger will stop execution and allow the values of the defined objects to be accessed. The output from this should be as follows.

Before starting simulation view the defined objects

```
idebug (cpu1) > view

platform
    Events:
        begin: Start of simulation event
        finish: Finish of simulation event
    cpu1: processor
        Events:
            modeswitch: Mode switch event
            exception: Exception event
        type = or1k
        id = 0 (0x00000000)
        addressException: Address exception
            Events:
                address: Address exception event trigger
            eear:  = 3735928559 (0xdeadbeef)
            epc:  = 3735928559 (0xdeadbeef)
            esr:  = 3735928559 (0xdeadbeef)
```

The new view object, `addressException`, is visible, containing an event and views of the `eear`, `epc` and `esr` registers. Note that the processor also contains automatically-created `modeswitch` and `exception` events. The `modeswitch` event is triggered when the processor switches mode using `vmirtSetMode`. The `exception` event is triggered when the processor updates its execution address using `vmirtSetPCException`.

Set an event point for the address exception event

```
idebug (cpu1) > event address
Created eventpoint 1 on /platform/cpu1/exception/address
```

Run the simulation

```
idebug (cpu1) > continue
Info 'cpu1', 0x0000000000010000: l.ori    r30,r0,0x0
Info 'cpu1', 0x0000000000010004: l.movhi  r1,0x8000
Info 'cpu1', 0x0000000000010008: l.movhi  r2,0x1234
Info 'cpu1', 0x000000000001000c: l.ori    r2,r2,0x5678
Info 'cpu1', 0x0000000000010010: l.ori    r3,r0,0x0
Info 'cpu1', 0x0000000000010014: l.sb     0x0(r1),r2
Info 'cpu1', 0x0000000000010018: l.sh     0x0(r1),r2
Info 'cpu1', 0x000000000001001c: l.sw     0x0(r1),r2
Info 'cpu1', 0x0000000000010020: l.addi   r3,r3,0x1
```

```
Info 'cpu1', 0x0000000000010024: l.sfeqi  r3,0xa
Info 'cpu1', 0x0000000000010028: l.bnf    0x00010014
Info 'cpu1', 0x000000000001002c: l.addi   r1,r1,0x1
Info 'cpu1', 0x0000000000010014: l.sb     0x0(r1),r2
Info 'cpu1', 0x0000000000010018: l.sh     0x0(r1),r2
```

Simulation stops on the event point i.e. an address exception

```
Eventpoint 1 for /platform/cpu1/exception/address triggered
0x00000200 in ?? ()
```

Examining the view provides all the information defined in the model. In this case it is simply the register values associated with an address exception.

```
idebug (cpu1) > view

platform
    Events:
        begin: Start of simulation event
        finish: Finish of simulation event
    cpu1: processor
        Events:
            modeswitch: Mode switch event
            exception: Exception event
        type = or1k
        id = 0 (0x00000000)
        exception: Address exception
            Events:
                address: Address exception event trigger
            eear:  = 2147483649 (0x80000001)
            epc:  = 65560 (0x00010018)
            esr:  = 32769 (0x00008001)

idebug (cpu1) >
```

19 Implementing Save/Restore

This section describes how to add save/restore support to the processor model. Note that save/restore is available only with Imperas Professional Tools; it is not supported by OVPsim.

When a processor has save/restore support implemented, it is possible to run simulations that restart from a previously-saved state, instead of having to restart a simulation from the beginning every time. There are various functions available in the OP interface for this:

1. `opProcessorStateSaveFile`/`opProcessorStateRestoreFile`: these two functions save and restore the state of a single processor using a file in a standard text format. They are typically used when the processor model is being used as part of a larger system that itself implements save/restore on other platform components.
2. `opProcessorStateSave`/`opProcessorStateRestore`: these two functions save and restore the state of a single processor using user-defined callbacks to implement the save/restore process. They are typically used when the processor model is being used as part of a larger system that itself implements save/restore on other platform components, and which has its own database format which must be supported.
3. `opRootModuleStateSaveFile`/`opRootModuleStateRestoreFile`: these two functions save and restore the state of an entire platform using a file in a standard text format. Saved information will include the state of all processors, memories and peripherals in the simulation, plus any other module-specific data defined in module save/restore callbacks. They are typically used when the entire simulation is implemented using OVP components.
4. `opRootModuleStateSave`/`opRootModuleStateRestore`: these two functions save and restore the state of an entire platform using user-defined callbacks to implement the save/restore process. Saved information will include the state of all processors, memories and peripherals in the simulation, plus any other module-specific data defined in module save/restore callbacks. They are typically used when the entire simulation is implemented using OVP components, but a custom save/restore format must be supported.

19.1 Example Save/Restore Implementation

A model for the OR1K processor with save/restore can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/17.or1kSaveRestore
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/17.or1kSaveRestore .
```

Compile the model, harness and application using the `make` command:

```
cd 17.or1kSaveRestore
make
```

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is based on the previous example, with the changes listed in following sections.

19.2 Save/Restore Mode Active - `or1kStructure.h`

The or1k structure has a new field, `inSaveRestore`, used to indicate whether save/restore is currently active:

```
typedef struct or1kS {

    . . . lines omitted . . .

    Bool           inSaveRestore;    // whether save/restore in progress

    . . . lines omitted . . .

} or1k, *or1kP;
```

19.3 Save/Restore Interface Functions - `or1kMain.c`

The interface to save/restore is implemented by two functions in `or1kMain.c`. Function `or1kSaveStateCB` is responsible for saving processor state. It is implemented like this:

```
VMI_SAVE_STATE_FN(or1kSaveStateCB) {

    or1kP or1k = (or1kP)processor;

    switch(phase) {

        case SRT_BEGIN:
            // start of save/restore process
            or1k->inSaveRestore = True;
            break;

        case SRT_BEGIN_CORE:
            // start of individual core
            break;

        case SRT_END_CORE:
            // end of individual core: save fields not covered by debug register
            // interface
            VMIRT_SAVE_FIELD(cxt, or1k, TTCRSetCount);
            VMIRT_SAVE_FIELD(cxt, or1k, timerRunning);
            VMIRT_SAVE_FIELD(cxt, or1k, resetInput);
            VMIRT_SAVE_FIELD(cxt, or1k, reset);
            break;

        case SRT_END:
            // end of save/restore process
            or1k->inSaveRestore = False;
            break;
```

```
        default:
            // not reached
            VMI_ABORT("unimplemented case"); // LCOV_EXCL_LINE
            break;
    }
}
```

The state save function is defined using the `VMI_SAVE_STATE_FN` macro:

```
#define VMI_SAVE_STATE_FN(_NAME) void _NAME( \
    vmiProcessorP       processor,  \
    vmiSaveContextP     cxt,        \
    vmiSaveRestorePhase phase,      \
    Uns32               srVersion   \
)
typedef VMI_SAVE_STATE_FN((*vmiSaveStateFn));
```

The `cxt` argument is an opaque type pointer to an object that implements primitive save operations (see below). The phase argument indicates one of four phases, as follows:

```
typedef enum vmiSaveRestorePhaseE {
    SRT_BEGIN,      // start save/restore for a processor (or SMP container)
    SRT_BEGIN_CORE, // start save/restore for a single SMP core
    SRT_END_CORE,   // end save/restore for a single SMP core
    SRT_END,        // end save/restore for a processor (or SMP container)
} vmiSaveRestorePhase;
```

When performing state save, the function is used in the following way:

1. An initial call is made, passing the processor, a context argument of type `vmiSaveContextP`, a phase argument of `SRT_BEGIN`, and a model-specific version. The model can use this phase to perform any model-specific preparation before starting the save.
2. A second call is made with the same arguments, except that the phase is `SRT_BEGIN_CORE`. The model can use this phase to perform any model-specific state save *before* standard state save occurs.
3. Standard information is then saved automatically. This includes:
    a. The processor program counter (and whether it is executing in a delay slot);
    b. The processor halted/executing state;
    c. The processor instruction count, and any associated state (for example, deration factor);
    d. Any processor registers accessible for read/write through the register interface that have not been specifically excluded using `noSaveRestore` in the register definition;
    e. The processor interrupt timer state (i.e. timer state set using `vmirtSetICountInterrupt`).
4. A third call is made with the same arguments, except that the phase is `SRT_END_CORE`. The model can use this phase to perform any model-specific state save *after* standard state save occurs.

5. A final call is made with the same arguments, except that the phase is SRT_END. The model can use this phase to perform any model-specific state finalization after save completes.

In the OR1K model, the callback phases are used as follows:

1. In phase SRT_BEGIN, a Boolean inSaveRestore is set to True to indicate that save/restore is in progress;
2. In phase SRT_END_CORE, some processor state not covered by the standard algorithm is saved (see below);
3. In phase SRT_END, inSaveRestore is set to False to indicate that save/restore is no longer active.

Most processor state held in read/write registers is saved or restored automatically and therefore does not need to be specifically handled in the save/restore callbacks. Action is required only to save/restore processor state that is *not* held in read/write registers. For example, in the OR1K model, there are three such state items:

1. The instruction count at which the TTCR register was last written, (TTCRSetCount);
2. The Boolean indicating whether the processor timer is currently running (timerRunning);
3. The Boolean indicating whether a reset should be triggered at the start of the next instruction (reset) and the latched value of the reset input (resetInput).

These four pieces of state are saved using the VMIRT_SAVE_FIELD macro:

```
            VMIRT_SAVE_FIELD(cxt, or1k, TTCRSetCount);
            VMIRT_SAVE_FIELD(cxt, or1k, timerRunning);
            VMIRT_SAVE_FIELD(cxt, or1k, resetInput);
            VMIRT_SAVE_FIELD(cxt, or1k, reset);
```

This macro expands to the following calls to the VMI run time function vmirtSave:

```
vmirtSave(cxt, “TTCRSetCount”, &or1k->TTCRSetCount, sizeof(or1k->TTCRSetCount));
vmirtSave(cxt, “timerRunning”, &or1k->timerRunning, sizeof(or1k->timerRunning));
vmirtSave(cxt, “resetInput”,   &or1k->TTCRSetCount, sizeof(or1k->resetInput));
vmirtSave(cxt, “reset”,        &or1k->TTCRSetCount, sizeof(or1k->reset));
```

The function vmirtSave saves the value of the indicated structure field in the given context. The saved value is preceded by a *key name* (for example, “TTCRSetCount”) which enables sanity checking during later restore.

The set of explicitly-saved data for this processor is quite small; in more complex processors, it is likely to be larger. Specifically, it is usually necessary to explicitly save and restore the following:

1. The value of model timers other than the default timer (for example, timers created using `vmirtCreateModelTimer`). To save and restore these, use functions `vmirtSaveModelTimer` and `vmirtRestoreModelTimer`.
2. The current set of virtual memory mappings, if those mappings cannot be derived from register values (e.g. a TLB).
3. The latched values of any processor inputs, and any values derived from them (e.g. `reset` and `resetInput` in the current example).
4. The contents of any model-local memories created using `vmirtNewDomain` and `vmirtMapMemory` (for example, TCM memories). To save and restore these, use functions `vmirtSaveDomain` and `vmirtRestoreDomain`.

Function `or1kRestoreStateCB` is responsible for restoring processor state. It is implemented like this:

```
VMI_RESTORE_STATE_FN(or1kRestoreStateCB) {

    or1kP or1k = (or1kP)processor;

    switch(phase) {

        case SRT_BEGIN:
            // start of save/restore process
            or1k->inSaveRestore = True;
            break;

        case SRT_BEGIN_CORE:
            // start of individual core
            break;

        case SRT_END_CORE:
            // end of individual core: save fields not covered by debug register
            // interface
            VMIRT_RESTORE_FIELD(cxt, or1k, TTCRSetCount);
            VMIRT_RESTORE_FIELD(cxt, or1k, timerRunning);
            VMIRT_RESTORE_FIELD(cxt, or1k, resetInput);
            VMIRT_RESTORE_FIELD(cxt, or1k, reset);
            // take any pending interrupt before the next instruction
            or1kInterruptNext(or1k);
            break;

        case SRT_END:
            // end of save/restore process
            or1k->inSaveRestore = False;
            break;

        default:
            // not reached
            VMI_ABORT("unimplemented case"); // LCOV_EXCL_LINE
            break;
    }
}
```

Note that `or1kRestoreStateCB` closely matches `or1kSaveStateCB` in its form, except that the `VMIRT_RESTORE_FIELD` macro is used in place of the `VMIRT_SAVE_FIELD` macro. This macro expands to calls to the VMI function `vmirtRestore`, each of which reads a named record from the context.

The only other significant difference is that after restoring model-specific state in the SRT_END_CORE phase, a call is make to or1kInterruptNext:

```
            VMIRT_RESTORE_FIELD(cxt, or1k, TTCRSetCount);
            VMIRT_RESTORE_FIELD(cxt, or1k, timerRunning);
            VMIRT_RESTORE_FIELD(cxt, or1k, resetInput);
            VMIRT_RESTORE_FIELD(cxt, or1k, reset);
            // take any pending interrupt before the next instruction
            or1kInterruptNext(or1k);
```

The purpose of this call is to make sure that if an interrupt or reset exception is required immediately after restore because of processor state changes, it will be taken.

Prototypes for or1kSaveStateCB and or1kRestoreStateCB have been added to file or1kFunctions.h.

19.3.1 Save/Restore Data Constraints

One important constraint on save/restore is that *the order of records read during restore must match the order in which the records were created during save*. Record names are checked against expected names during the restore process, and restore will fail if a difference is found. For example, modifying the order of restore field lines like this:

```
            VMIRT_RESTORE_FIELD(cxt, or1k, TTCRSetCount);
            VMIRT_RESTORE_FIELD(cxt, or1k, timerRunning);
            VMIRT_RESTORE_FIELD(cxt, or1k, reset);
            VMIRT_RESTORE_FIELD(cxt, or1k, resetInput);
```

Produces this error during restore:

```
Fatal (SRE1) Expected 'reset' but found 'resetInput' - terminating restore
Info Exiting
```

This error indicates that there is a serious problem in the design of the processor model save/restore interface and therefore terminates the simulation (processor state is very likely to be corrupted, making it unusable).

After performing a restore, the simulator also automatically checks that the visible processor register values are as expected; if not the restore will fail. For example, modifying the restore lines like this:

```
            VMIRT_RESTORE_FIELD(cxt, or1k, TTCRSetCount);
            VMIRT_RESTORE_FIELD(cxt, or1k, timerRunning);
            VMIRT_RESTORE_FIELD(cxt, or1k, resetInput);
            VMIRT_RESTORE_FIELD(cxt, or1k, reset);
            // deliberately corrupt PICSR after restore
            or1k->PICSR = 0x12345678;
```

Produces this error during restore:

```
Error (PC_RESTORE_CHK) Register 'PICSR' value mismatch after restore -
expected:0x00000000 actual:0x12345678
```

In this case, simulation is not terminated because the error is less serious than a restore that fails completely, but it is usually the case that further simulation is inadvisable. The return code for the restore function used (for example, `opProcessorStateRestoreFile`) will be `OP_SAVE_ERROR` if any such error is found. Value inconsistencies such as this usually occur when the value of a register is derived from some other processor state. To correct them, ensure that all such dependent register state is derived at the end of the `SRT_END_CORE` phase.

19.3.2 Multicore Processors

It is possible to create multicore processors using the VMI API (see *SMP Processor Hierarchies* in the *VMI Run Time Function Reference* document for more information). For such processors, the save/restore flow is as follows.

1. A single call is made to the save or restore function, passing the *root processor* and `phase` of `SRT_BEGIN`.
2. For each leaf processor a call is made to the save or restore function, passing the *leaf processor* and `phase` of `SRT_BEGIN_CORE`, followed by a second call with phase `SRT_END_CORE`.
3. Finally, a single call is made to the save or restore function, passing the *root processor* and `phase` of `SRT_END`.

19.4 Save/Restore Function Registration - `or1kAttrs.c`

The `modelAttrs` structure in `or1kAttrs.c` has been changed to include references to the save/restore functions, as follows:

```
const vmiIASAttr modelAttrs = {

    . . . lines omitted . . .

    ////////////////////////////////////////////////////////////////////////
    // SAVE/RESTORE ROUTINES
    ////////////////////////////////////////////////////////////////////////

    .saveCB    = or1kSaveStateCB,
    .restoreCB = or1kRestoreStateCB,
    .srVersion = 1,

    . . . lines omitted . . .

};
```

`srVersion` is the model-specific save/restore version number: it is this number that is passed as the `srVersion` parameter of the save callback. During restore, `srVersion` is *not* taken from the `modelAttrs` structure, but instead from the saved context structure (in other words, the restore callback is told the version number used when data was *saved*).

As models develop over time, it is often the case that the information used during save/restore needs to change. By making the restore function sensitive to `srVersion`, it is possible to support both current and previous version save files if required.

19.5 Save/Restore Mode Accesses - or1kExceptions.c

We stated previously that the processor interrupt timer state (i.e. timer state set using `vmirtSetICountInterrupt`) is saved and restored automatically and does not need to be handled explicitly. However, we *do* still need to save and restore the value of the `TTCR` field in the processor structure, because this value is used when calculating timer expiry delays. We handle this by modifying `getTTCR` and `setTTCR` as follows:

```
inline static Uns32 getTTCR(or1kP or1k) {

    Uns32 TTCR = or1k->TTCR;

    if(or1k->timerRunning && !or1k->inSaveRestore) {
        TTCR = TTCR - or1k->TTCRSetCount + getThisICount(or1k);
    }

    return TTCR;
}
```

If not in save/restore mode, `getTTCR` behaves as before. In in save/restore mode, `getTTCR` now returns the unmodified value of the `TTCR` field.

```
static void setTTCR(or1kP or1k, Uns32 TTCR) {

    // set raw value of TTCR
    or1k->TTCR = TTCR;

    // update fields dependent on TTCR only if save/restore is not active
    if(!or1k->inSaveRestore) {

        // record count at which TTCR was modified
        or1k->TTCRSetCount = getThisICount(or1k);

        // if the timer is running, calculate the cycle delay to any interrupt
        // (28 bits maximum) and schedule timer interrupt
        if(or1k->timerRunning) {
            Uns32 iCount = (or1k->TTMR_TP-TTCR-1) & 0xfffffff;
            vmirtSetICountInterrupt((vmiProcessorP)or1k, iCount);
        } else {
            vmirtClearICountInterrupt((vmiProcessorP)or1k);
        }
    }
}
```

If not in save/restore mode, `setTTCR` behaves as before. In in save/restore mode, `setTTCR` simply updates the of the `TTCR` field with the given value.

19.6 Save/Restore Test Harness - platform/harness.c

The test harness for this example, `platform/harness.c`, has been specially constructed to allow validation of save/restore. The platform instantiates *two* instances of the OR1K processor (called `procA` and `procB`) connected to a common memory bus. Then, simulation is run in a loop of the following steps:

1. A single instruction is executed on `procA`;

2. The state of `procA` is saved to a file using `opProcessorStateSaveFile`;
3. The processor state is loaded onto `procB` using `opProcessorStateRestoreFile`;
4. A single instruction is executed on `procB`;
5. The state of `procB` is saved to a file using `opProcessorStateSaveFile`;
6. The processor state is loaded onto `procA` using `opProcessorStateRestoreFile`, at which point execution restarts at step 1.

The effect of this harness is to run a simulation executing alternate instructions on `procA` and `procB`, transferring all state from one processor to the other before each instruction is executed. The loop is implemented in function simulate, as follows:

```
static Bool simulate(
    procPairP pair,
    Uns32     clocks,
    optNetP   intr0Net,
    Bool      intr0Value,
    optNetP   resetNet,
    Bool      resetValue
) {
    const char *stateFile = "checkpoint.txt";
    Uns32       i;

    // simulate on alternate processors, one instruction at a time
    for(i=0; i<clocks; i++) {

        // get next processor to simulate
        optProcessorP processor = pair->processors[pair->nextRun];

        // restore processor state before simulation if required
        if(pair->doRestore) {
            opProcessorStateRestoreFile(processor, stateFile);
        }

        // if not the first iteration, apply interrupt net stimulus
        if(!i) {
            opNetWrite(intr0Net, intr0Value);
            opNetWrite(resetNet, resetValue);
        }

        optStopReason stopReason = opProcessorSimulate(processor, 1);

        // save processor after simulation
        opProcessorStateSaveFile(processor, stateFile);

        // prepare for next iteration
        pair->nextRun   = !pair->nextRun;
        pair->doRestore = True;

        switch(stopReason) {

           case OP_SR_SCHED:
               // hit the scheduler limit
               break;

           case OP_SR_HALT:
               // processor halted in reset
               break;

           case OP_SR_EXIT:
```

```
                // processor has exited
                return False;

            case OP_SR_FINISH:
                // simulation must end
                return False;

            default:
                opPrintf("unexpected stopReason %u\n", stopReason);
                return False;
        }
    }

    // here when the required number of instructions have been executed
    return True;
}
```

The harness calls function simulate using this sequence:

```
    // run for 9 instructions
    simulate(&pair, 9, intr0Net,0, resetNet,0);

    // assert reset for 5 instructions
    simulate(&pair, 5, intr0Net,0, resetNet,1);

    // run for 9 instructions
    simulate(&pair, 9, intr0Net,0, resetNet,0);

    // assert interrupt for 1 instruction
    simulate(&pair, 1, intr0Net,1, resetNet,0);

    // run for 9 instructions
    simulate(&pair, 9, intr0Net,0, resetNet,0);

    // assert interrupt for 1 instruction
    simulate(&pair, 1, intr0Net,1, resetNet,0);

    // run processor until done
    simulate(&pair, -1, intr0Net,0, resetNet,0);
```

Refer to section 16.9 for more details on the purpose of this sequence.

19.7 Testing Save/Restore

Run the platform using the assembler executable file:

```
platform/harness.$IMPERAS_ARCH.exe –-trace \
    --program application\asmtest.OR1K.elf \
    --trace --tracechange --traceshowicount
```

The output from this should be as follows:

```
Info 1: 'platform/procA', 0x0000000000010000(_start): l.ori    r30,r0,0x0
Info   R30 deadbeef -> 00000000
Info 2: 'platform/procB', 0x0000000000010004(_start+4): l.addi
r1,r0,0xffffffff
Info   R1 00000000 -> ffffffff
Info 3: 'platform/procA', 0x0000000000010008(_start+8): l.mtspr  r0,r1,18432
Info   PICMR 00000000 -> ffffffff
```

```
Info 4: 'platform/procB', 0x000000000001000c(_start+c): l.ori    r1,r0,0x6
Info   R1 ffffffff -> 00000006
Info 5: 'platform/procA', 0x0000000000010010(_start+10): l.mtspr  r0,r1,17
Info   SR 00008001 -> 00008006
Info 6: 'platform/procB', 0x0000000000010014(_start+14): l.ori    r1,r0,0x4
Info   R1 00000006 -> 00000004
Info 7: 'platform/procA', 0x0000000000010018(loop1): l.addi   r1,r1,0xffffffff
Info   R1 00000004 -> 00000003
Info   SR 00008006 -> 00008406
Info 8: 'platform/procB', 0x000000000001001c(loop1+4): l.sfeqi  r1,0x0
Info 9: 'platform/procA', 0x0000000000010020(loop1+8): l.bnf    0x00010018
Info 15: 'platform/procA', 0x0000000000010024(loop1+c): *** FETCH EXCEPTION ***
Info   SR 00008406 -> 0000a401
Info   EPCR deadbeef -> 00010020
Info   ESR deadbeef -> 00008406
Info 16: 'platform/procB', 0x0000000000000100(.text+100): l.j      0x00010000
Info 17: 'platform/procA', 0x0000000000000104(.text+104): l.addi   r30,r30,0x1
Info   R30 00000000 -> 00000001
Info   SR 0000a401 -> 0000a001
Info 18: 'platform/procB', 0x0000000000010000(_start): l.ori    r30,r0,0x0
Info   R30 00000001 -> 00000000
Info 19: 'platform/procA', 0x0000000000010004(_start+4): l.addi
r1,r0,0xffffffff
Info   R1 00000003 -> ffffffff
Info 20: 'platform/procB', 0x0000000000010008(_start+8): l.mtspr  r0,r1,18432
Info 21: 'platform/procA', 0x000000000001000c(_start+c): l.ori    r1,r0,0x6
Info   R1 ffffffff -> 00000006
Info 22: 'platform/procB', 0x0000000000010010(_start+10): l.mtspr  r0,r1,17
Info   SR 0000a001 -> 00008006
Info 23: 'platform/procA', 0x0000000000010014(_start+14): l.ori    r1,r0,0x4
Info   R1 00000006 -> 00000004
Info 24: 'platform/procB', 0x0000000000010018(loop1): *** FETCH EXCEPTION ***
Info   SR 00008006 -> 00008001
Info   EPCR 00010020 -> 00010018
Info   ESR 00008406 -> 00008006
Info   PICSR 00000000 -> 00000001
Info   EXCPT 00000000 -> 00000006
Info   PICSR 00000001 -> 00000000
Info 25: 'platform/procA', 0x0000000000000800(.text+800): l.addi   r30,r30,0x1
Info   R30 00000000 -> 00000001
Info 26: 'platform/procB', 0x0000000000000804(.text+804): l.rfe
Info   SR 00008001 -> 00008006
Info 27: 'platform/procA', 0x0000000000010018(loop1): l.addi   r1,r1,0xffffffff
Info   R1 00000004 -> 00000003
Info   SR 00008006 -> 00008406
Info 28: 'platform/procB', 0x000000000001001c(loop1+4): l.sfeqi  r1,0x0
Info 29: 'platform/procA', 0x0000000000010020(loop1+8): l.bnf    0x00010018
Info 30: 'platform/procB', 0x0000000000010024(loop1+c): l.nop    0x0
Info 31: 'platform/procA', 0x0000000000010018(loop1): l.addi   r1,r1,0xffffffff
Info   R1 00000003 -> 00000002
Info 32: 'platform/procB', 0x000000000001001c(loop1+4): l.sfeqi  r1,0x0
Info 33: 'platform/procA', 0x0000000000010020(loop1+8): l.bnf    0x00010018
Info 34: 'platform/procB', 0x0000000000010024(loop1+c): *** FETCH EXCEPTION ***
Info   SR 00008406 -> 0000a401
Info   EPCR 00010018 -> 00010020
Info   ESR 00008006 -> 00008406
Info   PICSR 00000000 -> 00000001
Info   PICSR 00000001 -> 00000000
Info 35: 'platform/procA', 0x0000000000000800(.text+800): l.addi   r30,r30,0x1
Info   R30 00000001 -> 00000002
Info   SR 0000a401 -> 0000a001
Info 36: 'platform/procB', 0x0000000000000804(.text+804): l.rfe
Info   SR 0000a001 -> 00008406
```

```
Info 37: 'platform/procA', 0x0000000000010020(loop1+8): l.bnf    0x00010018
Info 38: 'platform/procB', 0x0000000000010024(loop1+c): l.nop    0x0
Info 39: 'platform/procA', 0x0000000000010018(loop1): l.addi   r1,r1,0xffffffff
Info   R1 00000002 -> 00000001
Info 40: 'platform/procB', 0x000000000001001c(loop1+4): l.sfeqi  r1,0x0
Info 41: 'platform/procA', 0x0000000000010020(loop1+8): l.bnf    0x00010018
Info 42: 'platform/procB', 0x0000000000010024(loop1+c): l.nop    0x0
Info 43: 'platform/procA', 0x0000000000010018(loop1): l.addi   r1,r1,0xffffffff
Info   R1 00000001 -> 00000000
Info 44: 'platform/procB', 0x000000000001001c(loop1+4): l.sfeqi  r1,0x0
Info   SR 00008406 -> 00008606
Info 45: 'platform/procA', 0x0000000000010020(loop1+8): l.bnf    0x00010018
Info 46: 'platform/procB', 0x0000000000010024(loop1+c): l.nop    0x0
Info 47: 'platform/procA', 0x0000000000010028(exit): l.nop    0x0
```

The application is the same as was used in example `14.or1kBehaviorExternalInterrupt`. Note that instructions in the application are executed alternately on processor `procA` and `procB`.

19.7.1 State File Format

When simulation completes, examine the state file checkpoint.txt in the run directory. The contents should be as follows:

```
VERSION
2
TYPE
or1k
MODEL_VERSION
00000001
. . . lines omitted . . .
START_CORE
00000000
START_REGISTERS
REGISTER
R1
00000000
REGISTER
R2
deadbeef
. . . lines omitted . . .
REGISTER
EXCPT
00000006
END_REGISTERS
END_CORE
START_CORE
00000000
TTCRSetCount
00000000
timerRunning
00
reset
00
END_CORE
START_CHECK
. . . lines omitted . . .
END_CHECK
IMPLICIT_TIMER
0000000000000000ffffffffffffffff
END
```

The format of this file is largely a set of of key/value pairs, some information of which is automatically-generated model and simulator data and some of which is explicitly-created fields (for example, the `TTCRSetCount`, `timerRunning` and `reset` entries).

20 Implementing Instruction Attributes

The OVP *Instruction Attribute Interface* simplifies the creation of tools that require detailed knowledge of the instruction set of the processor on which they are installed. For example, it might be required to know which registers are accessed and modified by an instruction and exactly how those registers are used. When the instruction attributes interface is implemented for a processor, this information is available using an API defined in the `ocl/oclia.h` header file[12].

Note: instruction attributes are only supported for the Imperas M*SDK product.

20.1 Instruction Attribute Access – `ocl/oclia.h`

The fundamental structure describing a processor instruction is accessed using the `octiaAttrP` opaque type pointer. It is possible to create a structure for a given instruction address either using the VMI or OP interfaces. In the VMI interface, use function `vmiiaGetAttrs`, defined in file `vmiInstructionAttrs.h` as follows:

```
octiaAttrP vmiiaGetAttrs(
    vmiProcessorP   processor,
    Addr            simPC,
    octiaDataSelect select,
    Bool            applyDFA
);
```

This function takes a processor argument and an instruction address for which instruction attributes are required. The kinds of data that should be returned is specified by the `select` argument, which is a bitfield enumeration of type `octiaDataSelect`, defined in `ocl/ocliaTypes.h`, described in detail in a later subsection. The `applyDFA` argument affects how the effects of an instruction are reported, as explained below.

In the OP interface, function `opProcessorInstructionAttributes` is used to obtain instruction attributes in a similar manner. The function is defined in `op.h` as follows:

```
octiaAttrP opProcessorInstructionAttributes (
    optProcessorP processor,
    Addr          simPC,
    Uns32         select,
    Bool          applyDFA
);
```

Generation of instruction attributes is performed as follows:

[12] The `ocl` directory contains *OVP common header files* that may be included and used in both VMI and OP contexts.

1. The *morpher callback* is called for the instruction address specified by the `simPC` argument (in either interface) using the current processor state[13], producing the same intermediate data structure from which the JIT compiler is normally driven.
2. If the `applyDFA` argument is `True`, the standard JIT compiler optimizations are then run on the intermediate data structure. This can cause removal or simplification of parts of the data structure.
3. Instead of using the intermediate data structure to generate JIT-compiled code, its contents are translated to an `ocl` data structure that can then be traversed using the instruction attributes API. By cross-referencing the intermediate data structure and the debug register interface, a list of registers *read* and *written* by the instruction is created.

The `octiaAttrP` structure is allocated from the heap. When it is no longer required, it should be deleted using function `ocliaFreeAttrs`:

```
void ocliaFreeAttrs(octiaAttrP attrs);
```

Much of the information available using the instruction attributes API is generated automatically from the processor morpher and debug register interface. The processor model requires enhancement only to handle cases where information cannot be automatically derived.

20.1.1 Information Available

The information returned in the octiaAttr structure is controlled by a bitfield enumeration of type `octiaDataSelect`:

```
typedef enum octiaDataSelectE {
    OCL_DS_NONE    = 0x00, ///< empty mask
    OCL_DS_NODES   = 0x01, ///< record node list
    OCL_DS_REG_R   = 0x02, ///< record debug interface registers read
    OCL_DS_REG_W   = 0x04, ///< record debug interface registers written
    OCL_DS_RANGE_R = 0x08, ///< record field ranges read (and not registers)
    OCL_DS_RANGE_W = 0x10, ///< record field ranges written (and not registers)
    OCL_DS_FETCH   = 0x20, ///< record fetch ranges
    OCL_DS_NEXTPC  = 0x40, ///< record next PC expressions
    OCL_DS_ADDRESS = 0x80  ///< record load/store address expressions
} octiaDataSelect;
```

Several classes of information can be returned by a single call by combining the required members of the enumeration with bitwise-or. The meaning of each enumeration member is given below.

20.1.1.1 Node List: `OCL_DS_NODES`

The `OCL_DS_NODES` member causes the intermediate node data structure to be recorded with the instruction attributes structure. This is of use only for debugging; it is shown

[13] For processors that support different instruction sets (e.g. ARM Thumb and AArch32 instructions) it might sometimes be required to obtain attributes for a currently-inactive state. In this case, it is necessary to temporarily force the processor into the required state while attribute generation is done.

when instruction attributes are printed using function `ocliaPrintAttrs`, and not discussed further here.

20.1.1.2 Registers Read: `OCL_DS_REG_R`

The `OCL_DS_REG_R` member causes registers *read* by an instruction to be recorded (in other words, registers used as *inputs* to that instruction). The list of registers can be traversed using functions `ocliaGetFirstReadReg` and `ocliaGetRegListNext`. Function `ocliaGetRegListReg` obtains a register object for a register list element; this can then be converted to the equivalent `vmiRegInfoP` structure using `vmiiaConvertRegInfo`, or to the equivalent `optRegP` structure using `opRegConvert`, depending on whether the VMI or OP API is in use. See the harness used with this example for more details.

20.1.1.3 Registers Written: `OCL_DS_REG_W`

The `OCL_DS_REG_W` member causes registers *written* by an instruction to be recorded (in other words, registers modified as *outputs* to that instruction). Usage is very similar to `OCL_DS_REG_R` above, except that function `ocliaGetFirstWrittenReg` is used instead of `ocliaGetFirstReadReg` to obtain the first written register list member.

20.1.1.4 Unknown Registers Read: `OCL_DS_RANGE_R`

As indicated previously, the instruction attributes API attempts to construct a list of registers read by an instruction so that they can be reported using the `OCL_DS_REG_R` selection member, described above. To do this, it tries to match up any `vmiReg` specified as part of an instruction implementation with a corresponding `raw` field specification in a `vmiRegInfo` structure for a register in the processor (see section 17.3 for more information on the `vmiRegInfo` structure). Sometimes, a `vmiReg` used as an input is found that does not match *any* `vmiRegInfo` structure in this way, which usually indicates that the processor instruction attributes interface requires enhancement, as described later in this chapter. The `OCL_DS_RANGE_R` member causes any such register range to be recorded. The list of unmatched ranges can be traversed using functions `ocliaGetFirstReadRange` and `ocliaGetRangeNext`. Functions `ocliaGetRangeLow` and `ocliaGetRangeHigh` return the low and high byte offsets of the `vmiReg` range that cannot be matched with a processor register, respectively. See the harness used with this example for more details.

20.1.1.5 Unknown Registers Written: `OCL_DS_RANGE_W`

The `OCL_DS_RANGE_W` member causes unmatched register ranges *written* by an instruction to be recorded (in other words, unmatched register ranges modified as *outputs* to that instruction). Usage is very similar to `OCL_DS_RANGE_R` above, except that function `ocliaGetFirstWrittenRange` is used instead of `ocliaGetFirstReadRange` to obtain the first list member.

20.1.1.6 Instruction Fetch Operations: `OCL_DS_FETCH`

The `OCL_DS_FETCH` member causes each fetch that was performed while translating an instruction to be recorded. The list of the fetches can then be traversed using functions `ocliaGetFirstFetchRecord` and `ocliaGetNextFetchRecord`. For each fetch, it is possible to get:

1. The low address (`ocliaGetFetchRecordLow`);
2. The high address (`ocliaGetFetchRecordHigh`); and
3. A pointer to the bytes that were fetched (`ocliaGetFetchRecordBytes`).

20.1.1.7 Next Instruction Expressions: OCL_DS_NEXTPC

The `OCL_DS_NEXTPC` member causes a list of potential *next instruction expressions* to be recorded. Each expression can be evaluated using `ocliaEvaluate` to yield a program counter to which control could be transferred once this instruction completes. For most instructions, there will be a single next instruction expression which will be the next instruction address. Unconditional jumps without delay slots will also have a single next instruction expression, which evaluates to the the jump target address. Conditional jumps will have two (or, occasionally, more) next instruction expressions, each of which evaluates to a potential next instruction address. All jumps with delay slots have two next instruction expressions, for reasons explained later in this chapter.

20.1.1.8 Load/Store Address Expressions: OCL_DS_ADDRESS

The `OCL_DS_ADDRESS` member causes a list of *load/store address expressions* to be recorded. Each address expression can be evaluated using `ocliaEvaluate` to yield an address to which a load or store might be made. Information is also recorded giving the load/store data size, the memory domain which is being accessed, and the list of registers required to calculate the address.

20.2 Example Instruction Attributes Implementation

A model for the OR1K processor with instruction attributes can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/18.or1kInstructionAttributes
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/18.or1kInstructionAttributes .
```

Compile the model, harness and application using the `make` command:

```
cd 18.or1kInstructionAttributes
make
```

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is based on the previous example, with the changes listed in following sections.

20.3 Baseline Instruction Attributes

Initially, we will see what information is generated by default using the instruction attributes API, and then as a second step enhance the model so that more information is provided. To enable basic instruction attributes support, a new callback function of type `vmiRegImplFn` must be defined. This type is defined by the following macro in `vmiDbg.h`:

```
#define VMI_REG_IMPL_FN(_NAME) void _NAME( \
    vmiProcessorP processor     \
)
typedef VMI_REG_IMPL_FN((*vmiRegImplFn));
```

This function takes a single processor argument. For the OR1K model, a function is defined in `or1kRegisters.c` using this macro as follows:

```
VMI_REG_IMPL_FN(or1kRegImpl) {
#if(ENABLE_ATTRS)
    . . . lines omitted . . .
#endif
}
```

Note that at this point we have compiled with `ENABLE_ATTRS` *undefined*; this means that the function body is empty for now. This *register implementation callback* is registered in `or1kAttrs.c` like this:

```
const vmiIASAttr modelAttrs = {

    . . . lines omitted . . .

    ////////////////////////////////////////////////////////////////////////
    // REGISTER ACCESS SUPPORT ROUTINES (DEBUGGER & SEMIHOSTING)
    ////////////////////////////////////////////////////////////////////////

    .regGroupCB      = or1kRegGroup,
    .regInfoCB       = or1kRegInfo,
    .regImplCB       = or1kRegImpl,
    .exceptionInfoCB = or1kExceptionInfo,
    .modeInfoCB      = or1kModeInfo,
    .getExceptionCB  = or1kGetException,
    .getModeCB       = or1kGetMode,
    .debugCB         = or1kDumpRegisters,

    . . . lines omitted . . .
};
```

Creating a register implementation callback like this is all that is required to enable instruction attributes support. We will now discuss the harness (which uses the attributes) before returning to the model later in this chapter.

20.4 Test Harness - `platform/harness.c`

The test harness for this example, `platform/harness.c`[14], has been designed to print out instruction attributes for every instruction in a program as it runs. The harness is described in detail here to show exactly how instruction attributes can be used.

The main simulation function is as follows:

```
static Bool simulate(optProcessorP processor, Uns32 clocks) {

    Uns32 i;

    for(i=0; i<clocks; i++) {

        showInstructionAttributes(processor);

        optStopReason stopReason = opProcessorSimulate(processor, 1);

        switch(stopReason) {

           case OP_SR_SCHED:
               // hit the scheduler limit
               break;

           case OP_SR_EXIT:
               // processor has exited
               return False;

           case OP_SR_FINISH:
               // simulation must end
               return False;

           case OP_SR_RD_PRIV:
           case OP_SR_WR_PRIV:
           case OP_SR_RD_ALIGN:
           case OP_SR_WR_ALIGN:
              // unhandled processor exception: simulation must end
              return False;

           default:
               opPrintf("unexpected stopReason %u\n", stopReason);
               return False;
        }
    }

    return True;
}
```

This simulation function calls `opProcessorSimulate` to execute instruction by instruction. Before each instruction is executed, function `showInstructionAttributes` is called, defined as follows:

```
static void showInstructionAttributes(optProcessorP processor) {

    Addr thisPC = opProcessorPC(processor);
```

[14] Note that the legacy ICM interface does not support instruction attributes, so this example contains an OP platform only.

```
    // select all attributes except nodes
    octiaDataSelect select = (
        OCL_DS_REG_R   |
        OCL_DS_REG_W   |
        OCL_DS_RANGE_R |
        OCL_DS_RANGE_W |
        OCL_DS_FETCH   |
        OCL_DS_NEXTPC  |
        OCL_DS_ADDRESS
    );

    // get instruction attributes for the current PC
    octiaAttrP attrs = opProcessorInstructionAttributes(
        processor, thisPC, select, False
    );

    if(attrs) {

        // print instruction disassembly
        opPrintf(
            "*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x"FMT_Ax" (%s)\n",
            thisPC,
            opProcessorDisassemble(processor, thisPC, OP_DSA_NORMAL)
        );

        // walk attributes
        walkAttrs(attrs);

        // free attributes
        ocliaFreeAttrs(attrs);
    }
}
```

This function calls `opProcessorInstructionAttributes`, requesting all attributes except nodes to be selected (`OCL_DS_NODES` is absent). It then disassembles the instruction and calls `walkAttrs` to show the attributes returned. Finally, it calls `ocliaFreeAttrs` to free the instruction attributes structure back to the heap. Function `walkAttrs` is implemented like this:

```
static void walkAttrs(octiaAttrP attrs) {

    printClass(attrs);
    printFetchRecords(attrs);

    printRegList(ocliaGetFirstReadReg(attrs), "read");
    printRangeList(ocliaGetFirstReadRange(attrs), "read");
    printRegList(ocliaGetFirstWrittenReg(attrs), "write");
    printRangeList(ocliaGetFirstWrittenRange(attrs), "write");

    printNextPCList(attrs);
    printAddressExpressionList(attrs);

    opPrintf("\n");
}
```

This function prints information of various types extracted from the attributes structure, as described in following sections.

20.4.1 Instruction Class

Each instruction can have a class associated with it, defined by the `octiaInstructionClass` enumeration in `ocl/ocliaTypes.h`:

```
typedef enum octiaInstructionClassE {
    OCL_IC_NONE         = 0x0,          ///< no class information
    OCL_IC_NOP          = 1ULL<<0,      ///< explicit NOP
    OCL_IC_INTEGER      = 1ULL<<1,      ///< instruction uses integer ALU
    OCL_IC_FLOAT        = 1ULL<<2,      ///< instruction uses FPU
    OCL_IC_DSP          = 1ULL<<3,      ///< instruction uses DSP
    OCL_IC_MULTIPLY     = 1ULL<<4,      ///< instruction implements multiply
    . . . lines omitted . . .
    OCL_IC_CUSTOM14     = 1ULL<<61,     ///< custom class 14
    OCL_IC_CUSTOM15     = 1ULL<<62,     ///< custom class 15
    OCL_IC_CUSTOM16     = 1ULL<<63      ///< custom class 16
} octiaInstructionClass;
```

The enumeration is a bitfield, containing up to 48 standard types and up to 16 custom types. Some instruction classes can be derived automatically (as we see below). Function `printClass` is called to print the class for a particular instruction:

```
#define CLASS_ENTRY(_NAME) case OCL_IC_##_NAME: opPrintf(#_NAME); break

static void printClass(octiaAttrP attrs) {

    octiaInstructionClass class;

    if((class=ocliaGetInstructionClass(attrs))) {

        octiaInstructionClass mask = 1;

        opPrintf("\n");
        opPrintf("  class ");

        do {

            if(class & mask) {

                switch(mask) {
                    CLASS_ENTRY(NOP       );
                    CLASS_ENTRY(INTEGER   );
                    CLASS_ENTRY(FLOAT     );
                    CLASS_ENTRY(DSP       );
                    CLASS_ENTRY(MULTIPLY  );
                    . . . lines omitted . . .
                    CLASS_ENTRY(CUSTOM14  );
                    CLASS_ENTRY(CUSTOM15  );
                    CLASS_ENTRY(CUSTOM16  );
                    default: opPrintf("*unknown*");
                }

                class &= ~mask;

                if(class) {
                    opPrintf("|");
                }
            }

            mask <<= 1;
```

```
        } while(class);

        opPrintf("\n");
    }
}
```

The function uses `ocliaGetInstructionClass` to obtain the class from the instruction attributes structure, and then uses a macro to implement a case statement converting each class member to a string.

20.4.2 Fetch Records

Function `printFetchRecords` prints the fetch records extracted from the instruction attributes structure:

```
static void printFetchRecords(octiaAttrP attrs) {

    octiaFetchRecordP fr;

    if((fr=ocliaGetFirstFetchRecord(attrs))) {

        opPrintf("\n");

        for(; fr; fr=ocliaGetNextFetchRecord(fr)) {

            Addr  fetchLow  = ocliaGetFetchRecordLow(fr);
            Addr  fetchHigh = ocliaGetFetchRecordHigh(fr);
            Uns8 *value     = ocliaGetFetchRecordBytes(fr);
            Uns32 bytes     = fetchHigh-fetchLow+1;
            Int32 i;

            opPrintf(
                "  fetch 0x"FMT_Ax":0x"FMT_Ax" (value:0x",
                fetchLow, fetchHigh
            );

            for(i=bytes-1; i>=0; i--) {
                opPrintf("%02x", value[i]);
            }

            opPrintf(")\n");
        }
    }
}
```

20.4.3 Read and Written Registers

Function `printRegList` prints a list of read or written registers:

```
static void printRegList(octiaRegListP regList, const char *type) {

    if(regList) {

        opPrintf("\n");

        for(; regList; regList=ocliaGetRegListNext(regList)) {
            optRegP reg = opRegConvert(ocliaGetRegListReg(regList));
            opPrintf("  %5s %s\n", type, opRegName(reg));
        }
    }
}
```

This is called twice from function `walkAttrs`, once to print read registers and once to print written registers:

```
static void walkAttrs(octiaAttrP attrs) {
    . . . lines omitted . . .
    printRegList(ocliaGetFirstReadReg(attrs), "read");
    . . . lines omitted . . .
    printRegList(ocliaGetFirstWrittenReg(attrs), "write");
    . . . lines omitted . . .
}
```

20.4.4 Unmatched Read and Written Ranges

Function `printRangeList` prints ranges within the processor structure that are read or written but do not match any register specification:

```
static void printRangeList(octiaRawRangeP range, const char *type) {

    if(range) {

        opPrintf("\n");

        for(; range; range=ocliaGetRangeNext(range)) {
            opPrintf(
                "  %5s "FMT_PTRd":"FMT_PTRd"\n",
                type,
                ocliaGetRangeLow(range),
                ocliaGetRangeHigh(range)
            );
        }
    }
}
```

This is also called twice from from function `walkAttrs`, once to print unmatched read registers and once to print unmatched written registers:

```
static void walkAttrs(octiaAttrP attrs) {
    . . . lines omitted . . .
    printRangeList(ocliaGetFirstReadRange(attrs), "read");
    . . . lines omitted . . .
    printRangeList(ocliaGetFirstWrittenRange(attrs), "write");
    . . . lines omitted . . .
}
```

20.4.5 Next PC Expressions

Function `printNextPCList` prints expressions calculating possible next PC addresses, together with supplementary information (any jump hint associated with the target address, and whether the jump has a delay slot instruction):

```
static void printNextPCList(octiaAttrP attrs) {

    octiaNextPCP nextPC;

    if((nextPC=ocliaGetFirstNextPC(attrs))) {

        opPrintf("\n");
```

```
        for(; nextPC; nextPC=ocliaGetNextNextPC(nextPC)) {

            octiaJumpHint hint      = ocliaGetNextPCHint(nextPC);
            const char   *hintText  = getHintText(ocliaGetNextPCHint(nextPC));
            octiaAddrExpP nextPCExp = ocliaGetNextPCAddrExp(nextPC);

            opPrintf("  next PC ");

            printAddrExp(nextPCExp, attrs);

            if(hintText) {
                opPrintf(" %s", hintText);
            }
            if((nextPCExp->type==OCL_ET_CONST) && (hint&OCL_JH_RELATIVE)) {
                opPrintf(" offset "FMT_Ad, ocliaGetNextPCOffset(nextPC));
            }
            if(ocliaGetNextPCDS(nextPC)) {
                opPrintf(" (DS)");
            }

            opPrintf("\n");
        }
    }
}
```

Function `printNextPCList` calls a helper function, `printAddrExp`, to evaluate and print the next PC expression value:

```
static void printAddrExp(octiaAddrExpP exp, octiaAttrP attrs) {

    if(exp->type==OCL_ET_UNKNOWN) {

        opPrintf("UNKNOWN");

    } else if(exp->type==OCL_ET_CONST) {

        opPrintf("0x"FMT_Ax, exp->c);

    } else if(exp->type==OCL_ET_REG) {

        optRegP reg = opRegConvert(exp->r);

        if(reg) {
            opPrintf("%s", opRegName(reg));
        } else {
            opPrintf("UNKNOWN_REGISTER");
        }

    } else if(exp->type==OCL_ET_EXTEND) {

        opPrintf(
            "(%cext-%u-to-%u ",
            exp->e.signExtend?'s':'z',
            exp->e.child->bits,
            exp->bits
        );
        printAddrExp(exp->e.child, 0);
        opPrintf(")");

    } else if(exp->type==OCL_ET_UNARY) {

        opPrintf("(%s ", exp->u.opName);
```

```
        printAddrExp(exp->u.child, 0);
        opPrintf(")");

    } else if(exp->type==OCL_ET_BINARY) {

        opPrintf("(%s ", exp->b.opName);
        printAddrExp(exp->b.child[0], 0);
        opPrintf(", ");
        printAddrExp(exp->b.child[1], 0);
        opPrintf(")");

    } else if(exp->type==OCL_ET_LOAD) {

        opPrintf("[");
        printAddrExp(exp->l.child, 0);
        opPrintf("]");

    } else {

        opPrintf("{unexpected expression type %u}", exp->type);
    }

    // print evaluated expression unless it is a constant
    if(attrs && (exp->type!=OCL_ET_CONST)) {

        Uns32 bytes = BITS_TO_BYTES(exp->bits);
        Uns8  result[bytes];
        Int32 i;

        // evaluate the expression
        ocliaEvaluate(attrs, exp, result);

        // print result
        opPrintf(" {current value:0x");
        for(i=bytes-1; i>=0; i--) {
            opPrintf("%02x", result[i]);
        }
        opPrintf("}");
    }
}
```

Each expression is a tree consisting of nodes that are either constants, registers, sign/zero extended expressions, unary expressions, binary expressions or memory reference expressions.

20.4.6 Address Expressions

Function `printAddressExpressionList` prints information about memory references, together with supplementary information about the type of access. It also prints all register values on which the memory reference expression depends:

```
static void printAddressExpressionList(octiaAttrP attrs) {

    octiaMemAccessP ma;

    // memory access type strings
    static const char *memAccessTypeString[] = {
        [OCL_MAT_LOAD]       = "load",
        [OCL_MAT_STORE]      = "store",
        [OCL_MAT_PRELOAD_LD] = "preload-for-load",
```

```
        [OCL_MAT_PRELOAD_ST] = "preload-for-store",
        [OCL_MAT_PRELOAD_EX] = "preload-for-fetch"
    };

    if((ma=ocliaGetFirstMemAccess(attrs))) {

        opPrintf("\n");

        for(; ma; ma=ocliaGetNextMemAccess(ma)) {

            octiaAddrExpP addrExp = ocliaGetMemAccessAddrExp(ma);
            octiaRegListP depend;

            // print characteristics of memory access
            opPrintf(
                "  %u-bit %s address (bits %u): ",
                ocliaGetMemAccessMemBits(ma),
                memAccessTypeString[ocliaGetMemAccessType(ma)],
                addrExp->bits
            );

            // print load/store address expression
            printAddrExp(addrExp, attrs);
            opPrintf("\n");

            // print all dependencies of this load/store (registers that must
            // be known before it can be evaluated)
            for(
                depend=ocliaGetMemAccessFirstDepend(ma);
                depend;
                depend=ocliaGetRegListNext(depend)
            ) {
                optRegP reg = opRegConvert(ocliaGetRegListReg(depend));
                opPrintf("    depend %s\n", opRegName(reg));
            }
        }
    }
}
```

20.5 Testing Baseline Instruction Attributes

Directory `18.or1kInstructionAttributes/application` contains the following example in file `asmtest.S`:

```
.org 0xc00

        ////////////////////////////////////////////////////////////////////////
        // SYS HANDLER (AT 0xc00)
        ////////////////////////////////////////////////////////////////////////

        l.rfe                       // return from exception

.org 0x10000
.global _start
_start:

        ////////////////////////////////////////////////////////////////////////
        // MOVHI INSTRUCTION TEST
        ////////////////////////////////////////////////////////////////////////

        l.movhi     r1,0x1234
```

```
        ////////////////////////////////////////////////////////////////////////
        // ARITHMETIC INSTRUCTION TESTS (SECOND ARGUMENT CONSTANT)
        ////////////////////////////////////////////////////////////////////////

        l.addi      r1,r2,1
        l.addic     r3,r4,2
        l.andi      r5,r6,1
        l.ori       r7,r8,1
        l.xori      r9,r10,1
        l.muli      r11,r12,1

        ////////////////////////////////////////////////////////////////////////
        // ARITHMETIC INSTRUCTION TESTS (SECOND ARGUMENT REGISTER)
        ////////////////////////////////////////////////////////////////////////

        l.add       r1,r2,r3
        l.addc      r3,r4,r5
        l.sub       r13,r14,r15
        l.and       r5,r6,r7
        l.or        r7,r8,r9
        l.xor       r9,r10,r11
        l.mul       r11,r12,r13
        l.addi      r1,r0,5
        l.addi      r2,r0,-7
        l.div       r3,r2,r1
        l.divu      r3,r2,r1

        ////////////////////////////////////////////////////////////////////////
        // SHIFT/ROTATE INSTRUCTION TESTS (SECOND ARGUMENT CONSTANT)
        ////////////////////////////////////////////////////////////////////////

        l.slli      r1,r2,1
        l.srli      r3,r4,2
        l.srai      r5,r6,3
        l.rori      r7,r8,4

        ////////////////////////////////////////////////////////////////////////
        // SHIFT/ROTATE INSTRUCTION TESTS (SECOND ARGUMENT REGISTER)
        ////////////////////////////////////////////////////////////////////////

        l.sll       r1,r2,r3
        l.srl       r3,r4,r5
        l.sra       r5,r6,r7
        l.ror       r7,r8,r7

        ////////////////////////////////////////////////////////////////////////
        // BRANCH INSTRUCTION TESTS
        ////////////////////////////////////////////////////////////////////////

        l.sfeqi     r1,0            // r1==0?
        l.bnf       1f              // go if condition false
        l.addi      r1,r1,0x12      // (delay slot)
        l.nop                       // (not executed)
1:      l.bf        1f              // go if condition true
        l.addi      r1,r1,0x12      // (delay slot)
        l.nop
1:      l.j         1f
        l.addi      r1,r1,0x12      // (delay slot)
        l.nop                       // (not executed)
1:      l.jal       1f
        l.addi      r1,r1,0x12      // (delay slot)
        l.nop                       // (not executed)
1:      l.movhi     r1,   hi(fwd1)  // r1 = fwd1 (high)
```

```
        l.ori       r1,r1,lo(fwd1)  // r1 = fwd1
        l.jr        r1
        l.addi      r1,r1,0x12      // (delay slot)
        l.nop                       // (not executed)
fwd1:   l.movhi     r1,   hi(fwd2)  // r1 = fwd2 (high)
        l.ori       r1,r1,lo(fwd2)  // r1 = fwd2
        l.jalr      r1
        l.addi      r1,r1,0x12      // (delay slot)
        l.nop                       // (not executed)
fwd2:

        ////////////////////////////////////////////////////////////////////////
        // SYS AND RFE INSTRUCTION TESTS
        ////////////////////////////////////////////////////////////////////////

        l.sys       0x1234

        ////////////////////////////////////////////////////////////////////////
        // SYSTEM REGISTER INSTRUCTION TESTS
        ////////////////////////////////////////////////////////////////////////

        l.mfspr     r2,r0,0x20      // get epc in r2
        l.addi      r2,r0,-1        // r2 = -1
        l.mtspr     r0,r2,0x20      // set epc from r2

        ////////////////////////////////////////////////////////////////////////
        // LOAD/STORE INSTRUCTION TESTS
        ////////////////////////////////////////////////////////////////////////

        l.movhi     r1,   hi(data)  // r1 = data (high)
        l.ori       r1,r1,lo(data)  // r1 = data
        l.lwz       r3,4(r1)
        l.lhz       r3,6(r1)
        l.lbz       r3,7(r1)
        l.lws       r3,4(r1)
        l.lhs       r3,6(r1)
        l.lbs       r3,7(r1)
        l.sw        0(r1),r3
        l.sh        2(r1),r3
        l.sb        3(r1),r3

        ////////////////////////////////////////////////////////////////////////
        // COMPARISON INSTRUCTION TESTS
        ////////////////////////////////////////////////////////////////////////

        l.sfeqi     r1,0
        l.sfeq      r1,r2

        ////////////////////////////////////////////////////////////////////////
        // NOP INSTRUCTION TEST
        ////////////////////////////////////////////////////////////////////////

        l.nop

.global exit
exit:
        l.nop

data:
        .word 0x12345678
        .word 0x89abcdef
```

This example begins execution at `_start` in supervisor mode. It then executes an instruction of every implemented type.

Run the platform using the assembler executable file:

```
platform/harness.$IMPERAS_ARCH.exe --program application\asmtest.OR1K.elf
```

The output from this should be as follows:

```
*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x10000 (l.movhi  r1,0x1234)

  fetch 0x10000:0x10003 (value:0x34122018)

  write R1

  next PC 0x10004 hint: (relative) offset 4

*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x10004 (l.addi   r1,r2,0x1)

  fetch 0x10004:0x10007 (value:0x0100229c)

   read R2

  write R1

  write 0:1

  next PC 0x10008 hint: (relative) offset 4

*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x10008 (l.addic  r3,r4,0x2)

  fetch 0x10008:0x1000b (value:0x020064a0)

   read R4

   read 0:0

  write R3

  write 0:1

  next PC 0x1000c hint: (relative) offset 4

. . . lines omitted . . .

*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x10100 (l.sfeqi  r1,0x0)

  fetch 0x10100:0x10103 (value:0x000001bc)

   read R1

  write 2:2

  next PC 0x10104 hint: (relative) offset 4

*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x10104 (l.sfeq   r1,r2)

  fetch 0x10104:0x10107 (value:0x001001e4)

   read R1
```

```
   read R2
  write 2:2
  next PC 0x10108 hint: (relative) offset 4
*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x10108 (l.nop    0x0)
  fetch 0x10108:0x1010b (value:0x00000015)
  next PC 0x1010c hint: (relative) offset 4
*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x1010c (l.nop    0x0)
  fetch 0x1010c:0x1010f (value:0x00000015)
  next PC 0x10110 hint: (relative) offset 4
Processor 'platform/cpu1' terminated at 'exit', address 0x1010c
```

Much useful information has been generated automatically. For example, consider the add instruction at address `0x10004`:

```
*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x10004 (l.addi   r1,r2,0x1)
  fetch 0x10004:0x10007 (value:0x0100229c)
   read R2
  write R1
  write 0:1
  next PC 0x10008 hint: (relative) offset 4
```

Here, the instruction fetch has been correctly identified, and the source register (`R2`) and destination register (`R1`) also identified automatically. The address of the next instruction (`0x10008`) is also correct. However, two bytes of the processor structure are being written that do not have a register correspondence that can be automatically identified:

```
  write 0:1
```

The written range `0:1` is in bytes from the start of the processor structure:

```
typedef struct or1kS {

    Bool          carryFlag;        // carry flag
    Bool          overflowFlag;     // overflow flag
    Bool          branchFlag;       // branch flag

    Uns32         regs[OR1K_REGS];  // basic registers

    . . . lines omitted

} or1k, *or1kP;
```

By inspection, byte 0 is the *carry flag* and byte 1 is the *overflow flag*. Writes to these registers cannot be automatically identified because there is no register defined in `or1kRegisters.c` that specifies them using the `raw` field of a `vmiRegInfo` structure. Instead, these flags are included in the *composed value* of the `SR` register (by function `fillSR` in `or1kUtils.c`). What we would therefore like to do is to enhance the model to indicate that the `carryFlag` and `overflowFlag` fields are part of the `SR` register, and that any reads or writes of them should be reported as an access to that register. This is exactly what the currently-compiled-out lines in function `or1kRegImpl` do:

```
#define OR1K_REG_IMPL_RAW(_REG, _FIELD, _BITS) \
    vmirtRegImplRaw(processor, _REG, _FIELD, _BITS)

VMI_REG_IMPL_FN(or1kRegImpl) {
    // specify that flag registers are in SR
    vmiRegInfoCP CPSR = vmirtGetRegByName(processor, "SR");
    OR1K_REG_IMPL_RAW(CPSR, OR1K_CARRY,    8);
    OR1K_REG_IMPL_RAW(CPSR, OR1K_OVERFLOW, 8);
    OR1K_REG_IMPL_RAW(CPSR, OR1K_BRANCH,   8);
}
```

Function `vmirtRegImplRaw` has this prototype:

```
void vmirtRegImplRaw(
    vmiProcessorP processor,
    vmiRegInfoCP  regDesc,
    vmiReg        r,
    Uns32         bits
);
```

It indicates that an unmatched access to a processor field identified by register `r` (of size `bits`) should be reported as an access to the given `regDesc` register. Moving further down the log file, we come to the `l.sfeqi` instruction at address `0x10068`:

```
*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x10068 (l.sfeqi  r1,0x0)

  fetch 0x10068:0x1006b (value:0x000001bc)

   read R1

  write 2:2

  next PC 0x1006c hint: (relative) offset 4
```

The written range `2:2` corresponds to the `branchFlag` field in the processor structure. This is also part of the composed value of the `SR` register, so identified in `or1kRegImpl` in the same way as the other two flags above. The flag value is then used in a subsequent branch instruction:

```
*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x1006c (l.bnf    0x00010078)

  class BRANCH_DS

  fetch 0x1006c:0x1006f (value:0x0300000c)
```

```
    read 2:2

  next PC 0x10078 (DS)
  next PC 0x10070

*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x10070 (l.addi   r1,r1,0x12)

  fetch 0x10070:0x10073 (value:0x1200219c)

   read R1

  write R1

  write 0:1

  next PC 0x10074 hint: (relative) offset 4
```

Note that the branch instruction is indicated as having *two* next instructions: address `0x10070` (which is the delay slot instruction) and address `0x10078` (which is the instruction address to execute *after the delay slot instruction completes*). Note that the delay slot instruction itself is indicated as having a next instruction address of `0x10074`, even though that address is never executed (control transfers to `0x10078` instead). This is because *instruction attributes are generated from instruction patterns only* and are independent of delay slot state. The same idiom is apparent for unconditional `l.j` instructions:

```
*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x10084 (l.j      0x00010090)

  class BRANCH_DS

  fetch 0x10084:0x10087 (value:0x03000000)

  next PC 0x10090 (DS)
  next PC 0x10088

*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x10088 (l.addi   r1,r1,0x12)

  fetch 0x10088:0x1008b (value:0x1200219c)

   read R1

  write R1

  write 0:1

  next PC 0x1008c hint: (relative) offset 4
```

Note that the `l.j` instruction has *two* next instruction expressions, even though the branch is in fact unconditional. This is because the jump has a delay slot instruction, and therefore both the next instruction address and the post-delay-slot address need to identified (for the reason stated above, the delay slot instruction does not report a post-delay-slot address as its next instruction). Unconditional jumps without delay slots report only a single next instruction address (see `l.rfe` below).

Direct jump-and-link instructions (`l.jal`) indicate that the link register (`R9`) is a destination:

```
*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x10090 (l.jal    0x0001009c)

  class BRANCH_DS

  fetch 0x10090:0x10093 (value:0x03000004)

  write R9

  next PC 0x1009c hint:call (DS)
  next PC 0x10094

*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x10094 (l.addi   r1,r1,0x12)

  fetch 0x10094:0x10097 (value:0x1200219c)

   read R1

  write R1

  write 0:1

  next PC 0x10098 hint: (relative) offset 4
```

Indirect jumps (e.g. `l.jr`) indicate that the jump target register is read. One next instruction expression is now a register instead of a constant:

```
*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x100a4 (l.jr     r1)

  class BRANCH_DS

  fetch 0x100a4:0x100a7 (value:0x00080044)

   read R1

  next PC R1 {current value:0x000100b0} (DS)
  next PC 0x100a8

*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x100a8 (l.addi   r1,r1,0x12)

  fetch 0x100a8:0x100ab (value:0x1200219c)

   read R1

  write R1

  write 0:1

  next PC 0x100ac hint: (relative) offset 4
```

In file `platform/harness.c`, function `printAddrExp` uses OCL function `ocliaEvaluate` to calculate the current value of any non-constant expression so that the value can be displayed in the trace:

```
static void printAddrExp(octiaAddrExpP exp, octiaAttrP attrs) {

    . . . lines omitted . . .

    // print evaluated expression unless it is a constant
```

```
    if(attrs && (exp->type!=OCL_ET_CONST)) {

        Uns32 bytes = BITS_TO_BYTES(exp->bits);
        Uns8  result[bytes];
        Int32 i;

        // evaluate the expression
        ocliaEvaluate(attrs, exp, result);

        // print result
        opPrintf(" {current value:0x");
        for(i=bytes-1; i>=0; i--) {
            opPrintf("%02x", result[i]);
        }
        opPrintf("}");
    }
}
```

The `l.sys` and `l.rfe` instructions produce the following trace sequence:

```
*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x100c4 (l.sys)

  fetch 0x100c4:0x100c7 (value:0x34120020)

  next PC 0x100c8 hint: (relative) offset 4

*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0xc00 (l.rfe)

  class BRANCH

  fetch 0xc00:0xc03 (value:0x00000024)

   read ESR
   read EPCR

  next PC EPCR {current value:0x000100c8} hint:returnint
```

The `l.rfe` instruction has a single next instruction address expression, obtained from the current value of the `EPCR` register. The `l.sys` instruction is also indicated to have a single next instruction, but this is incorrect (it is reported as `0x100c8`, but should be the system vector address `0xc00`). The problem here is the way that function `morphSys` was originally defined:

```
static OR1K_MORPH_FN(morphSYS) {

    vmimtArgProcessor();
    vmimtArgUns32(OR1K_EXCPT_SYS);
    vmimtArgUns32(4);
    vmimtCallAttrs((vmiCallFn)or1kTakeException, VMCA_EXCEPTION);
}
```

Function `or1kTakeException` uses `vmirtSetPCException` to modify the program counter, but this cannot be known by scanning the JIT data structures. The problem can be corrected by inserting an explicit jump to the exception vector as follows:

```
static OR1K_MORPH_FN(morphSYS) {

    vmimtArgProcessor();
```

```
    vmimtArgUns32(OR1K_EXCPT_SYS);
    vmimtArgUns32(4);
    vmimtCallAttrs((vmiCallFn)or1kTakeException, VMCA_EXCEPTION);

    // correct next instruction address in instruction attributes
    vmimtUncondJump(0, SYS_ADDRESS, VMI_NOREG, vmi_JH_CALLINT);
}
```

The new unconditional jump is never in fact executed: its only purpose is to indicate a correct next instruction address.

Towards the end of the trace file, instruction attributes are generated for a series of load and store instructions. For example:

```
*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x100ec (l.lhs    r3,0x6(r1))

  fetch 0x100ec:0x100ef (value:0x06006198)

   read R1

  write R3

  next PC 0x100f0 hint: (relative) offset 4

  16-bit load address (bits 32): (ADD R1, 0x6) {current value:0x00010116}
    depend R1
```

Each load and store indicates the size of the loaded data (16 bits in this case) and the address size (32 bits). There is also an expression indicating the address (`ADD R1, 0x6`) and the current value of that address expression. Finally, there is a list of registers on which the address expression depends (`R1` in this case).

20.6 Adding Instruction Classes

Some instruction types in the default trace have instruction class information automatically added. For example, the `l.jr` instruction has class `BRANCH_DS`, indicating that it is a branch with a delay slot:

```
*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x100a4 (l.jr     r1)

  class BRANCH_DS

  fetch 0x100a4:0x100a7 (value:0x00080044)

   read R1

  next PC R1 {current value:0x000100b0} (DS)
  next PC 0x100a8
```

Class information can provide useful information to client tools about the functional units required to execute an instruction. However, many instructions do not have class information generated for them by default. When required, class information may be added explicitly using function `vmimtInstructionClassAdd`, defined as follows:

```
void vmimtInstructionClassAdd(octiaInstructionClass value);
```

This function causes any instruction class specified using the argument to be added (by bitwise-or) to the indicated class of the instruction currently being translated. To implement explicit instruction classes, the `or1kInstructionInfo` structure in file `or1kDecode.h` has been updated as follows:

```
typedef struct or1kInstructionInfoS {
    const char           *opcode;       // opcode name
    const char           *format;       // disassembly format string
    or1kInstructionType   type;         // instruction type
    octiaInstructionClass class;        // instruction class
    Uns32                 thisPC;       // instruction address
    Uns32                 instruction;  // raw instruction
    Uns32                 r1;           // register 1
    Uns32                 r2;           // register 2
    Uns32                 r3;           // register 3
    Uns32                 c;            // constant value
    Uns32                 bytes;        // load/store bytes (1, 2 or 4)
    or1kExtend            extend;       // load sign/zero extension
    or1kCondition         cond;         // condition
    Bool                  iSuffix;      // does opcode require 'i' suffix?
} or1kInstructionInfo, *or1kInstructionInfoP;
```

The new class member is used in `or1kMorphInstruction`:

```
VMI_MORPH_FN(or1kMorphInstruction) {

    or1kP          or1k  = (or1kP)processor;
    or1kMorphState state = {{0}};

    // decode instruction
    or1kDecode(or1k, thisPC, &state.info);

#if(ENABLE_ATTRS)
    // add any explicit instruction class
    vmimtInstructionClassAdd(state.info.class);
#endif

    . . . lines omitted . . .
}
```

In file `or1kDecode.c`, the `opAttrs` structure now also has a new `class` member:

```
typedef struct opAttrsS {
    const char           *member;       // enumeration member name
    const char           *opcode;       // opcode name
    const char           *format;       // format string
    const char           *decode;       // decode string
    octiaInstructionClass class;        // instruction class
    regSpec               r1      : 8;  // register 1 specification
    regSpec               r2      : 8;  // register 2 specification
    regSpec               r3      : 8;  // register 3 specification
    constSpec             c       : 8;  // constant value specification
    Uns32                 bytes   : 8;  // load/store size specification
    or1kExtend            extend  : 8;  // load/store sign extension
    Bool                  isCond  : 1;  // is instruction conditional?
    Bool                  iSuffix : 1;  // does opcode require 'i' suffix?
} opAttrs, *opAttrsP;
```

The `class` member is set up by the attribute initialization macros. For example, system instructions (`l.sys` and `l.rfe`) are now explicitly identified:

```
#define ATTR_SET_SYS(_NAME, _OPCODE, _DECODE) \
    [OR1K_IT_##_NAME] = {           \
        member : #_NAME,            \
        opcode : _OPCODE,           \
        format : FMT_NONE,          \
        class  : OCL_IC_SYSTEM,     \
        decode : _DECODE            \
    }
```

Finally, function `or1kDecode` has been modified to assign the new class field in the `or1kInstructionInfo` structure using the `opAttrs` structure:

```
void or1kDecode(or1kP or1k, Uns32 thisPC, or1kInstructionInfoP info) {

    . . . lines omitted . . .

    // fill structure fields
    info->opcode      = attrs->opcode;
    info->format      = attrs->format;
    info->type        = type;
    info->class       = attrs->class;

    . . . lines omitted . . .
}
```

20.7 Testing Enhanced Instruction Attributes

The previous section identified various changes that could be made to the model to improve the instruction attributes output. The example in fact contains all the suggested changes: to enable them, we need to recompile the processor model with `ENABLE_ATTRS` defined:

```
make clean
make ENABLE_ATTRS=1
```

And then rerun the example:

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

Note that many more instructions now have class information, and that processor flags are now correctly reported as members of the `SR` register when they are used as sources and destinations:

```
*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x10008 (l.addic  r3,r4,0x2)

  class INTEGER

  fetch 0x10008:0x1000b (value:0x020064a0)

   read SR
   read R4
```

```
  write SR
  write R3

  next PC 0x1000c hint: (relative) offset 4
```

The next instruction address for the `l.sys` instruction is now also correct:

```
*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x100c4 (l.sys)

  class SYSTEM|BRANCH

  fetch 0x100c4:0x100c7 (value:0x34120020)

  next PC 0xc00 hint:callint
```

21 Implementing Fixed-Mapped Virtual Memory

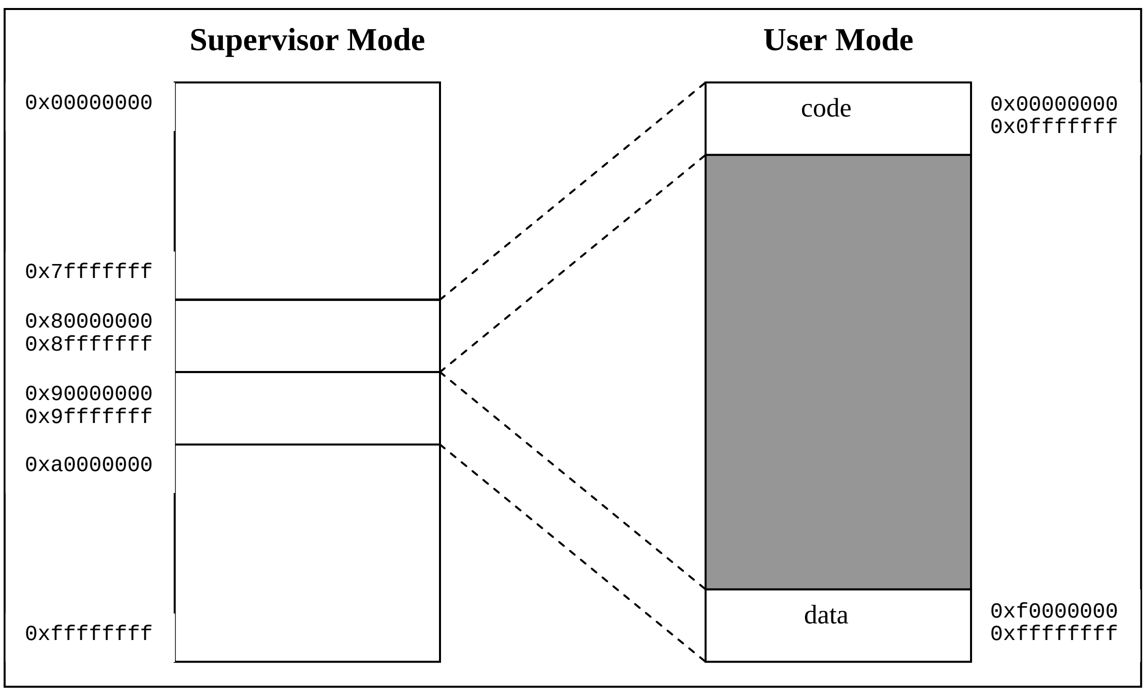
The OR1K example described so far in this document has only physical memory. It is also possible to implement *virtual memory* in a highly-efficient manner.

This chapter shows how to implement *fixed-mapped* virtual memory, where the virtual-to-physical address mappings are constant. Chapter 22 extends this to show how to implement *dynamic-mapped* virtual memory, where the virtual-to-physical address mappings can be changed at run time.

21.1 Example Memory Maps

The OR1K processor can be configured with a full TLB-based virtual memory system. Although it is perfectly possible to implement the full OR1K virtual memory algorithm using the VMI interface, the code to do this is too complex for an introductory example. Therefore, this chapter will implement a much simpler memory mapping scheme that demonstrates the required concepts without the complexity of the true virtual memory algorithm.

The fixed mapping scheme we will implement will look like this:



In other words, the example memory mapping scheme will be as follows:

1. In *supervisor* mode, the entire address space is mapped with full access permissions (read, write and execute)
2. In *user* mode, there are three distinct memory areas:
    a. The address range `0x00000000:0x0fffffff` is *code* memory which has *execute and read* permission, but is *not writable in user mode* (any attempt

to write to this address range should generate an exception). It is mapped to the physical address range `0x80000000:0x8fffffff` (so, for example, address `0x80000000` in supervisor mode addresses the same location as address `0x00000000` in user mode).

b. The address range `0xf0000000:0xffffffff` is *data* memory which has *read and write* permission, but is *not executable in user mode* (any attempt to execute code in this address range should generate an exception). It is mapped to the physical address range `0x90000000:0x9fffffff` (so, for example, address `0x90000000` in supervisor mode addresses the same location as address `0xf0000000` in user mode).

c. The address range `0x10000000:0xefffffff` is unmapped in user mode; any attempt to access this memory in any way should generate an exception.

21.2 The Template Fixed-Mapped Model

A template model for the OR1K processor implementing a fixed-mapped virtual memory scheme can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/19.or1kBehaviorVM
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/19.or1kBehaviorVM .
```

Compile the model, harness and application using the `make` command:

```
cd 19.or1kBehaviorVM
make OPT=1
```

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is based on the previous model, with the changes listed in following sections.

21.3 or1kVM.c

The fixed-mapped virtual memory scheme described in the previous section is created by defining a *virtual memory constructor* function for the OR1K processor. This is implemented as follows in file `or1kVM.c`:

```
VMI_VMINIT_FN(or1kVMInit) {

    // for this example, the SUPERVISOR memory domain will be the physical
    // domain; the USER memory domain will be a derived virtual domain
    memDomainP physicalDomain   = codeDomains[0];
```

```
    memDomainP supervisorDomain = physicalDomain;
    memDomainP userDomain       = vmirtNewDomain("user", 32);

    // create an initial mapping that makes physical addresses 0x80000000:
    // 0x8fffffff visible at 0x00000000:0x0fffffff in the USER memory space
    vmirtAliasMemory(
        physicalDomain,
        userDomain,
        0x80000000,
        0x8fffffff,
        0x00000000,
        0
    );

    // remove write permissions from addresses 0x00000000:0x0fffffff in the
    // user address space
    vmirtProtectMemory(
        userDomain,
        0x00000000,
        0x0fffffff,
        MEM_PRIV_RX,
        MEM_PRIV_SET
    );

    // create an initial mapping that makes physical addresses 0x90000000:
    // 0x9fffffff visible at 0xf0000000:0xffffffff in the USER memory space
    vmirtAliasMemory(
        physicalDomain,
        userDomain,
        0x90000000,
        0x9fffffff,
        0xf0000000,
        0
    );

    // remove execute permissions from addresses 0xf0000000:0xffffffff in the
    // USER memory space
    vmirtProtectMemory(
        userDomain,
        0xf0000000,
        0xffffffff,
        MEM_PRIV_RW,
        MEM_PRIV_SET
    );

    // supervisorDomain should be used for both instruction and data in
    // SUPERVISOR mode
    codeDomains[OR1K_MODE_SUPERVISOR] = supervisorDomain;
    dataDomains[OR1K_MODE_SUPERVISOR] = supervisorDomain;

    // userDomain should be used for both instruction and data in USER mode
    codeDomains[OR1K_MODE_USER] = userDomain;
    dataDomains[OR1K_MODE_USER] = userDomain;
}
```

The virtual memory constructor is defined using the `VMI_VMINIT_FN` macro, defined as follows in `vmiAttrs.h`:

```
#define VMI_VMINIT_FN(_NAME) void _NAME( \
    vmiProcessorP processor,            \
    memDomainPP   codeDomains,          \
    memDomainPP   dataDomains           \
```

```
)
```

The virtual memory constructor is called by the simulator after the processor constructor has been called. It is passed three arguments:

1. the newly-constructed processor;
2. an array of memory domain objects for instruction fetch (the *code* domains);
3. an array of memory domain objects for data access (the *data* domains).

The number of entries in each of the code and data domain arrays is the same as the number of *code dictionaries* that the processor has. The number of code dictionaries is determined by the length of the dictionary names array in file `or1kAttrs.c`:

```
static const char *dictNames[] = {"SUPERVISOR", "USER", 0};
```

For the OR1K, each of the code and data domain arrays will therefore contain two entries. Each of the entries is used as follows:

1. entry 0 of `codeDomains` is used when fetching instructions in *supervisor* mode;
2. entry 1 of `codeDomains` is used when fetching instructions in *user* mode;
3. entry 0 of `dataDomains` is used for loads and stores in *supervisor* mode;
4. entry 1 of `dataDomains` is used for loads and stores in *user* mode;

Each entry in `codeDomains` is seeded with a physical domain associated with the bus object that was connected to the `INSTRUCTION` bus port in the harness. Similarly, each entry in `dataDomains` is seeded with a physical domain associated with the bus object that was connected to the `DATA` bus port in the harness. However, the default entries in the `codeDomains` and `dataDomains` arrays can be overridden with new memory domain objects in the virtual memory constructor to specify different mappings for each processor mode, as described in detail below.

```
    memDomainP physicalDomain   = codeDomains[0];
```

This line gets the physical domain automatically associated with the OR1K processor when it was created.

```
    memDomainP supervisorDomain = physicalDomain;
```

In supervisor mode, we want to use the physical domain for all accesses.

```
    memDomainP userDomain       = vmirtNewDomain("user", 32);
```

In user mode, we do not want to use the physical domain, but instead we will use a new domain, with a 32-bit address width, created by calling `vmirtNewDomain`.

```
    vmirtAliasMemory(
        physicalDomain,
        userDomain,
        0x80000000,
        0x8fffffff,
        0x00000000,
```

```
        0
    );
```

This call to `vmirtAliasMemory` maps addresses `0x80000000:0x8fffffff` in `physicalDomain` to addresses `0x80000000:0x8fffffff` in `userDomain`. See the *VMI Run Time Function Reference* documentation for more information about `vmirtAliasMemory`.

```
    vmirtProtectMemory(
        userDomain,
        0x00000000,
        0x0fffffff,
        MEM_PRIV_RX,
        MEM_PRIV_SET
    );
```

This call to `vmirtProtectMemory` sets the access permissions on the address range `0x80000000:0x8fffffff` in `userDomain` to execute and read (but not write).

```
    vmirtAliasMemory(
        physicalDomain,
        userDomain,
        0x90000000,
        0x9fffffff,
        0xf0000000,
        0
    );

    vmirtProtectMemory(
        userDomain,
        0xf0000000,
        0xffffffff,
        MEM_PRIV_RW,
        MEM_PRIV_SET
    );
```

These two calls map addresses `0x90000000:0x9fffffff` in `physicalDomain` to addresses `0xf0000000:0xffffffff` in `userDomain`, and give that address range read and write (but not execute) permission.

```
    codeDomains[OR1K_MODE_SUPERVISOR] = supervisorDomain;
    dataDomains[OR1K_MODE_SUPERVISOR] = supervisorDomain;

    codeDomains[OR1K_MODE_USER] = userDomain;
    dataDomains[OR1K_MODE_USER] = userDomain;
```

These lines override the default domains to use for the various access types and processor modes, so that `supervisorDomain` is used for all supervisor mode accesses and `userDomain` is used for all user mode accesses.

21.4 VM Function Registration - `or1kAttrs.c`

A prototype for `or1kVMInit` has been added to `or1kFunctions.h`, and is referenced in the attribute structure in `or1kAttrs.c`:

```
const vmiIASAttr modelAttrs = {

    . . . lines omitted . . .

    ////////////////////////////////////////////////////////////////////////
    // CREATE/DELETE ROUTINES
    ////////////////////////////////////////////////////////////////////////

    .constructorCB = or1kConstructor,
    .vmInitCB      = or1kVMInit,
    .destructorCB  = or1kDestructor,

    . . . lines omitted . . .
```

21.4.1 Virtual Memory Test Harness - `platform/harness.c`

The test platform for this example, `platform/harness.c`, has been changed to enable simulated exceptions in the OR1K processor:

```
    const char *modelFile = "model."IMPERAS_SHRSUF;
    optProcessorP processor = opProcessorNew(
        mr, modelFile, "cpu1", 0,
        OP_PARAMS (
            OP_PARAM_BOOL_SET(OP_FP_SIMULATEEXCEPTIONS, 1)
        )
    );
```

Simulated exceptions are required because we want the processor to run exception handlers on illegal read, write or instruction fetch – if this is not enabled, then simulation would terminate on an illegal access.

21.5 Testing Fixed-Mapped Virtual Memory

Run the platform using the assembler executable file:

```
platform/harness.$IMPERAS_ARCH.exe –-trace --traceshowicount \
    --program application/asmtest.OR1K.elf
```

The output from this should be as follows:

```
Info 1: 'cpu1', 0x0000000000000000: l.ori    r31,r0,0x0
Info 2: 'cpu1', 0x0000000000000004: l.movhi  r1,0x8000
Info 3: 'cpu1', 0x0000000000000008: l.movhi  r2,0x0
Info 4: 'cpu1', 0x000000000000000c: l.ori    r2,r2,0xc0c
Info 5: 'cpu1', 0x0000000000000010: l.movhi  r3,0x0
Info 6: 'cpu1', 0x0000000000000014: l.ori    r3,r3,0xc2c
Info 7: 'cpu1', 0x0000000000000018: l.lwz    r4,0x0(r2)
Info 8: 'cpu1', 0x000000000000001c: l.sw     0x0(r1),r4
Info 9: 'cpu1', 0x0000000000000020: l.addi   r2,r2,0x4
Info 10: 'cpu1', 0x0000000000000024: l.addi   r1,r1,0x4
Info 11: 'cpu1', 0x0000000000000028: l.sfne   r2,r3
Info 12: 'cpu1', 0x000000000000002c: l.bf     0x00000018
. . . lines omitted . . .
Info 61: 'cpu1', 0x000000000000002c: l.bf     0x00000018
Info 62: 'cpu1', 0x0000000000000030: l.nop    0x0
Info 63: 'cpu1', 0x0000000000000034: l.mtspr  r0,r0,64
Info 64: 'cpu1', 0x0000000000000038: l.mtspr  r0,r0,32
```

```
Info 65: 'cpu1', 0x000000000000003c: l.rfe
Info 66: 'cpu1', 0x0000000000000000: l.movhi  r1,0xf000
Info 67: 'cpu1', 0x0000000000000004: l.ori    r2,r0,0x1234
Info 68: 'cpu1', 0x0000000000000008: l.sw     0x0(r1),r2
Info 69: 'cpu1', 0x000000000000000c: l.sw     0x0(r0),r2
Info 70: 'cpu1', 0x0000000000000300: l.mfspr  r1,r0,32
Info 71: 'cpu1', 0x0000000000000304: l.addi   r1,r1,0x4
Info 72: 'cpu1', 0x0000000000000308: l.mtspr  r0,r1,32
Info 73: 'cpu1', 0x000000000000030c: l.rfe
Info 74: 'cpu1', 0x0000000000000010: l.movhi  r1,0xf000
Info 75: 'cpu1', 0x0000000000000014: l.jalr   r1
Info 76: 'cpu1', 0x0000000000000018: l.nop    0x0
Info 77: 'cpu1', 0x00000000f0000000: *** FETCH EXCEPTION ***
Info 78: 'cpu1', 0x0000000000000400: l.mtspr  r0,r9,32
Info 79: 'cpu1', 0x0000000000000404: l.rfe
Info 80: 'cpu1', 0x000000000000001c: l.sys
Info 81: 'cpu1', 0x0000000000000c00: l.movhi  r1,0x9000
Info 82: 'cpu1', 0x0000000000000c04: l.lwz    r3,0x0(r1)
Info 83: 'cpu1', 0x0000000000000c08: l.nop    0x0
Processor 'cpu1' terminated at 'exit', address 0xc08
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 90000000   R2 : 00001234   R3 : 00001234
 R4 : 20000000   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : 0000001c   R10: deadbeef   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: deadbeef   R31: 00000000
 PC : 00000c0c   SR : 00008001   ESR: 00008000   EPC: 00000020
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:0 CF:0 OF:0
--------------- --------------- --------------- ---------------

processor has executed 83 instructions
```

The source code for this example is as follows:

```
        ////////////////////////////////////////////////////////////////////////
        // KERNEL START CODE (AT 0x0)
        ////////////////////////////////////////////////////////////////////////
.global _start
_start:
        l.ori       r31,r0,0        // r31 = 0 (stack pointer)

        // prepare to copy application to user space
        l.movhi     r1,0x8000       // r1 = 0x80000000
        l.movhi     r2,hi(appStart) // r2 = appStart
        l.ori       r2,r2,lo(appStart)
        l.movhi     r3,hi(appEnd)   // r3 = appEnd
        l.ori       r3,r3,lo(appEnd)

        // copy application to user space
loop:   l.lwz       r4,0(r2)        // r4 = word of application code
        l.sw        0(r1),r4        // copy to 0x80000000
        l.addi      r2,r2,4         // increment src pointer
        l.addi      r1,r1,4         // increment dst pointer
        l.sfne      r2,r3           // r2!=r3?
        l.bf        loop            // go if true
        l.nop                       // (delay slot instruction)

        // run user code
```

```
        l.mtspr     r0,r0,0x40      // clear esr
        l.mtspr     r0,r0,0x20      // clear epc (resume address in user space)
        l.rfe                       // return from exception (runs user code)

.org 0x300
        ////////////////////////////////////////////////////////////////////////
        // DATA PRIVILEGE EXCEPTION VECTOR (AT 0x300) - SKIP INSTRUCTION
        ////////////////////////////////////////////////////////////////////////
        l.mfspr     r1,r0,0x20      // get epc in r1
        l.addi      r1,r1,4         // increment address to skip faulting insn
        l.mtspr     r0,r1,0x20      // set epc
        l.rfe                       // return from exception

.org 0x400
        ////////////////////////////////////////////////////////////////////////
        // CODE PRIVILEGE EXCEPTION VECTOR (AT 0x400) - RESUME AT LINK ADDRESS
        ////////////////////////////////////////////////////////////////////////
        l.mtspr     r0,r9,0x20      // set epc from link register (r9)
        l.rfe                       // return from exception

.org 0xc00
        ////////////////////////////////////////////////////////////////////////
        // SYSCALL VECTOR (AT 0xc00) - TERMINATE PROGRAM
        ////////////////////////////////////////////////////////////////////////
        l.movhi     r1,0x9000       // r1 = 0x90000000
        l.lwz       r3,0(r1)        // load r3 from address 0x90000000
.global exit
exit:   l.nop

        ////////////////////////////////////////////////////////////////////////
        // USER APPLICATION (IN KERNEL MEMORY)
        ////////////////////////////////////////////////////////////////////////
appStart:
        l.movhi     r1,0xf000       // r1 = 0xf0000000
        l.ori       r2,r0,0x1234    // r2 = 0x00001234
        l.sw        0(r1),r2        // legal store to 0xf0000000
        l.sw        0(r0),r2        // attempt ILLEGAL store to 0x00000000
        l.movhi     r1,0xf000       // r1 = 0xf0000000
        l.jalr      r1              // attempt ILLEGAL jump to 0xf0000000
        l.nop                       // (delay slot instruction)
        l.sys       0               // exit program
appEnd:
```

Execution starts at label `_start`. The application (running in supervisor mode) first executes a loop to copy the code between labels `appStart` and `appEnd` to address `0x80000000` (so this code will become visible in the user address space at address `0x00000000`). When this is done (at instruction 63), the processor clears `esr` and `epc` and executes an `l.rfe` instruction (return form exception) to start executing at address `0x00000000` in *user* mode:

```
        l.mtspr     r0,r0,0x40      // clear esr
        l.mtspr     r0,r0,0x20      // clear epc (resume address in user space)
        l.rfe                       // return from exception (runs user code)
```

Note that the trace output now shows *user mode instruction addresses*:

```
Info 66: 'cpu1', 0x0000000000000000: l.movhi  r1,0xf000
Info 67: 'cpu1', 0x0000000000000004: l.ori    r2,r0,0x1234
Info 68: 'cpu1', 0x0000000000000008: l.sw     0x0(r1),r2
```

The application stores the value `0x1234` to address `0xf0000000` (legal in user mode) and then attempts the same store to address `0x00000000` (illegal in user mode, as this address has read and execute permissions only). This causes a data privilege exception at instruction 70. The data privilege exception handler simply increments `epcr` and returns from the exception (to skip the faulting instruction):

```
Info 69: 'cpu1', 0x000000000000000c: l.sw     0x0(r0),r2
Info 70: 'cpu1', 0x0000000000000300: l.mfspr  r1,r0,32
Info 71: 'cpu1', 0x0000000000000304: l.addi   r1,r1,0x4
Info 72: 'cpu1', 0x0000000000000308: l.mtspr  r0,r1,32
Info 73: 'cpu1', 0x000000000000030c: l.rfe
```

Next, the application attempts a call to address `0xf0000000` (also illegal in user mode, as this address has read and write permissions only). This causes an instruction privilege exception at instruction 78. The instruction privilege exception handler resumes execution at the link address (`0x1c` in the user address space):

```
Info 75: 'cpu1', 0x0000000000000014: l.jalr   r1
Info 76: 'cpu1', 0x0000000000000018: l.nop    0x0
Info 77: 'cpu1', 0x00000000f0000000: *** FETCH EXCEPTION ***
Info 78: 'cpu1', 0x0000000000000400: l.mtspr  r0,r9,32
Info 79: 'cpu1', 0x0000000000000404: l.rfe
```

Finally, the user mode program executes an `l.sys` instruction. The system call vector loads the contents of address `0x90000000` into register `r3` and then exits. On completion, register `r3` contains `0x1234`, proving that address `0x90000000` in supervisor mode is correctly mapped to address `0xf0000000` in user mode:

```
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 90000000   R2 : 00001234   R3 : 00001234
 R4 : 20000000   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : 0000001c   R10: deadbeef   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: deadbeef   R31: 00000000
 PC : 00000c0c   SR : 00008001   ESR: 00008000   EPC: 00000020
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:0 CF:0 OF:0
--------------- --------------- --------------- ---------------
```

22 Implementing a Dynamic-Mapped TLB

Chapter 18 showed how to implement *fixed-mapped* virtual memory. This chapter extends that example to show how to implement *dynamic-mapped* memory, where the virtual-to-physical address mappings can be changed at run time. Specifically, we implement a simple TLB structure.

22.1 General TLB Concepts

Most processors that implement virtual memory have hardware support for virtual-to-physical address translation, implemented using a *translation lookaside buffer* (TLB). The TLB is in typically an associative cache of a number of valid virtual-to-physical page mappings. If a memory access (fetch, read or write) uses a virtual address that is mapped in the TLB, the corresponding physical address location is obtained immediately and execution continues uninterrupted. Otherwise, if there is a *TLB miss*, a supervisor or kernel mode routine is typically called to handle the miss. The miss handler typically has access to a much larger table of virtual-to-physical address mappings in kernel memory: if the virtual address was present in this table but not in the TLB, the miss handler will eject an existing valid entry and replace it with the required entry from the page table; if the virtual address was not valid, a privilege exception will typically be generated. When an entry is being replaced, a key concept is the *replacement policy* that determines which existing valid entry is to be ejected[15].

Note that a TLB miss is often not handled the same way as a cache miss: typically, a cache miss simply stalls the processor while the cache line is filled, whereas a TLB miss usually requires an exception handler to be called to process it and modify the TLB contents

TLB entries typically hold more that just virtual-to-physical mappings: usually, each has some access permissions associated with it, and often an address space id (ASID) that restricts the validity of the entry to a subset of running processes (this prevents the TLB having to be completely flushed and refilled on a process switch). There is often also a choice of page sizes supported.

22.2 The Simple Example TLB

As mentioned in chapter 18, the OR1K processor can be configured with a full virtual memory system implementing a TLB. However, although this can be modeled using the VMI interface, it is too complex for an introductory example; instead, we will model a very simple TLB structure that demonstrates the key concepts but is easier to understand.

The simple TLB will extend the fixed-mapped virtual memory example of chapter 18 as follows:

1. Memory at addresses in the range `0x10000000:0xefffffff` in user memory space will be divided into 4096-byte pages that will be mapped on demand.

[15] In some processor architectures (e.g. ARM) the process of TLB maintenance is handled by special hardware.

2. The OR1K processor will implement a small TLB with four entries, so up to four different 4096-byte pages may be mapped concurrently.
3. At address `0x10000` in the supervisor address space, we will locate a page table structure. The data in this will be laid out as follows:
   a. `0x10000`: the number of entries in the table
   b. `0x10004`: virtual address for entry 1
   c. `0x10008`: virtual address for entry 2
   d. `0x1000c`: virtual address for entry 3
   e. (and so on up to the number of entries in the table)

   The page table will dynamically grow as more pages in the user address space need to be mapped.
4. The physical pages corresponding to the virtual addresses in the page table will be allocated in order starting at `0x10000000` in supervisor space. In other words, entry 1 in the table will be mapped to physical address `0x10000000`, entry 2 will be mapped to physical address `0x10001000`, entry 3 will be mapped to physical address `0x10002000`, and so on.
5. In order to update the processor TLB, we will modify the behavior of the `l.nop 99` instruction. When `l.nop 99` is executed, we will assume that:
   a. register `r1` holds a TLB index number (0-3) indicating the TLB entry to replace;
   b. register `r2` holds a user space virtual address;
   c. register `r3` holds a physical address.

   Executing the `l.nop 99` should discard any current mapping for the numbered TLB entry in the processor, and create a new mapping for that entry, mapping one page of data at the virtual address in the user memory domain to the physical address in the physical memory domain.
6. The replacement policy in this example will be simple round-robin.
7. In this simple example, all dynamically mapped pages will have read/write privilege, and there will be no way to invalidate an entry in the TLB (other than to replace it with a new mapping). We will also not implement any means of freeing an allocated physical page. All these would of course be supported in a real operating system, but would over complicate this example.

22.3 The Template Simple TLB Model

A template model for the OR1K processor implementing a simple TLB can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/20.or1kBehaviorTLB
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/20.or1kBehaviorTLB .
```

Compile the model, harness and application using the `make` command:

```
cd 20.or1kBehaviorTLB
make OPT=1
```

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is based on the previous model, with the changes listed in following sections.

22.4 Defining TLB Structures - or1kStructure.h

Some new #defines describe some aspects of the TLB:

```
#define OR1K_TLB_SIZE  4        // size of TLB
#define OR1K_PAGE_SIZE 4096     // size of TLB-mapped page
```

There is also a new type, tlbEntry, implementing a single entry in the TLB:

```
// simple TLB entry
typedef struct tlbEntryS {
    Uns32 va;                   // virtual address
    Uns32 pa;                   // physical address
    Bool  valid;                // is entry valid?
} tlbEntry, *tlbEntryP;
```

The processor structure has an array of TLB entries, and also a new memory domain field giving the physical domain for use during dynamic mapping:

```
typedef struct or1kS {

    . . . lines omitted . . .

    memDomainP physicalDomain;  // physical domain target for TLB mappings
    tlbEntry tlb[OR1K_TLB_SIZE];// simulated TLB

} or1k, *or1kP;
```

22.5 Implementing Virtual Memory - or1kVM.[ch]

The virtual memory constructor has been modified to save the physical domain for use in the TLB update function:

```
VMI_VMINIT_FN(or1kVMInit) {

    . . . lines omitted . . .

    // save physicalDomain on the OR1K structure for TLB usage
    or1kP or1k = (or1kP)processor;
    or1k->physicalDomain = physicalDomain;
}
```

A new function, or1kSetTLBEntry, is used to update the TLB mapping in the OR1K structure. The function takes as arguments the processor, a virtual address (va) and a physical address (pa):

```
void or1kSetTLBEntry(or1kP or1k, Uns32 tlbIndex, Uns32 va, Uns32 pa) {

    // get the current instruction count (for messages)
    Uns64 iCount = vmirtGetICount((vmiProcessorP)or1k);

    // clip tlbIndex, va and pa to valid values
    tlbIndex &= (OR1K_TLB_SIZE-1);
    va       &= ~(OR1K_PAGE_SIZE-1);
    pa       &= ~(OR1K_PAGE_SIZE-1);

    // get the TLB entry to update and the memory domain affected
    tlbEntryP  entry          = &or1k->tlb[tlbIndex];
    memDomainP physicalDomain = or1k->physicalDomain;
    memDomainP tlbDomain      = or1k->tlbDomain;

    // if TLB entry is already mapped, unmap it
    if(entry->valid) {

        vmiPrintf(
            FMT_64u": DELETE entry %u mapping (va:0x%08x pa:0x%08x)\n",
            iCount, tlbIndex, entry->va, entry->pa
        );

        vmirtUnaliasMemoryVM(
            tlbDomain, entry->va, entry->va+OR1K_PAGE_SIZE-1, True, 0
        );
    }

    // update the TLB entry
    entry->va    = va;
    entry->pa    = pa;
    entry->valid = True;

    vmiPrintf(
        FMT_64u": CREATE entry %u mapping (va:0x%08x pa:0x%08x)\n",
        iCount, tlbIndex, va, pa
    );

    // establish the new page mapping with read/write permissions
    vmirtAliasMemoryVM(
        physicalDomain, tlbDomain, pa, pa+OR1K_PAGE_SIZE-1, va, 0,
        MEM_PRIV_RW, True, 0
    );
}
```

The function first gets the TLB entry to update, and the physical and TLB domains:

```
    tlbEntryP  entry          = &or1k->tlb[tlbIndex];
    memDomainP physicalDomain = or1k->physicalDomain;
    memDomainP tlbDomain      = or1k->tlbDomain;
```

Next, it invalidates any current mapping for the TLB entry using `vmirtUnaliasMemoryVM`:

```
Bool vmirtUnaliasMemoryVM(
    memDomainP virtualDomain,
    Addr       virtualLowAddr,
    Addr       virtualHighAddr,
    Uns64      ASIDMaskOrG,
```

```
    Uns64      ASID
);
```

The arguments to this function are:

1. The domain in which to remove a virtual memory mapping.
2. The address range for which the mapping should be removed.
3. An entry indicating whether the entry is *globally-mapped* or *ASID-mapped* (see *TLB Modeling with ASID-Mapped Entries* later in this section). In this example, the TLB entries are assumed to be globally-mapped.
4. If the mapping to remove is ASID-managed, the corresponding ASID.

See the *VMI Run Time Function Reference* manual for more information about this function.

To monitor what is happening, the details of the deleted mapping are also printed out:

```
    if(entry->valid) {

        vmiPrintf(
            FMT_64u": DELETE entry %u mapping (va:0x%08x pa:0x%08x)\n",
            iCount, tlbIndex, entry->va, entry->pa
        );

        vmirtUnaliasMemoryVM(
            tlbDomain, entry->va, entry->va+OR1K_PAGE_SIZE-1, True, 0
        );
    }
```

Next, the TLB entry is updated to describe the new mapping, and the details of the new mapping are printed out:

```
    entry->va    = va;
    entry->pa    = pa;
    entry->valid = True;

    vmiPrintf(
        FMT_64u": CREATE entry %u mapping (va:0x%08x pa:0x%08x)\n",
        iCount, tlbIndex, va, pa
    );
```

Finally, the new mapping is established with read/write access permissions, using function `vmirtAliasMemoryVM`:

```
Bool vmirtAliasMemoryVM(
    memDomainP physicalDomain,
    memDomainP virtualDomain,
    Addr       physicalLowAddr,
    Addr       physicalHighAddr,
    Addr       virtualLowAddr,
    memMRUSetP mruSet,
    memPriv    privilege,
    Uns64      ASIDMaskOrG,
    Uns64      ASID
);
```

The arguments to this function are:

1. The *physical domain* to which to map.
2. The *virtual domain* in which the mapping should be created.
3. The address range in the physical domain of the region to map.
4. The base address in the virtual domain of the region to map.
5. A structure of type `memMRUSetP`, which automates the maintenance of region usage so that the *least-recently-used* TLB entry can be identified if required. In this example we are not interested in maintaining least-recently-used state (the replacement policy is round-robin) so this argument is `NULL`. See chapter 23 which discusses this in more detail.
6. An argument specifying the access privileges that the page should have.
7. An argument used to control whether an entry is *globally-mapped* or *ASID-mapped* (see *TLB Modeling with ASID-Mapped Entries* later in this section). In this example, the TLB entries are assumed to be globally-mapped.
8. If the mapping to create is ASID-managed, the corresponding ASID.

In this case, we create a globally-mapped entry, so `ASIDMaskOrG` is `True` and `ASID` is zero (the value is ignored for global entries):

```
    vmirtAliasMemoryVM(
        physicalDomain, tlbDomain, pa, pa+OR1K_PAGE_SIZE-1, va, 0,
        MEM_PRIV_RW, True, 0
    );
```

22.6 Adding TLB Update Instruction - `or1kMorph.c`

Function `morphNOP` has been modified to handle a `l.nop 99` instruction, calling the new TLB update function `or1kSetTLBEntry`. To enable an example application to be debugged, the `l.nop 98` instruction has also been subverted to print the current contents of register `r1`:

```
static void vmic_printVal(Uns32 val) {
    vmiPrintf("    fib returns %u\n", val);
}

static OR1K_MORPH_FN(morphNOP) {

    Uns32 code = state->info.c;

    // subvert nop 98 to print the contents of r1 and nop 99 to set a TLB entry
    if(code==98) {
        vmimtArgReg(OR1K_BITS, getGPR(1));
        vmimtCall((vmiCallFn)vmic_printVal);
    } else if(code==99) {
        vmimtArgProcessor();
        vmimtArgReg(OR1K_BITS, getGPR(1));    // r1: TLB entry index
        vmimtArgReg(OR1K_BITS, getGPR(2));    // r2: virtual address (va)
        vmimtArgReg(OR1K_BITS, getGPR(3));    // r3: physical address (pa)
        vmimtCall((vmiCallFn)or1kSetTLBEntry);
    }
}
```

It can often be a very useful technique to subvert certain opcodes to assist application debugging. Many processor instruction sets contain unassigned instructions for just this purpose. Make sure not to leave debugging hacks in the final model!

22.7 Testing the Simple TLB Model

Run the platform using the assembler executable file:

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

The output from this should be as follows:

```
    fib returns 1
369: CREATE entry 0 mapping (va:0x80001000 pa:0x10000000)
    fib returns 2
439: CREATE entry 1 mapping (va:0x80002000 pa:0x10001000)
    fib returns 1
    fib returns 3
549: CREATE entry 2 mapping (va:0x80003000 pa:0x10002000)
    fib returns 1
    fib returns 2
    fib returns 5
699: CREATE entry 3 mapping (va:0x80005000 pa:0x10003000)
    fib returns 1
    fib returns 2
    fib returns 1
    fib returns 3
    fib returns 8
919: DELETE entry 0 mapping (va:0x80001000 pa:0x10000000)
919: CREATE entry 0 mapping (va:0x80008000 pa:0x10004000)
    fib returns 1
1021: DELETE entry 1 mapping (va:0x80002000 pa:0x10001000)
1021: CREATE entry 1 mapping (va:0x80001000 pa:0x10000000)
    fib returns 2
1092: DELETE entry 2 mapping (va:0x80003000 pa:0x10002000)
1092: CREATE entry 2 mapping (va:0x80002000 pa:0x10001000)
    fib returns 1
    fib returns 3
. . . lines omitted . . .
    fib returns 233
62313: DELETE entry 1 mapping (va:0x80005000 pa:0x10003000)
62313: CREATE entry 1 mapping (va:0x800e9000 pa:0x1000b000)
    fib returns 610
62494: DELETE entry 2 mapping (va:0x8000d000 pa:0x10005000)
62494: CREATE entry 2 mapping (va:0x80262000 pa:0x1000d000)
Processor 'cpu1' terminated at 'exit', address 0xc00
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000262   R2 : 80262000   R3 : 00000001
 R4 : 15000000   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : 00000008   R10: deadbeef   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: 000001f3   R31: 00000000
 PC : 00000c04   SR : 00008201   ESR: 00008200   EPC: 0000000c
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:1 CF:0 OF:0
--------------- --------------- --------------- ---------------
```

```
processor has executed 62512 instructions
```

The source code for this example, in file `application/asmtest.S`, breaks down into three main sections. Firstly, there is a section that initializes the processor and copies the user program to `0x80000000` before entering user mode and starting the program. This is exactly as in the example in chapter 18:

```
        ////////////////////////////////////////////////////////////////////////
        // KERNEL START CODE (AT 0x0)
        ////////////////////////////////////////////////////////////////////////
.global _start
_start:
        l.ori       r31,r0,0        // r31 = 0 (stack pointer)
        l.ori       r30,r0,0        // r30 = 0 (TLB miss count)

        // prepare to copy application to user space
        l.movhi     r1,0x8000       // r1 = 0x80000000
        l.movhi     r2,hi(appStart) // r2 = appStart
        l.ori       r2,r2,lo(appStart)
        l.movhi     r3,hi(appEnd)   // r3 = appEnd
        l.ori       r3,r3,lo(appEnd)

        // copy application to user space
loop:   l.lwz       r4,0(r2)        // r4 = word of application code
        l.sw        0(r1),r4        // copy to 0x80000000
        l.addi      r2,r2,4         // increment src pointer
        l.addi      r1,r1,4         // increment dst pointer
        l.sfne      r2,r3           // r2!=r3?
        l.bf        loop            // go if true
        l.nop                       // (delay slot instruction)

        // run user code
        l.mtspr     r0,r0,0x40      // clear esr
        l.mtspr     r0,r0,0x20      // clear epc (resume address in user space)
        l.rfe                       // return from exception (runs user code)
```

The user mode application calculates `fib(15)` using the naive recursive algorithm previously described in this document. On every return from the `fib` function, it reads and writes to an address in user memory, calculated as `0x8000000 + result*4096`. In other words, if the `fib` function is about to return 1, it reads and writes to address `0x80001000`, if the `fib` function is about to return 2, it reads and writes to address `0x80001000`, and so on:

```
appStart:
        l.jal       fib             // calculate fib(15)
        l.addi      r1,r0,15        // r1 = 15 (delay slot)
        l.sys       0               // exit the application

fib:    l.sflesi    r1,1            // r1<=1? (signed)
        l.bf        done            // done if so, result is r1
        l.nop                       // (delay slot)

        l.addi      r31,r31,-12     // create stack frame
        l.sw        0(r31),r9       // save link register
        l.sw        4(r31),r1       // save input r1

        l.jal       fib             // calculate fib(N-1)
        l.addi      r1,r1,-1        // r1 = N-1 (delay slot)
```

```
        l.sw        8(r31),r1        // save fib(N-1)

        l.lwz       r1,4(r31)        // restore initial N
        l.jal       fib              // calculate fib(N-2)
        l.addi      r1,r1,-2         // r1 = N-2 (delay slot)

        l.lwz       r2,8(r31)        // restore fib(N-1)
        l.add       r1,r1,r2         // r1 = fib(N-2) + fib(N-1)

        l.lwz       r9,0(r31)        // restore link register
        l.addi      r31,r31,12       // destroy stack frame

        l.nop       98               // whats in r1?
        l.muli      r2,r1,PAGE_SIZE // r2 = fib * page size
        l.movhi     r3,0x8000        // r3 = user heap base (0x80000000)
        l.add       r2,r2,r3         // r2 = fib page
        l.lwz       r3,0(r2)         // r3 = old fib count
        l.addi      r3,r3,1          // increment count
        l.sw        0(r2),r3         // save new count

done:   l.jr        r9               // return, result in r1
        l.nop                        // (delay slot instruction)
appEnd:
```

The choice of application is completely arbitrary, and has been selected merely to cause writes to a somewhat random sequence of pages in the TLB mapped region starting at address `0x80000000` in the user address space. The subverted instruction `l.nop 98` has been used to print the return value from the fib function to make it easier to follow the flow of execution.

The third section of the example is the data privilege exception handler:

```
.org 0x300
        ////////////////////////////////////////////////////////////////////////
        // DATA PRIVILEGE EXCEPTION VECTOR (AT 0x300) - UPDATE TLB
        ////////////////////////////////////////////////////////////////////////
        l.sw        -4(r31),r1       // save r1
        l.sw        -8(r31),r2       // save r2
        l.sw        -12(r31),r3      // save r3
        l.sw        -16(r31),r4      // save r4
        l.sw        -20(r31),r5      // save r5
        l.sw        -24(r31),r6      // save r6
        l.sw        -28(r31),r7      // save r7

        l.addi      r30,r30,1        // increment TLB miss count

        l.mfspr     r2,r0,0x30       // r2 = eear
        l.addi      r1,r0,-PAGE_SIZE// r1 = page mask
        l.and       r2,r2,r1         // mask faulting va to page size
        l.movhi     r4,0x0001        // r4 = 0x10000 (page table address)
        l.lwz       r5,0(r4)         // r5 = table size
        l.addi      r7,r4,8          // r7 = current page table entry
        l.addi      r6,r0,0          // r6 = 0 (current entry index)

try:    l.sfeq      r6,r5            // last entry?
        l.bf        miss             // go if so (a miss)
        l.nop                        // (delay slot instruction)
        l.lwz       r1,0(r7)         // r1 = current entry va
        l.sfeq      r1,r2            // does va match entry?
        l.bf        hit              // go if so (a hit)
```

```
        l.nop                       // (delay slot instruction)
        l.addi      r7,r7,4         // r7 = next page table entry
        l.j         try             // try next entry
        l.addi      r6,r6,1         // r6 = next entry index (delay slot)

miss:   l.sw        0(r7),r2        // save va into page table
        l.addi      r5,r5,1         // increment table size
        l.sw        0(r4),r5        // save new table size

hit:    l.muli      r3,r6,PAGE_SIZE // r3 = current entry index * page size
        l.movhi     r5,0x1000       // r5 = heap base (0x10000000)
        l.add       r3,r3,r5        // r3 = pa
        l.lwz       r1,4(r4)        // r1 = tlbIndex
        l.nop       99              // set TLB entry
        l.addi      r1,r1,1         // increment tlbIndex
        l.andi      r1,r1,TLB_SIZE-1// clip to TLB size
        l.sw        4(r4),r1        // save tlbIndex for next call

        l.lwz       r1,-4(r31)      // restore r1
        l.lwz       r2,-8(r31)      // restore r2
        l.lwz       r3,-12(r31)     // restore r3
        l.lwz       r4,-16(r31)     // restore r4
        l.lwz       r5,-20(r31)     // restore r5
        l.lwz       r6,-24(r31)     // restore r6
        l.lwz       r7,-28(r31)     // restore r7
        l.rfe                       // return from exception
```

The handler does the following:

1. it gets the fault address for the read or write;
2. it determines whether there is already a mapping for that fault address in the page table at address `0x10000` in supervisor memory - if not, it allocates a new page from the heap starting at `0x10000000` in supervisor memory, and creates a new page table entry mapping the virtual page address to the new page;
3. finally, it executes `l.nop 99` to update the processor simulated TLB entry.

The entry index to update is generated in a round-robin fashion. The fault handler keeps count of the number of times it has been called in `r30`. Examining the test output, we see that the first four page accesses in the TLB allocated region fill entries 0, 1, 2 and 3 of the TLB in sequence:

```
    fib returns 1
369: CREATE entry 0 mapping (va:0x80001000 pa:0x10000000)
    fib returns 2
439: CREATE entry 1 mapping (va:0x80002000 pa:0x10001000)
    fib returns 1
    fib returns 3
549: CREATE entry 2 mapping (va:0x80003000 pa:0x10002000)
    fib returns 1
    fib returns 2
    fib returns 5
699: CREATE entry 3 mapping (va:0x80005000 pa:0x10003000)
```

After that, every TLB miss must first cause an existing mapping to be deleted before the new mapping is established:

```
    fib returns 1
    fib returns 2
```

```
    fib returns 1
    fib returns 3
    fib returns 8
919: DELETE entry 0 mapping (va:0x80001000 pa:0x10000000)
919: CREATE entry 0 mapping (va:0x80008000 pa:0x10004000)
    fib returns 1
1021: DELETE entry 1 mapping (va:0x80002000 pa:0x10001000)
1021: CREATE entry 1 mapping (va:0x80001000 pa:0x10000000)
    fib returns 2
1092: DELETE entry 2 mapping (va:0x80003000 pa:0x10002000)
1092: CREATE entry 2 mapping (va:0x80002000 pa:0x10001000)
. . . etc . . .
```

At the end of simulation, register `r30` holds the number of TLB misses – 0x1f3 (499):

```
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000262   R2 : 80262000   R3 : 00000001
 R4 : 15000000   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : 00000008   R10: deadbeef   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: 000001f3   R31: 00000000
 PC : 00000c04   SR : 00008201   ESR: 00008200   EPC: 0000000c
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:1 CF:0 OF:0
--------------- --------------- --------------- ---------------
```

22.8 TLB Modeling with Multiple Processor Modes

In this example, only the user mode address space was TLB mapped. In the general case, more than one address space may be TLB mapped. This introduces an extra level of complexity because memory mappings need to be maintained in all TLB mapped address spaces.

There are two ways to address this problem, described in the following subsections.

22.8.1 Apply Changes in All TLB-Mapped Domains

One solution is to *apply all TLB changes to every TLB-mapped domain*. For example, if both kernel and user domains are TLB mapped, each call to `or1kSetTLBEntry` would require *two* calls to `vmirtAliasMemoryVM`, one to establish the mapping and protections in the kernel mode domain and one to establish them in the user mode domain. This solution is the simplest to implement initially.

22.8.2 Maintain Multiple Copies of the TLB

An alternative solution requires *maintaining multiple copies of the TLB, one for each TLB-mapped domain*, as follows:

1. All TLB changes are made only in the copy of the TLB for the current processor mode, and all calls to `vmirtAliasMemoryVM` and `vmirtUnaliasMemoryVM` are applied only to the current domain (as in the example above).

2. When the processor switches mode, all mappings from the *previously-current* copy of the TLB are replicated in the *new current* TLB, and corresponding memory mappings in the new current memory domain are established using `vmirtAliasMemoryVM` and `vmirtUnaliasMemoryVM`.

Maintaining multiple copies of the TLB is often faster than the simpler approach of applying changes to all TLB-mapped domains, but requires careful coding to avoid obscure bugs.

22.9 TLB Modeling with ASID-Mapped Entries

It is common hardware practice to label TLB entries with *address-space identifiers* (ASIDs). ASIDs allow a TLB to be partitioned efficiently between several processes: the processor has a current ASID register, and only entries that match that register are considered candidates for matching.

ASID-mapped virtual memory pages are supported directly by the simulator. You can set the current ASID for the processor using:

```
void vmirtSetProcessorASID(vmiProcessorP processor, Uns64 ASID);
```

The ASID can be any 64-bit value. When pages are mapped and unmapped using `vmirtAliasMemoryVM` and `vmirtUnaliasMemoryVM`, they can either be specified to be *global* mappings (in which case the mapping is valid irrespective of ASID) or *ASID-managed* mappings (in which case the mapping is valid only whether the processor ASID matches the ASID specified when the virtual page was mapped). When the ASID is modified by `vmirtSetProcessorASID`, the simulator automatically invalidates any existing ASID-managed mappings for the old ASID: no special action needs to be taken in the model.

Functions `vmirtAliasMemoryVM` and `vmirtUnaliasMemoryVM` both take an argument of type `Uns64` called `ASIDMaskOrG` which controls whether a virtual memory mapping matches the current processor ASID or not. This parameter can take values with three meanings, as follows:

1. If `ASIDMaskOrG` is `True` (i.e. has value `1`) then the entry is *globally-mapped*. In this case, the `ASID` parameter to these functions is ignored.
2. If `ASIDMaskOrG` is `False` (i.e. has value `0`) then the entry is *fully-ASID-mapped*. In this case, the entry matches only if the current processor ASID exactly equals the given `ASID` value.
3. If `ASIDMaskOrG` is anything else, then it is treated as a bitmask, and the entry matches only if:
```
((currentASID & ASIDMaskOrG) == ASID)
```
where `currentASID` is the current processor ASID and `ASIDMaskOrG` and `ASID` are the parameters to `vmirtAliasMemoryVM` or `vmirtUnaliasMemoryVM`.

In most cases, the simple usage model implied by the first two cases above suffices. The third case can be useful when modeling processors that implement multiple levels of address translation (for example, an ASID and a VMID, if hardware virtualization is supported).

As an example, the following line creates a mapping valid only when the processor ASID is 34:

```
    vmirtAliasMemoryVM(
        physicalDomain, tlbDomain, pa, pa+OR1K_PAGE_SIZE-1, va, 0,
        MEM_PRIV_RW, False, 34
    );
```

As a second example, the following line creates a mapping valid only when *the least significant 8 bits* of the processor ASID contain the value 34 (all other bits of the processor ASID are ignored):

```
    vmirtAliasMemoryVM(
        physicalDomain, tlbDomain, pa, pa+OR1K_PAGE_SIZE-1, va, 0,
        MEM_PRIV_RW, 0xff, 34
    );
```

22.9.1 Managing Virtual Address Aliases with Different ASID

When processors support ASID-based mapping, it is common for their TLBs to be populated with entries that map the *same virtual address* with *different ASID*s. For example, an operating system may always place the first executable address of a program at the same virtual address, e.g. `0x80000000`. If the processor is running an OS with four currently-running user processes, there may therefore be four distinct mappings for virtual address `0x80000000` in the TLB, one for each user process, each with a different ASID.

As described above, the simulator automatically invalidates mappings for an *old* ASID on a processor ASID switch. It does not, however, automatically re-establish any mapping for the *new* ASID that may have been specified in the past: it is up to the model to do this. In general, it is most efficient to re-establish mappings using a *lazy* scheme, as described in the next subsection.

22.10 Lazy Mapping of TLB Entries

In the example described in this chapter, memory mappings are managed using `vmirtAliasMemoryVM` and `vmirtUnaliasMemoryVM` whenever a TLB entry is updated. However, these memory management functions are quite compute-intensive and should be used sparingly. If TLB updates establish mappings that are not used by the running program before being replaced, then the processor model will run slower than necessary because time will be wasted setting up memory mappings that are never actually used. This is very often the case for modal processors, since mappings are typically set up in *kernel* mode and used only in *user* mode.

For best performance, it is therefore best to establish virtual memory mappings on demand in a *lazy* fashion. This is done as follows:

1. Each TLB entry has an additional boolean field, `mapped`. This boolean indicates whether the TLB entry has already had a mapping established for it by `vmirtAliasMemoryVM`. Any entry with `mapped` of `False` definitely has no mapping established. Any entry with `mapped` of `True` may or may not have a current valid mapping (if it is an ASID-managed entry, a mapping may have been established but later invalidated by a processor ASID switch, as described in the previous section).

2. Each TLB entry can be in one of three states:
    a. `valid=0`, `mapped=0`: the TLB entry is not valid.
    b. `valid=1`, `mapped=1`: the TLB entry is valid and possibly mapped.
    c. `valid=1`, `mapped=0`: the TLB entry is valid but *not* currently mapped.

3. When a TLB entry is written, if `entry->mapped` is 1 then any existing mapping for that entry must be discarded using `vmirtUnaliasMemoryVM`. If the entry is an ASID-managed entry, then the `isGlobal` and `ASID` parameters must match those specified when the mapping was created.
   However, a new mapping is *not* established with `vmirtAliasMemoryVM` at this point: instead, `entry->mapped` is set to 0.

4. When the lazy scheme is in use, processor model exception handler callbacks will be called for processor addresses that are *valid but not mapped*. Therefore, each exception handler needs to allow for this by establishing a mapping for a valid but unmapped address if required. Template code for the OR1K could be as follows:

```
VMI_RD_PRIV_EXCEPT_FN(or1kRdPrivExceptionCB) {

    or1kP or1k = (or1kP)processor;

    // if the address is present in the TLB but not currently mapped, establish
    // the mappings
    if(or1kTLBMapRead(or1k, address, bytes)) {

        // here if the read address range is mapped and readable – redo the read
        *action = VMI_LOAD_STORE_CONTINUE;

    } else if(MEM_AA_IS_TRUE_ACCESS(attrs)) {

        // here if a true exception
        or1k->EEAR = (Uns32)address;
        or1kTakeException(or1k, OR1K_EXCPT_DPF, 0);
    }
}
```

Code for `or1kTLBMapRead` (not shown) would do the following:

1. Establish whether the address range `address:address+bytes-1` lies in a TLB-mapped page that allows read access and is valid. The matching entry may or may not

be already marked as mapped (it may be an ASID-managed entry that was automatically unmapped by an ASID switch).

2. If so, create the mapping using `vmirtAliasMemoryVM`, set `entry->mapped` to 1, and return `True`.
3. If not, return `False`.

Note the behavior of function `or1kRdPrivExceptionCB` when `or1kTLBMapRead` returns `True`. In this case, the function sets the by-ref parameter `action` to the value `VMI_LOAD_STORE_CONTINUE`. What this does is cause the failing read to be *retried* on return from the exception callback – since a TLB mapping has been established, this read should now succeed (but see the detailed description in section 12 for cases in which the access may still fail).

Also note that the TLB mapping and subsequent read are done for both *artifact* and *non-artifact* accesses, but any exception is taken only for *non-artifact* accesses. Refer to section 12 for a detailed description of how these access types are indicated by the `attrs` parameter to the exception callback.

23 Implementing a TLB LRU Replacement Policy

Chapter 22 showed how to implement a TLB with a round-robin replacement policy. Often, other replacement policies are used. One of the most common is the LRU replacement policy, where the entry to replace is the *least-recently-used* entry. The simulator has special support to enable an LRU replacement policy to be efficiently modeled, as demonstrated in this example.

23.1 Introduction to LRU Replacement Implementation

Before looking at the details of the LRU replacement policy implementation, it is useful to understand some of the associated concepts.

To model an LRU replacement policy, it is first required to have a *state variable* that represents the current order of entries in the table. For example, suppose that the TLB contains four entries, numbered 0, 1, 2 and 3. At a particular point in time, the four entries may have been accessed (in most-to-least-recent order) in any of 4! (i.e. 24) different ways. Suppose that the current state implies the following ordering:

0 3 1 2

If there is now a read of an address mapped by TLB entry 1, that entry should be promoted to the most-recently-used (MRU) position, yielding a new current state, implying the following ordering:

1 0 3 2

In general, it is possible to construct a *transition table* for each TLB entry that, when indexed by the current state, will return the new state:

```
newState = transitionTable[currentState];
```

When an entry has to be ejected from the table, the last entry implied by the state is the least recently used and should be selected (entry 2 in this example).

To implement an LRU replacement policy on a set of TLB entries, two things are therefore required:

1. A state variable that encodes the entry ordering for the set of LRU-managed entries.
2. A set of transition tables, one for each entry, used to get the next state when the current entry is promoted to the most-recently-used position.

Given this information, the simulator is able to manage the state variable automatically at each read, write or fetch access.

23.2 The Template LRU Replacement Policy Model

A template model for the OR1K processor implementing a TLB with LRU replacement policy can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/21.or1kBehaviorTLBMRU
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/21.or1kBehaviorTLBMRU .
```

Compile the model, harness and application using the `make` command:

```
cd 21.or1kBehaviorTLBMRU
make OPT=1
```

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is based on the previous model, with the changes listed in following sections.

23.3 Adding MRU Entry State - `or1kStructure.h`

The `tlbEntry` structure has been modified to contain a new field of type `memMRUSet` (defined in `vmiTypes.h`):

```
typedef struct tlbEntryS {
    Uns32     va;               // virtual address
    Uns32     pa;               // physical address
    Bool      valid;            // is entry valid?
    memMRUSet set;              // MRU management
} tlbEntry, *tlbEntryP;
```

The `memMRUSet` type has this definition:

```
typedef struct memMRUSetS {
    const Uns32 *nextState;     // LUT giving next state from current state
    Uns32       *currentState;  // reference to current state
} memMRUSet;
```

The `memMRUSet` structure has two entries:

1. A transition table, `nextState`. Given a current state, this is indexed to find the next state.
2. A pointer to a current state variable, `currentState`.

The or1k structure has a new `mruState` field, used to represent the current state of the TLB entry set:

```
typedef struct or1kS {

    . . . lines omitted . . .

    memDomainP physicalDomain;      // physical domain target for TLB mappings
    memDomainP tlbDomain;           // memory domain to which TLB mappings apply
    tlbEntry   tlb[OR1K_TLB_SIZE];  // simulated TLB
    Uns32      mruState;            // TLB MRU state variable

} or1k, *or1kP;
```

23.4 Using MRU Entry State - `or1kVM.c`

Function or1kVMInit has some new code to initialize the memMRUSet structures inside the TLB entries:

```
VMI_VMINIT_FN(or1kVMInit) {

    . . . lines omitted . . .

    Uns32 i;
    for(i=0; i<OR1K_TLB_SIZE; i++) {
        or1k->tlb[i].set.nextState    = vmirtGetMRUStateTable(OR1K_TLB_SIZE, i);
        or1k->tlb[i].set.currentState = &or1k->mruState;
    }
}
```

For each entry, two things are done:

1. The `nextState` field is initialized with a transition table returned by a call to function `vmirtGetMRUStateTable`. Given the number of entries in the TLB (`OR1K_TLB_SIZE`) and the index number of this entry (`i`), this function returns a transition table encoding state transitions when entry `i` is promoted to the most-recently-used slot.
2. The `currentState` field is initialized to point to the `mruState` field in the OR1K processor structure.

> Function `vmirtGetMRUStateTable` can be used to obtain transition tables for any number of entries up to and including 8. The transition table it returns is of type `const Uns32 *`.
> You do not have to use `vmirtGetMRUStateTable` to obtain the transition table – it is provided for convenience only, and any other transition table can be provided if desired.
> In this example, we have a single set of MRU-managed entries, and therefore there is a single state variable, `mruState`, in the processor structure. Multiple independent MRU-managed sets can be modeled: simply ensure that there is a separate state variable for each set in the processor model.

Function `or1kSetTLBEntry` has been modified so that, if it is passed a `tlbIndex` equal to the number of TLB entries (`OR1K_TLB_SIZE`), it selects the least-recently-used entry to discard:

```
void or1kSetTLBEntry(or1kP or1k, Uns32 tlbIndex, Uns32 va, Uns32 pa) {

    // get the current instruction count (for messages)
    Uns64 iCount = vmirtGetICount((vmiProcessorP)or1k);

    // either use the specified TLB index or, if tlbIndex is OR1K_TLB_SIZE,
    // replace the LRU entry
    if(tlbIndex==OR1K_TLB_SIZE) {
        tlbIndex = vmirtGetNthStateIndex(
            OR1K_TLB_SIZE, or1k->mruState, OR1K_TLB_SIZE-1
        );
    } else {
        tlbIndex &= (OR1K_TLB_SIZE-1);
    }
```

Function `vmirtGetNthStateIndex` has the following prototype (in file `vmiRt.h`):

```
Uns8 vmirtGetNthStateIndex(Uns32 numEntries, Uns32 state, Uns32 position);
```

Given the number of entries in a transition table, a current state and a *position*, this function returns the entry at the passed position for that state. A position of 0 implies the most-recently-used entry and a position of `numEntries-1` implies the least-recently-used entry (the function can also return the entry at any intermediate position between the most and least recently used, though this is seldom useful).

In this example, we provide the current MRU state from the processor structure and request the least-recently-used entry (position `OR1K_TLB_SIZE-1`).

> `vmirtGetNthStateIndex` works only with states managed by transition tables returned by `vmirtGetMRUStateTable`. If you use custom state tables, you will need to derive the least-recently-used state yourself.

In order to tell the simulator that memory accesses to a particular dynamic-mapped memory region should update an MRU state variable, there is one further change in function `or1kSetTLBEntry`:

```
    vmirtAliasMemoryVM(
        physicalDomain, tlbDomain, pa, pa+OR1K_PAGE_SIZE-1, va,
        &or1k->tlb[tlbIndex].set, MEM_PRIV_RW, True, 0
    );
```

The `mruSet` argument to `vmirtAliasMemoryVM` is a pointer to a `memMRUSet` structure – in this case, we select the structure in the current TLB entry.

23.5 Testing the LRU Replacement Policy Model

Run the platform using the assembler executable file:

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

The output from this should be as follows:

```
    fib returns 1
369: CREATE entry 0 mapping (va:0x80001000 pa:0x10000000)
    fib returns 2
436: CREATE entry 1 mapping (va:0x80002000 pa:0x10001000)
    fib returns 1
    fib returns 3
543: CREATE entry 2 mapping (va:0x80003000 pa:0x10002000)
    fib returns 1
    fib returns 2
    fib returns 5
690: CREATE entry 3 mapping (va:0x80005000 pa:0x10003000)
    fib returns 1
    fib returns 2
    fib returns 1
    fib returns 3
    fib returns 8
907: DELETE entry 3 mapping (va:0x80005000 pa:0x10003000)
907: CREATE entry 3 mapping (va:0x80008000 pa:0x10004000)
    fib returns 1
    fib returns 2
    fib returns 1
    fib returns 3
    fib returns 1
    fib returns 2
    fib returns 5
1192: DELETE entry 3 mapping (va:0x80008000 pa:0x10004000)
1192: CREATE entry 3 mapping (va:0x80005000 pa:0x10003000)
. . . lines omitted . . .
    fib returns 233
53940: DELETE entry 1 mapping (va:0x80005000 pa:0x10003000)
53940: CREATE entry 1 mapping (va:0x800e9000 pa:0x1000b000)
    fib returns 610
54118: DELETE entry 3 mapping (va:0x8000d000 pa:0x10005000)
54118: CREATE entry 3 mapping (va:0x80262000 pa:0x1000d000)
Processor 'cpu1' terminated at 'exit', address 0xc00
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000262   R2 : 80262000   R3 : 00000001
 R4 : 15000000   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : 00000008   R10: deadbeef   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: 00000156   R31: 00000000
 PC : 00000c04   SR : 00008201   ESR: 00008200   EPC: 0000000c
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:1 CF:0 OF:0
--------------- --------------- --------------- ---------------

processor has executed 54133 instructions
```

The source code for this example, in file `application/asmtest.S`, is identical to the previous example, except that `TLB_SIZE` is always used as the entry argument to the `l.nop 99` instruction to indicate that the LRU entry should be replaced:

```
hit:    l.muli      r3,r6,PAGE_SIZE // r3 = current entry index * page size
        l.movhi     r5,0x1000       // r5 = heap base (0x10000000)
        l.add       r3,r3,r5        // r3 = pa
        l.addi      r1,r0,TLB_SIZE  // r1 = TLB_SIZE (i.e. replace LRU entry)
        l.nop       99              // set TLB entry
```

When the application runs, it proceeds in a similar way to the run in chapter 22 until the point just after the `fib(8)` result; at this point, entry 3 is discarded (the `fib(5)` result entry) since this is the least-recently-used entry.

At the end of simulation, register `r30` holds the number of TLB misses – 0x156 (342):

```
--------------- --------------- --------------- ---------------
 R0 : 00000000   R1 : 00000262   R2 : 80262000   R3 : 00000001
 R4 : 15000000   R5 : deadbeef   R6 : deadbeef   R7 : deadbeef
 R8 : deadbeef   R9 : 00000008   R10: deadbeef   R11: deadbeef
 R12: deadbeef   R13: deadbeef   R14: deadbeef   R15: deadbeef
 R16: deadbeef   R17: deadbeef   R18: deadbeef   R19: deadbeef
 R20: deadbeef   R21: deadbeef   R22: deadbeef   R23: deadbeef
 R24: deadbeef   R25: deadbeef   R26: deadbeef   R27: deadbeef
 R28: deadbeef   R29: deadbeef   R30: 00000156   R31: 00000000
 PC : 00000c04   SR : 00008201   ESR: 00008200   EPC: 0000000c
 TCR: 00000000   TMR: 00000000   PSR: 00000000   PMR: 00000000
 BF:1 CF:0 OF:0
--------------- --------------- --------------- ---------------
```

Note that the number of misses in this example (342) is less than the number in the previous example (499), implying that the LRU replacement policy is giving some benefit over a simple round-robin policy.

24 Implementing QuantumLeap-Compatible Models

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

In order for processor models to run correctly under QuantumLeap, some care must be taken to indicate to the simulator instructions that need to be executed *atomically* or which access *shared state*. This chapter explains the changes required.

24.1 Introduction to Multiprocessor Simulation

The *OVP Processor Modeling Guide* describes how multiprocessor platforms may be simulated with QuantumLeap parallel simulation enabled. Refer to the chapter titled *Parallel Simulation: QuantumLeap* in that document before reading further.

24.2 QuantumLeap Requirements

The QuantumLeap parallel simulation algorithm accelerates multiprocessor simulation by distributing execution of cores in the platform over threads on multiple cores of the host machine. This means that simulation of multiple cores is performed in parallel, increasing performance. Unless informed otherwise, the simulator kernel assumes that instructions can execute in parallel, except in the following cases:

1. The instruction makes an embedded call to a *synchronizing* function from the VMI interface (for example, see the *VMI Run Time Function Reference* manual for a description of which functions in that API are synchronizing).
2. The instruction activates a memory read or write callback function.
3. The instruction is intercepted (for example, by a Semihost library).

In any of the cases described above, the simulator will assume that synchronous execution is required. It will ensure that all other parallel processor threads are stopped before allowing the current processor thread to proceed, thereby guaranteeing that the current processor sees a consistent system state that cannot be asynchronously modified by execution of another processor. The simulator also enforces synchronous execution during code morphing and intercepted function calls.

The algorithm automatically ensures safe, deterministic parallel simulation in the vast majority of cases. However, there are three scenarios that need to be handled explicitly in the processor model:

1. Identification of test-and-set or atomic swap instructions.
2. Identification of load/store exclusive constructs.
3. Identification of instructions that access shared register state.

These three cases are covered in the next sections.

24.3 Test-and-Set or Atomic Swap Instructions

Traditionally, instruction sets have implemented *test-and-set* or *swap* instructions that enable a memory location to be read and updated in a single atomic instruction. Using the basic VMI morph-time API, a swap instruction could be described like this:

```
    vmimtLoadRRO(
        32, 32, c, CPUX_TMP, CPUX_REG(ra), MEM_ENDIAN_BIG, True,
        MEM_CONSTRAINT_ALIGNED
    );
    vmimtStoreRRO(
        32, c, CPUX_REG(ra), CPUX_REG(rd), MEM_ENDIAN_BIG,
        MEM_CONSTRAINT_ALIGNED
    );
    vmimtMoveRR(CPUX_GBITS, CPUX_REG(rd), CPUX_TMP);
```

This sequence emits code that first loads the contents of a memory location into a processor temporary, then stores to the same location from a register, and finally moves the temporary to the same register. The overall effect is to swap the register and memory location.

The above emitted code sequence works correctly using the standard, single-threaded multiprocessor simulation algorithm. However, when QuantumLeap is enabled, there is a chance that another asynchronously-executing processor could modify the memory location between execution of the load and the store, leading to incorrect behavior (assuming the instruction is atomic in the real hardware). To prevent this occurring, the current instruction should be identified as atomic, using the `vmimtAtomic` function, as follows:

```
    vmimtAtomic();
    vmimtLoadRRO(
        32, 32, c, CPUX_TMP, CPUX_REG(ra), MEM_ENDIAN_BIG, True,
        MEM_CONSTRAINT_ALIGNED
    );
    vmimtStoreRRO(
        32, c, CPUX_REG(ra), CPUX_REG(rd), MEM_ENDIAN_BIG,
        MEM_CONSTRAINT_ALIGNED
    );
    vmimtMoveRR(CPUX_GBITS, CPUX_REG(rd), CPUX_TMP);
```

Function `vmimtAtomic` indicates to the simulator that *all other processors in a multiprocessor simulation must be stopped while this instruction executes*. This ensures that the memory location content cannot be updated by an asynchronously-executing processor between the load and store.

> Function `vmimtAtomic` can be called at any point in the emission of the current instruction. We could, for example, have inserted the call after the final `vmimtMoveRR` and achieved the same effect.

24.4 Load/Store Exclusive Constructs

Traditional *test-and-set* or *swap* instructions and now being replaced in more modern instruction sets with *load/store exclusive* blocks, because these scale better to highly-parallel systems. Load/store exclusive synchronization is typically implemented using a pair of instructions: an initial *load exclusive*, which loads from an address and sets up a monitor that detects any writes by other processors to that address, and a subsequent *store exclusive*, which commits a write to the address only if the monitor has not detected a write to the same address by another processor in the interim. Many features of load/store exclusive instructions are typically implementation-dependent (for example, the granularity of the exclusive region and the conditions that can cause an exclusive store to fail) but this is not significant here.

In following subsections, we will first describe how the basic load/store exclusive construct should be implemented assuming QuantumLeap is not in use, and then describe the changes required to support QuantumLeap.

24.4.1 Describing the Load Exclusive Instruction

The load exclusive instruction is typically implemented using a load which additionally sets a load-exclusive-active flag and records the load-exclusive address, like this:

```
    // load from address in ra
    vmimtLoadRRO(
        32, 32, 0, CPUX_REG(rd), CPUX_REG(ra), MEM_ENDIAN_BIG, True,
        MEM_CONSTRAINT_ALIGNED
    );

    // indicate load/store exclusive is active
    vmimtMoveRC(8, CPUX_LDREX_FLAG, 1);

    // record load/store exclusive address
    vmimtMoveRR(32, CPUX_LDREX_ADDRESS, CPUX_REG(ra));
```

(In this example, `CPUX_LDREX_FLAG` and `CPUX_LDREX_ADDRESS` are assumed to be architectural registers that indicate whether a load/store exclusive is active and the load/store exclusive address, respectively.)

24.4.2 Describing the Store Exclusive Instruction

The store exclusive instruction typically does the following:

1. Validates that the load-exclusive-active flag is set (otherwise the store is skipped);
2. Validates that the load-exclusive address matches (otherwise the store is skipped);
3. Performs the store;
4. Clears the load-exclusive-active flag.

A simple implementation could therefore be:

```
    vmiLabelP done = vmimtNewLabel();

    // skip store if load/store exclusive is not active
    vmimtCompareRCJumpLabel(8, vmi_COND_NE, CPUX_LDREX_FLAG, 1, done);

    // skip store if load/store address does not match
    vmimtCompareRR(32, vmi_COND_NE, CPUX_LDREX_ADDRESS, CPUX_REG(ra), CPUX_TMP);
    vmimtCondJumpLabel(CPUX_TMP, True, done);
```

```
    // store to address in ra
    vmimtStoreRRO(
        32, 0, CPUX_REG(ra), CPUX_REG(rd), MEM_ENDIAN_BIG,
        MEM_CONSTRAINT_ALIGNED
    );

    // jump to here if store should be skipped
    vmimtInsertLabel(done);

    // terminate load/store exclusive
    vmimtMoveRC(8, CPUX_LDREX_FLAG, 0);
```

24.4.3 Handling the Address Monitor

The basic load/store exclusive instructions have now been implemented to the level required for single-processor simulation. The implementation is not yet sufficient for a multiprocessor simulation, however, because there is no monitor installed on the load/store exclusive address to detect writes to that address *by other processors*.

When QuantumLeap is not active, there is no chance that another processor could write to the load/store exclusive address while the current processor is running because the simulation is single-threaded: if this processor is running, then all others must be suspended. However, if a load/store exclusive is in force when a processor reaches the end of its time-slice then it *is* possible for another processor to store to the monitored address *while this processor is suspended awaiting its next time slice*. This can be efficiently handled as follows:

1. If a load/store exclusive is in force when the processor reaches the end of its time slice, *install a memory callback on the load/store exclusive address to detect writes to that address by other processors*.
2. When a processor starts a new time slice, *remove any previously installed memory callback before resuming simulation.*

Here is a typical implementation of this algorithm:

```
//
// Callback to abort load/store exclusive on a conflicting write by another
// processor
//
static VMI_MEM_WATCH_FN(abortEA) {
    if(processor) {
        cpuxP cpux = (cpuxP)userData;
        cpux->ldrexActive = False;
        updateExclusiveAccessCallback(cpux, False);
    }
}

//
// Install or remove the exclusive access monitor callback
//
static void updateExclusiveAccessCallback(cpuxP cpux, Bool install) {

    memDomainP domain  = vmirtGetProcessorDataDomain((vmiProcessorP)cpux);
    Uns32      simLow  = cpux->ldrexAddress;
    Uns32      simHigh = simLow+3;
```

```
    // install or remove a watchpoint on the current exclusive access address
    if(install) {
        vmirtAddWriteCallback(domain, 0, simLow, simHigh, abortEA, cpux);
    } else {
        vmirtRemoveWriteCallback(domain, 0, simLow, simHigh, abortEA, cpux);
    }
}

//
// This is called on simulator context switch (when this processor is either
// about to start or about to stop simulation)
//
VMI_IASSWITCH_FN(cpuxContextSwitchCB) {
    cpuxP cpux = (cpuxP)processor;
    if(cpux->ldrexActive) {
        updateExclusiveAccessCallback(cpux, (state==RS_SUSPEND));
    }
}
```

In the algorithm, field `ldrexActive` corresponds to VMI register `CPUX_LDREX_FLAG`, and field `ldrexAddress` corresponds to VMI register `CPUX_LDREX_ADDRESS`.

The context-switch function `cpuxContextSwitchCB` is defined using the macro `VMI_IASSWITCH_FN`, define in file `vmiAttrs.h`:

```
#define VMI_IASSWITCH_FN(_NAME) void _NAME( \
    vmiProcessorP  processor,   \
    vmiIASRunState state        \
)
```

Parameter `state` indicates the new state of the processor:

```
typedef enum vmiIASRunStateE {
    RS_RUN,         // processor about to be run
    RS_SUSPEND      // processor has just been suspended
} vmiIASRunState;
```

When a processor is about to start its time slice, the context switch function is called with a state of `RS_RUN`; when it has completed its time slice, the context switch function is called with a state of `RS_SUSPEND`. The context-switch callback needs to be specified in the processor model attributes structure using the switchCB field:

```
const vmiIASAttr modelAttrs =  {

    ... fields omitted ...

    .switchCB = cpuxContextSwitchCB,

    ... fields omitted ...
};
```

24.4.4 Load/Store Exclusive with QuantumLeap

The preceding subsections describe how to implement load/store exclusive instructions in the absence of QuantumLeap simulation. When QuantumLeap simulation is enabled, the algorithm as described is no longer sufficient because it is no longer true that other processors are stopped while the load/store exclusive block is active.

To make the algorithm compatible with QuantumLeap, the only change required is *to indicate that the initial load exclusive instruction is atomic*: the simulator is then able to correctly detect writes to an exclusive block by other processors. The load-exclusive instruction therefore should be changed like this:

```
    // indicate load-exclusive is atomic
    vmimtAtomic();

    // load from address in ra
    vmimtLoadRRO(
        32, 32, 0, CPUX_REG(rd), CPUX_REG(ra), MEM_ENDIAN_BIG, True,
        MEM_CONSTRAINT_ALIGNED
    );

    // initiate load/store exclusive is active
    vmimtMoveRC(8, CPUX_LDREX_FLAG, 1);

    // save load/store exclusive address
    vmimtMoveRR(32, CPUX_LDREX_ADDRESS, CPUX_REG(ra));
```

24.5 Accessing Shared Register State

Apart from the explicit synchronization instructions described above, the only other area in which care needs to be taken when making models compatible with QuantumLeap is in *accesses to shared register state*. It is sometimes the case (particularly in multicore models) that a particular register is accessible to more than one core. If such a shared register is accessed in a read-modify-write fashion, or is updated non-atomically by several VMI morph-time calls in a single simulated instruction, then use `vmimtAtomic` to ensure that all other processors are stopped while the updates occur to prevent them from seeing invalid interim register state. Note that there is generally no need to use `vmimtAtomic` if the shared register is written or read in its entirety: such accesses will be atomic.

Take special care to ensure that embedded calls are not accessing dynamically-changing shared state in an uncontrolled way. If an embedded call needs to access such state, use `vmimtAtomic` together with `vmimtCall`/`vmimtCallResult` to ensure other processors are stopped while the call takes place. As mentioned above, many calls in the VMI API are *synchronizing*: that is, in a QuantumLeap simulation they will cause the current thread to suspend until all other asynchronously-executing threads have been safely stopped. This often means that embedded calls that access shared state are in fact implicitly synchronizing and do not need to be explicitly identified as such with `vmimtAtomic`.

24.6 Enabling QuantumLeap in a processor model

When the requirements described above have been met, the model must notify the simulator that it can support parallel simulation. To do this, set to `True` the `QLQualified` field in the vmiProcessorInfo structure for the model (see section 4.2.6.11). Note that unless this field is set, the simulator will not run in parallel mode.

25 Function Address Semihosting

Semihosting allows behavior that would normally occur on a simulated system to be implemented using features of the host system instead. As a simple example, a real platform might contain a UART peripheral to receive output. When simulating this system, it is generally more convenient not to simulate the UART at all but instead to intercept any `write` call that a processor makes and redirect the output to the simulator log instead.

This section will describe a semihosting support library for the OR1K processor when used with the popular Newlib library.

25.1 Interception

Semihosting is based on a more fundamental concept: *interception*. Using Imperas technology, it is possible to define *intercept libraries*, which are loadable shared objects (on Linux[16]) or dynamic linked libraries (on Windows).

The intercept library can specify alternate behavior for a particular instruction type (for example a `TRAP` or `SYS` instruction), or when execution reaches a particular address (for example, an interrupt vector address), or when a particular function is executed (for example, a call to `write`).

Interception requires no application image file modification or special application compilation modes (except that, for *function address interception*, the application must be compiled with function symbols present). There can be several intercept libraries available for use with a processor (for example, there might be a Newlib semihosting intercept library and a uClibc semihosting intercept library). It is even possible to cascade multiple intercept libraries for a single processor[17].

Intercept libraries exist entirely separately from the processor model. This means that you do not need access to the processor model source to create a new intercept library. To access the processor model registers, the intercept library uses the Debug interface, also used to support *gdb* RSP (see chapter 17). This of course implies that the Debug interface must be implemented as a prerequisite before an intercept library can be created.

25.2 The Template Semihosting Library

A template model for the OR1K Newlib semihosting intercept library can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/22.or1kSemiHosting
```

Take a copy of the template model:

[16] Imperas Professional tools are available on both Linux and Windows operating systems and fully support the OVP APIs.
[17] This feature is available only in the Imperas Professional tools.

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/22.or1kSemiHosting .
```

Compile the model, harness, semihost library and application using the `make` command:

```
cd 22.or1kSemiHosting
make
```

Note that the harness, semihost library and application can be compiled individually if required using these commands:

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

The processor model is the same as that described in example `18.or1kInstructionAttributes`, with one small change, described in section 25.4.

There is a new directory, `semihosting`, which contains the source file for the Newlib semihosting intercept library (`or1kNewlib.c`). This is compiled to a Linux shared library, `or1kNewlib.so`, or Windows dll, `or1kNewlib.dll`. File `or1kNewlib.c` is described in the next section.

25.3 File `semihosting/or1kNewlib.c`

Every intercept library has an object of type `vmiosAttr` describing the interceptions it performs. The structure type is defined in `vmiosAttrs.h`:

```
typedef struct vmiosAttrS {

    ////////////////////////////////////////////////////////////////////////
    // VERSION
    ////////////////////////////////////////////////////////////////////////

    const char           *versionString;    // version string
    vmiModelType          modelType;        // type of model (enum)
    vmiInterceptType      interceptType;    // required in an intercept library
    const char           *packageName;      // package name
    Uns32                 objectSize;       // size in bytes of VMIOS object

    ////////////////////////////////////////////////////////////////////////
    // MODEL STATUS
    ////////////////////////////////////////////////////////////////////////

    vmiVisibility         visibility;       // model visibility (enum)
    vmiReleaseStatus      releaseStatus;    // model release status (enum)

    ////////////////////////////////////////////////////////////////////////
    // SAVE/RESTORE ROUTINES
    ////////////////////////////////////////////////////////////////////////

    vmiosSaveStateFn      saveCB;           // model state save callback
    vmiosRestoreStateFn   restoreCB;        // model state restore callback
    Uns32                 srVersion;        // model save/restore version

    ////////////////////////////////////////////////////////////////////////
    // CONSTRUCTOR/DESTRUCTOR ROUTINES
```

```
    ////////////////////////////////////////////////////////////////////////

    vmiosConstructorFn    constructorCB;    // constructor
    vmiosPostSimulateFn   postSimulateCB;   // post-simulation, pre-destruction
    vmiosDestructorFn     destructorCB;     // destructor

    ////////////////////////////////////////////////////////////////////////
    // INSTRUCTION INTERCEPT ROUTINES
    ////////////////////////////////////////////////////////////////////////

    vmiosMorphFn          morphCB;          // morph override callback
    vmiosNextPCFn         nextPCCB;         // get next instruction address
    vmiosDisassFn         disCB;            // disassemble instruction

    ////////////////////////////////////////////////////////////////////////
    // FORMAL PARAMETERS iterators to find parameters accepted by this model
    ////////////////////////////////////////////////////////////////////////

    vmiosParamSpecFn      paramSpecsCB;     // callback for next formal param
    vmiosParamValueSizeFn paramValueSizeCB; // callback to get size of table

    ////////////////////////////////////////////////////////////////////////
    // Debugger assistance
    ////////////////////////////////////////////////////////////////////////

    vmiosStepLineBeginFn   stepLineBeginCB;  // callback to begin next operation
    vmiosStepLineIterateFn stepLineIterateCB;// callback to see if next is done

    ////////////////////////////////////////////////////////////////////////
    // Model VLNV
    ////////////////////////////////////////////////////////////////////////

    vmiVlnvInfo           vlnv;             // vendor/library/name/version

    ////////////////////////////////////////////////////////////////////////
    // ADDRESS INTERCEPT DEFINITIONS
    ////////////////////////////////////////////////////////////////////////

    vmiosInterceptDesc    intercepts[];     // null-terminated intercept list

} vmiosAttr;
```

25.3.1 OR1K Newlib Semihosting `vmiosAttr` Definition

For the OR1K Newlib semihosting intercept library, the `vmiosAttr` structure instance is as follows:

```
vmiosAttr modelAttrs = {

    ////////////////////////////////////////////////////////////////////////
    // VERSION
    ////////////////////////////////////////////////////////////////////////

    .versionString = VMI_VERSION,            // version string
    .modelType     = VMI_INTERCEPT_LIBRARY,  // model type
    .interceptType = VMI_IT_INTERCEPT,       // intercept type
    .packageName   = "Newlib",               // description
    .objectSize    = sizeof(vmiosObject),    // size in bytes of OSS object

    ////////////////////////////////////////////////////////////////////////
    // CONSTRUCTOR/DESTRUCTOR ROUTINES
    ////////////////////////////////////////////////////////////////////////
```

```
    .constructorCB = constructor,            // object constructor
    .destructorCB  = destructor,             // object destructor

    ////////////////////////////////////////////////////////////////////////
    // ADDRESS INTERCEPT DEFINITIONS
    ////////////////////////////////////////////////////////////////////////

    .intercepts    =
    {
        // ---------------- ---------- ----------- -----------------
        // Name             Address    Attributes   Callback
        // ---------------- ---------- ----------- -----------------
        { "_close",         0,         OSIA_OPAQUE, closeInt        },
        { "__exit",         0,         OSIA_OPAQUE, exitInt         },
        { "_fstat",         0,         OSIA_OPAQUE, fstatInt        },
        { "_gettimeofday",  0,         OSIA_OPAQUE, gettimeofdayInt },
        { "_ioctl",         0,         OSIA_OPAQUE, ioctlInt        },
        { "_lseek",         0,         OSIA_OPAQUE, lseekInt        },
        { "_lstat",         0,         OSIA_OPAQUE, lstatInt        },
        { "_open",          0,         OSIA_OPAQUE, openInt         },
        { "_read",          0,         OSIA_OPAQUE, readInt         },
        { "_stat",          0,         OSIA_OPAQUE, statInt         },
        { "_time",          0,         OSIA_OPAQUE, timeInt         },
        { "_times",         0,         OSIA_OPAQUE, timesInt        },
        { "_unlink",        0,         OSIA_OPAQUE, unlinkInt       },
        { "_write",         0,         OSIA_OPAQUE, writeInt        },
        { 0                                                         },
    }
};
```

In detail, the sections of this file are described below.

```
    .versionString = VMI_VERSION,            // version string
    .modelType     = VMI_INTERCEPT_LIBRARY,  // model type
    .interceptType = VMI_IT_INTERCEPT,       // intercept type
```

Each intercept library contains a reference to the current VMI version (from `vmiVersion.h`) and the type of model (in `vmiTypes.h`) so that the simulator can verify interface compatibility. `interceptType` controls when the library must be loaded and initialized; a processor extension at the same time as the processor model, an intercept library later during elaboration.

```
    .packageName   = "Newlib",               // description
```

This is a descriptive name (used for message reporting only).

```
    .objectSize    = sizeof(vmiosObject),    // size in bytes of OSS object
```

The intercept library defines a custom structure, used to hold all data required for an instance of that library; this defines the size of that structure so that it can be automatically allocated by the simulator when the library is instantiated. For the Newlib semihosting intercept library, the structure is defined like this:

```
#define FILE_DES_NUM 128
#define REG_ARG_NUM  3
```

```
typedef struct vmiosObjectS {

    // first few argument registers (standard ABI)
    vmiRegInfoCP args[REG_ARG_NUM];

    // return register (standard ABI)
    vmiRegInfoCP result;

    // stack pointer (standard ABI)
    vmiRegInfoCP sp;

    // __impure_ptr address and domain
    Addr       impurePtrAddr;
    memDomainP impurePtrDomain;

    // file descriptor table
    Int32 fileDescriptors[FILE_DES_NUM];

} vmiosObject;
```

(The fields of this structure will be covered in more detail on following sections.)

Next, the `vmiosAttr` structure contains references to *constructor* and *destructor* functions, called when an instance of the intercept library is created and destroyed, respectively:

```
    .constructorCB = constructor,            // object constructor
    .destructorCB  = destructor,             // object destructor
```

These functions will be covered in detail in a later subsection.

Finally, the `vmiosAttr` structure contains a list of *address intercept* definitions:

```
    .intercepts    =
    {
        // ---------------- ---------- ------------ -----------------
        // Name             Address    Attributes   Callback
        // ---------------- ---------- ------------ -----------------
        { "_close",         0,         OSIA_OPAQUE, closeInt        },
        { "__exit",         0,         OSIA_OPAQUE, exitInt         },
        { "_fstat",         0,         OSIA_OPAQUE, fstatInt        },
        { "_gettimeofday",  0,         OSIA_OPAQUE, gettimeofdayInt },
        { "_ioctl",         0,         OSIA_OPAQUE, ioctlInt        },
        { "_lseek",         0,         OSIA_OPAQUE, lseekInt        },
        { "_lstat",         0,         OSIA_OPAQUE, lstatInt        },
        { "_open",          0,         OSIA_OPAQUE, openInt         },
        { "_read",          0,         OSIA_OPAQUE, readInt         },
        { "_stat",          0,         OSIA_OPAQUE, statInt         },
        { "_time",          0,         OSIA_OPAQUE, timeInt         },
        { "_times",         0,         OSIA_OPAQUE, timesInt        },
        { "_unlink",        0,         OSIA_OPAQUE, unlinkInt       },
        { "_write",         0,         OSIA_OPAQUE, writeInt        },
        { 0                                                         },
    }
```

Each entry in this null-terminated table of intercept definitions is of type `vmiosInterceptDesc`:

```
typedef enum vmiosInterceptAttrE {
    OSIA_NONE   = 0x0,              // no special attributes
    OSIA_OPAQUE = 0x1,              // opaque intercept (otherwise transparent)
    OSIA_THREAD = 0x2,              // run in thread (otherwise synchronous)
} vmiosInterceptAttr;

typedef struct vmiosInterceptDescS {
    const char        *name;        // for interception by name
    Addr               simAddress;  // for interception by address
    vmiosInterceptAttr attrs;       // intercepted function attributes
    vmiosInterceptFn   interceptCB; // interception callback
    void              *userData;    // client-specific data pointer
    Bool               skipPrologue;// Use gdb to find the prologue
} vmiosInterceptDesc;
```

In detail, each entry has the following fields:

1. A function name: if non-`NULL`, this field specifies the *name of a function* in the application executable to which this row applies. For example, the first row in the Newlib semihosting intercept library applies to function `_close`.
2. A function address: if non-zero, this field specifies an *address* within the application executable to which this row applies. This field isn't used in this example, but is useful when interception of a known address, such as an exception handler address, is required.
3. A bitfield enumeration, `attrs`, comprised of bitwise-or of the following members:
   `OSIA_OPAQUE`: this indicates whether the action performed by this intercept should *replace* any default behavior specified by the processor model (if present) or be performed *in addition* to the default behavior specified by the processor model (if absent). Semihosting libraries in general specify replacement behaviors, so this field is usually present.
   `OSIA_THREAD`: this option is available only with Imperas Professional products. It indicates that the intercepted function should be run in a separate hardware thread, improving simulator performance.
4. An *intercept function*, which specifies the new behavior for the intercepted address. For example, the first row in the Newlib semihosting intercept library associates intercept function `closeInt` with the application function `_close`. Intercept functions are covered in more detail below.
5. A client-specific data pointer, passed as an argument to the intercept function. This is unused in the current example.
6. A Boolean, `skipPrologue`: if `False`, then the exact symbol address is intercepted; if `True`, then a gdb for the processor is invoked by the simulator to calculate the intercept address *after any function prologue*. This is not required for this example.

The template Newlib semihosting intercept library supplies opaque function address intercepts for each of the following functions in the application: `_close`, `__exit`, `_fstat`, `_gettimeofday`, `_ioctl`, `_lseek`, `_lstat`, `_open`, `_read`, `_stat`, `_time`, `_times`, `_unlink` and `_write`. Because these functions are all *opaquely* intercepted, whenever the simulator executes at any of these function addresses, it will perform actions specified by the corresponding intercept functions instead of the normal processor model behavior.

25.3.2 OR1K Newlib Semihosting Constructor Definition

The constructor for the OR1K Newlib semihosting intercept library has this definition:

```
static VMIOS_CONSTRUCTOR_FN(constructor) {

    Uns32 i;

    // first few argument registers (standard ABI)
    object->args[0] = vmiosGetRegDesc(processor, "R3");
    object->args[1] = vmiosGetRegDesc(processor, "R4");
    object->args[2] = vmiosGetRegDesc(processor, "R5");

    // return register (standard ABI)
    object->result = vmiosGetRegDesc(processor, "R11");

    // stack pointer (standard ABI)
    object->sp = vmiosGetRegDesc(processor, "R1");

    // __impure_ptr address
    object->impurePtrDomain = vmirtAddressLookup(
        processor, ERRNO_REF, &object->impurePtrAddr
    );

    // initialize stdin, stderr and stdout
    object->fileDescriptors[0] = vmiosGetStdin(processor);
    object->fileDescriptors[1] = vmiosGetStdout(processor);
    object->fileDescriptors[2] = vmiosGetStderr(processor);

    // initialize remaining file descriptors
    for(i=3; i<FILE_DES_NUM; i++) {
        object->fileDescriptors[i] = -1;
    }
}
```

The constructor first obtains *register description* objects that enable certain named registers with the processor model to be read and written. Because we are writing function address intercepts in this library, we need to be able to access several registers used in the standard processor ABI: the first few function argument registers (`R3`, `R4` and `R5` for the OR1K), the function result register (`R11` for the OR1K) and the stack pointer (`R1` for the OR1K):

```
    // first few argument registers (standard ABI)
    object->args[0] = vmiosGetRegDesc(processor, "R3");
    object->args[1] = vmiosGetRegDesc(processor, "R4");
    object->args[2] = vmiosGetRegDesc(processor, "R5");

    // return register (standard ABI)
    object->result = vmiosGetRegDesc(processor, "R11");

    // stack pointer (standard ABI)
    object->sp = vmiosGetRegDesc(processor, "R1");
```

The `vmiRegInfoCP` object returned by `vmiosGetRegDesc` is in fact a register descriptor supplied by the debug interface, created in chapter 17.

Next, to support correct error return from functions implemented in the Newlib library, the constructor obtains the address of a special symbol, `__impure_ptr`, that is always defined in any Newlib application:

```
    object->impurePtrDomain = vmirtAddressLookup(
        processor, ERRNO_REF, &object->impurePtrAddr
    );
```

Newlib allows reentrant calls, so that instead of having a single `errno` variable to signal library errors there is instead a pointer, `__impure_ptr`, that points to the current `errno` value to update. Hence, if we are to semihost a call that could set `errno`, we need to obtain the address in `__impure_ptr` to determine what errno address to write.

Finally, the constructor initializes a file descriptor map for the semihost intercept library. This maps from file numbers expected by the application to native file pointers. Files 0, 1 and 2 are the standard input, standard output and standard error, respectively:

```
    object->fileDescriptors[0] = vmiosGetStdin(processor);
    object->fileDescriptors[1] = vmiosGetStdout(processor);
    object->fileDescriptors[2] = vmiosGetStderr(processor);
```

Other files are initially closed:

```
    for(i=3; i<FILE_DES_NUM; i++) {
        object->fileDescriptors[i] = -1;
    }
```

> See the *VMI OS Support Function Reference* manual for more information about all functions with the `vmios` prefix.

25.3.3 OR1K Newlib Semihosting Destructor Definition

The destructor for the OR1K Newlib semihosting intercept library is currently a void function:

```
static VMIOS_DESTRUCTOR_FN(destructor) {
}
```

Typically, the destructor would be used to print out statistics gathered by the intercept library and free any temporary structures that were allocated.

25.3.4 Function Address Intercept Example: `closeInt`

To understand how to write a function address intercept callback, see function `closeInt` in `or1kNewlib.c`, which supplies alternate behavior to be performed when the application `_close` function is executed. This function should close a file descriptor passed as the only argument. It is implemented like this:

```
static VMIOS_INTERCEPT_FN(closeInt) {

    Int32 fd;
```

```
    // obtain function arguments
    getArg(processor, object, 0, &fd);

    // implement close
    Int32 fdMap  = mapFileDescriptor(processor, object, fd);
    Int32 result = vmiosClose(processor, fdMap);

    // null out the semihosted file descriptor if success
    if(!result) {
        object->fileDescriptors[fd] = -1;
    }

    // return result
    setErrnoAndResult(processor, object, result, context);
}
```

All function address intercept callbacks should be defined using the `VMIOS_INTERCEPT_FN` macro, defined in file `vmiOSAttrs.h`:

```
#define VMIOS_INTERCEPT_FN(_NAME) void _NAME( \
    vmiProcessorP processor,        \
    vmiosObjectP  object,           \
    Addr          thisPC,           \
    const char   *context,          \
    void         *userData,         \
    Bool          atOpaqueIntercept \
)
typedef VMIOS_INTERCEPT_FN((*vmiosInterceptFn));
```

The function address intercept callback is called at *morph time* and should use the VMI Morph Time Function API to generate code to implement required behavior. The callback is passed six arguments:

1. The processor that is about to execute code at the intercepted address;
2. The current function intercept object;
3. The intercepted address;
4. A context string, which gives the function name being intercepted (e.g. “`_close`”).
5. The client-specific data pointer associated with the row of the function address intercept table defining this interception.
6. An indication of whether the current address is already opaquely intercepted. This may be required when intercept libraries are cascaded – for example, an intercept of a function call may expect there to be a corresponding return later, but this won’t be the case if the call has been opaquely intercepted already.

This function first gets the first argument (argument 0) of _close, using a utility function `getArg`:

```
    Int32 fd;
    getArg(processor, object, 0, &fd);
```

getArg itself simply accesses a register value with the standard OR1K ABI, using function vmiosRegRead, which obtains the value of the register using the Debug interface:

```
static void getArg(
    vmiProcessorP processor,
    vmiosObjectP  object,
    Uns32         index,
    void         *result
) {
    if(index>=REG_ARG_NUM) {

        vmiMessage("P", "OR1K_ANS_NEWLIB",
            "No more than %u function arguments supported",
            REG_ARG_NUM
        );

        vmirtFinish(-1);

    } else {

        vmiosRegRead(processor, object->args[index], result);
    }
}
```

Next, closeInt calls another utility function, mapFileDescriptor, which translates from an application file number to the equivalent native one, and closes that file using the vmios-prefixed function vmiosClose:

```
    Int32 fdMap  = mapFileDescriptor(processor, object, fd);
    Int32 result = vmiosClose(processor, fdMap);
```

mapFileDescriptor is very simple, and uses the table of file descriptors in the intercept library instance object to perform the mapping:

```
static Int32 mapFileDescriptor(
    vmiProcessorP processor,
    vmiosObjectP  object,
    Uns32         i
) {
    if(i>=FILE_DES_NUM) {
        return -1;
    } else {
        return object->fileDescriptors[i];
    }
}
```

A mapping table used to translate from application to native file descriptors instead of using native descriptors directly for two reasons.
Firstly, it ensures that the application sees the full range of expected file numbers 0, 1, 2, … FILE_DES_NUM-1, irrespective of what native files are in use.
Secondly, it prevents badly-behaved simulated applications closing or otherwise modifying unrelated native files of which it should have no knowledge.
Always use a mapping table to translate from application to native file descriptors in semihosting intercept libraries.

If the file close succeeded, `closeInt` then nulls out the corresponding entry in the file descriptor table:

```
    if(!result) {
        object->fileDescriptors[fd] = -1;
    }
```

Finally, a utility function `setErrnoAndResult` is called. This does two things:

1. It updates the current `errno` so that it is consistent with the command just executed;
2. It assigns the result of `vmiosClose` to the appropriate result register in the OR1K ABI:

```
    setErrnoAndResult(processor, object, result, context);
```

`setErrnoAndResult` is defined as:

```
static void setErrnoAndResult(
    vmiProcessorP processor,
    vmiosObjectP  object,
    Int32         result,
    const char   *context
) {
    if(!object->impurePtrDomain) {

        vmiMessage("P", "OR1K_ICF_NEWLIB",
            "Interception of '%s' failed - %s not found "
            "(application does not appear to be compiled with newlib "
            "or has no symbols)",
            context, ERRNO_REF
        );

        vmirtFinish(-1);

    } else if(result<0) {

        memDomainP domain     = object->impurePtrDomain;
        memEndian  endian     = vmirtGetProcessorDataEndian(processor);
        Int32      errnoValue = -result;
        Uns32      impurePtrAddr;

        result = -1;

        // swap errno endianness if required
        if(endian != ENDIAN_NATIVE) {
            errnoValue = swap4(errnoValue);
        }

        // read __impure_ptr value
        vmirtReadNByteDomain(
            domain, object->impurePtrAddr, &impurePtrAddr,
            sizeof(impurePtrAddr), 0, False
        );

        // swap errno address endianness if required
        if(endian != ENDIAN_NATIVE) {
            impurePtrAddr = swap4(impurePtrAddr);
```

```
        }

        // write back errno
        vmirtWriteNByteDomain(
            domain, impurePtrAddr+OR1K_ERRNO_OFFSET, &errnoValue,
            sizeof(errnoValue), 0, True
        );
    }

    vmiosRegWrite(processor, object->result, &result);
}
```

`setErrnoAndResult` reads the address in the `__impure_ptr` variable using `vmirtReadNByteDomain` and stores the required `errno` value to this address using `vmirtWriteNByteDomain`. It also performs any endianness conversions required if (as for the OR1K) the simulated processor endianness differs from the host.

> The template Newlib semihosting intercept library has been designed to be relatively easy to port to any new processor: changes should be limited to the constructor (where details of the processor ABI will need to be encoded) and perhaps `getArg` (if parameter values are passed on the stack instead of in registers).

25.4 Semihosting Function Return - `or1kSemiHost.c`

The OR1K model contains one new file, `or1kSemiHost.c`, which implements a single function that is required to support opaque function interception: `or1kIntReturnCB`.

```
VMI_INT_RETURN_FN(or1kIntReturnCB) {
    vmimtUncondJumpReg(0, OR1K_REG(OR1K_LINK), VMI_NOREG, vmi_JH_RETURN);
}
```

This intercept return function generates code that forces a return from an opaquely-intercepted function address after one instruction. For the OR1K, this is done simply by jumping to the link address. *Note that this function does not implement any true processor behavior: it is required merely to allow opaque function intercepts to work – therefore, there should be no attempt to model true hardware features such as delay slots.*

The new function is prototyped in `or1kFunctions.h` and referenced in `or1kAttrs.h`:

```
const vmiIASAttr modelAttrs = {

    . . . lines omitted . . .

    ////////////////////////////////////////////////////////////////////////
    // IMPERAS INTERCEPTED FUNCTION SUPPORT ROUTINES
    ////////////////////////////////////////////////////////////////////////

    .intReturnCB = or1kIntReturnCB,

    . . . lines omitted . . .
};
```

25.5 File `platform/harness.c`

To use the new semihost library, `platform/harness.c` has been changed as follows:

```
    // create a processor instance
    const char *modelFile = "model."IMPERAS_SHRSUF;
    optProcessorP processor = opProcessorNew(mr, modelFile, "cpu1", 0, 0);

    // attach Newlib semihost library to processor
    const char *semihostFile = "semihosting/model."IMPERAS_SHRSUF;
    opProcessorExtensionNew(processor, semihostFile, "Newlib", 0);
```

25.6 Flow of Control for Opaque Address Intercepts

When an opaquely-intercepted function is encountered by the simulator, the flow of control will appear as follows:

1. The call to the intercepted address will be as normal.
2. The simulator will appear to execute one instruction at the intercepted function address. However, this single instruction will perform the entire behavior specified by the interception library for that address.
3. After the instruction completes, the processor will immediately return to the instruction after the call to the intercepted address – in other words, the entire functionality at the intercepted address will appear to have been replaced by a single instruction.

25.7 Testing the Semihosting Intercept Library

Run the platform using the C program executable file:

```
platform/harness.$IMPERAS_ARCH.exe –-trace –-traceshowicount \
    --program application/application.OR1K.elf
```

The output from this should be as follows:

```
Info 1: 'cpu1', 0x0000000000000100: l.addi   r2,r0,0x0
Info 2: 'cpu1', 0x0000000000000104: l.addi   r3,r0,0x0
Info 3: 'cpu1', 0x0000000000000108: l.addi   r4,r0,0x0
Info 4: 'cpu1', 0x000000000000010c: l.addi   r5,r0,0x0
Info 5: 'cpu1', 0x0000000000000110: l.addi   r6,r0,0x0
. . . lines omitted . . .
Info 340: 'platform/cpu1', 0x0000000000009854: l.ori    r3,r4,0x0
Info 341: 'platform/cpu1', 0x0000000000009858: l.addi   r4,r0,0x0
Info 342: 'platform/cpu1', 0x000000000000985c: l.sw     0x0(r10),r4
Info 343: 'platform/cpu1', 0x0000000000009860: l.jal    0x00009bc4
Info 344: 'platform/cpu1', 0x0000000000009864: l.ori    r4,r5,0x0
Info 345: 'platform/cpu1', 0x0000000000009bc4: *** INTERCEPT *** (_fstat)
Info 346: 'platform/cpu1', 0x0000000000009868: l.sfnei  r11,0xffffffff
. . . lines omitted . . .
Info 1394: 'platform/cpu1', 0x00000000000099b4: l.sw     0x0(r10),r4
Info 1395: 'platform/cpu1', 0x00000000000099b8: l.ori    r4,r5,0x0
Info 1396: 'platform/cpu1', 0x00000000000099bc: l.jal    0x00009c48
Info 1397: 'platform/cpu1', 0x00000000000099c0: l.ori    r5,r6,0x0
Info 1398: 'platform/cpu1', 0x0000000000009c48: *** INTERCEPT *** (_write)
main called
Info 1399: 'platform/cpu1', 0x00000000000099c4: l.sfnei  r11,0xffffffff
. . . lines omitted . . .
```

```
Info 2231: 'platform/cpu1', 0x0000000000001718: l.jal    0x00009b3c
Info 2232: 'platform/cpu1', 0x000000000000171c: l.ori    r3,r10,0x0
Info 2233: 'platform/cpu1', 0x0000000000009b3c: *** INTERCEPT *** (__exit)
processor has executed 2233 instructions
```

The application starts running normally. Then, after 345 instructions, there is a call to function `_fstat`, which is intercepted. The entire behavior of the intercepted `_fstat` appears to occur in a single instruction, and at instruction 346 execution has returned from `_fstat`. At instruction 1398, there is an intercepted call to `_write`, which actually writes the string `main called` to the standard output, before returning after one instruction. There are various other intercepted calls in this run, finishing with an intercepted call to `_exit` at instruction 2233, which terminates the current processor, ending simulation.

Note that intercepted calls are automatically reported in the trace output to make it clear where behavior is deviating from the standard processor model.

25.8 Intercepts and Multicore Processors

It is possible to create multicore processors using the VMI API (see *SMP Processor Hierarchies* in the *VMI Run Time Function Reference* document for more information). When such processors are instantiated, a hierarchy of `optProcessorP` objects is created beneath a container `optProcessorP` root object (or `icmProcessorP` objects, when using the legacy ICM API). For example, instantiating an ARM Cortex-A57MPx4 processor variant creates a single `optProcessorP` root object representing the container, plus four further `optProcessorP` objects linked as children of this, one for each of the cores in the container processor. In this case, using the OP API, it is possible to get the first leaf core by calling function `opProcessorChild` on the root processor, and each subsequent leaf by applying `opProcessorSiblingNext` to the previous leaf core. Note that, in general, there could be more than two levels of hierarchy (for example OVP MIPS processor models can have to four hierarchy levels).

When installing intercept libraries on such a multicore processor, it is possible to do so *at any level in the hierarchy*: the rule is that an intercept library applies to *all leaf cores below that level*. For example, installing an intercept library on the Cortex-A57MPx4 root object would cause it to be instanced on each of the four leaf cores, but installing it on a leaf would apply to that leaf only.

When intercept libraries are instanced hierarchically, the default behavior is to allocate a `vmiosObject` for *each leaf level core*, and perform all the steps described in previous sections (for example, execute the constructor, morpher function, destructor, and so on) for each of those leaf levels independently. If (for example) an intercept library was installed at the root level of a Cortex-A57MPx4, the effect would be equivalent to installing the same intercept library on each of the leaf cores independently.

Sometimes, it is required that intercept libraries at leaf levels are not entirely independent, but instead are able to communicate. The default behavior described above can be overridden using the `allLevels` member of the `vmiosAttr` structure:

```
typedef struct vmiosAttrS {

    ////////////////////////////////////////////////////////////////////////
    // VERSION & TYPE
    ////////////////////////////////////////////////////////////////////////

    const char          *versionString;       // version string (THIS MUST BE FIRST)
    vmiModelType         modelType;           // type of model (enum)
    vmiInterceptType     interceptType;       // required in an intercept library
    const char          *packageName;         // package name
    Uns32                objectSize;          // size in bytes of VMIOS object
    Bool                 allLevels;           // apply at all levels (leaf if False)

    . . . fields omitted for clarity

} vmiosAttr;
```

When `allLevels` is set to `True` in a `vmiosAttr` structure definition, and that intercept library is applied to a non-leaf level of the hierarchy, the behavior is as follows.

1. a `vmiosObject` is allocated for the top-level processor first (remember that this is not the case when `allLevels` is set to `False`: with that setting, structures are allocated only at leaf levels);
2. The constructor at the top level is called.
3. A second `vmiosObject` is allocated for the *first child* of the top-level processor, and the constructor is called for that object. The constructor in this case is passed a `parent` argument, which is a pointer to the previously-allocated top-level `vmiosObject`.
4. Object allocation continues as the processor hierarchy is covered in an in-order traversal.

To clarify, the `vmiosConstructorFn` function type is defined in `vmiOSAttrs.h` like this:

```
#define VMIOS_CONSTRUCTOR_FN(_NAME) void _NAME( \
    vmiProcessorP processor,      \
    vmiosObjectP  object,         \
    vmiosObjectP  parent,         \
    void         *parameterValues \
)
typedef VMIOS_CONSTRUCTOR_FN((*vmiosConstructorFn));
```

The second and third arguments are the current object and parent object, respectively. Typically, the constructor will set up any required intercept object hierarchy; the example below initializes a `parent` member on each `vmiosObject` structure for later use, and also a `root` member that holds the `vmiosObject` at the highest hierarchy level:

```
typedef struct vmiosObjectS {
    vmiosObjectP root;
    vmiosObjectP parent;
} vmiosObject;

static VMIOS_CONSTRUCTOR_FN(constructor) {

    // save root on object
    object->root = parent ? parent->root : object;
```

```
    // save parent on object
    object->parent = parent;
}
```

These members can be used at run time to access data that needs to be *shared* across intercept library instances.

26 Using Intercept Libraries for Instruction Set Enhancement

The previous chapter described how to use Imperas intercept library technology to implement semihosting libraries. When using Imperas Professional products, it is also possible to use this technology to implement enhancements to processor model instruction sets, registers or ports, *even when the source of the processor model is unavailable*. When used in this way, intercept libraries are referred to as *extension libraries*.

Note that this feature is not available in OVPsim.

26.1 The Template Instruction Set Enhancement Library

A template model for the OR1K processor with its instruction set enhanced by an external intercept library can be found at:

```
$IMPERAS_HOME/Examples/Models/Processor/23.or1kExchange
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/23.or1kExchange .
```

Compile the model, harness and application using the `make` command:

```
cd 23.or1kExchange
make
```

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is the same as that described in example `22.or1kSemiHosting`.

There is a new directory, `exchange`, which contains the source file for the enhanced instruction set extension library. This is compiled to a Linux shared library, `model.so`, or Windows dll, `model.dll`. File `or1kExchange.c` is described in the next section.

26.2 File exchange/or1kExchange.c

The standard OR1K processor contains no single instruction that enables a register to be exchanged with a memory location. It might be reasonable to want to enhance the instruction set to provide this capability, if the OR1K is to be used in a multiprocessor platform.

This example adds an exchange instruction using an intercept library, so that the core functionality of the basic model is unaffected. As well as a new instruction, the example also adds four new registers, as follows:

`exchCount`: counts the number of exchange instructions executed.
`exchAddress`: records the memory address used by the previous exchange.
`exchRDValue`: records the value *read* by the previous exchange.
`exchAddress`: records the value *written* by the previous exchange.

The `vmiosAttr` structure implementing the additional exchange instruction is defined as follows:

```
vmiosAttr modelAttrs = {

    ////////////////////////////////////////////////////////////////////////
    // VERSION
    ////////////////////////////////////////////////////////////////////////

    .versionString = VMI_VERSION,           // version string
    .modelType     = VMI_INTERCEPT_LIBRARY, // model type
    .interceptType = VMI_IT_PROC_EXTENSION, // type of intercept
    .packageName   = "Exchange",            // description
    .objectSize    = sizeof(vmiosObject),   // size in bytes of OSS object

    ////////////////////////////////////////////////////////////////////////
    // CONSTRUCTOR/DESTRUCTOR ROUTINES
    ////////////////////////////////////////////////////////////////////////

    .constructorCB = constructor,           // object constructor

    ////////////////////////////////////////////////////////////////////////
    // INSTRUCTION INTERCEPT ROUTINES
    ////////////////////////////////////////////////////////////////////////

    .morphCB       = exchangeMorph,         // morph callback
    .nextPCCB      = exchangeNextPC,        // get next instruction address
    .disCB         = exchangeDisass,        // disassemble instruction

    ////////////////////////////////////////////////////////////////////////
    // DEBUGGER INTEGRATION SUPPORT ROUTINES
    ////////////////////////////////////////////////////////////////////////

    .regGroupCB    = exchangeIterRegGroup,  // regGroup callback
    .regInfoCB     = exchangeIterRegInfo,   // regInfo callback

    ////////////////////////////////////////////////////////////////////////
    // PORT ACCESS ROUTINES
    ////////////////////////////////////////////////////////////////////////

    .netPortSpecsCB = exchangeIterNetPorts, // net port callback

    ////////////////////////////////////////////////////////////////////////
    // ADDRESS INTERCEPT DEFINITIONS
    ////////////////////////////////////////////////////////////////////////
    .intercepts    = {{0}}
};
```

When implementing supplementary instructions in an extension library, a *morpher callback*, a *disassembly callback* and *a next PC* callback are required. If the extension library implements new registers, a *register group iterator* callback and a *register iterator callback* are required. This extension library also implements a net port exposing the value of the `exchCount` register can be externally; the net port is defined using a *net port iterator*. Note that it is also possible to implement *bus* and *fifo* ports if required, using similar iterators.

Address intercepts are not required, so the address intercept table for this extension library is empty.

26.2.1 OR1K Newlib Semihosting `vmiosAttr` Definition

The `vmiosObject` for this extension library is defined as follows:

```
#define OR1K_GPR_NUM 32

typedef struct vmiosObjectS {

    // handles for the OR1K GPRs
    vmiRegInfoCP or1kRegs[OR1K_GPR_NUM];

    // enhanced instruction decode table
    vmidDecodeTableP table;

    // new 32-bit registers implemented by this extension library
    Uns32 exchCount;
    Uns32 exchAddress;
    Uns32 exchRDValue;
    Uns32 exchWRValue;

    // 32-bit temporaries implemented by this extension library
    Uns32 exchTmp;

    // net written with value of exchCount
    Uns32 exchCountNetHandle;

    // descriptions of registers and register groups
    vmiRegGroup localGroups[LR_EXCH_GROUP_NUM];
    vmiRegInfo  localRegs[LR_EXCH_NUM];

    // descriptions of nets
    vmiNetPort  localNetPorts[LR_EXCH_NET_NUM];

} vmiosObject;
```

Field `or1kRegs` is initialized in the constructor to hold a `vmiRegInfo` handle to each of the OR1K GPRs. Field `table` is also initialized in the constructor, and holds a decode table to decode supplementary instructions implemented by this intercept library.

Fields `exchCount`, `exchAddress`, `exchRDValue` and `exchWRValue` hold the values of extension registers implemented in this extension library. Field `exchTmp` is used to hold a temporary value used during execution of an exchange instruction. Field `exchCountNetHandle` is used to hold a handle to a net exposing the value of `exchCount` externally.

Field `localGroups` defines register groups implemented by this model; in this case, there is a single group, identified by the member `LR_EXCH_GROUP` of the `localGroup` enumeration:

```
typedef enum localGroupE {
    LR_EXCH_GROUP,      // register counting number of exchanges
    LR_EXCH_GROUP_NUM,  // KEEP LAST: for sizing
} localGroup;
```

Field `localRegs` defines registers implemented by this model; in this case, there are four registers, identified in the `localReg` enumeration:

```
typedef enum localRegE {
    LR_EXCH_COUNT,      // register counting number of exchanges
    LR_EXCH_ADDR,       // register recording last exchange address
    LR_EXCH_RD,         // register recording last exchange value read
    LR_EXCH_WR,         // register recording last exchange value written
    LR_EXCH_NUM,        // KEEP LAST: for sizing
} localReg;
```

Field `localNetPorts` defines net ports implemented by this model; in this case, there is one net port, identified in the `localNet` enumeration:

```
typedef enum localNetE {
    LR_EXCH_NET,        // net to which exchCount is written
    LR_EXCH_NET_NUM,    // KEEP LAST: for sizing
} localNet;
```

26.2.2 Constructor Definition

The constructor has this definition:

```
static VMIOS_CONSTRUCTOR_FN(constructor) {

    Uns32 i;

    // get handles to the OR1K GPRs
    for(i=0; i<OR1K_GPR_NUM; i++) {
        char regName[8];
        sprintf(regName, "R%u", i);
        object->or1kRegs[i] = vmiosGetRegDesc(processor, regName);
    }

    // create enhanced instruction decoder
    object->table = createDecodeTable();

    // fill local group descriptors
    for(i=0; i<LR_EXCH_GROUP_NUM; i++) {

        vmiRegGroupP this = &object->localGroups[i];

        // this gives group names for exchange unit register groups
        static const char *localGroupNames[] = {
            [LR_EXCH_GROUP] = "Exchange_Unit",
        };

        this->name = localGroupNames[i];
```

```
    }

    // type used to describe extension registers
    typedef struct extRegDescS {
        const char *name;           // register name
        const char *description;    // register description
        Uns32      *value;          // pointer to register value
    } extRegDesc, *extRegDescP;

    // describe extension registers
    extRegDesc extRegs[] = {
        [LR_EXCH_COUNT] = {
            name        : "exchCount",
            description : "count of exchange instructions executed",
            value       : &object->exchCount
        },
        [LR_EXCH_ADDR] = {
            name        : "exchAddress",
            description : "last exchange address",
            value       : &object->exchAddress
        },
        [LR_EXCH_RD] = {
            name        : "exchRDValue",
            description : "last value read by exchange",
            value       : &object->exchRDValue
        },
        [LR_EXCH_WR] = {
            name        : "exchWRValue",
            description : "last value written by exchange",
            value       : &object->exchWRValue
        },
    };

    // fill local register descriptors
    for(i=0; i<LR_EXCH_NUM; i++) {

        vmiRegInfoP this = &object->localRegs[i];
        extRegDescP desc = &extRegs[i];

        this->name        = desc->name;
        this->description = desc->description;
        this->group       = &object->localGroups[LR_EXCH_GROUP];
        this->gdbIndex    = LOCAL_INDEX+i;
        this->access      = vmi_RA_RW;
        this->bits        = 32;
        this->raw         = vmimtGetExtReg(processor, desc->value);
    }

    // fill local net port descriptors
    {
        vmiNetPortP this = &object->localNetPorts[0];

        this->name   = "exchCountNet";
        this->type   = vmi_NP_OUTPUT;
        this->handle = &object->exchCountNetHandle;
    }
}
```

The constructor first obtains *register description* objects for each of the OR1K general-purpose registers, and saves them in the `or1kRegs` array for later use:

```
    for(i=0; i<OR1K_GPR_NUM; i++) {
```

```
        char regName[8];
        sprintf(regName, "R%u", i);
        object->or1kRegs[i] = vmiosGetRegDesc(processor, regName);
    }
```

Then it creates a decoder table to decode extra instructions implemented by this library:

```
    object->table = createDecodeTable();
```

Function `createDecodeTable` uses the standard VMI *decoder* API functions to create a new decode table for a single exchange instruction:

```
static vmidDecodeTableP createDecodeTable(void) {

    vmidDecodeTableP table = vmidNewDecodeTable(OR1K_BITS, OR1K_EIT_LAST);

    // handle exchange instruction
    DECODE_ENTRY(0, EXW, "|111101..........................|");

    return table;
}
```

We have selected an arbitrary unused instruction prefix for the new instruction. Refer to chapter 5 for detailed information about the VMI decoder function API.

Next, the constructor initializes information about the register groups that the model implements:

```
    // fill local group descriptors
    for(i=0; i<LR_EXCH_GROUP_NUM; i++) {

        vmiRegGroupP this = &object->localGroups[i];

        // this gives group names for exchange unit register groups
        static const char *localGroupNames[] = {
            [LR_EXCH_GROUP] = "Exchange_Unit",
        };

        this->name = localGroupNames[i];
    }
```

In this case, the single group is called `Exchange_Unit`.

Next, the constructor defines a local array of structures (`extRegs`) giving the name, description and address in the extension library object of each of the four extension registers that it implements:

```
    // type used to describe extension registers
    typedef struct extRegDescS {
        const char *name;           // register name
        const char *description;    // register description
        Uns32      *value;          // pointer to register value
    } extRegDesc, *extRegDescP;

    // describe extension registers
    extRegDesc extRegs[] = {
```

```
        [LR_EXCH_COUNT] = {
            name        : "exchCount",
            description : "count of exchange instructions executed",
            value       : &object->exchCount
        },
        [LR_EXCH_ADDR] = {
            name        : "exchAddress",
            description : "last exchange address",
            value       : &object->exchAddress
        },
        [LR_EXCH_RD] = {
            name        : "exchRDValue",
            description : "last value read by exchange",
            value       : &object->exchRDValue
        },
        [LR_EXCH_WR] = {
            name        : "exchWRValue",
            description : "last value written by exchange",
            value       : &object->exchWRValue
        },
    };
```

Next, the constructor uses the `extRegs` array to populate a list of `vmiRegInfo` register descriptions about the registers that it implements:

```
    // fill local register descriptors
    for(i=0; i<LR_EXCH_NUM; i++) {

        vmiRegInfoP this = &object->localRegs[i];
        extRegDescP desc = &extRegs[i];

        this->name        = desc->name;
        this->description = desc->description;
        this->group       = &object->localGroups[LR_EXCH_GROUP];
        this->gdbIndex    = LOCAL_INDEX+i;
        this->access      = vmi_RA_RW;
        this->bits        = 32;
        this->raw         = vmimtGetExtReg(processor, desc->value);
    }
```

Here, each of the four local registers are given a name and description extracted from the `extRegs` array. Each register is specified to be a member of the new group, initialized previously. The registers are defined to have indices 0x1000-0x1003, and are all described as read/write and of 32-bit size. Finally, the `raw` field is set to a `vmiReg` value returned from function `vmimtGetExtReg` (defined in header file `vmiMt.h`). This function creates a register descriptor that targets a given address when in a given processor context. In this case, we use it to inform the simulator that the values of the new registers can be found in the `exchCount`, `exchAddress`, `exchRDValue` and `exchWRValue` fields in the extension library object.

See chapter 17 for more detailed information about how registers are described using `vmiRegInfo` objects.

Finally, the constructor initializes the single net port implemented by this extension library:

```
    // fill local net port descriptors
    {
        vmiNetPortP this = &object->localNetPorts[0];

        this->name   = "exchCountNet";
        this->type   = vmi_NP_OUTPUT;
        this->handle = &object->exchCountNetHandle;
    }
```

See chapter 16 for more detailed information about how net ports are described using `vmiNetPort` objects.

26.2.3 The Morpher Callback: `exchangeMorph`

This example implements a *morpher callback function* that specifies behavior for the new exchange instruction. It is defined as follows:

```
static VMIOS_MORPH_FN(exchangeMorph) {

    // decode the instruction to get the type
    UnsPS                 instruction = vmicxtFetch4Byte(processor, thisPC);
    or1kEnhancedInstrType type        = vmidDecode(object->table, instruction);

    if(type==OR1K_EIT_EXW) {

        // instruction is enhanced exchange
        emitExchange(processor, object, instruction);

        // indicate standard behavior has been replaced
        *opaque = True;
    }

    // no intercept callback specified
    return 0;
}
```

The intercept library morpher callback is called *before* the standard processor model callback. It first decodes the instruction at the current program counter address:

```
    UnsPS                 instruction = vmicxtFetch4Byte(processor, thisPC);
    or1kEnhancedInstrType type        = vmidDecode(object->table, instruction);
```

If the instruction is the new exchange instruction, the function then calls a subfunction, `emitExchange`, which uses morph-time primitives to describe the instruction behavior (see the next subsection):

```
    if(type==OR1K_EIT_EXW) {

        // instruction is enhanced exchange
        emitExchange(processor, object, instruction);
```

It then indicates that the extension instruction is an *opaque* intercept:

```
        *opaque = True;
```

What this means is that the behavior in the extension library will *replace* any default behavior in the processor model - if `opaque` was instead set to `False`, then this would be a *transparent* intercept, and behavior specified in the extension library would be performed *in addition* to the standard behavior in the processor model.

The function finally returns a `NULL` pointer value, indicating that no call to a supplemental intercept function is required (all behavior is implemented by morph-time calls emitted previously).

26.2.4 The Exchange Instruction Morpher Callback: `emitExchange`

Code to implement the exchange instruction is emitted by function `emitExchange`, defined as follows:

```
static void emitExchange(
    vmiProcessorP processor,
    vmiosObjectP  object,
    Uns32         instruction
) {
    // get processor endianness for loads and stores
    memEndian     endian     = vmirtGetProcessorDataEndian(processor);
    memConstraint constraint = MEM_CONSTRAINT_ALIGNED;

    // extract instruction fields
    Uns32 ra = OPEX_A(instruction);
    Uns32 rb = OPEX_B(instruction);
    Int16 i  = OPEX_I(instruction);

    // create vmiReg objects addressing extension registers and temporaries
    // from processor context
    vmiReg exchCount   = vmimtGetExtReg (processor, &object->exchCount);
    vmiReg exchAddress = vmimtGetExtReg (processor, &object->exchAddress);
    vmiReg exchRDValue = vmimtGetExtReg (processor, &object->exchRDValue);
    vmiReg exchWRValue = vmimtGetExtReg (processor, &object->exchWRValue);
    vmiReg exchTmp     = vmimtGetExtTemp(processor, &object->exchTmp);

    // increment count of exchange instructions executed
    vmimtBinopRC(32, vmi_ADD, exchCount, 1, 0);

    // copy rb and ra processor GPRs to exchWRValue and exchAddress
    vmimtGetR(processor, 32, exchWRValue, object->or1kRegs[rb]);
    vmimtGetR(processor, 32, exchAddress, object->or1kRegs[ra]);

    // adjust address, including constant offset
    vmimtBinopRC(32, vmi_ADD, exchAddress, i, 0);

    // load exchTmp from exchAddress
    vmimtLoadRRO(32, 32, 0, exchTmp, exchAddress, endian, False, constraint);

    // store exchWRValue to exchAddress
    vmimtStoreRRO(32, 0, exchAddress, exchWRValue, endian, constraint);

    // copy exchTmp to exchRDValue
    vmimtMoveRR(32, exchRDValue, exchTmp);

    // copy exchTmp to processor GPR
    vmimtSetR(processor, 32, object->or1kRegs[rb], exchTmp);

    // write exchCountNet if required
```

```
    if(object->exchCountNetHandle) {
        vmimtArgProcessor();
        vmimtArgUns32(object->exchCountNetHandle);
        vmimtArgReg(32, exchCount);
        vmimtCallResultAttrs(
            (vmiCallFn)vmirtWriteNetPort, 0, VMI_NOREG, VMCA_NA
        );
    }
}
```

Because this instruction is going to access processor memory, the callback first gets the processor endianness, and specifies how misaligned load/store addresses should be handled:

```
    memEndian     endian     = vmirtGetProcessorDataEndian(processor);
    memConstraint constraint = MEM_CONSTRAINT_ALIGNED;
```

Next, it uses functions `vmimtGetExtReg` and `vmimtGetExtTemp` to obtain `vmiReg` descriptors for extension library structure fields that implement extension registers and temporaries that will be used in this instruction:

```
    vmiReg exchCount   = vmimtGetExtReg (processor, &object->exchCount);
    vmiReg exchAddress = vmimtGetExtReg (processor, &object->exchAddress);
    vmiReg exchRDValue = vmimtGetExtReg (processor, &object->exchRDValue);
    vmiReg exchWRValue = vmimtGetExtReg (processor, &object->exchWRValue);
    vmiReg exchTmp     = vmimtGetExtTemp(processor, &object->exchTmp);
```

The first action of the instruction is to increment `exchCount`, which records the number of times the instruction has been executed:

```
    vmimtBinopRC(32, vmi_ADD, exchCount, 1, 0);
```

Next, function `vmimtGetR` is used to copy current values from OR1K GPRs into extension library registers:

```
    vmimtGetR(processor, 32, exchWRValue, object->or1kRegs[rb]);
    vmimtGetR(processor, 32, exchAddress, object->or1kRegs[ra]);
```

This is required because the extension library cannot directly access OR1K registers: it has no visibility of OR1K data structures. Having retrieved the base address register, code is then emitted to add the constant offset (`i`) encoded in the instruction to form the full target address:

```
    vmimtBinopRC(32, vmi_ADD, exchAddress, i, 0);
```

Next, code is emitted to load the value at the target address into a temporary:

```
    vmimtLoadRRO(32, 32, 0, exchTmp, exchAddress, endian, False, constraint);
```

The next step is to emit code to store the value from the OR1K GPR indexed by `rb` to the same address:

```
    vmimtStoreRRO(32, 0, exchAddress, exchWRValue, endian, constraint);
```

Next, the value previously loaded into a temporary is copied to both the OR1K GPR indexed by `rb` and the extension library `exchRDValue` register. To access the OR1K GPR, function `vmimtSetR` is used; note that this is required because the extension library cannot directly access OR1K registers:

```
    vmimtMoveRR(32, exchRDValue, exchTmp);
    vmimtSetR(processor, 32, object->or1kRegs[rb], exchTmp);
```

> Note that the value loaded from memory is first assigned only to a temporary, and committed to the result register only when the instruction completes. This means that register state will not be updated if the store of the new value causes an exception (for example, because the targeted address is read-only).

The final step is to write the new value of `exchCount` to the `exchCountNet` port, if that port is connected:

```
    if(object->exchCountNetHandle) {
        vmimtArgProcessor();
        vmimtArgUns32(object->exchCountNetHandle);
        vmimtArgReg(32, exchCount);
        vmimtCallResultAttrs(
            (vmiCallFn)vmirtWriteNetPort, 0, VMI_NOREG, VMCA_NA
        );
    }
```

26.2.5 The Next Instruction Callback: exchangeNextPC

This example also implements a *next PC callback function* that specifies the next instruction address after for the new exchange instruction. It is defined as follows:

```
//
// Return instruction address after passed program counter
//
static VMIOS_NEXT_PC_FN(exchangeNextPC) {

    // decode the instruction to get the type
    Uns32                 instruction = vmicxtFetch4Byte(processor, thisPC);
    or1kEnhancedInstrType type        = vmidDecode(object->table, instruction);

    if(type==OR1K_EIT_EXW) {
        *nextPC = thisPC+4;
        return True;
    } else {
        return False;
    }
}
```

The callback decodes the instruction at the passed address. If it is the exchange instruction, it sets the `nextPC` byref argument to the address of the instruction following the instruction and returns `True`; otherwise, it returns `False`.

The next instruction address callback is in fact only required if *the next instruction address differs from the address that would be calculated by the base model*, which is not the case for the OR1K model (since all instructions are four bytes long). The callback could therefore have been omitted for this example (and specified as 0 in the attribute structure). It has been specified in this case for example purposes only.

26.2.6 The Disassembler Callback: `exchangeDisass`

This example also implements a *disassembler callback function* that specifies disassembly for the new exchange instruction. It is defined as follows:

```
static VMIOS_DISASSEMBLE_FN(exchangeDisass) {

    Uns32                 instruction = vmicxtFetch4Byte(processor, thisPC);
    or1kEnhancedInstrType type        = vmidDecode(object->table, instruction);

    if(type==OR1K_EIT_EXW) {

        static char buffer[256];

        // extract instruction fields
        Uns32 ra = OPEX_A(instruction);
        Uns32 rb = OPEX_B(instruction);
        Int16 i  = OPEX_I(instruction);

        sprintf(buffer, "%-8s 0x%x(r%u),r%u", "l.exw", i, ra, rb);

        return buffer;

    } else {

        return 0;
    }
}
```

Once again, the callback first decodes the instruction at the passed address:

```
    Uns32                 instruction = vmicxtFetch4Byte(processor, thisPC);
    or1kEnhancedInstrType type        = vmidDecode(object->table, instruction);
```

If the instruction is the new exchange instruction, a disassembly string is created in a static buffer and returned. The general approach is just the same as for standard instruction disassembly callbacks; refer to chapter 6 for more details. We have assumed that the new instruction takes arguments in the same format as the existing `l.sw` (store word) instruction:

```
    if(type==OR1K_EIT_EXW) {

        static char buffer[256];

        // extract instruction fields
        Uns32 ra = OPEX_A(instruction);
        Uns32 rb = OPEX_B(instruction);
        Int16 i  = OPEX_I(instruction);

        sprintf(buffer, "%-8s 0x%x(r%u),r%u", "l.exw", i, ra, rb);
```

```
        return buffer;
```

If the new instruction is *not* the new exchange instruction, the function returns a null pointer to indicate that standard processor model disassembly should be performed.

26.2.7 The Register Group Iterator Callback: `exchangeIterRegGroup`

This example implements a *register group iterator callback function* that enables register groups defined in the extension library to be iterated. It is defined as follows:

```
static VMIOS_REG_GROUP_FN(exchangeIterRegGroup) {

    vmiRegGroupP first = object->localGroups;

    if(!prev) {
        return first;
    } else {
        localGroup nextIndex = (prev-first)+1;
        return nextIndex<LR_EXCH_GROUP_NUM ? first+nextIndex : 0;
    }
}
```

Given an argument of type `vmiRegGroupCP` (i.e. a pointer to a member of the `localGroups` array) this function should return the *next* register group description in the array, or `NULL` if there are no more register group descriptions. If called with a `NULL` argument, it should return the first register group description in the array.

26.2.8 The Register Iterator Callback: `exchangeIterRegInfo`

This example implements a *register iterator callback function* that enables registers defined in the extension library to be iterated. It is defined as follows:

```
static VMIOS_REG_INFO_FN(exchangeIterRegInfo) {

    vmiRegInfoP first = object->localRegs;

    if(type!=VMIRIT_NORMAL) {
        return 0;
    } else if(!prev) {
        return first;
    } else {
        localReg nextIndex = (prev-first)+1;
        return nextIndex<LR_EXCH_NUM ? first+nextIndex : 0;
    }
}
```

Given an argument of type `vmiRegInfoCP` (i.e. a pointer to a member of the `localRegs` array) this function should return the *next* register description in the array, or `NULL` if there are no more register group descriptions. If called with a `NULL` argument, it should return the first register description in the array.

26.3 The Harness File, `platform/harness.c`

The platform is similar to previous examples. One new line is included to install the new intercept library on the processor model using function `opProcessorExtensionNew`:

```
    // attach exchange instruction semihost library to processor
    const char *exchangeFile = "exchange/model."IMPERAS_SHRSUF;
    opProcessorExtensionNew(processor, exchangeFile, "exchange", 0);
```

Note that the new intercept library is installed *in addition* to the standard semihost library on the processor model itself. This ability to install multiple intercept libraries is only available with the Imperas Professional products.

Lines have also been included to connect to the `exchCountNet` port on the extension library, and install a callback to monitor changes on that net:

```
    ////////////////////////////////////////////////////////////////////////////
    // ADD NET MONITOR OF EXCHANGE COUNTER
    ////////////////////////////////////////////////////////////////////////////

    // create exchCountNet net
    optNetP exchCountNet = opNetNew(mr, "exchCountNet", 0, 0);

    // connect exchCountNet net to the processor
    opObjectNetConnect(processor, exchCountNet, "exchCountNet");

    // install callback monitoring net
    opNetWriteMonitorAdd(exchCountNet, exchCountMonitor, 0);

    ////////////////////////////////////////////////////////////////////////////
```

Note that from the OP interface, the net port defined in the extension library appears just like other processor ports. The callback function is defined as follows:

```
static OP_NET_WRITE_FN(exchCountMonitor) {
    opPrintf(">>> exchCount net written with %u\n", value);
}
```

26.4 Testing the Intercept Library

Run the platform using the assembler program executable file:

```
platform/harness.$IMPERAS_ARCH.exe –-trace –-tracechange --traceshowicount\
    --program application/asmtest.OR1K.elf
```

The example is a very simple one: it saves a value on the stack and then uses the new exchange instruction to exchange that with a value in a register. Then, it makes a second call to exchange a new value with the previously-saved value. Because the standard assembler does not know about the new exchange instruction, it is specified as a raw bit pattern using the `.word` assembler directive:

```
        ////////////////////////////////////////////////////////////////////////
        // MAIN ROUTINE
        ////////////////////////////////////////////////////////////////////////
.global _start
_start:
        l.addi      r31,r0,0        // initialize stack pointer to 0
        l.addi      r1,r0,0x100     // r1 = 0x100
        l.sw        -4(r31),r1      // save r1 value on stack
```

```
        l.addi      r1,r0,0x200      // r1 = 0x200
        .word       0xf7ff0ffc       // exch r1,-4(r31)
        l.lwz       r2,-4(r31)       // get current value of -4(r31)
        l.addi      r1,r0,0x300      // r1 = 0x300
        .word       0xf7ff0ffc       // exch r1,-4(r31)
        l.lwz       r2,-4(r31)       // get current value of -4(r31)
        ////////////////////////////////////////////////////////////////////////
        // EXIT FROM POINT TEST
        ////////////////////////////////////////////////////////////////////////
.global exit
exit:
        l.nop
```

The output from this should be as follows:

```
Info 1: 'platform/cpu1', 0x0000000001000074(_start): l.addi   r31,r0,0x0
Info   R31 deadbeef -> 00000000
Info 2: 'platform/cpu1', 0x0000000001000078(_start+4): l.addi   r1,r0,0x100
Info   R1 00000000 -> 00000100
Info 3: 'platform/cpu1', 0x000000000100007c(_start+8): l.sw     0xfffffffc(r31),r1
Info 4: 'platform/cpu1', 0x0000000001000080(_start+c): l.addi   r1,r0,0x200
Info   R1 00000100 -> 00000200
Info 5: 'platform/cpu1', 0x0000000001000084(_start+10): l.exw    0xfffffffc(r31),r1
>>> exchCount net written with 1
Info   R1 00000200 -> 00000100
Info   exchCount 00000000 -> 00000001
Info   exchAddress 00000000 -> fffffffc
Info   exchRDValue 00000000 -> 00000100
Info   exchWRValue 00000000 -> 00000200
Info 6: 'platform/cpu1', 0x0000000001000088(_start+14): l.lwz    r2,0xfffffffc(r31)
Info   R2 deadbeef -> 00000200
Info 7: 'platform/cpu1', 0x000000000100008c(_start+18): l.addi   r1,r0,0x300
Info   R1 00000100 -> 00000300
Info 8: 'platform/cpu1', 0x0000000001000090(_start+1c): l.exw    0xfffffffc(r31),r1
>>> exchCount net written with 2
Info   R1 00000300 -> 00000200
Info   exchCount 00000001 -> 00000002
Info   exchRDValue 00000100 -> 00000200
Info   exchWRValue 00000200 -> 00000300
Info 9: 'platform/cpu1', 0x0000000001000094(_start+20): l.lwz    r2,0xfffffffc(r31)
Info   R2 00000200 -> 00000300
Info 10: 'platform/cpu1', 0x0000000001000098(exit): l.nop    0x0
Processor 'platform/cpu1' terminated at 'exit', address 0x1000098
processor has executed 10 instructions
```

Note that trace is emitted for the extension instruction and extension registers just as if they were core processor instructions and registers, and that the callback installed on the `exchCount` net reflects the value of the new `exchCount` register as it changes.

26.5 Extension Libraries and Multicore Processors

It is possible to create multicore processors using the VMI API (see *SMP Processor Hierarchies* in the *VMI Run Time Function Reference* document for more information) and section 25.8 explained how intercept library instances can be configured to create structure only at the *leaf* hierarchy level or at *all* hierarchy levels.

The same method can be used when intercept libraries are used for processor extension. If the `allLevels` Boolean in the `vmiosAttr` structure is `False`, then `vmiosObject` structures are created at leaf levels only, and these objects are effectively independent of each other. If `allLevels` is `True`, then `vmiosObject` structures are created at every level at or below the instantiation level, and these structures can be linked together in the constructors as shown in section 25.8. In addition, the register and port iterator callbacks are called at *each hierarchy level*: this means that it is possible to add registers and ports that are common to subtrees of the processor hierarchy, not just at the leaf level. For example, the following code adds net port `rootPort` at the root level only, and net port `subPort` at each non-root level:

```
#define MAX_PORTS 2

typedef struct vmiosObjectS {

    vmiosObjectP root;
    vmiosObjectP parent;

    Uns32 rootPortHandle;
    Uns32 subPortHandle;

    vmiNetPort ports[MAX_PORTS];

} vmiosObject;

static VMIOS_CONSTRUCTOR_FN(constructor) {

    // save root on object
    object->root = parent ? parent->root : object;

    // save parent on object
    object->parent = parent;

    // get first port to fill
    vmiNetPortP thisNP = &object->ports[0];

    // add root-level net port
    if(!parent) {
        thisNP->name   = "rootPort";
        thisNP->type   = vmi_NP_INPUT;
        thisNP->handle = &object->rootPortHandle;
        thisNP++;
    }

    // add sub-level net port
    if(parent) {
        thisNP->name   = "subPort";
        thisNP->type   = vmi_NP_INPUT;
        thisNP->handle = &object->subPortHandle;
        thisNP++;
    }
}

static VMIOS_NET_PORT_SPECS_FN(iterNetPorts) {

    vmiNetPortP first = &object->ports[0];
    vmiNetPortP this  = prev ? prev+1 : first;
    Uns32       index = this-first;

    if((index<MAX_PORTS) && this->name) {
        return this;
    } else {
        Return NULL;
    }
}
```

26.6 Pre-Morph and Post-Morph Callbacks

Adding extension instructions to processor models as described in previous sections entails replacement of *all* behavior described using the processor model `morphCB` function with the new behavior from the extension library. Usually this is desirable, but occasionally it is not: for example, the ARC processor model implements a *zero-overhead loop* construct, which enables loop behavior based only on the *address* of an instruction. Each instruction for the ARC processor is therefore dependent on information derived both from the instruction *and* from control registers for the zero-overhead loop. When extending this processor instruction set, we want to *replace* the instruction information but *retain* the zero-overhead loop behavior from the base model.

From VMI version 7.1.0, additional *pre-morph* and *post-morph* callback functions, with the same prototype as the existing `morphCB` callback, have been added to the `vmiIASAttr` structure to handle such cases:

```
typedef struct vmiIASAttrS {

    . . . fields omitted for clarity . . .

    ////////////////////////////////////////////////////////////////////////
    // SIZE ATTRIBUTES
    ////////////////////////////////////////////////////////////////////////

    const char    **dictNames;          // null-terminated dictionary name list
    Uns32           cpuSize;            // full size of CPU object
    Uns32           blockStateSize;     // size of block state structure
    Uns32           instrStateSize;     // size of instruction state structure

    . . . fields omitted for clarity . . .

    ////////////////////////////////////////////////////////////////////////
    // MORPHER CORE ROUTINES
    ////////////////////////////////////////////////////////////////////////

    vmiStartEndBlockFn startBlockCB;    // called before block translate
    vmiStartEndBlockFn endBlockCB;      // called after block translate
    vmiMorphFn         preMorphCB;      // called at start of each instruction
    vmiMorphFn         morphCB;         // morph callback
    vmiMorphFn         postMorphCB;     // called at end of each instruction
    vmiPostOpaqueFn    postOpaqueCB;    // called after opaque intercept
    vmiFetchSnapFn     fetchSnapCB;     // fetch address snap callback
    vmiRdWrSnapFn      rdSnapCB;        // read alignment snap function
    vmiRdWrSnapFn      wrSnapCB;        // write alignment snap function

    . . . fields omitted for clarity . . .
                                        // update functions (for save/restore)
} vmiIASAttr;
```

When translating an instruction, the flow is as follows:

1. The `preMorphCB` callback, if specified, is called. This should emit code required at the *start* of every instruction (even if base model code is replaced by an extension library).
2. The `morphCB` callback is called, unless replaced in an extension library.

3. The `postMorphCB` callback, if specified, is called. This should emit code required at the *end* of every instruction (even if base model code is replaced by an extension library).

All three functions have the same prototype:

```
#define VMI_MORPH_FN(_NAME) void _NAME( \
    vmiProcessorP processor,    \
    Addr          thisPC,       \
    Uns8          inDelaySlot,  \
    Bool          firstInBlock, \
    Bool          emitTrace,    \
    void         *blockState,   \
    void         *instrState    \
)
typedef VMI_MORPH_FN((*vmiMorphFn));
```

The final two arguments (`blockState` and `instrState`) allow communication between callbacks as instruction translation proceeds.

The `blockState` argument is a pointer to a block of scratch memory of size defined by the `blockStateSize` field in the `vmiIASAttr` structure. This memory is initialized to zero at the start of each *code block* and passed to each of the callbacks described above as translation proceeds. It is therefore useful when information about translated code needs to be passed between calls that translate *different instructions* in a code block.

The `instrState` argument is a pointer to a block of scratch memory of size defined by the `instrStateSize` field in the `vmiIASAttr` structure. This memory is initialized to zero at the start of each *instruction* and passed to each of the callbacks described above as translation proceeds. It is therefore useful when information about translated code needs to be passed between the *pre-morph, morph and post-morph functions for a single instruction.*

The OVP ARC model uses these functions to implement zero-overhead loops. The `vmiIASAttr` structure is defined as follows:

```
const vmiIASAttr modelAttrs = {

    ////////////////////////////////////////////////////////////////////////
    // VERSION & SIZE ATTRIBUTES
    ////////////////////////////////////////////////////////////////////////

    .versionString  = VMI_VERSION,
    .modelType      = VMI_PROCESSOR_MODEL,
    .releaseStatus  = VMI_OVP,
    .dictNames      = dictNames,
    .cpuSize        = sizeof(arc),
    .blockStateSize = sizeof(arcBlockState),
    .instrStateSize = sizeof(arcMorphState),

    . . . fields omitted for clarity . . .

    ////////////////////////////////////////////////////////////////////////
    // MORPHER CORE ROUTINES
```

```
    ////////////////////////////////////////////////////////////////////////

    .preMorphCB  = arcPreMorphInstruction,
    .morphCB     = arcMorphInstruction,
    .postMorphCB = arcPostMorphInstruction,
    .fetchSnapCB = arcFetchSnap,
    .rdSnapCB    = arcRdWrSnapCB,
    .wrSnapCB    = arcRdWrSnapCB,

    . . . fields omitted for clarity . . .
};
```

`blockStateSize` is set to the size of a structure used to hold per-block information:

```
typedef struct arcBlockStateS {
    vmiReg branchFlag;          // register to test for branch
    Bool   branchIfTrue;        // branch if flag true?
    Bool   isUncondBranchDS;    // is delay-slot branch unconditional?
    Uns32  blockIndex;          // index number of instruction in block
} arcBlockState;
```

`instrStateSize` is set to the size of a structure used to hold per-instruction information:

```
typedef struct arcMorphStateS {
    arcInstructionInfo info;        // instruction description (from decoder)
    arcMorphAttrCP     attrs;       // instruction attributes
    arcBlockStateP     blockState;  // current block state
    arcP               arc;         // current processor
    Uns32              nextPC;      // next instruction address in sequence
    vmiLabelP          skipLabel;   // label to skip instruction body
    Uns32              tempIdx;     // current temporary index
    Bool               inDelaySlot; // instruction is in delay slot?
    Bool               atZOL;       // instruction at zero-overhead address?
    Bool               pclSet;      // is PCL valid?
    Bool               isEX;        // is this an EX instruction
} arcMorphState;
```

Function `arcPreMorphInstruction` decodes the instruction and sets up initial state for zero-overhead loops. It also handles annulled delay slot instructions:

```
VMI_MORPH_FN(arcPreMorphInstruction) {

    arcP           arc           = (arcP)processor;
    arcBlockStateP arcBlockState = blockState;
    arcMorphStateP state         = instrState;

    // get instruction and instruction type
    arcDecode(arc, thisPC, &state->info);

    // get morpher attributes for the decoded instruction and initialize other
    // state fields
    state->attrs      = &dispatchTable[state->info.type];
    state->arc        = arc;
    state->blockState = arcBlockState;

    // indicate whether the current instruction terminates a zero-overhead loop
    state->atZOL = (nextPC==AUX_REG(arc, lp_end));

    // do actions required when starting an instruction at the zero-overhead
```

```
    // loop end address
    if(state->atZOL) {
        arcEmitStartZOL(state);
    }

    // skip actions after this point if annulling
    arcEmitSkipIfAnnul();
}
```

Function `arcMorphInstruction` performs instruction translation (if not replaced in an extension library):

```
VMI_MORPH_FN(arcMorphInstruction) {

    arcMorphStateP state = instrState;

    if(state->attrs->morphCB) {
        // translate the instruction
        morphImplemented(state);
    } else if(state->info.type==ARC_IT_LAST) {
        // take IllegalInstruction exception
        emitIllegalInstruction(state);
    } else {
        // here if no morph callback specified
        vmiMessage("F", CPU_PREFIX"_UII",   // LCOV_EXCL_LINE
            SRCREF_FMT "unimplemented instruction",
            SRCREF_ARGS(processor, state->info.thisPC)
        );
    }
}
```

Function `arcPostMorphInstruction` handles termination of zero-overhead loops:

```
VMI_MORPH_FN(arcPostMorphInstruction) {

    arcMorphStateP state = instrState;

    // do actions required and end of instruction at the zero-overhead loop end
    // address
    arcEmitEndZOL(state);
}
```

Functions `arcEmitStartZOL` and `arcEmitEndZOL` optimize zero-overhead loop overhead in cases where loops are disabled or not active at the current address. They are quite complex and included here for completeness, but not explained in detail:

```
void arcEmitStartZOL(arcMorphStateP state) {

    Uns32     bits       = ARC_GPR_BITS;
    vmiLabelP noLPUpdate = vmimtNewLabel();
    Uns32     lpcMask    = state->arc->lpcMask;

    // when executing this block, validate that lp_end has the same value
    // that it has when code for the block was generated
    vmimtValidateBlockMaskR(ARC_GPR_BITS, ARC_AUX_REG(lp_end), -1);

    // tag this block to avoid unnecessary deletion when lp_end changes
    vmimtTagBlock(VBT_1);
```

```
    // ARC700 and ARCv2 implement STATUS32.L flag, disabling zero-overhead loops
    if(isARC700v2(state->arc)) {

        // assume zero-overhead loop is disabled
        vmimtMoveRC(8, ARC_ZOL_BRANCH, 0);

        // go to the label if L bit is set
        vmimtCondJumpLabel(ARC_L, True, noLPUpdate);
    }

    // define flags to detect non-zero condition, when loop count is decremented
    vmiFlags flags = {
        VMI_NOFLAG,
        {
            [vmi_CF] = VMI_NOFLAG,
            [vmi_PF] = VMI_NOFLAG,
            [vmi_ZF] = ARC_ZOL_BRANCH,
            [vmi_SF] = VMI_NOFLAG,
            [vmi_OF] = VMI_NOFLAG
        },
        vmi_FN_ZF
    };

    // decrement loop count and perhaps mask it, generating non-zero flag
    if(state->arc->lpcMask==-1) {
        vmimtBinopRC(bits, vmi_SUB, ARC_LP_COUNT, 1, &flags);
    } else {
        vmimtBinopRC(bits, vmi_SUB, ARC_LP_COUNT, 1, 0);
        vmimtBinopRC(bits, vmi_AND, ARC_LP_COUNT, lpcMask, &flags);
    }

    if(isARC600(state->arc)) {

        // ARC600 loop terminates if pre-decrement value is either 0 or 1, so
        // include detection of *post-decrement* value -1
        vmiReg tf = ARC_TEMP(state->tempIdx+1);
        vmimtCompareRC(bits, vmi_COND_NE, ARC_LP_COUNT, lpcMask, tf);
        vmimtBinopRR(bits, vmi_AND, ARC_ZOL_BRANCH, tf, 0);

    } else if(state->inDelaySlot) {

        // on ARC700 and ARCv2, if in a delay slot, only branch if STATUS32.DE
        // is zero
        vmimtBinopRR(8, vmi_ANDN, ARC_ZOL_BRANCH, ARC_DE, 0);
    }

    // here if zero-overhead loops are disabled
    vmimtInsertLabel(noLPUpdate);
}

void arcEmitEndZOL(arcMorphStateP state) {

    if(state->atZOL) {
        vmimtCondJumpReg(
            ARC_ZOL_BRANCH,
            True,
            0,
            ARC_AUX_REG(lp_start),
            VMI_NOREG,
            vmi_JH_NONE
        );
    }
}
```

27 Processor Configuration

A processor model is configured by variables which can be set by the platform or simulation environment. They are called *model parameters.* Model parameters are collected into a structure which is initialized by the simulator then passed to the model's constructor function. The structure exists only when the constructor is executing. To specify the type and constraints of each parameter, the model must provide an iterator function which returns each parameter specification in turn. This allows the simulator to set and check each parameter and allows other tools to discover the configuration interface of the model.

27.1 Example of a Configurable Processor

A model for the OR1K processor with model parameters can be found in:

```
$IMPERAS_HOME/Examples/Models/Processor/24.or1kConfigurable
```

Take a copy of the template model:

```
cp –r $IMPERAS_HOME/Examples/Models/Processor/24.or1kConfigurable .
```

Compile the model, harness and application using the `make` command:

```
cd 24.or1kConfigurable
make
```

Note that the harness and application can be compiled individually if required using these commands:

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

The processor model is based on the model of example `18.or1kInstructionAttributes`, with the changes listed in following sections.

27.2 The Parameters Structure

The parameters structure is defined in `or1kParameters.h`:

```
typedef struct or1kParamValuesS {

    VMI_BOOL_PARAM(verbose);
    VMI_UNS32_PARAM(extinterrupts);
    VMI_STRING_PARAM(extintlogfile);

} or1kParamValues, *or1kParamValuesP;
```

The example defines boolean, integer and string parameters. This is the complete list of available parameter types:

macro	data type
VMI_BOOL_PARAM	boolean
VMI_INT32_PARAM	32 bit signed
VMI_UNS32_PARAM	32 bit unsigned
VMI_INT64_PARAM	64 bit signed
VMI_UNS64_PARAM	64 bit unsigned
VMI_PTR_PARAM	pointer
VMI_DBL_PARAM	floating point
VMI_STRING_PARAM	null terminated string
VMI_ENUM_PARAM	null terminated string
VMI_ENDIAN_PARAM	null terminated string

27.3 Parameter Specification - or1kParameters.c

The parameters are specified in `or1kParameters.c`.

27.3.1 Structure Size

The simulator allocates this structure so needs to know its size. A function must be defined (using the `VMI_PROC_PARAM_TABLE_SIZE_FN)` and set in the model attributes table:

```
//
// Get the size of the parameter values table
//
VMI_PROC_PARAM_TABLE_SIZE_FN(or1kParamValueSize) {
    return sizeof(paramValues);
}

//
// Add function to the model attributes table
//
const vmiIASAttr modelAttrs = {
    ...
    .paramValueSizeCB = or1kParamValueSize,
    ...
};
```

27.3.2 Specification Objects

A function must be defined to supply the attribute specifications to the simulator (using the `VMI_PROC_PARAM_SPECS_FN` macro) and set in the model attributes table. In this model, the parameter specifications are built as a static list, and the function iterates over the list.

Each element of the list is initialized using macros defined in `vmiParameters.h`:

```
//
// Table of parameter specs
//
static vmiParameter formals[] = {
    VMI_BOOL_PARAM_SPEC  (or1kParamValues, verbose, 0,
                  "Verbose mode"),

    VMI_UNS32_PARAM_SPEC (or1kParamValues,extinterrupts, 1, 1, 8,
                  "Number of external interrupts" ),
```

```
    VMI_STRING_PARAM_SPEC(or1kParamValues,extintlogfile, 0,
                  "Event log file" ),

    VMI_END_PARAM
};

//
// Function to iterate over the parameter specs
//
VMI_PROC_PARAM_SPECS_FN(or1kGetParamSpec) {
    if(!prev) {
        return formals;
    } else {
        prev++;
        if (prev->name)
            return prev;
        else
            return 0;
    }
}

//
// Add function to the model attributes table
//
const vmiIASAttr modelAttrs = {
    ...
    .paramSpecsCB     = or1kGetParamSpec,
    ...
};
```

This is the complete list of parameter specification macros corresponding to the parameter definition macros:

macro	data type	limits
VMI_BOOL_PARAM_SPEC	boolean	0 or 1
VMI_INT32_PARAM_SPEC	32 bit signed	specified min / max
VMI_UNS32_PARAM_SPEC	32 bit unsigned	specified min / max
VMI_INT64_PARAM_SPEC	64 bit signed	specified min / max
VMI_UNS64_PARAM_SPEC	64 bit unsigned	specified min / max
VMI_PTR_PARAM_SPEC	pointer	NULL if not specified
VMI_DBL_PARAM_SPEC	floating point	specified min / max
VMI_STRING_PARAM_SPEC	null terminated string	any string (NULL if not specified)
VMI_ENUM_PARAM_SPEC	null terminated string	string must be a member of the specified list
VMI_ENDIAN_PARAM_SPEC	null terminated string	"big" or "little"

27.3.3 Using the Parameters

The parameter structure is allocated and assigned by the simulator then passed to the model constructor. In this example the constructor is in `or1kMain.c`:

```
//
// OR1K processor constructor
//
VMI_CONSTRUCTOR_FN(or1kConstructor) {

    or1kP            or1k   = (or1kP)processor;
```

```
    or1kParamValuesP params = parameterValues;

    ...

    // copy parameters to the model instance
    or1k->numExtInts = params->extinterrupts;
    or1k->noisy      = params->verbose;

    ...

    // open a log file and report or not
    if (params->extintlogfile) {
        or1k->logFile = fopen(params->extintlogfile, "w");
        if (params->verbose) {
            if (or1k->logFile) {
                ... (etc)
            }
        }
    }
}
```

The values `noisy`, `numExtInts` and `logFile` are saved on the model instance, so can be used subsequently anywhere in the processor model. `numExtInts` is used to control the number of external interrupt nets:

```
////////////////////////////////////////////////////////////////////////////////
// NET PORTS
////////////////////////////////////////////////////////////////////////////////

//
// Template net port list
//
const static vmiNetPort netPorts[] = {
    {"reset", vmi_NP_INPUT, (void*)0,   or1kExternalReset    },
    {"intr0", vmi_NP_INPUT, (void*)1,   or1kExternalInterrupt},
    {"intr1", vmi_NP_INPUT, (void*)2,   or1kExternalInterrupt},
    {"intr2", vmi_NP_INPUT, (void*)4,   or1kExternalInterrupt},
    {"intr3", vmi_NP_INPUT, (void*)8,   or1kExternalInterrupt},
    {"intr4", vmi_NP_INPUT, (void*)16,  or1kExternalInterrupt},
    {"intr5", vmi_NP_INPUT, (void*)32,  or1kExternalInterrupt},
    {"intr6", vmi_NP_INPUT, (void*)64,  or1kExternalInterrupt},
    {"intr7", vmi_NP_INPUT, (void*)128, or1kExternalInterrupt},
};

//
// Get the number of processor net ports (reset port plus external
// interrupts)
//
static Uns32 getNumNetPorts(or1kP or1k) {
    return 1 + or1k->numExtInts;
}

//
// Allocate net port specifications
//
static void newNetPorts(or1kP or1k) {

    Uns32 numNetPorts = getNumNetPorts(or1k);
    Uns32 i;

    or1k->netPorts = STYPE_CALLOC_N(vmiNetPort, numNetPorts);
```

```
    for(i=0; i<numNetPorts; i++) {
        or1k->netPorts[i] = netPorts[i];
    }
}

//
// Get the next net port
//
VMI_NET_PORT_SPECS_FN(or1kGetNetPortSpec) {

    or1kP or1k = (or1kP)processor;

    if(!prev) {

        // first port
        return or1k->netPorts;

    } else {

        // port other than the first
        Uns32 numNetPorts = getNumNetPorts(or1k);
        Uns32 prevIndex   = (prev-or1k->netPorts);
        Uns32 thisIndex   = prevIndex+1;

        return (thisIndex<numNetPorts) ? &or1k->netPorts[thisIndex] : 0;
    }
}
```

The `logFile` variable is used to control logging of interrupt events to a special file:

```
static VMI_NET_CHANGE_FN(externalInterrupt) {

    or1kP or1k     = (or1kP)processor;
    Uns32 deviceId = (Uns32)userData;

    if(newValue) {
        if(or1k->logFile) {
            fprintf(or1k->logFile, "Taking interrupt with ID=0x%x\n", deviceId);
        }
        ...
    }
    ....
}
```

27.4 Parameter Function Registration - `or1kAttrs.c`

The new parameter callbacks are registered in `or1kAttrs.c`:

```
const vmiIASAttr modelAttrs = {

    . . . lines omitted . .

    ////////////////////////////////////////////////////////////////////////
    // PARAMETER SUPPORT ROUTINES
    ////////////////////////////////////////////////////////////////////////

    .paramSpecsCB     = or1kGetParamSpec,
    .paramValueSizeCB = or1kParamValueSize,

    . . . lines omitted . .
```

```
};
```

27.5 Using a parameterized model

The example platform (`platform/harness.c`) creates two model instances. `procA` uses default parameter values, with simulated exceptions enabled:

```
    optProcessorP procA = opProcessorNew(
        mr,
        "model."IMPERAS_SHRSUF,
        "procA",
        0,
        OP_PARAMS (
            OP_PARAM_BOOL_SET(OP_FP_SIMULATEEXCEPTIONS, 1)
        )
    );
```

`procB` has its parameters overridden:

```
    optProcessorP procB = opProcessorNew(
        mr,
        "model."IMPERAS_SHRSUF,
        "procB",
        0,
        OP_PARAMS (
            OP_PARAM_BOOL_SET(OP_FP_SIMULATEEXCEPTIONS, 1),
            OP_PARAM_BOOL_SET(OP_FP_VERBOSE, 1),
            OP_PARAM_UNS32_SET("extinterrupts", 2),
            OP_PARAM_STRING_SET("extintlogfile", "test.log")
        )
    );
```

Consequently, `procB` can be connected to two interrupt nets while `procA` can be connected to only one.

```
    // create nets
    optNetP n1 = opNetNew(mr, "n1", 0, 0);
    optNetP n2 = opNetNew(mr, "n2", 0, 0);
    optNetP n3 = opNetNew(mr, "n3", 0, 0);

    // connect nets to processors
    opObjectNetConnect(procA, n1, "intr0");
    opObjectNetConnect(procB, n2, "intr0");
    opObjectNetConnect(procB, n3, "intr1");
```

To demonstrate that configuration has worked, the harness includes a function that shows the net ports present on a given processor instance:

```
static void queryNetPorts(optProcessorP processor) {

    const char *name = opObjectHierName(processor);

    if(!opObjectNetPortNext(processor, 0)) {

        opPrintf("%s HAS NO NET PORTS\n", name);

    } else {
```

```
        opPrintf("%s NET PORTS\n", name);

        optNetPortP netPort = 0;

        while((netPort=opObjectNetPortNext(processor, netPort))) {
            opPrintf(
                "  %s (%s)\n",
                opObjectName(netPort),
                opNetPortTypeString(netPort)
            );
        }
    }

    opPrintf("\n");
}
```

This is called for each processor before simulation starts:

```
    // print net ports
    queryNetPorts(procA);
    queryNetPorts(procB);
```

27.6 Testing Processor Configuration

Run the platform using the assembler executable file:

```
platform/harness.$IMPERAS_ARCH.exe –-trace \
    --program application/asmtest.OR1K.elf
```

The output from this should be as follows:

```
Opened log file 'test.log'
platform/procA NET PORTS
  reset (input)
  intr0 (input)
platform/procB NET PORTS
  reset (input)
  intr0 (input)
  intr1 (input)
Info 'platform/procA', 0x0000000000010000(_start): l.ori    r30,r0,0x0
Info 'platform/procA', 0x0000000000010004(_start+4): l.ori    r1,r0,0x7
Info 'platform/procA', 0x0000000000010008(_start+8): l.mtspr  r0,r1,17
Info 'platform/procA', 0x000000000001000c(_start+c): l.addi   r1,r0,0xffffffff
Info 'platform/procA', 0x0000000000010010(_start+10): l.mtspr  r0,r1,18432
Info 'platform/procA', 0x0000000000010014(_start+14): l.ori    r1,r0,0x8
Info 'platform/procA', 0x0000000000010018(loop1): l.addi   r1,r1,0xffffffff
Info 'platform/procA', 0x000000000001001c(loop1+4): l.sfeqi  r1,0x0
Info 'platform/procA', 0x0000000000010020(loop1+8): l.bnf    0x00010018
Info 'platform/procB', 0x0000000000010000(_start): l.ori    r30,r0,0x0
Info 'platform/procB', 0x0000000000010004(_start+4): l.ori    r1,r0,0x7
Info 'platform/procB', 0x0000000000010008(_start+8): l.mtspr  r0,r1,17
Info 'platform/procB', 0x000000000001000c(_start+c): l.addi   r1,r0,0xffffffff
Info 'platform/procB', 0x0000000000010010(_start+10): l.mtspr  r0,r1,18432
Info 'platform/procB', 0x0000000000010014(_start+14): l.ori    r1,r0,0x8
Info 'platform/procB', 0x0000000000010018(loop1): l.addi   r1,r1,0xffffffff
Info 'platform/procB', 0x000000000001001c(loop1+4): l.sfeqi  r1,0x0
Info 'platform/procB', 0x0000000000010020(loop1+8): l.bnf    0x00010018
Interrupting A & B
Info 'platform/procA', 0x0000000000010024(loop1+c): *** FETCH EXCEPTION ***
Info 'platform/procB', 0x0000000000010024(loop1+c): *** FETCH EXCEPTION ***
```

```
Info 'platform/procA', 0x0000000000000800(.text+800): l.addi   r30,r30,0x1
Info 'platform/procA', 0x0000000000000804(.text+804): l.rfe
Info 'platform/procA', 0x0000000000010020(loop1+8): l.bnf    0x00010018
Info 'platform/procA', 0x0000000000010024(loop1+c): l.nop    0x0
Info 'platform/procA', 0x0000000000010018(loop1): l.addi   r1,r1,0xffffffff
Info 'platform/procA', 0x000000000001001c(loop1+4): l.sfeqi  r1,0x0
Info 'platform/procA', 0x0000000000010020(loop1+8): l.bnf    0x00010018
Info 'platform/procA', 0x0000000000010024(loop1+c): l.nop    0x0
Info 'platform/procA', 0x0000000000010018(loop1): l.addi   r1,r1,0xffffffff
Info 'platform/procB', 0x0000000000000800(.text+800): l.addi   r30,r30,0x1
Info 'platform/procB', 0x0000000000000804(.text+804): l.rfe
Info 'platform/procB', 0x0000000000010020(loop1+8): l.bnf    0x00010018
Info 'platform/procB', 0x0000000000010024(loop1+c): l.nop    0x0
Info 'platform/procB', 0x0000000000010018(loop1): l.addi   r1,r1,0xffffffff
Info 'platform/procB', 0x000000000001001c(loop1+4): l.sfeqi  r1,0x0
Info 'platform/procB', 0x0000000000010020(loop1+8): l.bnf    0x00010018
Info 'platform/procB', 0x0000000000010024(loop1+c): l.nop    0x0
Info 'platform/procB', 0x0000000000010018(loop1): l.addi   r1,r1,0xffffffff
Interrupting B
Info 'platform/procA', 0x000000000001001c(loop1+4): l.sfeqi  r1,0x0
Info 'platform/procB', 0x000000000001001c(loop1+4): *** FETCH EXCEPTION ***
Info 'platform/procA', 0x0000000000010020(loop1+8): l.bnf    0x00010018
Info 'platform/procA', 0x0000000000010024(loop1+c): l.nop    0x0
Info 'platform/procA', 0x0000000000010018(loop1): l.addi   r1,r1,0xffffffff
Info 'platform/procA', 0x000000000001001c(loop1+4): l.sfeqi  r1,0x0
Info 'platform/procA', 0x0000000000010020(loop1+8): l.bnf    0x00010018
Info 'platform/procA', 0x0000000000010024(loop1+c): l.nop    0x0
Info 'platform/procA', 0x0000000000010018(loop1): l.addi   r1,r1,0xffffffff
Info 'platform/procA', 0x000000000001001c(loop1+4): l.sfeqi  r1,0x0
Info 'platform/procA', 0x0000000000010020(loop1+8): l.bnf    0x00010018
Info 'platform/procA', 0x0000000000010024(loop1+c): l.nop    0x0
Info 'platform/procB', 0x0000000000000800(.text+800): l.addi   r30,r30,0x1
Info 'platform/procB', 0x0000000000000804(.text+804): l.rfe
Info 'platform/procB', 0x000000000001001c(loop1+4): l.sfeqi  r1,0x0
Info 'platform/procB', 0x0000000000010020(loop1+8): l.bnf    0x00010018
Info 'platform/procB', 0x0000000000010024(loop1+c): l.nop    0x0
Info 'platform/procB', 0x0000000000010018(loop1): l.addi   r1,r1,0xffffffff
Info 'platform/procB', 0x000000000001001c(loop1+4): l.sfeqi  r1,0x0
Info 'platform/procB', 0x0000000000010020(loop1+8): l.bnf    0x00010018
Info 'platform/procB', 0x0000000000010024(loop1+c): l.nop    0x0
Info 'platform/procB', 0x0000000000010018(loop1): l.addi   r1,r1,0xffffffff
processor A has executed 30 instructions
processor B has executed 30 instructions
```

Note the lines at the start of the run showing net ports present on each instance, revealing that `procA` has a single interrupt input but `procB` has two such inputs:

```
platform/procA NET PORTS
  reset (input)
  intr0 (input)
platform/procB NET PORTS
  reset (input)
  intr0 (input)
  intr1 (input)
```

After the simulation completes, the `procB` processor-specific test file `test.log` contains this:

```
Taking interrupt with ID=0x2
Taking interrupt with ID=0x1
```

```
Finished
```

28 Making High-Performance Processor Models

Previous sections in this document have shown some of the techniques required to make high performance processor models. This section summarizes some key points, not all of which have been mentioned previously.

Do as much work as possible at morph time. If it is possible to simplify an instruction by making a decision at morph time, always do so. As an example, for some OR1K control register accesses it was possible to determine at morph time which control register was being accessed, and special code was generated to do such accesses. As another example, instructions that have different behavior in different modes (e.g. user and supervisor) need only morph the behavior required for the current mode in a modal processor.

Use morph time constructs wherever possible and reasonable in preference to embedded calls. With every instruction, there is a fundamental choice of implementation strategy: should it be implemented by a morpher primitive, or by an embedded call to a C function? The interface in `vmiMt.h` contains a rich set of morph time primitives, and these should almost always be used in preference to embedded calls. Embedded calls are slower, and the DFA optimizer in the simulator cannot propagate optimizations across calls, which means the code around the call is also less efficient.
The one exception is when an instruction requires many morph primitives to implement – in this case, a call to a native function may be faster (and the JIT compiled code will be smaller).

In particular, use `vmimtUncondJump` etc in preference to `vmirtSetPC` in an embedded call. Jumps described using the morph-time functions are hugely faster.

In particular, use `vmimtLoadRRO` etc in preference to `vmirtReadNByteDomain` etc in an embedded call. Loads and stores described using the morph-time functions are hugely faster.

Position the most-frequently-used simulated registers in the first 128 bytes of the processor structure. This section of the processor model is accessed in JIT code using a byte offset from the processor object, so the generated code is most compact.

Always use `VMI_REG_TEMP` to describe intermediates that are not true processor registers. The JIT code generator can often assign these entirely to native registers or optimize away references to them altogether.

Do not explicitly assign the program counter on every instruction. The program counter can be found when required using `vmirtGetPC` etc, so do not maintain its value explicitly in the processor structure.

Derive complex register values on demand. Infrequently-accessed but frequently-changing registers like status registers should have their values assembled on demand when required. This is especially true if the registers contain flags.

Master flag register values as bytes in the processor structure. Do not master them as bits in a status register, as this will require complex code to correct the status register on every flag change.

Use `vmirtAliasMemoryVM`/`vmirtUnaliasMemoryVM` to model virtual memory pages. The simulator will generate JIT code in such regions so that it is *physically* mapped and relocatable, which dramatically improves simulation performance of VM systems.

Use `vmirtAliasMemory`/`vmirtUnaliasMemory` to model fixed-mapped pages. The simulator will generate JIT code in such regions so that it is *virtually* mapped and not relocatable, which is even faster than relocatable physically mapped sections, provided that the mapping remains constant.

28.1 Processor Model Efficiency Analysis

When a processor model is fully implemented there are a some further checks that can be made to ensure that it is working effectively as possible. The first step is to enable dictionary statistics at the end of a simulation run using the `--verbosedict` command line parameter. For example:

```
platform/harness.$IMPERAS_ARCH.exe --verbosedict --program
application/asmtest.OR1K.elf
```

When this attribute is used, additional summary statistics are printed at the end of simulation for each of the code dictionaries. Here is the result for an OR1K processor running ucLinux:

```
Info ---------------------------------------------------
Info CPU 'plat/cpuA' STATISTICS
Info ---------------------------------------------------
Info DICTIONARY 'SUPERVISOR' STATISTICS
Info Simulated code size     : 95,132 bytes
Info Native code size        : 306,278 bytes
Info     bloat factor        : 3.2
Info Writeback fragment size : 3,705 bytes
Info     of which small      : 3,093 bytes
Info     of which large      : 612 bytes
Info     large fragments     : 97
Info Active code blocks      : 4,138
Info Pair code blocks        : 155 (3.7% of active)
Info Pair native code size   : 36,931 bytes
Info Block ejections
Info     colliding           : 6
Info     bad memory window   : 4
Info     remorph icount      : 3
Info     set mode block      : 4
Info Mispredicted branches   : 6,916
Info MEMORY STATISTICS
Info Static mapped memory    : 0 bytes
```

```
Info Dynamic mapped memory    : 4,530,440 bytes
Info Master regions (by size)
Info     smaller than 1Kb     : 1
Info     smaller than 4Kb     : 2
Info     at least 4Kb         : 37
Info Slave regions (by size)
Info     smaller than 1Kb     : 8
Info     smaller than 4Kb     : 0
Info     at least 4Kb         : 0
Info Memory model efficiency
Info     fast region lookups : 4,336
Info     uncached reads       : 36
Info     - which set region  : 36
Info     - which set function: 36
Info     uncached writes      : 784
Info     - which set region  : 359
Info     - which set function: 39
Info     window misses        : 82
Info ---------------------------------------------------
Info DICTIONARY 'USER' STATISTICS
Info Simulated code size      : 40,544 bytes
Info Native code size         : 129,838 bytes
Info     bloat factor         : 3.2
Info Writeback fragment size : 957 bytes
Info     of which small       : 819 bytes
Info     of which large       : 138 bytes
Info     large fragments      : 23
Info Active code blocks       : 1,675
Info Pair code blocks         : 8 (0.5% of active)
Info Pair native code size    : 1,339 bytes
Info Block ejections
Info     colliding            : 11
Info Mispredicted branches    : 1,966
Info MEMORY STATISTICS
Info Static mapped memory     : 0 bytes
Info Dynamic mapped memory    : 4,530,440 bytes
Info Master regions (by size)
Info     smaller than 1Kb     : 1
Info     smaller than 4Kb     : 2
Info     at least 4Kb         : 37
Info Slave regions (by size)
Info     smaller than 1Kb     : 8
Info     smaller than 4Kb     : 0
Info     at least 4Kb         : 0
Info Memory model efficiency
Info     fast region lookups : 4,336
Info     uncached reads       : 36
Info     - which set region  : 36
Info     - which set function: 36
Info     uncached writes      : 784
Info     - which set region  : 359
Info     - which set function: 39
Info     window misses        : 82
Info ---------------------------------------------------
```

The fields of most interest are the *bloat factor* and the categories listed beneath *Block ejections*.

'**bloat factor'** gives the ratio of *JIT-translated code size* to the *original simulated code size* for all currently-active code blocks (for example, if a simulated instruction sequence of 20 bytes gets translated into a native block of 100 bytes, the bloat factor is 5).

Typically, JIT-translated code is somewhat more verbose than the original source, so this number is usually greater than 1. If the ratio is much higher than 5 or 6, then it could mean that some instruction translations are producing very long native code equivalents, which could be a cause of inefficiency. Create small (probably assembler) test cases to help identify which instructions may be suffering from poor translation.

'**Block ejections**' indicates how many code blocks were ejected during simulation, grouped by the reason for the ejection. If a large number of code blocks are ejected during a simulation run, that could indicate an error in the model (an inappropriate use of block masks, for example). By writing small, probably assembler, test cases and using `--verbosedict`, individual instructions and/or families of instructions can be verified for efficiency.

28.1.1 Processor Model Profiling

When using the Imperas Professional products with the `CPUMAN_MULTI` personality, it is possible to enable *model instruction profiling* to determine which simulated instructions take most execution time. This feature requires the `IMP_CPUDEV` license key in addition to your regular license key. The profiler works by sampling the processor simulated program counter at regular intervals during the simulation and accumulating samples for each instruction type. Profiling can be enabled in one of three ways:

1. by using the option `–profile <number>` on the simulator command line;
2. by setting the `Int32` parameter `profile` to `<number>` on the root module, when using an OP platform;
3. by setting environment variable `IMPERAS_PROFILE` to `<number>` when running a simulation.

In each case, `<number>` is the maximum number of instruction categories that should be reported at the end of simulation. As an example, running an OR1K simulation of an H.264 encoding algorithm with the environment variable setting `IMPERAS_PROFILE=20` produces this output at the end of simulation:

```
-------------------------------------------------------------------
PROFILE REPORT
-------------------------------------------------------------------
    l.addi              3.84s (or1k_1/cpuA)
    l.sb                3.15s (or1k_1/cpuA)
    l.lwz               2.16s (or1k_1/cpuA)
    l.lbz               1.92s (or1k_1/cpuA)
    l.sw                0.96s (or1k_1/cpuA)
    l.lhz               0.55s (or1k_1/cpuA)
    l.add               0.46s (or1k_1/cpuA)
    l.slli              0.44s (or1k_1/cpuA)
    l.or                0.28s (or1k_1/cpuA)
    l.and               0.21s (or1k_1/cpuA)
    (JIT translation)   0.20s
    l.ori               0.20s (or1k_1/cpuA)
    l.sub               0.19s (or1k_1/cpuA)
    l.sh                0.13s (or1k_1/cpuA)
    l.srli              0.09s (or1k_1/cpuA)
    l.nop               0.07s (or1k_1/cpuA)
    l.srai              0.06s (or1k_1/cpuA)
    l.sfeqi             0.06s (or1k_1/cpuA)
    l.movhi             0.05s (or1k_1/cpuA)
```

```
    l.bf                  0.05s (or1k_1/cpuA)
    TOTAL TIME           15.45s
--------------------------------------------------------------------
```

From this, it is possible to see that most time was taken in the `l.addi` instruction (which happens to be by far the most frequently-used instruction type in this example), but generally loads and stores are dominant. Time taken by the JIT morpher is also shown in a special category.

Simple profiling is the least intrusive and therefore gives the most accurate indication of where time is spent. It is possible to enable somewhat more intrusive profiling that also counts instructions and therefore can generate a report giving the mean time spent in a particular instruction type. To do this, specify a *negative* value for `IMPERAS_PROFILE`; for example, setting `IMPERAS_PROFILE=-20` produces this output on the same example:

```
--------------------------------------------------------------------
PROFILE REPORT
--------------------------------------------------------------------
    l.addi                5.43s      6,246,115,414   0.9ps/instruction (or1k_1/cpuA)
    l.sb                  4.34s      1,634,569,969   2.7ps/instruction (or1k_1/cpuA)
    l.lwz                 2.72s      1,220,710,977   2.2ps/instruction (or1k_1/cpuA)
    l.lbz                 1.81s      1,589,078,874   1.1ps/instruction (or1k_1/cpuA)
    l.sw                  0.87s        614,735,643   1.4ps/instruction (or1k_1/cpuA)
    l.lhz                 0.71s        330,262,102   2.1ps/instruction (or1k_1/cpuA)
    l.add                 0.69s        981,748,236   0.7ps/instruction (or1k_1/cpuA)
    l.slli                0.66s        833,395,483   0.8ps/instruction (or1k_1/cpuA)
    l.and                 0.46s        235,084,866   2.0ps/instruction (or1k_1/cpuA)
    (JIT translation)     0.38s            107,923   3.4ns/instruction
    l.or                  0.23s        267,618,753   0.9ps/instruction (or1k_1/cpuA)
    l.sub                 0.20s        277,832,028   0.7ps/instruction (or1k_1/cpuA)
    l.ori                 0.18s        440,171,168   0.4ps/instruction (or1k_1/cpuA)
    l.movhi               0.15s        252,801,812   0.6ps/instruction (or1k_1/cpuA)
    l.nop                 0.13s        171,563,151   0.8ps/instruction (or1k_1/cpuA)
    l.sh                  0.11s         40,420,735   2.7ps/instruction (or1k_1/cpuA)
    l.j                   0.10s         82,737,470   1.2ps/instruction (or1k_1/cpuA)
    l.srli                0.09s        267,001,754   0.3ps/instruction (or1k_1/cpuA)
    l.bf                  0.08s      2,292,386,744   0.0ps/instruction (or1k_1/cpuA)
    l.srai                0.07s         81,752,183   0.9ps/instruction (or1k_1/cpuA)
    TOTAL                19.82s     20,717,653,845
--------------------------------------------------------------------
```

In this report, the columns for each instruction type are as follows:

1. the instruction type (for example, `l.addi`);
2. the total execution time for instructions of that type (for example, `5.43s`);
3. the number of instructions of that type executed (for example, `6,246,115,414`);
4. the time per instruction of that type (for example, `0.9ps/instruction`).

The speed of JIT translation per instruction is also shown: in this case, it is 3.4ns per instruction. A large JIT translation time per instruction might indicate an inefficiency in the processor model morpher.

Note that simulation is slower than in the previous report because of the extra overhead required for event counting, but the general time distribution pattern is similar.