



## RISC-V Model Custom Extension Guide

### Imperas Software Limited

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



Author:	Imperas Software Limited
Version:	1.4
Filename:	OVP_RISCV_Model_Custom_Extension_Guide.doc
Project:	RISC-V Model Custom Extension Guide
Last Saved:	Monday, 12 April 2021
Keywords:	

## Copyright Notice

Copyright © 2021 Imperas Software Limited All rights reserved. This software and documentation contain information that is the property of Imperas Software Limited. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Imperas Software Limited, or as expressly provided by the license agreement.

## Right to Copy Documentation

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

## Destination Control Statement

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

## Disclaimer

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

Table of Contents

<b>1</b>	<b>Introduction.....</b>	<b>7</b>
<b>2</b>	<b>Building a RISC-V Model .....</b>	<b>8</b>
2.1	INTRODUCTION .....	8
2.2	USING LOCAL RISC-V MODEL SOURCE DIRECTORY.....	8
2.3	USING VLNV LIBRARY RISC-V MODEL SOURCE.....	8
2.4	USING A LINKED MODEL .....	9
<b>3</b>	<b>Introduction to Custom Extensions with Linked Model .....</b>	<b>10</b>
3.1	LINKED MODEL CREATION .....	10
3.2	CUSTOM CONFIGURATION OPTIONS.....	10
3.3	ADDING CUSTOM INSTRUCTIONS.....	10
3.4	ADDING CUSTOM CSRS.....	10
3.5	ADDING CUSTOM EXCEPTIONS .....	10
3.6	ADDING CUSTOM LOCAL INTERRUPTS.....	10
3.7	ADDING CUSTOM FIFO PORTS .....	10
3.8	ADDING TRANSACTIONAL MEMORY.....	10
3.9	BUILDING THE LINKED MODEL .....	10
<b>4</b>	<b>Linked Model Creation .....</b>	<b>12</b>
4.1	FILE EXTENSIONCONFIG.H .....	13
4.2	FILE RISCVINFO.C.....	13
4.3	FILE RISCVCONFIGLIST.C.....	14
<b>5</b>	<b>Custom Configuration Options.....</b>	<b>16</b>
5.1	FUNDAMENTAL CONFIGURATION .....	19
5.2	INTERRUPTS AND EXCEPTIONS.....	20
5.3	INSTRUCTION AND CSR BEHAVIOR .....	20
5.4	CLIC .....	21
5.5	MEMORY SUBSYSTEM.....	22
5.6	FLOATING POINT.....	22
5.7	VECTOR EXTENSION .....	23
5.8	BIT MANIPULATION EXTENSION .....	24
5.9	CRYPTOGRAPHIC EXTENSION .....	25
5.10	HYPERVERSOR EXTENSION .....	25
5.11	DEBUG MODE.....	26
5.12	TRIGGER MODULE .....	27
5.13	MULTICORE VARIANTS .....	28
5.14	CSR DEFAULT VALUES .....	28
5.15	CSR MASKS.....	28
<b>6</b>	<b>Adding Custom Instructions (addInstructions).....</b>	<b>29</b>
6.1	INTERCEPT ATTRIBUTES .....	29
6.2	OBJECT TYPE AND CONSTRUCTOR.....	30
6.3	INSTRUCTION DECODE.....	30
6.4	INSTRUCTION DISASSEMBLY.....	33
6.5	INSTRUCTION TRANSLATION.....	34
6.5.1	<i>Required VMI Morph-Time Function Knowledge</i> .....	38
6.6	EXAMPLE EXECUTION.....	39
<b>7</b>	<b>Adding Custom CSRs (addCSRs).....</b>	<b>42</b>
7.1	CSR TYPE DEFINITIONS.....	42
7.2	EXTENSION-SPECIFIC CONFIGURATION .....	43
7.3	INTERCEPT ATTRIBUTES .....	44

7.4	OBJECT TYPE AND CONSTRUCTOR.....	44
7.5	EXAMPLE EXECUTION.....	53
<b>8</b>	<b>Adding Custom Exceptions (addExceptions) .....</b>	<b>57</b>
8.1	EXCEPTION CODE .....	57
8.2	INTERCEPT ATTRIBUTES .....	57
8.3	OBJECT TYPE AND CONSTRUCTOR.....	57
8.4	INSTRUCTION DECODE.....	59
8.5	INSTRUCTION DISASSEMBLY.....	60
8.6	INSTRUCTION TRANSLATION.....	60
8.7	EXAMPLE EXECUTION.....	60
<b>9</b>	<b>Adding Custom Local Interrupts (addLocalInterrupts).....</b>	<b>63</b>
9.1	ENABLING LOCAL INTERRUPT PORTS .....	63
9.2	INTERRUPT CODES .....	63
9.3	INTERCEPT ATTRIBUTES .....	64
9.4	OBJECT TYPE AND CONSTRUCTOR.....	64
9.5	EXAMPLE EXECUTION.....	66
<b>10</b>	<b>Adding Custom FIFOs (fifoExtensions) .....</b>	<b>68</b>
10.1	INTERCEPT ATTRIBUTES .....	68
10.2	OBJECT TYPE AND CONSTRUCTOR.....	69
10.3	EXTENSION CSRS .....	70
10.4	EXTENSION FIFO PORTS.....	73
10.5	INSTRUCTION DECODE.....	74
10.6	INSTRUCTION DISASSEMBLY.....	74
10.7	INSTRUCTION TRANSLATION.....	75
<b>11</b>	<b>Adding Transactional Memory (tmExtensions).....</b>	<b>78</b>
11.1	INTERCEPT ATTRIBUTES .....	78
11.2	INTERCEPT PARAMETERS.....	79
11.3	OBJECT TYPE, CONSTRUCTOR AND POST-CONSTRUCTOR.....	80
11.4	EXTENSION CSRS .....	82
11.5	CONTEXT SWITCH MONITOR (RISCVSWITCH).....	83
11.6	TRANSACTIONAL LOAD AND STORE FUNCTIONS (RISCVTLOAD AND RISCVTSTORE) .....	84
11.7	TRAP AND EXCEPTION RETURN NOTIFIERS (RISCVTRAPNOTIFIER AND RISCVERETNOTIFIER).....	85
11.8	INSTRUCTION DECODE.....	86
11.9	INSTRUCTION DISASSEMBLY.....	87
11.10	INSTRUCTION TRANSLATION.....	87
11.11	MEMORY MODEL IMPLEMENTATION GUIDELINES .....	91
<b>12</b>	<b>Appendix: Standard Instruction Patterns .....</b>	<b>92</b>
12.1	PATTERN RVIP_RD_RS1_RS2 .....	92
12.2	PATTERN RVIP_RD_RS1_SI.....	92
12.3	PATTERN RVIP_RD_RS1_SHIFT.....	92
12.4	PATTERN RVIP_RD_RS1_RS2_RS3.....	92
12.5	PATTERN RVIP_RD_RS1_RS3_SHIFT.....	92
12.6	PATTERN RVIP_FD_FS1_FS2 .....	93
12.7	PATTERN RVIP_FD_FS1_FS2_RM.....	93
12.8	PATTERN RVIP_FD_FS1_FS2_FS3_RM.....	93
12.9	PATTERN RVIP_RD_FS1_FS2 .....	93
12.10	PATTERN RVIP_VD_VS1_VS2_M.....	93
12.11	PATTERN RVIP_VD_VS1_SI_M.....	93
12.12	PATTERN RVIP_VD_VS1_UI_M.....	94

12.13 PATTERN RVIP_VD_VS1_RS2_M.....	94
12.14 PATTERN RVIP_VD_VS1_FS2_M.....	94
12.15 PATTERN RVIP_RD_VS1_RS2 .....	94
12.16 PATTERN RVIP_RD_VS1_M.....	94
12.17 PATTERN RVIP_VD_RS2.....	94
12.18 PATTERN RVIP_FD_VS1.....	95
12.19 PATTERN RVIP_VD_FS2.....	95
<b>13 Appendix: Base Model Interface Service Functions .....</b>	<b>96</b>
13.1 FUNCTION REGISTEREXTCB .....	97
13.2 FUNCTION GETEXTCLIENTDATA .....	98
13.3 FUNCTION GETEXTCONFIG .....	99
13.4 FUNCTION GETXLENMODE .....	100
13.5 FUNCTION GETXLENARCH .....	101
13.6 FUNCTION GETXREGNAME .....	102
13.7 FUNCTION GETFREGNAME.....	103
13.8 FUNCTION GETVREGNAME.....	104
13.9 FUNCTION SETTMODE .....	105
13.10 FUNCTION GETTMODE .....	106
13.11 FUNCTION GETDATAENDIAN.....	107
13.12 FUNCTION READCSR .....	108
13.13 FUNCTION WRITECSR.....	109
13.14 FUNCTION READBASECSR .....	110
13.15 FUNCTION WRITEBASECSR .....	111
13.16 FUNCTION HALT.....	112
13.17 FUNCTION RESTART .....	113
13.18 FUNCTION UPDATEINTERRUPT .....	114
13.19 FUNCTION UPDATEDISABLE .....	115
13.20 FUNCTION TESTINTERRUPT .....	116
13.21 FUNCTION ILLEGALINSTRUCTION .....	117
13.22 FUNCTION ILLEGALVERBOSE .....	118
13.23 FUNCTION VIRTUALINSTRUCTION .....	119
13.24 FUNCTION VIRTUALVERBOSE .....	120
13.25 FUNCTION TAKEEXCEPTION.....	121
13.26 FUNCTION TAKERESET.....	122
13.27 FUNCTION FETCHINSTRUCTION .....	123
13.28 FUNCTION DISASSINSTRUCTION .....	124
13.29 FUNCTION INSTRUCTIONENABLED .....	125
13.30 FUNCTION MORPHEXTERNAL .....	126
13.31 FUNCTION MORPHILLEGAL.....	127
13.32 FUNCTION MORPHVIRTUAL.....	128
13.33 FUNCTION GETVMIREG .....	129
13.34 FUNCTION GETVMIREGFS .....	130
13.35 FUNCTION WRITEREGSIZE .....	131
13.36 FUNCTION WRITEREG.....	132
13.37 FUNCTION GETFPFLAGSMT .....	133
13.38 FUNCTION GETDATAENDIANMT .....	134
13.39 FUNCTION CHECKLEGALRMMT.....	135
13.40 FUNCTION MORPHTRAPTVM .....	136
13.41 FUNCTION MORPHVOP .....	137
13.42 FUNCTION NEWCSR .....	140
13.43 FUNCTION UPDATELDSTDOMAIN .....	141

13.44	FUNCTION NEWTLBENTRY .....	142
13.45	FUNCTION FREETLBENTRY .....	144
<b>14</b>	<b>Appendix: Extension Object Interface Functions .....</b>	<b>145</b>
14.1	FUNCTION RDSNAPCB.....	146
14.2	FUNCTION WRSNAPCB.....	148
14.3	FUNCTION SUPPRESSMEMEXCEPT .....	149
14.4	FUNCTION CUSTOMNMI.....	150
14.5	FUNCTION TRAPNOTIFIER.....	151
14.6	PRETRAPFUNCTION TRAPPRENOTIFIER .....	152
14.7	FUNCTION ERETNOTIFIER.....	153
14.8	FUNCTION RESETNOTIFIER.....	154
14.9	FUNCTION FIRSTEXCEPTION .....	155
14.10	FUNCTION GETINTERRUPTPRI.....	156
14.11	FUNCTION HALTRESTARTNOTIFIER .....	157
14.12	FUNCTION LRSCABORTFN .....	158
14.13	FUNCTION PREMORPH.....	159
14.14	FUNCTION POSTMORPH.....	160
14.15	FUNCTION SWITCHCB .....	161
14.16	FUNCTION TLOAD .....	163
14.17	FUNCTION TSTORE .....	165
14.18	FUNCTION PMAENABLE .....	167
14.19	FUNCTION PMACHECK .....	168
14.20	FUNCTION VMTRAP .....	169
14.21	FUNCTION SETDOMAINNOTIFIER .....	171
14.22	FUNCTION FREEENTRYNOTIFIER .....	172
14.23	FUNCTION RESTRICTIONS CB.....	173

## 1 Introduction

The RISC-V architecture is designed to be extensible. Standard features are grouped into subsets which may or may not be present in a particular implementation, and in addition the architecture permits custom extensions (for example, non-standard instructions or control and status registers, CSRs). This presents an issue when a *model* of such a custom processor is required – how can such a model be easily developed and maintained as the architecture evolves?

This document describes a methodology to simplify enhancement of the generic OVP RISC-V processor model with custom extensions. These extensions can add instructions, registers (including CSRs), net ports, bus ports, FIFO ports and documentation to the base model.

Any of the techniques described in the *Imperas Binary Interception Technology User Guide* can be used to enhance a processor model.

In addition, the OVP RISC-V processor model provides some RISC-V specific integration functionality that simplifies addition of many kinds of feature. This document, from chapter 3, describes these model-specific integration facilities, using examples from a template custom RISC-V model implemented in the `vendor.com` library.

## 2 Building a RISC-V Model

### 2.1 Introduction

A RISC-V processor model can be built from source provided in a riscvOVPsim / riscvOVPsimPlus download or within a VLNV library provided in an OVPsim or Imperas or other installation.

This chapter describes how this source can be built and how the subsequent model binary image can be used in a simulation.

### 2.2 Using Local RISC-V Model Source Directory

To build a source model in a standalone directory an installation of OVPsim ([www.ovpworld.org](http://www.ovpworld.org)) or the Imperas Professional products ([www.imperas.com](http://www.imperas.com)) are required.

As part of these installations a Makefile build system is provided that can be used in a Linux shell or an MSYS shell on Windows.

Below is shown the command line for building the source that is provided within a riscv-ovpsim download from GitHub

```
% cd riscvOVPsim/source
% make -f $IMPERAS_HOME/ImperasLib/buildutils/Makefile.host
```

For example, this may then be executed using the IMPERAS\_ISS by selecting the processor as shown below

```
% $IMPERAS_ISS --processorfile /d/Imperas/work/riscvOVPsim/source/model.so \
--variant RV32I \
--program application.elf
```

### 2.3 Using VLNV Library RISC-V Model Source

To build a source model in a VLNV library an installation of OVPsim ([www.ovpworld.org](http://www.ovpworld.org)) or the Imperas Professional products ([www.imperas.com](http://www.imperas.com)) are required.

As part of these installations a Makefile build system is provided that can be used in a Linux shell or an MSYS shell on Windows.

To show the building of a VLNV library, let us assume that there is a local library source directory at /home/user1/LocalLib/source containing the RISC-V processor source in a standard VLNV structure, for example as vendor/processor/riscv/1.0/model

The following instructions will build the model into an output VLNV binary library directory that, for example we can specify the root as

/home/user1/lib/\$IMPERAS\_ARCH/ImperasLib:

```
% mkdir -p /home/user1/lib/$IMPERAS_ARCH/ImperasLib
```



```
% make -C $IMPERAS_HOME/ImperasLib/buildutils/Makefile.library \  
  VLNVSRC=/home/user1/LocalLib/source \  
  VLNVROOT=/home/user1/lib/$IMPERAS_ARCH/ImperasLib
```

To use this model in a simulation platform or with the IMPERAS\_ISS the following argument should be added to the simulator command line

```
-vlnvroot lib/$IMPERAS_ARCH/ImperasLib
```

For example, this may then be executed using the IMPERAS\_ISS by selecting the processor as shown below

```
% $IMPERAS_ISS --processorvendor vendor -processor name riscv \  
  --vlnvroot /home/user1/lib/$IMPERAS_ARCH/ImperasLib \  
  --variant RV32I \  
  --program application.elf
```

## 2.4 Using a Linked Model

The previous sections have described how a RISC-V model binary image can be created from any source code that is available and used in simulation.

However, when wanting to customize and/or extend the RISC-V processor model some important things should be considered:

- 1) How easy will it be to maintain the RISC-V model?
- 2) How will the RISC-V model be adapted to changes to the specifications?
  - a. Incorporating new specification versions
  - b. Adding new core extensions
- 3) How will the RISC-V model be tested and verified?

The solution to the above points is to use a linked model.

By creating a linked model, the base RISC-V model is used unchanged. This solves the previous issues:

- 1) Imperas maintain the base RISC-V model.
- 2) Imperas will update the RISC-V model to add new extensions and for new versions or modifications of specifications.

As part of doing this configuration parameters are provided that allow any of the current or previous versions to be selected that will configure the processor operations.
- 3) Imperas test and verify all aspects of the RISC-V model so that it can be used as a golden reference

## 3 Introduction to Custom Extensions with Linked Model

Modeling custom features is done by specifying a *custom configuration* and adding a *custom extension library* to the basic RISC-V processor model, using a *linked model*. A linked model is a RISC-V model that refers to all source files in the base model using *links* rather than copying them. This has the advantage that if the base model is updated (to fix bugs or provide new features) then the linked model will automatically inherit those features – in effect, the linked model implements a skin around the base model. This document discusses these features in turn, referring to the `vendor.com` template, as follows:

### 3.1 *Linked Model Creation*

Chapter 4 describes what must be done to create a new linked model.

### 3.2 *Custom Configuration Options*

Chapter 5 describes in detail the custom configuration options that can be specified for a linked model. These options control the availability of standard feature subsets.

### 3.3 *Adding Custom Instructions*

Chapter 6 describes how custom instructions can be added to a RISC-V model.

### 3.4 *Adding Custom CSRs*

Chapter 7 describes how custom CSRs can be added to a RISC-V model, and how the behavior of existing CSRs in the base model can be modified.

### 3.5 *Adding Custom Exceptions*

Chapter 8 describes how custom exceptions can be added to a RISC-V model.

### 3.6 *Adding Custom Local Interrupts*

Chapter 9 describes how custom local interrupts can be added to a RISC-V model.

### 3.7 *Adding Custom FIFO Ports*

Chapter 10 is an advanced example describing how custom FIFO ports and supporting instructions can be added to a RISC-V model.

### 3.8 *Adding Transactional Memory*

Chapter 11 is an advanced example describing how transactional memory and supporting instructions can be added to a RISC-V model.

### 3.9 *Building the linked model*

The linked model is built using the standard Makefile system provided in a product installation, located at `$IMPERAS_HOME`.

To show the building of the linked model let's assume that the `vendor.com` directory, from `$IMPERAS_HOME/ImperasLib/source`, will be copied into a local library source

directory at `/home/user1/LocalLib/source`. The following instructions will build the model into an output directory `/home/user1/lib/$IMPERAS_ARCH/ImperasLib`:

```
% mkdir -p /home/user1/lib/$IMPERAS_ARCH/ImperasLib
% make -C $IMPERAS_HOME/ImperasLib/buildutils/Makefile.library \
    VLNVSRC=/home/user1/LocalLib/source \
    VLNVROOT=/home/user1/lib/$IMPERAS_ARCH/ImperasLib
```

To use this model in a simulation platform the following argument should be added to the simulator command line:

```
-vlnvroot lib/$IMPERAS_ARCH/ImperasLib
```

The model can be used with the IMPERAS\_ISS by inclusion on the command line, using:

```
-processorvendor vendor.com vlnvroot lib/$IMPERAS_ARCH/ImperasLib
```

## 4 Linked Model Creation

To create a custom RISC-V model, create a new vendor directory based on the `vendor.com` template. Beneath the `vendor.com` directory, there are two subdirectories of significance to this process:

1. Directory `processor/riscv/1.0/model` contains the linked RISC-V processor model. Most files in this directory have names of the form `link.baseName.c`, where `baseName.c` is the name of a source file in the base RISC-V model. Within each file, there is a `#include` directive that includes the base model file; for example, file `link.riscvAttrs.c` contains this line:

```
#include "riscv.ovpworld.org/processor/riscv/1.0/model/riscvAttrs.c"
```

In addition, there are three other code files:

- a. `extensionConfig.h`: this contains an enumeration of the various extensions that can be applied to the model. The RISC-V model in the `vendor.com` directory has six separate extensions, each identified by a member of the `extensionID` enumeration;
  - b. `riscvConfigList.c`: This defines the RISC-V variants implemented by this linked model, together with a list of extension libraries that implement particular custom features for each variant;
  - c. `riscvInfo.c`: This identifies the available extensions by name and specifies various other standard RISC-V processor features (for example, which debugger to use).
2. Directory `intercept` contains source of each extension library that adds additional custom features to the base RISC-V model. In the `vendor.com` example, there are six extension libraries, each of which adds a particular class of feature so that the features can be described in stand-alone chapters of this document. A typical custom processor will have a single extension library combining features from one or more of these examples.

To create a new customized RISC-V linked model, first copy the `vendor.com` `intercept` and `processor` directories into a new vendor directory. Depending on the nature of the extensions being added, remove all except one of the subdirectories under the `intercept` directory. These directories implement separate extensions, as follows:

1. `addCSRs`: adds custom CSRs;
2. `addInstructions`: adds four instructions operating on general-purpose registers;
3. `addExceptions`: adds custom exceptions and one simple instruction;
4. `addLocalInterrupts`: adds 2 local interrupts with custom priorities;
5. `fifoExtensions`: adds FIFO ports, custom instructions and documentation;
6. `tmExtensions`: adds custom instructions, exceptions, documentation and implements transactional memory.

Having copied the vendor.com template directories, files in the processor/riscv/1.0/model directory need to be updated, as follows:

#### 4.1 *File extensionConfig.h*

Modify this to contain a single enumeration entry for the new extension, for example:

```
typedef enum extensionIDE {
    EXTID_CUSTOM = 1234,
} extensionID;
```

When multiple extensions are present on a RISC-V processor, the code here is used to distinguish between them so that context information for an extension can be obtained by client code. The value of the enumeration member is arbitrary but must be unique amongst all extensions added to a RISC-V processor.

#### 4.2 *File riscvInfo.c*

This file provides generic information about a processor (not RISC-V specific). Modify it to contain a single vmiVlnvInfo structure for the extension, and a single vmiVlnvInfoList structure referencing it, for example:

```
static const vmiVlnvInfo custom = {
    .vendor      = "custom.com",
    .library     = "intercept",
    .name        = "customExt",
    .version     = "1.0",
};

static const vmiVlnvInfoList customEntry = {
    next : 0,
    info : &custom,
};
```

Then update the info32 and info64 vmiProcessorInfo definitions to include the new customEntry list in the mandatoryExtensions field, and correct the vendor and family fields to match the new vendor name. For example, the info32 entry should be modified like this:

```
static const vmiProcessorInfo info32 = {
    .vlnv.vendor      = "custom.com",
    .vlnv.library     = "processor",
    .vlnv.name        = "riscv",
    .vlnv.version     = "1.0",

    .semihost.vendor  = "riscv.ovpworld.org",
    .semihost.library = "semihosting",
    .semihost.name    = "pk",
    .semihost.version = "1.0",

    .helper.vendor    = "imperas.com",
    .helper.library   = "intercept",
    .helper.name      = "riscv32CpuHelper",
    .helper.version   = "1.0",

    .mandatoryExtensions = &customEntry,

    .elfCode          = 243,
    .endianFixed      = True,
    .endian            = MEM_ENDIAN_LITTLE,
```

```

        .gdbPath           = "$IMPERAS_HOME/lib/$IMPERAS_ARCH/gdb/riscv-none-embed-
gdb" VMI_EXE_SUFFIX,
        .gdbInitCommands = "set architecture riscv:rv32",
        .family           = "Custom",
        .QLQualified      = True
    };

```

The `vendor.com` template example shows how it is possible to add *multiple* extension libraries to a processor, by forming a list of `vmiVlnvInfo` structures, if required.

### 4.3 File `riscvConfigList.c`

This file provides RISC-V-specific information about the processor variants implemented. Each variant is implemented by a single structure of type `riscvConfig`; the multiple possible variants are in a null-terminated list which is returned by function `riscvGetConfigList`. The `vendor.com` template model contains two variants, `RV32X` and `RV64X`, and there is a configuration for each:

```

static const riscvConfig configList[] = {

    {
        .name           = "RV32X",
        .arch           = ISA_U|RV32GC|ISA_X,
        .user_version   = RVUV_DEFAULT,
        .priv_version   = RVPV_DEFAULT,
        .tval_ii_code   = True,
        .ASID_bits      = 9,
        .local_int_num  = 7,           // enable local interrupts 16-22
        .unimp_int_mask = 0x1f0000, // int16-int20 absent
        .extensionConfigs = allExtensions,
    },

    {
        .name           = "RV64X",
        .arch           = ISA_U|RV64GC|ISA_X,
        .user_version   = RVUV_DEFAULT,
        .priv_version   = RVPV_DEFAULT,
        .tval_ii_code   = True,
        .ASID_bits      = 9,
        .local_int_num  = 7,           // enable local interrupts 16-22
        .unimp_int_mask = 0x1f0000, // int16-int20 absent
        .extensionConfigs = allExtensions,
    },

    {0} // null terminator
};

```

The `riscvConfig` structure allows many aspects of the RISC-V processor to be configured without additional extension library effort, so for some processors all required behavior may be described without the requirement for any extension library component. The `riscvConfig` structure is discussed in detail in section 5.

For a new processor with a single variant, modify the null-terminated list to contain a single entry with the correct name with a minimal set of field initializations:

```

static const riscvConfig configList[] = {

    {
        .name           = "variant1",
        .arch           = ISA_U|RV32GC|ISA_X,
        .user_version   = RVUV_DEFAULT,

```

```
.priv_version      = RVPV_DEFAULT,  
.extensionConfigs = allExtensions,  
},  
{0} // null terminator  
};
```

The `extensionConfigs` field is a pointer to a null-terminated list of `riscvExtConfig` structures. Each structure contains an extension id and a pointer to an extension-specific configuration structure (which can often be null, but see following chapters for more detail). To add a single extension, modify the template code like this:

```
static riscvExtConfigCP allExtensions[] = {  
  
    &(const riscvExtConfig){  
        .id      = EXTID_CUSTOM,  
        .userData = 0  
    },  
  
    0 // KEEP LAST: terminator  
};
```

Note that the value `EXTID_CUSTOM` was defined previously in file `extensionConfig.h`.

## 5 Custom Configuration Options

Many features of a custom RISC-V variant can be defined by fields in the configuration structure of type `riscvExtConfig` for that variant. This section describes the configuration structure and the possible field settings. The defaults specified in this structure can generally be overridden by model parameters if required.

The structure is defined in file `riscvConfig.h` in the base model. It contains configuration options, important CSR defaults and CSR write masks:

```
typedef struct riscvConfigS {
    const char      *name;           // variant name

    // fundamental variant configuration
    riscvArchitecture arch;         // variant architecture
    riscvArchitecture archMask;     // read/write bits in architecture
    riscvArchitecture archFixed;   // fixed bits in architecture
    riscvUserVer     user_version;  // user-level ISA version
    riscvPrivVer     priv_version;  // privileged architecture version
    riscvVectVer     vect_version;  // vector architecture version
    riscvBitManipVer bitmanip_version; // bitmanip architecture version
    riscvBitManipSet bitmanip_absent; // bitmanip absent extensions
    riscvCryptoVer   crypto_version; // cryptographic architecture version
    riscvCryptoSet   crypto_absent;  // cryptographic absent extensions
    riscvHypVer      hyp_version;    // hypervisor architecture version
    riscvDebugVer    dbg_version;    // debugger architecture version
    riscvZfinxVer    Zfinx_version;  // Zfinx version
    riscvFp16Ver     fp16_version;   // 16-bit floating point version
    riscvFSMode      mstatus_fs_mode; // mstatus.FS update mode
    riscvDMMode      debug_mode;     // is Debug mode implemented?
    riscvDERETMode   debug_eret_mode; // debug mode MRET, SRET or DRET action
    const char      **members;       // cluster member variants

    // configuration not visible in CSR state
    Uns64          reset_address;    // reset vector address
    Uns64          nmi_address;      // NMI address
    Uns64          debug_address;    // debug vector address
    Uns64          dexc_address;     // debug exception address
    Uns64          unimp_int_mask;   // mask of unimplemented interrupts
    Uns64          force_mideleg;    // always-delegated M-mode interrupts
    Uns64          force_sideleg;    // always-delegated S-mode interrupts
    Uns64          no_ideleg;        // non-delegated interrupts
    Uns64          no_e deleg;       // non-delegated exceptions
    Uns64          ecode_mask;       // implemented bits in xcause.ecode
    Uns64          ecode_nmi;        // exception code for NMI
    Uns32          counteren_mask;   // counter-enable implemented mask
    Uns32          noinhibit_mask;   // counter no-inhibit mask
    Uns32          local_int_num;    // number of local interrupts
    Uns32          lr_sc_grain;      // LR/SC region grain size
    Uns32          PMP_grain;        // PMP region grain size
    Uns32          PMP_registers;    // number of implemented PMP registers
    Uns32          PMP_max_page;     // maximum size of PMP page to map
    Uns32          Sv_modes;         // bit mask of valid Sv modes
    Uns32          numHarts;         // number of hart contexts if MPCore
    Uns32          tvec_align;       // trap vector alignment (vectored mode)
    Uns32          ELEN;             // ELEN (vector extension)
    Uns32          SLEN;             // SLEN (vector extension)
    Uns32          VLEN;             // VLEN (vector extension)
    Uns32          SEW_min;          // minimum SEW (vector extension)
    Uns32          ASID_cache_size;  // ASID cache size
    Uns16          tinfo;            // tinfo default value (all triggers)
    Uns8           ASID_bits;        // number of implemented ASID bits
    Uns8           VMID_bits;        // number of implemented VMID bits
    Uns8           trigger_num;      // number of implemented triggers
}
```



```

Uns8      mcontext_bits;    // implemented bits in mcontext
Uns8      scontext_bits;    // implemented bits in scontext
Uns8      mvalue_bits;      // implemented bits in textra.mvalue
Uns8      svalue_bits;      // implemented bits in textra.svalue
Uns8      mcontrol_maskmax; // configured value of mcontrol.maskmax
Uns8      dcsr_ebreak_mask; // mask of writable dcsr.ebreak* bits
Bool      isPSE;            // whether a PSE (internal use only)
Bool      endianFixed;      // endianness is fixed (UBE/SBE/MBE r/o)
Bool      ABI_d;            // ABI uses D registers for parameters
Bool      MXL_writable;      // writable bits in misa.MXL
Bool      SXL_writable;      // writable bits in mstatus.SXL
Bool      UXL_writable;      // writable bits in mstatus.UXL
Bool      VSXL_writable;     // writable bits in mstatus.VSXL
Bool      Zvlsseg;          // Zvlsseg implemented?
Bool      Zvamo;            // Zvamo implemented?
Bool      Zvediv;           // Zvediv implemented?
Bool      Zvqmac;           // Zvqmac implemented?
Bool      unitStrideOnly;    // only unit-stride operations supported
Bool      noFaultOnlyFirst; // fault-only-first instructions absent?
Bool      updatePTEA;        // hardware update of PTE A bit?
Bool      updatePTED;        // hardware update of PTE D bit?
Bool      unaligned;         // whether unaligned accesses supported
Bool      unalignedAMO;      // whether AMO supports unaligned
Bool      wfi_is_nop;        // whether WFI is treated as NOP
Bool      mtvec_is_ro;       // whether mtvec is read-only
Bool      cycle_undefined;   // whether cycle CSR is undefined
Bool      time_undefined;    // whether time CSR is undefined
Bool      instret_undefined; // whether instret CSR is undefined
Bool      tinfo_undefined;   // whether tinfo CSR is undefined
Bool      tcontrol_undefined; // whether tcontrol CSR is undefined
Bool      mcontext_undefined; // whether mcontext CSR is undefined
Bool      scontext_undefined; // whether scontext CSR is undefined
Bool      mscontext_undefined; // whether mscontext CSR is undefined
Bool      hcontext_undefined; // whether hcontext CSR is undefined
Bool      amo_trigger;       // whether triggers used with AMO
Bool      no_hit;            // whether tdata1.hit is unimplemented
Bool      no_sselect_2;      // whether textra.sselect=2 is illegal
Bool      d_requires_f;      // whether misa D requires F to be set
Bool      mstatus_FS_zero;    // whether mstatus.FS hardwired to zero
Bool      xret_preserves_lr; // whether xRET preserves current LR
Bool      require_vstart0;    // require vstart 0 if uninterruptible?
Bool      align_whole;        // whole register load aligned to hint?
Bool      vill_trap;         // trap instead of setting vill?
Bool      enable_CSR_bus;     // enable CSR implementation bus
Bool      mcounteren_present; // force mcounteren to be present
Bool      PMP_undefined;     // force all PMP registers undefined
Bool      external_int_id;    // enable external interrupt ID ports
Bool      tval_zero;          // whether [smultval are always zero
Bool      tval_zero_ebreak;   // whether [smultval always zero on ebreak
Bool      tval_ii_code;       // instruction bits in [smultval for
                                // illegal instruction exception?

// CLIC configuration
Uns32     CLICLEVELS;        // number of CLIC interrupt levels
Bool      externalCLIC;      // is CLIC externally implemented?
Bool      CLICANDBASIC;      // whether implements basic mode also
Uns8      CLICVERSION;       // CLIC version
Uns8      CLICINTCTLBITS;    // bits implemented in clicintctl[i]
Uns8      CLICCFGBITS;       // bits implemented for cliccfg.nmbits
Uns8      CLICCFGLBITS;      // bits implemented for cliccfg.nlbits
Bool      CLICSELHVEC;        // selective hardware vectoring?
Bool      CLICXNXTI;         // *nxti CSRs implemented?
Bool      CLICXCSW;          // *scratchcs* CSRs implemented?
Bool      tvt_undefined;     // whether *vtv CSRs are undefined
Bool      intthresh_undefined; // whether *intthresh CSRs undefined
Bool      mclicbase_undefined; // whether mclicbase CSR is undefined

// Hypervisor configuration
Uns8      GEILEN;            // number of guest external interrupts
Bool      xtinst_basic;      // only pseudo-instruction in xtinst
    
```

```

// CSR register values
struct {
    CSR_REG_DECL (mvendorid);    // mvendorid value
    CSR_REG_DECL (marchid);     // marchid value
    CSR_REG_DECL (mimpid);      // mimpid value
    CSR_REG_DECL (mhartid);     // mhartid value
    CSR_REG_DECL (mtvec);        // mtvec value
    CSR_REG_DECL (mstatus);     // mstatus reset value
    CSR_REG_DECL (mclibase);    // mclibase value
} csr;

// CSR register masks
struct {
    CSR_REG_DECL (mtvec);        // mtvec mask
    CSR_REG_DECL (stvec);        // stvec mask
    CSR_REG_DECL (utvec);        // utvec mask
    CSR_REG_DECL (mtvt);        // mtvec mask
    CSR_REG_DECL (stvt);        // stvec mask
    CSR_REG_DECL (utvt);        // utvec mask
    CSR_REG_DECL (tdata1);      // tdata1 mask
    CSR_REG_DECL (mip);         // mip mask
    CSR_REG_DECL (sip);         // sip mask
    CSR_REG_DECL (uip);         // uip mask
    CSR_REG_DECL (hip);         // hip mask
} csrMask;

// custom documentation
const char    **specificDocs;   // custom documentation
riscvDocFn    restrictionsCB;   // custom restrictions

// extension configuration information
riscvExtConfigCPP extensionConfigs; // null-terminated list of extension
// configurations

} riscvConfig;

```

Structures of this type should be defined in file `riscvConfigList.c` in the linked model, as shown in section 4. *Always use field initialization by name (as shown in that section) when specifying field values, to avoid any dependency on field order.*

Fields in this structure are described below, in functional groups. Having copied the template model, modify any custom definitions required by referring to these tables.

## 5.1 Fundamental Configuration

Fields here are applicable to all RISC-V variants. All field defaults can be overridden using a model parameter of the same name unless otherwise specified.

Field	Type	Description
name	String	This is the name of the variant and must always be specified as a non-null string. It is used to select this configuration when it matches the <code>variant</code> parameter specified for the processor instantiation.
arch	riscvArchitecture	This specifies the processor architecture (whether it is 32 or 64 bit, and which standard extensions are present). The value is a bitmask enumeration with values defined in file <code>riscvVariant</code> in the base model. The default value is typically formed by bitwise-or of values from this file, for example, the value: <code>ISA_U RV32GC ISA_X</code> specifies an RV32 processor with C,D,F,U and custom extensions (X). The value in a processor instance can be indirectly modified by parameters <code>misa_MXL</code> , <code>misa_Extensions</code> and <code>add_Extensions</code> .
archMask	riscvArchitecture	This specifies writable architecture bits in the <code>misa</code> CSR. Non-writable bits will be fixed to 1 or 0 depending on the value in the <code>arch</code> field. The value in a processor instance can be indirectly modified by parameters <code>misa_MXL_mask</code> , <code>misa_Extensions_mask</code> and <code>add_Extensions_mask</code> .
archFixed	riscvArchitecture	This specifies bits in the <code>misa</code> CSR that may <i>never</i> be overridden by parameters. For example, a value of <code>ISA_V</code> means that the vector extension may never be configured or deconfigured. It is usually easier to specify as a bitwise-negation of features that <i>may</i> be configured or deconfigured. This field cannot be overridden by a parameter.
user_version	riscvUserVer	This specifies the implemented Unprivileged Architecture version, defined by the <code>riscvUserVer</code> enumeration: <pre>typedef enum riscvUserVerE {     RVUV_2_2,     RVUV_2_3,     RVUV_20190305,     RVUV_DEFAULT = RVUV_20190305, } riscvUserVer;</pre>
priv_version	riscvPrivVer	This specifies the implemented Privileged Architecture version, defined by the <code>riscvPrivVer</code> enumeration: <pre>typedef enum riscvPrivVerE {     RVPV_1_10,     RVPV_1_11,     RVPV_20190405,     RVPV_1_12,     RVPV_MASTER = RVPV_1_12,     RVPV_DEFAULT = RVPV_20190405, } riscvPrivVer;</pre>
endianFixed	Bool	This specifies that MBE, SBE and UBE fields in <code>mstatus</code> are read-only (data endianness is fixed). This parameter only has effect for privileged version <code>RVPC_1_12</code> and later.

## 5.2 Interrupts and Exceptions

Fields here control interrupt and exception state. All field defaults can be overridden using a model parameter of the same name.

Field	Type	Description
reset_address	Uns64	This specifies the address to jump to on reset.
nmi_address	Uns64	This specifies the address to jump to on NMI.
unimp_int_mask	Uns64	This is a bitmask specifying interrupts that are <i>not</i> implemented. It determines the net ports created and the writable values of some CSRs (e.g. mie, mideleg).
force_mideleg	Uns64	This is a bitmask specifying interrupts that are always delegated from M-mode to lower execution levels.
force_sideleg	Uns64	This is a bitmask specifying interrupts that are always delegated from S-mode to lower execution levels.
no_ideleg	Uns64	This is a bitmask specifying interrupts that can never be delegated to lower execution levels.
no_edeleg	Uns64	This is a bitmask specifying exceptions that can never be delegated to lower execution levels.
ecode_mask	Uns64	This is a mask specifying writable bits in <code>xcause.ecode</code> .
ecode_nmi	Uns64	This defines the cause reported by an NMI interrupt.
local_int_num	Uns32	This defines the number of local interrupts implemented. For RV32, the maximum number is 16 and for RV64 the maximum number is 48. If the CLIC is implemented, the maximum number is 4080 in both cases.
tvec_align	Uns32	This specifies the hardware-enforced alignment of <code>xtvec</code> CSRs. For example, a value of 64 implies that <code>mtvec</code> is masked to enforce 64-byte alignment.
mtvec_is_ro	Bool	This specifies whether <code>mtvec</code> is a read-only register.
xret_preserves_lr	Bool	This specifies whether an <code>xRET</code> instruction preserves the current value of the link register (LR).
tval_zero	Bool	This specifies whether <code>xtval</code> registers are always zero.
tval_ii_code	Bool	This specifies whether <code>xtval</code> registers are filled with the instruction value for Illegal Instruction exceptions. If <code>False</code> , <code>xtval</code> registers are zeroed instead.

## 5.3 Instruction and CSR Behavior

Fields here define instruction and CSR behavior. All field defaults can be overridden using a model parameter of the same name.

Field	Type	Description
wfi_is_nop	Bool	This specifies whether the <code>WFI</code> instruction is treated as a <code>NOP</code> (if <code>False</code> , it will cause execution of the current core to suspend waiting for an interrupt).
cycle_undefined	Bool	This specifies whether the <code>cycle</code> CSR is undefined (its behavior must be emulated by a trap).
time_undefined	Bool	This specifies whether the <code>time</code> CSR is undefined (its behavior must be emulated by a trap).
instret_undefined	Bool	This specifies whether the <code>instret</code> CSR is undefined (its behavior must be emulated by a trap).
counteren_mask	Uns32	This is a bitmask specifying writable bits in <code>mcounteren/scounteren</code> registers.
noinhibit_mask	Uns32	This is a bitmask specifying counters that may <i>not</i> be inhibited using <code>mcountinhibit</code> (in other words, 1 bits in this mask are always 0 in <code>mcountinhibit</code> ).

## 5.4 CLIC

Fields here define the *Core Local Interrupt Controller (CLIC)* configuration. All field defaults can be overridden using a model parameter of the same name.

Field	Type	Description
CLICLEVELS	Uns32	If zero, this specifies that the CLIC is unimplemented, otherwise it must be a number in the range 2-256, specifying the number of levels implemented.
externalCLIC	Bool	If the CLIC is implemented (CLICLEVELS!=0), this specifies whether the internal CLIC model is used or whether the CLIC is modeled by an external memory-mapped component. If implemented externally, five new net ports are created that must be connected to the external CLIC model: <ol style="list-style-type: none"> <li>1. <code>irq_id_i</code>: input port written with highest-priority pending interrupt;</li> <li>2. <code>irq_lev_i</code>: input port written with level of highest-priority pending interrupt;</li> <li>3. <code>irq_sec_i</code>: input port written with execution level of highest-priority pending interrupt;</li> <li>4. <code>irq_shv_i</code>: input port written with indication of whether the highest-priority pending interrupt uses selective hardware vectoring;</li> <li>5. <code>irq_i</code>: active-high input indicating that an external CLIC interrupt is pending.</li> </ol>
CLICANDBASIC	Bool	If the CLIC is implemented internally (CLICLEVELS!=0 and externalCLIC=False), this specifies whether basic interrupt control is also implemented. If the CLIC is not implemented internally, the field is ignored.
CLICVERSION	Uns32	If the CLIC is implemented (CLICLEVELS!=0), this specifies the CLIC version. If the CLIC is not implemented, the field is ignored.
CLICINTCTLBITS	Uns32	If the CLIC is implemented internally (CLICLEVELS!=0 and externalCLIC=False), this specifies the number of bits implemented in <code>clicintctl[i]</code> . If the CLIC is not implemented internally, the field is ignored.
CLICCFGMBITS	Uns32	If the CLIC is implemented internally (CLICLEVELS!=0 and externalCLIC=False), this specifies the number of bits implemented in <code>cliccfg.nmbits</code> . If the CLIC is not implemented internally, the field is ignored.
CLICCFGNBITS	Uns32	If the CLIC is implemented internally (CLICLEVELS!=0 and externalCLIC=False), this specifies the number of bits implemented in <code>cliccfg.nlbits</code> . If the CLIC is not implemented internally, the field is ignored.
CLICSELHVEC	Bool	If the CLIC is implemented internally (CLICLEVELS!=0 and externalCLIC=False), this specifies whether <i>selective hardware vectoring</i> is supported. If the CLIC is not implemented internally, the field is ignored.
CLICXNXTI	Bool	If the CLIC is implemented (CLICLEVELS!=0), this specifies whether <code>xnxti</code> CSRs are implemented. If the CLIC is not implemented, the field is ignored.
CLICXCSW	Bool	If the CLIC is implemented (CLICLEVELS!=0), this specifies whether <code>xscratchcsx</code> CSRs are implemented. If the CLIC is not implemented, the field is ignored.
tvt_undefined	Bool	If the CLIC is implemented (CLICLEVELS!=0), this specifies whether <code>xtvt</code> CSRs are implemented – if <code>False</code> , then <code>xtvec</code> registers are used instead. If the CLIC is not implemented, the field is ignored.
intthresh_undefined	Bool	If the CLIC is implemented (CLICLEVELS!=0), this specifies whether <code>xinthreash</code> CSRs are undefined. If the CLIC is not implemented, the field is ignored.
mclicbase_undefined	Bool	If the CLIC is implemented (CLICLEVELS!=0), this specifies whether the <code>mclicbase</code> CSR is undefined. If the CLIC is not implemented, the field is ignored.

## 5.5 Memory Subsystem

Fields here define memory model features. All field defaults can be overridden using a model parameter of the same name.

Field	Type	Description
lr_sc_grain	Uns32	This defines the lock granularity for LR and SC instructions. It must be a power of 2 in the range 1..(1<<16).
PMP_grain	Uns32	This defines the minimum granularity for PMP regions in the standard format (0: 4 bytes, 1: 8 bytes, etc).
PMP_registers	Uns32	This defines the number of PMP registers present (maximum of 64). A value of 0 indicates PMP is absent.
PMP_decompose	Bool	This indicates whether unaligned PMP accesses are decomposed into individual aligned accesses (thus allowing accesses that straddle region boundaries).
Sv_modes	Uns32	This is a bitmask specifying supported virtual memory modes in the standard format (for example 1<<8 is Sv39).
ASID_bits	Uns32	This defines the number of bits implemented in the ASID (maximum of 9 for RV32 and 16 for RV64). A value of 0 indicates that ASID is not implemented.
updatePTEA	Bool	This indicates whether hardware update of page table entry A bit is implemented.
updatePTED	Bool	This indicates whether hardware update of page table entry D bit is implemented.
unaligned	Bool	This indicates whether unaligned accesses are supported.
unalignedAMO	Bool	This indicates whether unaligned accesses are supported for AMO operations.

## 5.6 Floating Point

Fields here are applicable when floating point is implemented (D or F extensions present). All field defaults can be overridden using a model parameter of the same name, except for `fp16_version`, for which parameter `zfh` must be used in the vector extension is not configured.

Field	Type	Description
mstatus_fs_mode	riscvFSMode	This field specifies how the implementation-dependent <code>mstatus.FS</code> field is updated by the model. The possible behaviors are specified by the <code>riscvFSMode</code> enumeration: <pre>typedef enum riscvFSModeE {     RVFS_WRITE_NZ,     RVFS_WRITE_ANY,     RVFS_ALWAYS_DIRTY, } riscvFSMode;</pre>
d_requires_f	Bool	This field specifies whether <code>misa.D</code> can only be set if <code>misa.F</code> is also set.
Zfinx_version	riscvZfinxVer	This field specifies whether, the <code>Zfinx</code> option is specified, and if so with what version. The possible values are specified by the <code>riscvZfinxVer</code> enumeration: <pre>typedef enum riscvZfinxVerE {     RVZFINX_NA, // Zfinx not implemented     RVZFINX_0_4, // Zfinx version 0.4 } riscvZfinxVer;</pre>
fp16_version	riscvFP16Ver	This parameter allows a format to be selected for 16-bit floating point. Values are defined by the <code>riscvFP16Ver</code> enumeration: <pre>typedef enum riscvFP16VerE {     RVFP16_NA, // 16-bit FP not supported     RVFP16_IEEE754, // IEEE half-precision format     RVFP16_BFLOAT16 // BFLOAT16 format } riscvFP16Ver;</pre>

## 5.7 Vector Extension

Fields here are applicable when the Vector Extension (V) is implemented. All field defaults can be overridden using a model parameter of the same name.

Field	Type	Description
vect_version	riscvVectVer	This specifies the implemented Vector Extension version, defined by the <code>riscvVectVer</code> enumeration: <pre>typedef enum riscvVectVerE {     RVVV_0_7_1,     RVVV_0_7_1_P,     RVVV_0_8_20190906,     RVVV_0_8_20191004,     RVVV_0_8_20191117,     RVVV_0_8_20191118,     RVVV_0_8,     RVVV_0_9,     RVVV_1_0_20210130,     RVVV_MASTER,     RVVV_LAST,     RVVV_DEFAULT = RVVV_1_0_20210130, } riscvVectVer;</pre>
ELEN	Uns32	This specifies the maximum vector element size, in bits (32 or 64).
SLEN	Uns32	This specifies the vector stride, in bits. From Vector Extension version 1.0, this must match <code>VLEN</code> .
VLEN	Uns32	This specifies the vector size, in bits (a power of two in the range 32 to 65536).
SEW_min	Uns32	This specifies the minimum vector element size, in bits. If zero, a value of 8 is assumed.
Zvlsseg	Bool	This specifies whether the <code>Zvlsseg</code> extension is implemented.
Zvamo	Bool	This specifies whether the <code>Zvamo</code> extension is implemented.
Zvediv	Bool	This specifies whether the <code>Zvediv</code> extension is implemented. <i>Note: this must currently always be False because the base model does not implement this extension.</i>
Zvqmac	Bool	This specifies whether the <code>Zvqmac</code> extension is implemented.
unitStrideOnly	Bool	This specifies whether only unit-stride vector loads and stores are supported. If <code>True</code> , then any other vector load/store instruction will be reported as an Illegal Instruction. <i>Note that such behavior is not compliant with the Vector Extension specification.</i>
noFaultOnlyFirst	Bool	This specifies whether <i>fault-only-first</i> vector loads are unimplemented. If <code>True</code> , then any other vector fault-only-first instruction will be reported as an Illegal Instruction. <i>Note that such behavior is not compliant with the Vector Extension specification.</i>
require_vstart0	Bool	This specifies whether vector instructions that cannot be interrupted by a memory access fault require the <code>vstart</code> CSR to be zero. If <code>True</code> , then attempting to execute a non-memory vector instruction with <code>vstart != 0</code> will cause an Illegal Instruction exception.
align_whole	Bool	This specifies whether whole-register load addresses must be aligned using the encoded <code>EEW</code> .
vill_trap	Bool	This specifies whether illegal <code>vtype</code> values cause a trap.

## 5.8 Bit Manipulation Extension

Fields here are applicable when the Bit Manipulation Extension (B) is implemented. All field defaults can be overridden using a model parameter of the same name.

Field	Type	Description
bitmanip_version	riscvBitManipVer	This specifies the implemented Bit Manipulation Extension version, defined by the <code>riscvBitManipVer</code> enumeration: <pre>typedef enum riscvBitManipVerE {     RVBV_0_90,     RVBV_0_91,     RVBV_0_92,     RVBV_0_93,     RVBV_LAST,     RVBV_DEFAULT = RVBV_0_92, } riscvBitManipVer;</pre>
bitmanip_absent	riscvBitManipSet	By default, all Bit Manipulation Extension instruction subsets are implemented. This field is a bitmask allowing <i>unimplemented</i> subsets to be specified: <pre>typedef enum riscvBitManipSetE {     RVBS_Zba = (1&lt;&lt;0),     RVBS_Zbb = (1&lt;&lt;1),     RVBS_Zbc = (1&lt;&lt;2),     RVBS_Zbe = (1&lt;&lt;3),     RVBS_Zbf = (1&lt;&lt;4),     RVBS_Zbm = (1&lt;&lt;5),     RVBS_Zbp = (1&lt;&lt;6),     RVBS_Zbr = (1&lt;&lt;7),     RVBS_Zbs = (1&lt;&lt;8),     RVBS_Zbt = (1&lt;&lt;9),     RVBS_Zbbp = RVBS_Zbb RVBS_Zbp } riscvBitManipSet;</pre> <p>The presence or absence of each of these subsets can be individually specified using parameters <code>Zba</code>, <code>Zbb</code>, <code>Zbc</code>, <code>Zbe</code>, <code>Zbf</code>, <code>Zbm</code>, <code>Zbp</code>, <code>Zbr</code>, <code>Zbs</code> and <code>Zbt</code>.</p>



## 5.9 Cryptographic Extension

Fields here are applicable when the Cryptographic Extension (K) is implemented. All field defaults can be overridden using a model parameter of the same name.

Field	Type	Description
crypto_version	riscvCryptoVer	This specifies the implemented Cryptographic Extension version, defined by the <code>riscvCryptoVer</code> enumeration: <pre>typedef enum riscvCryptoVerE {     RVKV_0_7_2,     RVKV_0_8_1,     RVKV_LAST,     RVKV_DEFAULT = RVKV_0_7_2 } riscvCryptoVer;</pre>
crypto_absent	riscvCryptoSet	By default, all Cryptographic Extension instruction subsets are implemented. This field is a bitmask allowing <i>unimplemented</i> subsets to be specified: <pre>typedef enum riscvCryptoSetE {     RVKS_Zkb = (1&lt;&lt;0),     RVKS_Zkg = (1&lt;&lt;1),     RVKS_Zkr = (1&lt;&lt;2),     RVKS_Zknd = (1&lt;&lt;3),     RVKS_Zkne = (1&lt;&lt;4),     RVKS_Zknh = (1&lt;&lt;5),     RVKS_Zksed = (1&lt;&lt;6),     RVKS_Zksh = (1&lt;&lt;7), } riscvCryptoSet;</pre> <p>The presence or absence of each of these subsets can be individually specified using parameters <code>Zkb</code>, <code>Zkg</code>, <code>Zkr</code>, <code>Zknd</code>, <code>Zkne</code>, <code>Zknh</code>, <code>Zksed</code> and <code>Zksh</code>.</p>

## 5.10 Hypervisor Extension

Fields here are applicable when the Hypervisor Extension (H) is implemented. All field defaults can be overridden using a model parameter of the same name.

Field	Type	Description
hyp_version	riscvHypVer	This specifies the implemented Hypervisor Extension version, defined by the <code>riscvHypVer</code> enumeration: <pre>typedef enum riscvHypVerE {     RVHV_0_6_1,     RVHV_LAST,     RVHV_DEFAULT = RVHV_0_6_1, } riscvHypVer;</pre>
VMID_bits	Uns32	This defines the number of bits implemented in the VMID (maximum of 7 for RV32 and 14 for RV64).
GEILEN	Uns32	This specifies the number of guest external interrupts implemented (maximum of 31 for RV32 and 63 for RV64).
xtinst_basic	Bool	If <code>True</code> , this specifies that only pseudo-instructions are reported by <code>htinst/mtinst</code> ; if <code>False</code> , then true instructions can be reported as well.

## 5.11 Debug Mode

Fields here are applicable when Debug mode is required. All field defaults can be overridden using a model parameter of the same name.

Field	Type	Description
debug_version	riscvDebugVer	This specifies the implemented Debug version, defined by the <code>riscvDebugVer</code> enumeration: <pre>typedef enum riscvDebugVerE {     RVDBG_0_13_2, // 0.13.2-DRAFT     RVDBG_0_14_0, // 0.14.0-DRAFT     RVDBG_DEFAULT = RVDBG_0_14_0 } riscvDebugVer;</pre>
debug_mode	riscvDMMode	This field specifies how Debug mode is implemented. The possible behaviors are specified by the <code>riscvDMMode</code> enumeration: <pre>typedef enum riscvDMModeE {     RVDM_NONE,     RVDM_VECTOR,     RVDM_INTERRUPT,     RVDM_HALT, } riscvDMMode;</pre>
debug_address	Uns64	This specifies the address to jump to on a debug event (when <code>debug_mode</code> is <code>RVDM_VECTOR</code> ).
dexc_address	Uns64	This specifies the address to jump to on a debug exception (when <code>debug_mode</code> is <code>RVDM_VECTOR</code> ).
debug_eret_mode	riscvDERETMode	This specifies the required behavior when <code>MRET</code> , <code>SRET</code> or <code>URET</code> instructions are executed in Debug mode. The possible behaviors are specified by the <code>riscvDERETMode</code> enumeration: <pre>typedef enum riscvDERETModeE {     RVDRM_NOP, // treat as NOP     RVDRM_JUMP, // jump to dexc_address     RVDRM_TRAP, // trap to dexc_address } riscvDERETMode</pre> <p>For <code>RVDRM_JUMP</code>, the instruction is considered to have retired. For <code>RVDRM_TRAP</code>, the instruction is <i>not</i> considered to have retired.</p>

## 5.12 Trigger Module

Fields here are applicable when the Trigger Module is configured. All field defaults can be overridden using a model parameter of the same name.

Field	Type	Description
trigger_num	Uns32	This field specifies how many trigger registers are implemented. If <code>trigger_num</code> is 0, the trigger module is not configured.
tinfo	Uns32	This specifies the implemented trigger types. Currently, only trigger types 0 and 2 are supported by the base model, so <code>tinfo</code> should be set to either 0x4 or 0x5.
mcontext_bits	Uns32	This specifies the number of implemented bits in the <code>mcontext</code> register.
scontext_bits	Uns32	This specifies the number of implemented bits in the <code>scontext</code> register.
mvalue_bits	Uns32	This specifies the number of implemented bits in the <code>textra32/textra64</code> <code>mvalue</code> field. If zero, the <code>mselect</code> field in the same register is tied to zero.
svalue_bits	Uns32	If Supervisor mode is present, this specifies the number of implemented bits in the <code>textra32/textra64</code> <code>svalue</code> field. If zero, the <code>sselect</code> field in the same register is tied to zero.
mcontrol_maskmax	Uns32	This specifies the value of the <code>mcontrol.maskmax</code> field.
tinfo_undefined	Bool	This specifies that the <code>tinfo</code> register is undefined, and that accesses to it will trap to Machine mode.
tcontrol_undefined	Bool	This specifies that the <code>tcontrol</code> register is undefined, and that accesses to it will trap to Machine mode.
mcontext_undefined	Bool	This specifies that the <code>mcontext</code> register is undefined, and that accesses to it will trap to Machine mode.
scontext_undefined	Bool	This specifies that the <code>scontext</code> register is undefined, and that accesses to it will trap to Machine mode.
mscontext_undefined	Bool	This specifies that the <code>mscontext</code> register is undefined, and that accesses to it will trap to Machine mode Debug Version 0.14.0 and later).
hcontext_undefined	Bool	This specifies that the <code>hcontext</code> register is undefined, and that accesses to it will trap to Machine mode (when Hypervisor extension is present).
amo_trigger	Bool	This specifies whether AMO operations cause load/store triggers to be activated.
no_hit	Bool	This specifies whether the optional <code>hit</code> bit in <code>tdata1</code> is unimplemented.
no_sselect_2	Bool	If Supervisor mode is present, this specifies whether the <code>sselect</code> field in <code>textra32/textra64</code> registers is unable to hold value 2 (indicating match by ASID is not allowed).

### 5.13 Multicore variants

Fields here are applicable only for multicore processor variants (SMP or AMP). All field defaults can be overridden using a model parameter of the same name.

Field	Type	Description
<code>numHarts</code>	Uns32	This field specifies how many processors are implemented beneath the root level in a multicore variant. A value of 0 specifies that the processor is not a multicore variant.
<code>members</code>	Uns32	For a multicore variant ( <code>numHarts!=0</code> ), a null value for this parameter specifies that the processor is an SMP processor. If <code>numHarts!=0</code> and <code>members</code> is non-null, then <code>members</code> must be a list of <code>numHarts</code> strings, each string specifying the variant name of a <i>cluster member</i> . The cluster member name must be the name of another variant in the current processor configuration list. The processor hierarchy will then be constructed with a heterogeneous set of variants, where the <i>n</i> th hart is of the type corresponding to the <i>n</i> th element of <code>members</code> .

### 5.14 CSR Default Values

The `csr` field contains default values for some CSRs that are typically read-only or not otherwise fully configurable by other options mentioned above. All field defaults can be overridden using a model parameter of the same name unless otherwise stated.

Field	Type	Description
<code>csr.mvendorid</code>	Uns64	This field specifies the default value of the <code>mvendorid</code> CSR.
<code>csr.marchid</code>	Uns64	This field specifies the default value of the <code>marchid</code> CSR.
<code>csr.mimpid</code>	Uns64	This field specifies the default value of the <code>mimpid</code> CSR.
<code>csr.mhartid</code>	Uns64	This field specifies the default value of the <code>mhartid</code> CSR.
<code>csr.mtvec</code>	Uns64	This field specifies the reset value of the <code>mtvec</code> CSR.
<code>csr.mstatus</code>	Uns64	This field specifies the default value of the <code>mstatus</code> CSR. Fields <code>mstatus.FS</code> and <code>mstatus.VS</code> can be overridden using parameters <code>mstatus_FS</code> and <code>mstatus_VS</code> , respectively.
<code>csr.mclibase</code>	Uns64	This field specifies the default value of the <code>mclibase</code> CSR (significant only if the CLIC is present and internally implemented).

### 5.15 CSR Masks

The `csrMask` field defines write masks for some CSRs. These masks are all ignored if the field is zero, otherwise they define writable bits for CSRs with the same name.

Field	Type	Description
<code>csrMask.mtvec</code>	Uns64	This field specifies the write mask of the <code>mtvec</code> CSR.
<code>csrMask.stvec</code>	Uns64	This field specifies the write mask of the <code>stvec</code> CSR.
<code>csrMask.utvec</code>	Uns64	This field specifies the write mask of the <code>utvec</code> CSR.
<code>csrMask.mtvt</code>	Uns64	This field specifies the write mask of the <code>mtvt</code> CSR.
<code>csrMask.stvt</code>	Uns64	This field specifies the write mask of the <code>stvt</code> CSR.
<code>csrMask.utvt</code>	Uns64	This field specifies the write mask of the <code>utvt</code> CSR.

## 6 Adding Custom Instructions (addInstructions)

The `addInstructions` extension demonstrates how to add custom instructions to a RISC-V model. The extension adds four instructions in the custom space of a RISC-V processor. Each instruction takes two 32-bit GPR inputs (`rs1` and `rs2`) and writes a result to a target GPR (`rd`), as follows:

### chacha20qr1

```
rd = ((rs1 ^ rs2) << 16) | ((rs1 ^ rs2) >> (32-16))
```

### chacha20qr2

```
rd = ((rs1 ^ rs2) << 12) | ((rs1 ^ rs2) >> (32-12))
```

### chacha20qr3

```
rd = ((rs1 ^ rs2) << 8) | ((rs1 ^ rs2) >> (32-8))
```

### chacha20qr4

```
rd = ((rs1 ^ rs2) << 7) | ((rs1 ^ rs2) >> (32-7))
```

All behavior of this extension object is implemented in file `addInstructionsExtensions.c`. Sections will be discussed in turn below.

### 6.1 Intercept Attributes

The behavior of the extension is defined using the standard `vmiosAttr` structure:

```
vmiosAttr modelAttrs = {
    ///////////////////////////////////////////////////////////////////
    // VERSION
    ///////////////////////////////////////////////////////////////////
    .versionString = VMI_VERSION,           // version string
    .modelType     = VMI_INTERCEPT_LIBRARY, // type
    .packageName  = "addCSRs",             // description
    .objectSize   = sizeof(vmiosObject),   // size in bytes of OSS object

    ///////////////////////////////////////////////////////////////////
    // CONSTRUCTOR/DESTRUCTOR ROUTINES
    ///////////////////////////////////////////////////////////////////
    .constructorCB = addInstructionsConstructor, // object constructor

    ///////////////////////////////////////////////////////////////////
    // INSTRUCTION INTERCEPT ROUTINES
    ///////////////////////////////////////////////////////////////////
    .morphCB       = addInstructionsMorph,     // instruction morph callback
    .disCB         = addInstructionsDisassemble, // disassemble instruction

    ///////////////////////////////////////////////////////////////////
    // ADDRESS INTERCEPT DEFINITIONS
    ///////////////////////////////////////////////////////////////////
    .intercepts    = {{0}}
}
```

In this extension, there is a constructor, a JIT translation (morpher) function and a disassembly function.

## 6.2 Object Type and Constructor

The object type is defined as follows:

```
typedef struct vmiosObjectS {  
    // Info for associated processor  
    riscvP      riscv;  
  
    // extended instruction decode table  
    vmidDecodeTableP decode32;  
  
    // extension callbacks  
    riscvExtCB   extCB;  
  
} vmiosObject;
```

To be compatible with the integration facilities described in this document, an extension object must contain the `riscv`, `decode32` and `extCB` fields shown here. It may also contain other instance-specific fields. The constructor initializes the fields as follows:

```
static VMIOS_CONSTRUCTOR_FN(addInstructionsConstructor) {  
    riscvP riscv = (riscvP)processor;  
  
    object->riscv = riscv;  
  
    // prepare client data  
    object->extCB.clientData = object;  
  
    // register extension with base model using unique ID  
    riscv->cb.registerExtCB(riscv, &object->extCB, EXTID_ADDINST);  
}
```

The `extCB` field defines one side of the interface between the extension object and the base RISC-V model. It is filled by the extension object constructor with callback function pointers and other data that are used by the base model to communicate with the extension object. To complement this, the base model itself implements a set of *interface functions* that can be called from the extension object to query and modify the base model state. These are discussed in following sections.

The constructor must initialize the `clientData` field within the `extCB` structure with the extension object and then call the `registerExtCB` interface function to register this extension object with the model. The last argument to this function is an identification number that should uniquely identify this extension object in the case that multiple libraries are installed on one RISC-V processor.

## 6.3 Instruction Decode

Because this extension object is implementing new instructions, it needs to define a utility function to decode those new instructions. Provided that the instructions follow the standard RISC-V pattern in terms of operand locations and types, an interface function is available to simplify this process.

The first step is to define an enumeration with an entry for each new instruction:

```
typedef enum riscvExtITypeE {
    // extension instructions
    EXT_IT_CHACHA20QR1,
    EXT_IT_CHACHA20QR2,
    EXT_IT_CHACHA20QR3,
    EXT_IT_CHACHA20QR4,

    // KEEP LAST
    EXT_IT_LAST
} riscvExtIType;
```

Then, information about each instruction is given in a table of `riscvExtInstrAttrs` entries, with one entry for each instruction. Members of the table should be initialized using the `EXT_INSTRUCTION` macro, defined (like all interface function types and macros) in file `riscvModelCallbackTypes.h` in the RISC-V model source:

```
const static riscvExtInstrAttrs attrsArray32[] = {
    EXT_INSTRUCTION(
        EXT_IT_CHACHA20QR1, "chacha20qr1", RVANY, RVIP_RD_RS1_RS2, FMT_R1_R2_R3,
        "|0000000|.....|.....|000|.....|0001011|"
    ),
    EXT_INSTRUCTION(
        EXT_IT_CHACHA20QR2, "chacha20qr2", RVANY, RVIP_RD_RS1_RS2, FMT_R1_R2_R3,
        "|0000000|.....|.....|001|.....|0001011|"
    ),
    EXT_INSTRUCTION(
        EXT_IT_CHACHA20QR3, "chacha20qr3", RVANY, RVIP_RD_RS1_RS2, FMT_R1_R2_R3,
        "|0000000|.....|.....|010|.....|0001011|"
    ),
    EXT_INSTRUCTION(
        EXT_IT_CHACHA20QR4, "chacha20qr4", RVANY, RVIP_RD_RS1_RS2, FMT_R1_R2_R3,
        "|0000000|.....|.....|011|.....|0001011|"
    )
};
```

Each instruction is described by 6 arguments to the `EXT_INSTRUCTION` macro:

1. The instruction enumeration member name (e.g. `EXT_IT_CHACHA20QR1`);
2. The instruction opcode, as a string (e.g. "chacha20qr1");
3. The applicable architecture (using enumeration values defined in file `riscvVariant.h` in the base model). Value `RVANY` means no constraint; other values can constrain an instruction to a particular `XLEN` (e.g. `RV32` or `RV64`) or to a particular extension code (e.g. `RVANYB`) or a combination of the two (e.g. `RV32B`). Given this constraint, the base model will automatically check that the constraint is valid and generate an `Illegal Instruction` exception if it is not.
4. The instruction operands. In this case, the value `RVIP_RD_RS1_RS2` means an instruction targeting GPR `Rd` and taking GPRs `Rs1` and `Rs2` as arguments, with all registers in the standard positions in a RISC-V instruction. Other operand layouts can be specified instead, as defined by the following enumeration in `riscvModelCallbackTypes.h`:

```
typedef enum riscvExtInstrPatternE {
```

```

// GPR INSTRUCTIONS
RVIP_RD_RS1_RS2, // op xd, xs1, xs2
RVIP_RD_RS1_SI, // op xd, xs1, imm
RVIP_RD_RS1_SHIFT, // op xd, xs1, shift
RVIP_RD_RS1_RS2_RS3, // op xd, xs1, xs2, xs3
RVIP_RD_RS1_RS3_SHIFT, // op xd, xs1, xs2, shift

// FPR INSTRUCTIONS
RVIP_FD_FS1_FS2, // op fd, fs1, fs2
RVIP_FD_FS1_FS2_RM, // op fd, fs1, fs2, rm
RVIP_FD_FS1_FS2_FS3_RM, // op fd, fs1, fs2, fs3, rm
RVIP_RD_FS1_FS2, // op xd, fs1, fs2

// VECTOR INSTRUCTIONS
RVIP_VD_VS1_VS2_M, // op vd, vs1, vs2, vm
RVIP_VD_VS1_SI_M, // op vd, vs1, simm, vm
RVIP_VD_VS1_UI_M, // op vd, vs1, uimm, vm
RVIP_VD_VS1_RS2_M, // op vd, vs1, rs2, vm
RVIP_VD_VS1_FS2_M, // op vd, vs1, fs2, vm
RVIP_RD_VS1_RS2, // op xd, vs1, vs2
RVIP_RD_VS1_M, // op xd, vs1, vm
RVIP_VD_RS2, // op vd, xs2
RVIP_FD_VS1, // op fd, vs1
RVIP_VD_FS2, // op vd, fs2

RVIP_LAST // KEEP LAST: for sizing
} riscvExtInstrPattern;

```

See section 12 for a detailed description of each of these.

- How arguments should be shown when the instruction is disassembled, as described by a format string defined in `riscvDisassembleFormats.h` in the base model. In this case, the value `FMT_R1_R2_R3` specifies that the three GPR arguments should be shown as a comma-separated list.
- The instruction pattern, in terms of ones, zeros and dot (don't care) bits, separated by vertical bar characters which are ignored. For example, the `chacha20qr1` instruction is defined using this pattern string (the comment identifies standard RISC-V instruction fields):

```

| dec | rs2 | rs1 | fn3 | rd | dec |
" |0000000|.....|.....|000|.....|0001011|"

```

The table of instructions is used by the `fetchInstruction` interface function, as follows:

```

static riscvExtIType decode(
    riscvP riscv,
    vmiosObjectP object,
    riscvAddr thisPC,
    riscvExtInstrInfoP info
) {
    return riscv->cb.fetchInstruction(
        riscv, thisPC, info, &object->decode32, attrsArray32, EXT_IT_LAST, 32
    );
}

```

Arguments to `cb.fetchInstruction` are:

- The RISC-V processor object;
- The instruction address;



3. A pointer to an object of type `riscvExtInstrInfo` which is filled by `cb.fetchInstruction` with details of the decoded instruction;
4. A pointer to a decode table structure used by `cb.fetchInstruction`;
5. A pointer to the instruction table, defined above;
6. An indication of the number of entries in the instruction table;
7. The size (in bits) of instructions in the table. In this case, instructions are 32 bits.

Function `cb.fetchInstruction` returns either the index number of the decoded instruction or `EXT_IT_LAST`, if the instruction is not specified in this instruction table.

The `riscvExtInstrInfo` type which is filled by `cb.fetchInstruction` has this definition:

```
typedef struct riscvExtInstrInfoS {
    riscvAddr      thisPC;           // instruction address
    Uns32          instruction;      // instruction word
    Uns8           bytes;           // instruction bytes
    const char     *opcode;         // opcode name
    const char     *format;         // disassembly format string
    riscvArchitecture arch;        // architecture requirements
    riscvRegDesc   r[4];            // argument registers
    riscvRegDesc   mask;           // mask register (vector instructions)
    riscvRMDesc    rm;             // rounding mode
    Uns64          c;              // constant value
    void           *userData;       // client-specific data
} riscvExtInstrInfo;
```

Fields `thisPC`, `instruction`, `bytes`, `arch`, `opcode` and `format` are always filled. Fields `r`, `mask`, `rm` and `c` are filled if applicable to the operand pattern, otherwise they are zeroed. Field `userData` is available for the extension object to use if it requires to pass further data to other stages (disassembly or JIT instruction translation).

All instructions added by this extension have operands specified by `RVIP_RD_RS1_RS2`, so in this case, `r[0]`, `r[1]` and `r[2]` are filled with register descriptions extracted from the instruction, and other fields are zero.

Typically, the decode function will be used unmodified in a new extension object, except for changing its name: only the instruction type enumeration and `attrsArray32` entries should change.

## 6.4 Instruction Disassembly

The decode routine described in the previous section can be used in combination with the `disassInstruction` interface function to implement instruction disassembly in standard form:

```
static VMIOS_DISASSEMBLE_FN(addInstructionsDisassemble) {
    riscvP      riscv = (riscvP)processor;
    const char  *result = 0;
    riscvExtInstrInfo info;

    // action is only required if the instruction is implemented by this
```

```

// extension
if(decode(riscv, object, thisPC, &info) != EXT_IT_LAST) {
    result = riscv->cb.disassInstruction(riscv, &info, attrs);
}

return result;
}

```

This function calls `decode`. If the result is not `EXT_IT_LAST`, then the instruction was successfully decoded by this extension object, and interface function `disassInstruction` is called to produce the disassembly string. This takes three arguments:

1. The RISC-V processor instance;
2. The `riscvExtInstrInfo` argument block (filled by function `decode`);
3. The disassembly attributes passed to `chachaDisassemble` (whether normal or uncooked disassembly is required).

This will produce disassembly for instructions in this extension that conforms exactly to the disassembly generated by base model instructions. For example:

```

Info `iss/cpu0`, 0x00000000000102b4(processWord+8): 01010413 addi    s0,sp,16
Info `iss/cpu0`, 0x00000000000102b8(processWord+c): 00050513 mv      a0,a0
Info `iss/cpu0`, 0x00000000000102bc(processWord+10): 00058593 mv      a1,a1
Info `iss/cpu0`, 0x00000000000102c0(processWord+14): 00b5050b chacha20qr1 a0,a0,a1
Info `iss/cpu0`, 0x00000000000102c4(processWord+18): 00b5150b chacha20qr2 a0,a0,a1
Info `iss/cpu0`, 0x00000000000102c8(processWord+1c): 00b5250b chacha20qr3 a0,a0,a1
Info `iss/cpu0`, 0x00000000000102cc(processWord+20): 00b5350b chacha20qr4 a0,a0,a1
Info `iss/cpu0`, 0x00000000000102d0(processWord+24): 00b5050b chacha20qr1 a0,a0,a1
Info `iss/cpu0`, 0x00000000000102d4(processWord+28): 00b5150b chacha20qr2 a0,a0,a1
Info `iss/cpu0`, 0x00000000000102d8(processWord+2c): 00b5250b chacha20qr3 a0,a0,a1
Info `iss/cpu0`, 0x00000000000102dc(processWord+30): 00b5350b chacha20qr4 a0,a0,a1
Info `iss/cpu0`, 0x00000000000102e0(processWord+34): 00050513 mv      a0,a0
Info `iss/cpu0`, 0x00000000000102e4(processWord+38): 00c12403 lw      s0,12(sp)
Info `iss/cpu0`, 0x00000000000102e8(processWord+3c): 01010113 addi    sp,sp,16

```

Typically, the disassembly function will be used unmodified in a new extension object, except for changing its name.

## 6.5 Instruction Translation

Information about each instruction is given in a table of `riscvExtMorphAttr` entries, with one entry for each instruction. This type is defined in `riscvModelCallbackTypes.h` as follows:

```

typedef struct riscvExtMorphAttrS {
    extMorphFn      morph; // function to translate one instruction
    octiaInstructionClass iClass; // supplemental instruction class
    Uns32          variant; // required variant
    void           *userData; // client-specific data
} riscvExtMorphAttr;

```

Field `morph` specifies a function to be called to translate the instruction. Field `iClass` is used to specify an instruction *class*; this information is used by some supplemental tools (such as timing estimators) but is not required if those tools are not used. Field `variant` is a model-specific variant code: this can be used to specify whether an instruction is implemented in a particular instantiation of this extension object (useful if the library can

be configured to support multiple supplementary instructions whose presence is determined by a configuration register, for example). Field `userData` can hold any extension-specific data.

In this example, the table is specified like this:

```
const static riscvExtMorphAttr dispatchTable[] = {
    [EXT_IT_CHACHA20QR1] = {morph:emitCHACHA20QR, userData:(void *)16},
    [EXT_IT_CHACHA20QR2] = {morph:emitCHACHA20QR, userData:(void *)12},
    [EXT_IT_CHACHA20QR3] = {morph:emitCHACHA20QR, userData:(void *) 8},
    [EXT_IT_CHACHA20QR4] = {morph:emitCHACHA20QR, userData:(void *) 7},
};
```

Here, the same callback function (`emitCHACHA20QR`) is used for all four extension instructions, with behavior controlled using an integer rotate passed using the `userData` field.

The JIT translation function is specified like this:

```
static VMIOS_MORPH_FN(addInstructionsMorph) {
    riscvP          riscv = (riscvP)processor;
    riscvExtMorphState state = {riscv:riscv, object:object};

    // decode instruction
    riscvExtIType type = decode(riscv, object, thisPC, &state.info);

    // action is only required if the instruction is implemented by this
    // extension
    if(type != EXT_IT_LAST) {
        // fill translation attributes
        state.attrs = &dispatchTable[type];

        // translate instruction
        riscv->cb.morphExternal(&state, 0, opaque);
    }

    // no callback function is required
    return 0;
}
```

This function defines a structure of type `riscvExtMorphState`, which is used to encapsulate all information required to translate a single instruction:

```
typedef struct riscvExtMorphStateS {
    riscvExtInstrInfo  info;          // decoded instruction information
    riscvExtMorphAttrCP attrs;       // instruction attributes
    riscvP             riscv;        // current processor
    vmiosObjectP      object;        // current extension object
} riscvExtMorphState;
```

The structure is declared with `riscv` and `object` fields initialized:

```
riscvExtMorphState state = {riscv:riscv, object:object};
```

Then, the `info` field is filled with information about the current instruction:

```
riscvExtIType type = decode(riscv, object, thisPC, &state.info);
```

If the instruction is implemented by this extension object, the `attrs` field is filled with the relevant entry from table `dispatchTable`, and the interface function `morphExternal` is called to perform the translation:

```
// fill translation attributes
state.attrs = &dispatchTable[type];

// translate instruction
riscv->cb.morphExternal(&state, 0, opaque);
```

Function `morphExternal` takes three arguments:

1. The `riscvExtMorphState` structure;
2. A *disable reason string*. If this string is non-NULL, then the instruction is not translated, but instead an Illegal Instruction exception is triggered, with the given string reported as a reason in verbose mode. In this example, the string is NULL, so the instruction is always translated, but typically code at this point would validate the `variant` in the `riscvExtMorphAttr` entry against current configuration in the extension object, and return a non-NULL disable reason string if required.
3. The `opaque` function argument passed to `chachaMorph`.

Interface function `morphExternal` performs standard checks for instruction validity (for example, if the instruction architecture is defined as RV64 then an Illegal Instruction exception is raised if the current XLEN is 32). If the instruction is valid, the appropriate callback function from the `dispatchTable` is called to generate JIT code to implement the instruction.

Typically, the JIT translation function will be used unmodified in a new extension object, except for changing its name.

Each translation callback function is passed a single argument of type `riscvExtMorphStateP`, which holds all information required to generate code for the current instruction. In this case, function `emitCHACHA20QR` is implemented like this:

```
static EXT_MORPH_FN(emitCHACHA20QR) {
    riscvP riscv = state->riscv;

    // get abstract register operands
    riscvRegDesc rd = getRVReg(state, 0);
    riscvRegDesc rs1 = getRVReg(state, 1);
    riscvRegDesc rs2 = getRVReg(state, 2);

    // get VMI registers for abstract operands
    vmiReg rdA = getVMIREg(riscv, rd);
    vmiReg rs1A = getVMIREg(riscv, rs1);
    vmiReg rs2A = getVMIREg(riscv, rs2);

    // emit embedded call to perform operation
    UnsPS rotl = (UnsPS)state->attrs->userData;
    Uns32 bits = 32;
    vmimtArgReg(bits, rs1A);
    vmimtArgReg(bits, rs2A);
    vmimtArgUns32(rotl);
    vmimtCallResult((vmiCallFn)qrN_c, bits, rdA);
}
```

```
// handle extension of result if 64-bit XLEN
writeRegSize(riscv, rd, bits, True);
}
```

The first part of this function extracts RISC-V *abstract register definitions* from the passed state object for registers `r[0]`, `r[1]` and `r[2]`:

```
riscvRegDesc rd = getRVReg(state, 0);
riscvRegDesc rs1 = getRVReg(state, 1);
riscvRegDesc rs2 = getRVReg(state, 2);
```

The abstract register description combines register index (0-31), type information (X, F or V register) with size (8, 16, 32 or 64 bits). Function `getRVReg` is a simple utility function:

```
inline static riscvRegDesc getRVReg(riscvExtMorphStateP state, Uns32 argNum) {
    return state->info.r[argNum];
}
```

The next part of the function translates abstract register definitions into VMI register definitions:

```
vmiReg rdA = getVMIReg(riscv, rd);
vmiReg rs1A = getVMIReg(riscv, rs1);
vmiReg rs2A = getVMIReg(riscv, rs2);
```

Function `getVMIReg` is a simple wrapper function around an interface function of the same name:

```
inline static vmiReg getVMIReg(riscvP riscv, riscvRegDesc r) {
    return riscv->cb.getVMIReg(riscv, r);
}
```

Once the registers have been converted to VMI register descriptions, all available VMI morph-time operations can be used (see the *VMI Morph Time Function Reference Manual* and *OVP Processor Modeling Guide*). For simplicity, it is usually easiest to implement extension instructions as *embedded calls*, in which each extension instruction is implemented by a simple function. To do this, first define a utility function to implement the instruction operation. In this case, the utility function is this:

```
static Uns32 qrN_c(Uns32 rs1, Uns32 rs2, Uns32 rotl) {
    return ((rs1 ^ rs2) << rotl) | ((rs1 ^ rs2) >> (32-rotl));
}
```

This function takes two 32-bit register inputs (`rs1` and `rs2`) and constant rotation. It returns operation results as described at the start of this chapter. Within the translation callback function, a call to this utility function is emitted. First, the constant rotation amount is extracted from the `userData` field in the `riscvExtMorphAttr` entry:

```
UnsPS rotl = (UnsPS)state->attrs->userData;
```

The two GPR sources and the constant amount are passed as arguments to the utility function, and a call to that emitted, and the result is assigned to register `rd`:

```
Uns32 bits = 32;
vmimtArgReg(bits, rs1A);
vmimtArgReg(bits, rs2A);
vmimtArgUns32(rotl);
vmimtCallResult((vmiCallFn)qrN_c, bits, rdA);
```

The instructions operate on data of fixed width (32 bits). If run on a RISC-V with XLEN of 64, the 32-bit result must be extended to 64 bits. This extension is specified as follows:

```
writeRegSize(riscv, rd, bits, True);
```

Function `writeRegSize` is again a simple wrapper round an interface function of the same name:

```
inline static void writeRegSize(
    riscvP      riscv,
    riscvRegDesc r,
    Uns32       srcBits,
    Bool        signExtend
) {
    riscv->cb.writeRegSize(riscv, r, srcBits, signExtend);
}
```

The `writeRegSize` interface function will extend the value in abstract register `r` from the `srcBits` to the full register size encoded in the abstract register. The extension can either be zero-extension or sign-extension (in this case, sign-extension).

### 6.5.1 Required VMI Morph-Time Function Knowledge

When using embedded functions to implement extension functions, knowledge of only a small subset of the VMI Morph-Time Function function API is required. The necessary functions are those that specify function arguments, together with `vmimtCallResultAttrs` and its aliases:

```
//
// Add various argument types to the stack frame
//
void vmimtArgProcessor(void);
void vmimtArgUns32(Uns32 arg);
void vmimtArgUns64(Uns64 arg);
void vmimtArgFlt64(Flt64 arg);
void vmimtArgReg(vmiRegArgType argType, vmiReg r);
void vmimtArgRegSimAddress(Uns32 bits, vmiReg r);
void vmimtArgSimAddress(Addr arg);
void vmimtArgSimPC(Uns32 bits);
void vmimtArgNatAddress(const void *arg);

//
// Deprecated name for argument type function
//
#define vmimtArgDouble vmimtArgFlt64

//
// Make a call with all current stack frame arguments. If 'rd' is not VMI_NOREG,
// the function result (of size bits) is assigned to this register. Argument
// 'attrs' is used to define optimization attributes of a called function
// (see the comment preceding the definition of that type).
```

```

//
void vmimtCallResultAttrs(
    vmiCallFn    arg,
    Uns32        bits,
    vmiReg       rd,
    vmiCallAttrs attrs
);

//
// Backwards-compatible vmimtCall
//
#define vmimtCall(_ARG) \
    vmimtCallResultAttrs(_ARG, 0, VMI_NOREG, VMCA_NA)

//
// Backwards-compatible vmimtCallResult
//
#define vmimtCallResult(_ARG, _BITS, _RD) \
    vmimtCallResultAttrs(_ARG, _BITS, _RD, VMCA_NA)

//
// Backwards-compatible vmimtCallAttrs
//
#define vmimtCallAttrs(_ARG, _ATTRS) \
    vmimtCallResultAttrs(_ARG, 0, VMI_NOREG, _ATTRS)

```

Refer to the *VMI MorphTime Function Reference* manual for detailed information about these.

## 6.6 Example Execution

The extension can be exercised using a simple assembler program:

```

//
// GPR aliases
//
#define r_zero    0
#define r_ra     1
#define r_sp     2
#define r_gp     3
#define r_tp     4
#define r_t0     5
#define r_t1     6
#define r_t2     7
#define r_s0     8
#define r_s1     9
#define r_a0    10
#define r_a1    11
#define r_a2    12
#define r_a3    13
#define r_a4    14
#define r_a5    15
#define r_a6    16
#define r_a7    17
#define r_s2    18
#define r_s3    19
#define r_s4    20
#define r_s5    21
#define r_s6    22
#define r_s7    23
#define r_s8    24
#define r_s9    25
#define r_s10   26
#define r_s11   27
#define r_t3    28
#define r_t4    29
#define r_t5    30
#define r_t6    31

```

```

//
// CHACHA20QRN <N>, <rd>, <rs1>, <rs2>
//
#define CHACHA20QRN(_N, _RD, _RS1, _RS2) .word ( \
    (0x0b << 0) | \
    (_RD << 7) | \
    ((_N-1) << 12) | \
    (_RS1 << 15) | \
    (_RS2 << 20) | \
    (0x01 << 25) \
)

//
// CHACHA20QR1-4 <rd> <rs1>, <rs2>
//
#define CHACHA20QR1(_RD, _RS1, _RS2) CHACHA20QRN(1, _RD, _RS1, _RS2)
#define CHACHA20QR2(_RD, _RS1, _RS2) CHACHA20QRN(2, _RD, _RS1, _RS2)
#define CHACHA20QR3(_RD, _RS1, _RS2) CHACHA20QRN(3, _RD, _RS1, _RS2)
#define CHACHA20QR4(_RD, _RS1, _RS2) CHACHA20QRN(4, _RD, _RS1, _RS2)

START_TEST:

    li        s0, 0x12345678
    li        s1, 0xaabbccdd

    // validate CHACHA instructions
    CHACHA20QR1(r_s2, r_s0, r_s1)
    CHACHA20QR2(r_s2, r_s0, r_s1)
    CHACHA20QR3(r_s2, r_s0, r_s1)
    CHACHA20QR4(r_s2, r_s0, r_s1)

EXIT_TEST

```

This can be run using the `iss.exe` simulator like this:

```

iss.exe \
--trace                               \|
--tracechange                          \|
--tracemode                             \|
--traceshowicount                       \|
--addressbits      32                  \|
--processorvendor  vendor.com          \|
--processorname    riscv                \|
--variant          RV64X                \|
--program          test.elf             \|
--extlib           iss/cpu0=riscv.ovpworld.org/intercept/customControl/1.0 \|

```

Which produces this output:

```

Info 1: 'iss/cpu0', 0x0000000080000000(_start): Machine 4000206f j      80002400
Info 2: 'iss/cpu0', 0x0000000080002400(START_TEST): Machine 12345437 lui    s0,0x12345
Info  s0 0000000000000000 -> 0000000012345000
Info 3: 'iss/cpu0', 0x0000000080002404(START_TEST+4): Machine 6784041b addiw  s0,s0,1656
Info  s0 0000000012345000 -> 0000000012345678
Info 4: 'iss/cpu0', 0x0000000080002408(START_TEST+8): Machine 000ab4b7 lui    s1,0xab
Info  s1 0000000000000000 -> 0000000000ab000
Info 5: 'iss/cpu0', 0x000000008000240c(START_TEST+c): Machine bbd4849b addiw  s1,s1,-1091
Info  s1 000000000000ab000 -> 0000000000aabbd
Info 6: 'iss/cpu0', 0x0000000080002410(START_TEST+10): Machine 00c49493 sllli  s1,s1,0xc
Info  s1 000000000000aabbd -> 00000000aabbd000
Info 7: 'iss/cpu0', 0x0000000080002414(START_TEST+14): Machine cdd48493 addi   s1,s1,-803
Info  s1 00000000aabbd000 -> 00000000aabbccdd
Info 8: 'iss/cpu0', 0x0000000080002418(START_TEST+18): Machine 0294090b chacha20qr1 s2,s0,s1
Info  s2 0000000000000000 -> ffffffff9aa5b88f
Info 9: 'iss/cpu0', 0x000000008000241c(START_TEST+1c): Machine 0294190b chacha20qr2 s2,s0,s1

```



```
Info s2 ffffffff9aa5b88f -> ffffffff9aa5b88
Info 10: 'iss/cpu0', 0x0000000080002420(START_TEST+20): Machine 0294290b chacha20qr3 s2,s0,s1
Info s2 ffffffff9aa5b88 -> ffffffff8f9aa5b8
Info 11: 'iss/cpu0', 0x0000000080002424(START_TEST+24): Machine 0294390b chacha20qr4 s2,s0,s1
Info s2 ffffffff8f9aa5b8 -> 0000000047cd52dc
Info 12: 'iss/cpu0', 0x0000000080002428(START_TEST+28): Machine 4501 li a0,0
Info 13: 'iss/cpu0', 0x000000008000242a(START_TEST+2a): Machine custom0
```

Note the four custom instructions being executed and that result values are sign-extended from bit 31.

## 7 Adding Custom CSRs (addCSRs)

The `addCSRs` extension demonstrates how to add custom CSRs to a RISC-V model. The extension adds five CSRs in the custom space of a RISC-V processor. Three of the CSRs are implemented using plain registers, and one of these is read-only. The other two CSRs are implemented using callback functions. In detail, the CSRs are:

### `custom_rw1_32`

This is a 32-bit M-mode CSR implemented as a plain register with a write mask (some bits are not writeable). When accessed with XLEN 64, the value is zero extended from 32 to 64 bits.

### `custom_rw1_64`

This is a 64-bit M-mode CSR implemented as a plain register with a write mask (some bits are not writeable). When accessed with XLEN 32, the most-significant 32 bits are zero.

### `custom_ro1`

This is a 64-bit M-mode read-only CSR implemented as a plain register.

### `custom_rw3_32`

This is a 32-bit M-mode CSR implemented using callback functions.

### `custom_rw4_64`

This is a 64-bit M-mode CSR implemented using callback functions.

In addition, the example shows how to modify the behavior of the existing `mstatus` CSR to add an extra field to it.

All behavior of this extension object is implemented in file `addCSRsExtensions.c` and header files `addCSRsCSR.h` and `addCSRsConfig.h`. Sections will be discussed in turn below.

### 7.1 CSR Type Definitions

Utility structures are defined for each of the CSRs implemented by this extension, in file `addCSRsCSR.h`. The structures use macros from file `riscvCSR.h` in the base model. 32-bit CSRs are defined using macros `CSR_REG_TYPE_32` (to define the structure type name) and `CSR_REG_STRUCT_DECL_32` (to define a union allowing the CSR to be accessed either using fields or as an entire 32-bit value). For example, here is the definition of custom CSR `custom_rw1_32` from file `addCSRsCSR.h`:

```
// 32-bit view
typedef struct {
    Uns32 F1 : 8;
    Uns32 _u1 : 8;
    Uns32 F3 : 8;
    Uns32 _u2 : 8;
} CSR_REG_TYPE_32(custom_rw1_32);

// define 32 bit type
```

```
CSR_REG_STRUCT_DECL_32(custom_rw1_32);

// define write masks
#define WM32_custom_rw1_32 0x00ff00ff
#define WM64_custom_rw1_32 0x00ff00ff
```

This CSR has two true fields (F1 and F3) and two unused field that are always zero. The last two `#define` lines specify *write masks* for the CSR, when it is written with XLEN of 32 and 64. In this case, the values are the same and allow change only to fields F1 and F3.

For 64-bit CSRs, equivalent macros `CSR_REG_TYPE_64` and `CSR_REG_STRUCT_DECL_64` are used instead. For example, here is the definition of custom CSR `custom_rw2_64` from file `addCSRsCSR.h`:

```
// 64-bit view
typedef struct {
    Uns32 F1 : 8;
    Uns32 _u1 : 8;
    Uns32 F3 : 8;
    Uns32 _u2 : 8;
    Uns32 F5 : 8;
    Uns32 _u3 : 8;
    Uns32 F7 : 8;
    Uns32 _u4 : 8;
} CSR_REG_TYPE_64(custom_rw2_64);

// define 32 bit type
CSR_REG_STRUCT_DECL_64(custom_rw2_64);

// define write masks
#define WM32_custom_rw2_64 0x00ff00ff
#define WM64_custom_rw2_64 0xff00ff0000ff00ffULL
```

In this case the write masks for XLEN of 32 and 64 differ, because only when XLEN is 64 are the most-significant bits accessible.

## 7.2 Extension-Specific Configuration

File `addCSRsConfig.h` defines a structure specifying extension-specific configuration information. In this case, the extension allows one CSR default value to be configurable:

```
typedef struct addCSRsConfigS {
    // extension CSR register values in configuration
    struct {
        CSR_REG_DECL(custom_rol);
    } csr;
} addCSRsConfig;

DEFINE_S (addCSRsConfig);
DEFINE_CS(addCSRsConfig);
```

The extension-specific configuration structure is used in the definition of variant configurations in file `riscvConfigList.h` of the linked model:

```
static riscvExtConfigCP allExtensions[] = {
    // example adding CSRs
    &(const riscvExtConfig){
```

```

        .id          = EXTID_ADDCSR,
        .userData = &(const addCSRsConfig){
            .csr = {
                .custom_ro1 = {u32 : {bits : 0x12345678}}
            }
        },
        . . . fields omitted . . .
    };

```

This specifies that the default value for custom CSR `custom_ro1` should be `0x12345678`.

### 7.3 Intercept Attributes

The behavior of the extension is defined using the standard `vmiosAttr` structure in `addCSRSExtensions.c`:

```

vmiosAttr modelAttrs = {

    ///////////////////////////////////////////////////////////////////
    // VERSION
    ///////////////////////////////////////////////////////////////////

    .versionString = VMI_VERSION,           // version string
    .modelType     = VMI_INTERCEPT_LIBRARY, // type
    .packageName  = "addCSRs",             // description
    .objectSize   = sizeof(vmiosObject),   // size in bytes of OSS object

    ///////////////////////////////////////////////////////////////////
    // CONSTRUCTOR/DESTRUCTOR ROUTINES
    ///////////////////////////////////////////////////////////////////

    .constructorCB = addCSRsConstructor,    // object constructor

    ///////////////////////////////////////////////////////////////////
    // ADDRESS INTERCEPT DEFINITIONS
    ///////////////////////////////////////////////////////////////////

    .intercepts    = {{0}}
}

```

In this extension, there is a constructor to register the custom CSRs with the base model.

### 7.4 Object Type and Constructor

The object type is defined as follows:

```

typedef struct vmiosObjectS {

    // Info for associated processor
    riscvP      riscv;

    // configuration (including CSR reset values)
    addCSRsConfig  config;

    // extension CSR info
    addCSRsCSRs    csr;           // extension CSR values
    riscvCSRAttrs  csrs[XCSR_ID(LAST)]; // extension CSR definitions
    Bool           mstatus30;     // modified mstatus bit 30

    // extension callbacks
    riscvExtCB     extCB;

} vmiosObject;

```

To be compatible with the integration facilities described in this document, an extension object must contain the `riscv`, `config`, `csr`, `csrs` and `extCB` fields shown here. It may also contain other instance-specific fields: in this case, a field `mstatus30` is added, to hold the value of bit 30 of the `mstatus` register, which is a custom field added by this extension.

The `config` field holds configuration-specific information (used, for example, to hold initial values for read-only CSR fields when these are configuration dependent (see section 7.2).

The `csr` field holds current values of each CSR defined by this extension. The `addCSRsCSRs` value container type is defined in file `addCSRsCSRs.h` like this:

```
typedef struct addCSRsCSRs {
    CSR_REG_DECL (custom_rw1_32);    // 0xBC0
    CSR_REG_DECL (custom_rw2_64);    // 0xBC1
    CSR_REG_DECL (custom_rw3_32);    // 0xBC2
    CSR_REG_DECL (custom_rw4_64);    // 0xBC3
    CSR_REG_DECL (custom_rol);       // 0xFC0
} addCSRsCSRs;
```

The `csrs` field holds description information for each CSR defined by this extension (required by the debugger and when tracing register value changes, for example). The `riscvCSRAttrs` type is defined in the base model:

```
typedef struct riscvCSRAttrs {
    // COMMON FIELDS
    const char    *name;              // register name
    const char    *desc;              // register description
    void          *object;            // client-specific object
    Uns32         csrNum;              // CSR number (includes privilege and r/w
access)
    riscvArchitecture arch;           // required architecture (presence)
    riscvArchitecture access;         // required architecture (access)
    riscvPrivVer   version;           // minimum specification version
    Bool          wEndBlock;          // whether write terminates this block
    Bool          wEndRM;              // whether write invalidates RM assumption
    Bool          noSaveRestore;      // whether to exclude from save/restore
    Bool          noTraceChange;      // whether to exclude from trace change
    Bool          TVMT;               // whether trapped by mstatus.TVM
    Bool          writeRd;             // whether write updates Rd
    riscvCSRPresentFn presentCB;      // CSR present callback
    riscvCSRReadFn  readCB;           // read callback
    riscvCSRReadFn  readWriteCB;      // read callback (in r/w context)
    riscvCSRWriteFn writeCB;          // write callback
    riscvCSRWStateFn wstateCB;        // adjust JIT code generator state

    // 32-BIT FIELDS
    vmiReg        reg32;              // register
    vmiReg        writeMaskV32;       // variable write mask
    Uns32         writeMaskC32;       // constant write mask

    // 64-BIT FIELDS
    vmiReg        reg64;              // register
    vmiReg        writeMaskV64;       // variable writable bit mask
    Uns64         writeMaskC64;       // constant writable bit mask
} riscvCSRAttrs;
```

Field `csrs` is an array of these structures, of size `XCSR_ID(LAST)`, which is a member of the `extCSRId` enumeration in `addCSRExtensions.c`:

```
typedef enum extCSRIdE {
    // custom CSRs, plain registers
    XCSR_ID (custom_rw1_32),    // 0xBC0
    XCSR_ID (custom_rw2_64),    // 0xBC1
    XCSR_ID (custom_ro1),      // 0xFC0

    // custom CSRs, implemented by callbacks
    XCSR_ID (custom_rw3_32),    // 0xBC3
    XCSR_ID (custom_rw4_64),    // 0xBC4

    // base CSR with additional fields
    XCSR_ID (mstatus),          // 0x300

    // keep last (used to define size of the enumeration)
    XCSR_ID (LAST)
} extCSRId;
```

This enumeration contains one member for each custom CSR added by this extension, an entry for the standard `mstatus` CSR (whose behavior is being modified) and a final `LAST` member for sizing.

The constructor initializes the fields in the `vmiosObject` structure as follows:

```
static VMIOS_CONSTRUCTOR_FN(addCSRConstructor) {
    riscvP riscv = (riscvP)processor;

    object->riscv = riscv;

    // prepare client data
    object->extCB.clientData = object;
    object->extCB.resetNotifier = CSRReset;

    // register extension with base model using unique ID
    riscv->cb.registerExtCB(riscv, &object->extCB, EXTID_ADDCSR);

    // copy configuration from template
    object->config = *getExtConfig(riscv);

    // initialize CSRs
    addCSRCSRInit(object);
}
```

The `extCB` field defines the interface between the extension object and the base RISC-V model. The constructor must initialize the `clientData` field within that structure with the extension object and then call the `registerExtCB` interface function to register this extension object with the model. The last argument to this function is an identification number that should uniquely identify this extension object in the case that multiple libraries are installed on one RISC-V processor.

The `resetNotifier` field is initialized with a notifier function that is called whenever a processor reset occurs. In this case, the notifier writes reset values to various CSRs:

```
static RISC_V_RESET_NOTIFIER_FN(CSRReset) {
```

```

vmiosObjectP object = clientData;

// reset custom mstatus field
object->mstatus30 = 0;

// reset custom CSRs by index
riscv->cb.writeCSR(riscv, 0xBC0, 0);
riscv->cb.writeCSR(riscv, 0xBC1, 0x1234);
riscv->cb.writeCSR(riscv, 0xBC2, 0);
riscv->cb.writeCSR(riscv, 0xBC3, 0);
}

```

The reset notifier sets the `mstatus30` field to 0. It then calls the `writeCSR` interface function to reset all custom CSRs added by this extension to initial values. This interface function will write a CSR value given the index number of the CSR:

```

#define RISCV_WRITE_CSR_NUM_FN(_NAME) Uns64 _NAME( \
    riscvP riscv, \
    Uns32 csrNum, \
    Uns64 newValue \
)
typedef RISCV_WRITE_CSR_NUM_FN((*riscvWriteCSRNumFn));

typedef struct riscvModelCBS {
    . . . fields omitted . . .
    riscvWriteCSRNumFn writeCSR;
    . . . fields omitted . . .
} riscvModelCB;

```

The `config` field is initialized by copying default values from the *extension configuration* for this variant. The purpose of this is to allow different variants to be defined which have different default values for the extension registers. Function `getExtConfig` retrieves configuration information for the current processor:

```

static addCSRsConfigCP getExtConfig(riscvP riscv) {

    riscvExtConfigCP cfg = riscv->cb.getExtConfig(riscv, EXTID_ADDCSR);

    VMI_ASSERT(cfg, "ADDCSR config not found");

    return cfg->userData;
}

```

Function `addCSRsCSRInit` initializes the custom CSRs:

```

static void addCSRsCSRInit(vmiosObjectP object) {

    riscvP riscv = object->riscv;
    extCSRId id;

    // initialize CSR values that have configuration values defined
    WR_XCSR(object, custom_rol, object->config.csr.custom_rol.u64.bits);

    // register each CSR with the base model using the newCSR interface
    // function
    for(id=0; id<XCSR_ID(LAST); id++) {

        extCSRAttrsCP src = &csrs[id];
        riscvCSRAttrs *dst = &object->csrs[id];

        riscv->cb.newCSR(dst, &src->baseAttrs, riscv, object);

    }

    // perform initial CSR reset
}

```

```
CSRReset(riscv, object);
}
```

This function first initializes the *current* value of read-only CSR `custom_rol` using the *default* value from the configuration:

```
WR_XCSR(object, custom_rol, object->config.csr.custom_rol.u64.bits);
```

The macro `WR_XCSR` is defined in file `riscvModelCallbackTypes.h` in the base model and is used to write the value of an entire extension CSR. The `for` loop iterates over all members of the `extCSRId` enumeration, filling one entry in the `csrs` array in the extension object from a matching entry in a static configuration template structure, `csrs`, using the `newCSR` interface function:

```
for(id=0; id<XCSR_ID(LAST); id++) {
    extCSRAttrsCP src = &csrs[id];
    riscvCSRAttrs *dst = &object->csrs[id];
    riscv->cb.newCSR(dst, &src->baseAttrs, riscv, object);
}
```

Copying the CSR definitions from a template into the `vmiosObject` structure in this way allows the definitions to be modified on an instance-specific basis if required (for example, using model parameters). The `csrs` template structure is defined like this:

```
static const extCSRAttrs csrs[XCSR_ID(LAST)] = {
    //
    // -----
    // CSRs IMPLEMENTED AS PLAIN REGISTERS
    // -----
    //
    //          name          num    arch extension attrs    description
rCB rwCB wCB
    XCSR_ATTR_TC(custom_rw1_32, 0xBC0, 0, 0, 0,0,0,0, "32-bit R/W CSR
(plain)", 0, 0, 0),
    XCSR_ATTR_TC(custom_rw2_64, 0xBC1, 0, 0, 0,0,0,0, "XLEN R/W CSR
(plain)", 0, 0, 0),
    XCSR_ATTR_TC(custom_rol, 0xFC0, 0, 0, 0,0,0,0, "R/O CSR (plain)",
0, 0, 0),
    //
    // -----
    // CSRs IMPLEMENTED WITH CALLBACKS
    // -----
    //
    //          name          num    arch extension attrs    description
rCB          rwCB wCB
    XCSR_ATTR_TC(custom_rw3_32, 0xBC2, 0, 0, 0,0,0,0, "32-bit R/W CSR
(cb)", custom_rw3_32R, 0, custom_rw3_32W),
    XCSR_ATTR_TC(custom_rw4_64, 0xBC3, 0, 0, 0,0,0,0, "XLEN R/W CSR (cb)",
custom_rw4_64R, 0, custom_rw4_64W),
    //
    // -----
    // MODIFIED BEHAVIOR OF BASE CSR
    // -----
    //
    //          name          num    arch extension attrs    description
rCB          rwCB wCB
```



```
XCSR_ATTR_P__(mstatus, 0x300, 0, 0, 0,0,0,0, "Machine Status",
mstatusR, 0, mstatusW ),
};
```

Each entry for a *new* CSR in the template is filled with CSR name, number, extension requirements, attributes, description and callbacks using macro `XCSR_ATTR_TC_` defined in file `riscvModelCallbackTypes.h` in the base model:

```
#define XCSR_ATTR_TC_( \
    _ID, _NUM, _ARCH, _EXT, _ENDB, _ENDRM, _NOTR, _TVMT, _DESC, _RCB, _RWCB, _WCB \
) [XCSR_ID(_ID)] = { \
    .extension = _EXT, \
    .baseAttrs = { \
        name : #_ID, \
        desc : _DESC, \
        csrNum : _NUM, \
        arch : _ARCH, \
        wEndBlock : _ENDB, \
        wEndRM : _ENDRM, \
        noTraceChange : _NOTR, \
        TVMT : _TVMT, \
        readCB : _RCB, \
        readWriteCB : _RWCB, \
        writeCB : _WCB, \
        reg32 : XCSR_REG32_MT(_ID), \
        writeMaskC32 : WM32_##_ID, \
        reg64 : XCSR_REG64_MT(_ID), \
        writeMaskC64 : WM64_##_ID \
    } \
}
```

This macro defines a CSR with a constant write mask and optional callbacks. There are other similar macro variants for CSRs with no write mask, or with configurable write masks. The macro takes the following arguments:

1. The CSR *identifier*. This is used to construct both the CSR enumeration member name and the CSR name string (for reporting).
2. The CSR number, using standard RISC-V CSR numbering conventions.
3. Any architectural restrictions for the CSR, specified in the same way as architectural restrictions on instructions, discussed previously.
4. The extension identifier (see above).
5. *End-block* attribute: whether writes to the CSR should terminate a code block.
6. *End-rounding* attribute: whether writes to the CSR modify rounding mode.
7. *No-trace* attribute: whether changes to the CSR should not be reported when trace change is enabled.
8. *TVMT* attribute: whether accesses to the CSR are trapped by `mstatus.TVM`.
9. A CSR description string (used in documentation generation).
10. An optional read callback function.
11. An optional read-modify-write callback function (only for CSRs that have special behavior in this case).
12. An optional write callback function.

Often, CSRs can be implemented as plain registers with no other associated behavior. In such cases, the callback fields in the template structure can be null. In this example, CSRs `custom_rw1_32`, `custom_rw2_64` and `custom_rol` are all implemented as plain registers.

When a CSR must have behavior associated with it, it must be implemented using callbacks. In this example, CSRs `custom_rw3_32` and `custom_rw4_64` are implemented in this way with read and write callbacks.

A *read* callback is called whenever the CSR is read (either by a true model access or in another way, for example by the debugger or when tracing CSR values). Read callbacks are defined using the `RISCV_CSR_READFN` macro, defined in `riscvCSRTypes.h` in the base model like this:

```
#define RISCV_CSR_READFN(_NAME) Uns64 _NAME( \
    riscvCSRAttrsCP attrs, \
    riscvP riscv \
)
typedef RISCV_CSR_READFN(*riscvCSRReadFn);
```

A CSR read callback is passed a description of the CSR being read (a pointer of type `riscvCSRAttrsCP`) and the RISC-V processor doing the read. The example read function for CSR `custom_rw3_32` is:

```
static RISCV_CSR_READFN(custom_rw3_32R) {
    vmiosObjectP object = attrs->object;
    Int32 result = RD_XCSR(object, custom_rw3_32);
    return result;
}
```

This function reads the entire value of the CSR using the `RD_XCSR` macro and returns it. In a real case, the callback would do more than this, because the same effect can be achieved with a plain register read.

Within a read (or write) callback function, the current extension object of type `vmiosObjectP` can be found using the expression `attrs->object`. This means that the callback can easily refer to any custom structures in the extension object. Whether this is a true access (by a processor) or an artifact access (by a debugger or for tracing) is indicated by the `artifactAccess` flag on the base processor, allowing the callback to modify its behavior in these cases. For example, to perform an operation only when a non-artifact access, the callback could contain an `if` statement:

```
if(!riscv->artifactAccess) {
    // behavior only if a true processor access
}
```

CSR *write* callbacks are defined using the `RISCV_CSR_WRITEFN` macro, defined in `riscvCSRTypes.h` like this:

```
#define RISCV_CSR_WRITEFN(_NAME) Uns64 _NAME( \
    riscvCSRAttrsCP attrs, \
    riscvP riscv, \
    Uns64 newValue \
)
typedef RISCV_CSR_WRITEFN(*riscvCSRWriteFn);
```

A CSR write callback is passes a description of the CSR being written, the RISC-V processor doing the read, and the new CSR value. The example write function for CSR `custom_rw3_32` is:

```
static RISC_V_CSR_WRITEFN(custom_rw3_32W) {
    vmiosObjectP object = attrs->object;

    WR_XCSR(object, custom_rw3_32, newValue);

    return newValue;
}
```

This function writes the entire value of the CSR using the `WR_XCSR` macro and returns it. In a real case, the callback would do more than this, because the same effect can be achieved with a plain register write.

The CSR installation process automatically handles CSR access constraints based on address. For example, custom CSR `custom_rw3_32` is known to allow read/write access because its address (`0xBC0`) is in the custom read/write range, whereas CSR `custom_ro1` is known to be read-only because its address (`0xFC0`) is in the custom read-only range.

If a CSR is defined in an extension with the same number as a standard CSR in the base model, then the extension implementation will override that in the base. This allows extension objects to modify the behavior of standard CSRs in custom ways. In this example, CSR `mstatus` is redefined so that an extra one-bit field can be added to it (bit 30). The CSR is redefined using the `XCSR_ATTR_P__` macro, defined in file `riscvModelCallbackTypes.h` in the base model:

```
#define XCSR_ATTR_P__( \
    _ID, _NUM, _ARCH, _EXT, _ENDB, _ENDRM, _NOTR, _TVMT, _DESC, _RCB, _RWCB, _WCB \
) [XCSR_ID(_ID)] = { \
    .extension = _EXT, \
    .baseAttrs = { \
        name      : #_ID, \
        desc      : _DESC, \
        csrNum    : _NUM, \
        arch      : _ARCH, \
        wEndBlock : _ENDB, \
        wEndRM    : _ENDRM, \
        noTraceChange : _NOTR, \
        TVMT      : _TVMT, \
        readCB    : _RCB, \
        readWriteCB : _RWCB, \
        writeCB   : _WCB, \
        writeMaskC32 : -1, \
        writeMaskC64 : -1 \
    } \
}
```

This macro defines a CSR which is implemented using callbacks only (there is no field for it in the `csr` structure in the extension object). In this case, `mstatus` is reimplemented using a read callback function (`mstatusR`) and write callback function (`mstatusW`). Function `mstatusR` is defined like this:

```
static RISCV_CSR_READFN(mstatusR) {
    vmiosObjectP object = attrs->object;

    // get value from base model
    mstatusU result = {u64 : riscv->cb.readBaseCSR(riscv, CSR_ID(mstatus))};

    // fill custom field from extension object
    result.f.custom1 = object->mstatus30;

    // return composed result
    return result.u64;
}
```

This function first uses the interface function `readBaseCSR` to read the value of `mstatus` from the base model into a union of type `mstatusU`:

```
mstatusU result = {u64 : riscv->cb.readBaseCSR(riscv, CSR_ID(mstatus))};
```

The `readBaseCSR` interface function has this prototype:

```
#define RISCV_READ_BASE_CSR_FN(_NAME) Uns64 _NAME( \
    riscvP    riscv, \
    riscvCSRId id \
)
typedef RISCV_READ_BASE_CSR_FN((*riscvReadBaseCSRFn));

typedef struct riscvModelCBS {
    . . . fields omitted . . .
    riscvReadBaseCSRFn    readBaseCSR;
    . . . fields omitted . . .
} riscvModelCB;
```

The `riscvCSRId` type defines the CSRs known to the base model. Type `mstatusU` is defined in the extension object like this:

```
typedef union {
    Uns64 u64;
    struct {
        Uns64 standard1 : 30;
        Uns64 custom1   : 1;
        Uns64 standard2 : 33;
    } f;
} mstatusU;
```

This bitfield structure defines the location of the new `custom1` field within the (otherwise opaque) `mstatus` CSR. Having read the `mstatus` value from the base model, function `mstatusR` inserts the value of the `custom1` field from the master value held in the extension object, and returns the composed value:

```
// fill custom field from extension object
result.f.custom1 = object->mstatus30;

// return composed result
return result.u64;
```

Function `mstatusW` is defined like this:

```
static RISCV_CSR_WRITEFN(mstatusW) {
```

```

vmiosObjectP object = attrs->object;

// assign value to mstatusU for field extraction
mstatusU result = {u64 : newValue};

// extract custom field
object->mstatus30 = result.f.custom1;

// set value in base model
result.u64 = riscv->cb.writeBaseCSR(riscv, CSR_ID(mstatus), newValue);

// fill custom field from extension object
result.f.custom1 = object->mstatus30;

// return composed result
return result.u64;
}

```

This function first saves the value being written to a union of type `mstatusU` and extracts the `custom1` field from that into the extension object field:

```

// assign value to mstatusU for field extraction
mstatusU result = {u64 : newValue};

// extract custom field
object->mstatus30 = result.f.custom1;

```

It then calls the interface function `writeBaseCSR` to write the value of `mstatus` in the base model:

```

result.u64 = riscv->cb.writeBaseCSR(riscv, CSR_ID(mstatus), newValue);

```

The `writeBaseCSR` interface function has this prototype:

```

#define RISC_V_WRITE_BASE_CSR_FN(_NAME) Uns64 _NAME( \
    riscvP      riscv, \
    riscvCSRId id, \
    Uns64      newValue \
)
typedef RISC_V_WRITE_BASE_CSR_FN((*riscvWriteBaseCSRFn));

typedef struct riscvModelCBS {
    . . . fields omitted . . .
    riscvWriteBaseCSRFn      writeBaseCSR;
    . . . fields omitted . . .
} riscvModelCB;

```

Interface function `writeBaseCSR` returns the new value of the base model `mstatus` CSR, allowing for non-writable bits. To compose a result from the `mstatusw` function, the `mstatus30` bit is inserted into this return value:

```

// fill custom field from extension object
result.f.custom1 = object->mstatus30;

// return composed result
return result.u64;

```

## 7.5 Example Execution

The extension can be exercised using a simple assembler program:

```

START_TEST:

    // set up default machine-mode exception handler
    SETUP_M_HANDLER customMHandler

    // read CSR initial values
    csrr    s1, 0xBC0
    csrr    s1, 0xBC1
    csrr    s1, 0xBC2
    csrr    s1, 0xBC3
    csrr    s1, 0xFC0

    // write CSRs
    li      s1, -1
    csrwr   0xBC0, s1
    csrwr   0xBC1, s1
    csrwr   0xBC2, s1
    csrwr   0xBC3, s1
    csrwr   0xFC0, s1

    // test mstatus with custom field at bit 30
    csrr    s1, mstatus
    li      s1, -1
    csrwr   mstatus, s1
    csrr    s1, mstatus
    EXIT_TEST

.align 6

customMHandler:

    // save gp, a0, t0 (gp in scratch)
    csrrw   gp, mscratch, gp
    SX      a0, 0(gp)
    SX      t0, 8(gp)

    // calculate faulting instruction size in t0
    csrr    a0, mepc
    lhu     a0, 0(a0)
    andi    a0, a0, 3
    addi    a0, a0, -3
    li      t0, 2
    bnez    a0, 1f
    addi    t0, t0, 2

1:
    csrr    a0, mepc          // skip instruction
    add     a0, a0, t0
    csrwr   mepc, a0

    // restore registers and return
    LX      a0, 0(gp)
    LX      t0, 8(gp)
    csrrw   gp, mscratch, gp
    mret

```

This program attempts to read and write each custom CSR and the modified `mstatus` CSR. There is a simple exception handler to trap illegal accesses. This can be run using the `iss.exe` simulator like this:

```

iss.exe \
  --trace \
  --tracechange \
  --tracemode \
  --traceshowicount \
  --addressbits 32 \
  --processorvendor vendor.com \
  --processorname riscv \
  --variant RV64X \

```

## RISC-V Model Custom Extension Guide

```
--program      test.elf \
--extlib       iss/cpu0=riscv.ovpworld.org/intercept/customControl/1.0 \
```

It produces the following output:

```
Info 1: 'iss/cpu0', 0x0000000080000000(_start): Machine 4000206f j      80002400
Info 2: 'iss/cpu0', 0x0000000080002400(START_TEST): Machine 00000297 auipc  t0,0x0
Info  t0 0000000000000000 -> 0000000080002400
Info 3: 'iss/cpu0', 0x0000000080002404(START_TEST+4): Machine 08028293 addi  t0,t0,128
Info  t0 0000000080002400 -> 0000000080002480
Info 4: 'iss/cpu0', 0x0000000080002408(START_TEST+8): Machine 30529073 csrwr  mtvec,t0
Info  mtvec 0000000000000000 -> 0000000080002480
Info 5: 'iss/cpu0', 0x000000008000240c(START_TEST+c): Machine fffff297 auipc  t0,0xfffff
Info  t0 0000000080002480 -> 000000008000140c
Info 6: 'iss/cpu0', 0x0000000080002410(START_TEST+10): Machine 5f428293 addi  t0,t0,1524
Info  t0 000000008000140c -> 0000000080001a00
Info 7: 'iss/cpu0', 0x0000000080002414(START_TEST+14): Machine 34029073 csrwr  mscratch,t0
Info  mscratch 0000000000000000 -> 0000000080001a00
Info 8: 'iss/cpu0', 0x0000000080002418(START_TEST+18): Machine bc0024f3 csrwr  s1,custom_rw1_32
Info 9: 'iss/cpu0', 0x000000008000241c(START_TEST+1c): Machine bc1024f3 csrwr  s1,custom_rw2_64
Info  s1 0000000000000000 -> 0000000000000034
Info 10: 'iss/cpu0', 0x0000000080002420(START_TEST+20): Machine bc2024f3 csrwr  s1,custom_rw3_32
Info  s1 0000000000000034 -> 0000000000000000
Info 11: 'iss/cpu0', 0x0000000080002424(START_TEST+24): Machine bc3024f3 csrwr  s1,custom_rw4_64
Info 12: 'iss/cpu0', 0x0000000080002428(START_TEST+28): Machine fc0024f3 csrwr  s1,custom_rol
Info  s1 0000000000000000 -> 000000012345678
Info 13: 'iss/cpu0', 0x000000008000242c(START_TEST+2c): Machine 54fd      li      s1,-1
Info  s1 0000000012345678 -> ffffffff
Info 14: 'iss/cpu0', 0x000000008000242e(START_TEST+2e): Machine bc049073 csrwr  custom_rw1_32,s1
Info  custom_rw1_32 0000000000000000 -> 0000000000ff00ff
Info 15: 'iss/cpu0', 0x0000000080002432(START_TEST+32): Machine bc149073 csrwr  custom_rw2_64,s1
Info  custom_rw2_64 0000000000000034 -> ff00ff0000ff00ff
Info 16: 'iss/cpu0', 0x0000000080002436(START_TEST+36): Machine bc249073 csrwr  custom_rw3_32,s1
Info  custom_rw3_32 0000000000000000 -> ffffffff
Info 17: 'iss/cpu0', 0x000000008000243a(START_TEST+3a): Machine bc349073 csrwr  custom_rw4_64,s1
Info  custom_rw4_64 0000000000000000 -> ffffffff
Info 18: 'iss/cpu0', 0x000000008000243e(START_TEST+3e): Machine fc049073 csrwr  custom_rol,s1
Info  mstatus 0000000200000000 -> 0000000200001800
Info  mepc 0000000000000000 -> 000000008000243e
Info  mcause 0000000000000000 -> 0000000000000002
Info  mtval 0000000000000000 -> 00000000fc049073
Info 19: 'iss/cpu0', 0x0000000080002480(customMHandler): Machine 340191f3 csrwr  gp,mscratch,gp
Info  gp 0000000000000000 -> 0000000080001a00
Info  mscratch 0000000080001a00 -> 0000000000000000
Info 20: 'iss/cpu0', 0x0000000080002484(customMHandler+4): Machine 00alb023 sd      a0,0(gp)
Info 21: 'iss/cpu0', 0x0000000080002488(customMHandler+8): Machine 0051b423 sd      t0,8(gp)
Info 22: 'iss/cpu0', 0x000000008000248c(customMHandler+c): Machine 34102573 csrwr  a0,mepc
Info  a0 0000000000000000 -> 000000008000243e
Info 23: 'iss/cpu0', 0x0000000080002490(customMHandler+10): Machine 00055503 lhu      a0,0(a0)
Info  a0 000000008000243e -> 0000000000009073
Info 24: 'iss/cpu0', 0x0000000080002494(customMHandler+14): Machine 890d      andi    a0,a0,3
Info  a0 0000000000009073 -> 0000000000000003
Info 25: 'iss/cpu0', 0x0000000080002496(customMHandler+16): Machine 1575      addi    a0,a0,-3
Info  a0 0000000000000003 -> 0000000000000000
Info 26: 'iss/cpu0', 0x0000000080002498(customMHandler+18): Machine 4289      li      t0,2
Info  t0 0000000080001a00 -> 0000000000000002
Info 27: 'iss/cpu0', 0x000000008000249a(customMHandler+1a): Machine e111      bnez    a0,8000249e
Info 28: 'iss/cpu0', 0x000000008000249c(customMHandler+1c): Machine 0289      addi    t0,t0,2
Info  t0 0000000000000002 -> 0000000000000004
Info 29: 'iss/cpu0', 0x000000008000249e(customMHandler+1e): Machine 34102573 csrwr  a0,mepc
Info  a0 0000000000000000 -> 000000008000243e
Info 30: 'iss/cpu0', 0x00000000800024a2(customMHandler+22): Machine 9516      add     a0,a0,t0
Info  a0 000000008000243e -> 0000000080002442
Info 31: 'iss/cpu0', 0x00000000800024a4(customMHandler+24): Machine 34151073 csrwr  mepc,a0
Info  mepc 000000008000243e -> 0000000080002442
Info 32: 'iss/cpu0', 0x00000000800024a8(customMHandler+28): Machine 0001b503 ld      a0,0(gp)
Info  a0 0000000080002442 -> 0000000000000000
Info 33: 'iss/cpu0', 0x00000000800024ac(customMHandler+2c): Machine 0081b283 ld      t0,8(gp)
Info  t0 0000000000000004 -> 0000000080001a00
Info 34: 'iss/cpu0', 0x00000000800024b0(customMHandler+30): Machine 340191f3 csrwr  gp,mscratch,gp
Info  gp 0000000080001a00 -> 0000000000000000
Info  mscratch 0000000000000000 -> 0000000080001a00
```

```
Info 35: 'iss/cpu0', 0x00000000800024b4(customMHandler+34): Machine 30200073 mret
Info mstatus 000000200001800 -> 000000200000080
Info 36: 'iss/cpu0', 0x0000000080002442(START_TEST+42): Machine 300024f3 csrr s1,mstatus
Info s1 ffffffff -> 000000200000080
Info 37: 'iss/cpu0', 0x0000000080002446(START_TEST+46): Machine 54fd li s1,-1
Info s1 000000200000080 -> ffffffff
Info 38: 'iss/cpu0', 0x0000000080002448(START_TEST+48): Machine 30049073 csrwr mstatus,s1
Info mstatus 000000200000080 -> 800000240227888
Info 39: 'iss/cpu0', 0x000000008000244c(START_TEST+4c): Machine 300024f3 csrr s1,mstatus
Info s1 ffffffff -> 800000240227888
Info 40: 'iss/cpu0', 0x0000000080002450(START_TEST+50): Machine 4501 li a0,0
Info 41: 'iss/cpu0', 0x0000000080002452(START_TEST+52): Machine custom0
```

In the trace output, note that:

1. CSR `custom_rw2_64` has initial value `0x34`, defined by the reset notifier (line 9). Although the reset notifier attempted to write `0x1234` to this CSR, the CSR write mask prevented read-only bits from being reset to non-zero values.
2. Attempting to write the read-only custom CSR `custom_ro1` causes an exception (line 18).
3. Standard CSR `mstatus` now has a writable custom field at bit 30 (lines 38 and 39).



## 8 Adding Custom Exceptions (addExceptions)

The `addExceptions` extension demonstrates how to add custom exceptions to a RISC-V model. The extension adds a custom exception with cause 24, and also a custom instruction that triggers the exception.

All behavior of this extension object is implemented in file `addExceptionExtensions.c`. Sections will be discussed in turn below.

### 8.1 Exception Code

The new exception code is defined by the `riscvExtException` enumeration:

```
typedef enum riscvExtExceptionE {
    EXT_E_EXCEPT24 = 24,
} riscvExtException;
```

### 8.2 Intercept Attributes

The behavior of the extension is defined using the standard `vmiosAttr` structure:

```
vmiosAttr modelAttrs = {

    ///////////////////////////////////////////////////////////////////
    // VERSION
    ///////////////////////////////////////////////////////////////////

    .versionString = VMI_VERSION,           // version string
    .modelType     = VMI_INTERCEPT_LIBRARY, // type
    .packageName  = "addCSRs",             // description
    .objectSize   = sizeof(vmiosObject),    // size in bytes of OSS object

    ///////////////////////////////////////////////////////////////////
    // CONSTRUCTOR/DESTRUCTOR ROUTINES
    ///////////////////////////////////////////////////////////////////

    .constructorCB = addExceptionsConstructor, // object constructor

    ///////////////////////////////////////////////////////////////////
    // INSTRUCTION INTERCEPT ROUTINES
    ///////////////////////////////////////////////////////////////////

    .morphCB      = addExceptionsMorph,      // instruction morph callback
    .disCB        = addExceptionsDisassemble, // disassemble instruction

    ///////////////////////////////////////////////////////////////////
    // ADDRESS INTERCEPT DEFINITIONS
    ///////////////////////////////////////////////////////////////////

    .intercepts   = {{0}}
}
```

In this library, there is a constructor, a JIT translation (morpher) function and a disassembly function.

### 8.3 Object Type and Constructor

The object type is defined as follows:

```
typedef struct vmiosObjectS {
```

```

// Info for associated processor
riscvP      riscv;

// extended instruction decode table
vmidDecodeTableP decode32;

// extension callbacks
riscvExtCB  extCB;

} vmiosObject;

```

See chapter 6 for a detailed description of these fields, which are required when instructions are being added by an extension. The constructor initializes the fields as follows:

```

static VMIOS_CONSTRUCTOR_FN(addExceptionsConstructor) {

    riscvP riscv = (riscvP)processor;

    object->riscv = riscv;

    // prepare client data
    object->extCB.clientData = object;

    // install custom exceptions
    object->extCB.firstException = firstException;

    // install notifier when trap is taken
    object->extCB.trapNotifier = takeTrap;

    // register extension with base model using unique ID
    riscv->cb.registerExtCB(riscv, &object->extCB, EXTID_ADDEXCEPT);
}

```

In addition to initializations that were explained in chapter 6, the constructor also initializes the `firstException` and `trapNotifier` fields in the interface object.

Function `firstException` returns the first exception description in a null-terminated list of exceptions implemented by this extension. It is defined like this:

```

//
// Fill one member of exceptions
//
#define EXT_EXCEPTION(_NAME, _DESC) { \
    name:#_NAME, code:EXT_E##_NAME, description:_DESC, \
}

//
// Table of exception descriptions
//
static const vmiExceptionInfo exceptions[] = {
    EXT_EXCEPTION (EXCEPT24, "Custom Exception 24"),
    {0}
};

//
// Return the first exception implemented by the derived model
//
static RISCVP_FIRST_EXCEPTION_FN(firstException) {
    return exceptions;
}

```

Each exception has a name, an index number (the cause number) and a description. In general, any number of exceptions can be specified in the list.

Function `takeTrap` is called whenever the RISC-V processor takes a trap of any kind (interrupt or exception). It gives the extension object a chance to update state that is dependent on that trap. In this case, `takeTrap` simply reports whenever the new exception is taken:

```
static RISC_V_TRAP_NOTIFIER_FN(takeTrap) {
// vmiosObjectP object = clientData;
Uns64 pc = getPC(riscv);
Uns32 code = RD_CSR_FIELD(riscv, mcause, ExceptionCode);

if(code==EXT_E_EXCEPT24) {
vmiMessage("I", CPU_PREFIX "_TRAP",
SRCREF_FMT "TRAP:%u MODE:%u INTERRUPT:%u",
SRCREF_ARGS(riscv, pc),
code,
mode,
RD_CSR_FIELD(riscv, mcause, Interrupt)
);
}
}
```

Macro `RD_CSR_FIELD` is defined in `riscvCSR.h` in the base model. It returns the value of a field within a standard CSR structure.

Function `takeTrap` is defined using the `RISC_V_TRAP_NOTIFIER_FN` macro defined in `riscvModelCallbacks.h` in the base model:

```
#define RISC_V_TRAP_NOTIFIER_FN(_NAME) void _NAME( \
riscvP riscv, \
riscvMode mode, \
void *clientData \
)
typedef RISC_V_TRAP_NOTIFIER_FN((*riscvTrapNotifierFn));
```

The trap notifier is passed the executing RISC-V processor, the mode to which the trap is being taken and a `clientData` opaque pointer. This third argument is the `vmiosObjectP` pointer for the extension object; the commented-out line should be included if access to the intercept object is required in the notifier (in this case, it is not).

## 8.4 Instruction Decode

This extension implements a single new instruction that triggers the new custom exception. Adding new instructions is discussed in detail in chapter 6, so only brief details are given here.

The instruction type enumeration is:

```
typedef enum riscvExtITypeE {
// extension instructions
EXT_IT_EXCEPT24,
// KEEP LAST
```

```

EXT_IT_LAST
} riscvExtIType;

```

The instruction table is this:

```

const static riscvExtInstrAttrs attrsArray32[] = {
    //
    | dec | rs2 | rs1 | fn3 | rd | dec |
    EXT_INSTRUCTION(EXT_IT_EXCEPT24, "except24", RVANY, RVIP_RD_RS1_RS2, FMT_NONE,
    "|0000001|00000|00000|100|00000|0001011|")
}

```

The instruction disassembly uses the value `FMT_NONE` to specify the instruction should be disassembled without arguments.

## 8.5 *Instruction Disassembly*

Disassembly uses an identical function to that described in chapter 6, so it is not described here.

## 8.6 *Instruction Translation*

The translation attribute table table is specified like this:

```

const static riscvExtMorphAttr dispatchTable[] = {
    [EXT_IT_EXCEPT24] = {morph:emitEXCEPT24},
};

```

In this case, JIT translation function `emitEXCEPT24` is implemented like this:

```

static EXT_MORPH_FN(emitEXCEPT24) {
    vmimtArgProcessor();
    vmimtCall((vmiCallFn)takeExcept24);
}

```

This emits code to call function `takeExcept24`, passing the current processor as an argument. Function `takeExcept24` calls interface function `takeException`, which causes the processor to take a standard exception with the numeric cause passed as the second argument:

```

static void takeExcept24(riscvP riscv) {
    riscv->cb.takeException(riscv, EXT_E_EXCEPT24, 0);
}

```

## 8.7 *Example Execution*

The extension can be exercised using a simple assembler program:

```

#define EXCEPT24 .word ( \
    (0x0b << 0) | \
    (0 << 7) | \
    (4 << 12) | \
    (0 << 15) | \
    (0 << 20) | \
    (0x01 << 25) \
)

```

```

START_TEST:

    // set up default machine-mode exception handler
    SETUP_M_HANDLER customMHandler

    // validate EXCEPT24 instruction
    EXCEPT24

    // check medeleg
    li      s0, -1
    csrw    medeleg, s0

    EXIT_TEST

.align 6

customMHandler:

    // save gp, a0, t0 (gp in scratch)
    csrrw   gp, mscratch, gp
    SX      a0, 0(gp)
    SX      t0, 8(gp)

    // calculate faulting instruction size in t0
    csrr    a0, mepc
    lhu     a0, 0(a0)
    andi    a0, a0, 3
    addi    a0, a0, -3
    li      t0, 2
    bnez    a0, 1f
    addi    t0, t0, 2

1:
    csrr    a0, mepc      // skip instruction
    add     a0, a0, t0
    csrw    mepc, a0

    // restore registers and return
    LX      a0, 0(gp)
    LX      t0, 8(gp)
    csrrw   gp, mscratch, gp
    mret
    
```

This can be run using the `iss.exe` simulator like this:

```

iss.exe \
--trace                               \|
--tracechange                          \|
--tracemode                            \|
--traceshowicount                      \|
--addressbits 32                       \|
--processorvendor vendor.com           \|
--processorname riscv                  \|
--variant RV64X                        \|
--program test.elf                    \|
--extlib iss/cpu0=riscv.ovpworld.org/intercept/customControl/1.0 \|
    
```

Which produces this output:

```

Info 1: 'iss/cpu0', 0x0000000080000000(_start): Machine 4000206f j      80002400
Info 2: 'iss/cpu0', 0x0000000080002400(START_TEST): Machine 00000297 auipc  t0,0x0
Info  t0 0000000000000000 -> 0000000080002400
Info 3: 'iss/cpu0', 0x0000000080002404(START_TEST+4): Machine 04028293 addi  t0,t0,64
Info  t0 0000000080002400 -> 0000000080002440
Info 4: 'iss/cpu0', 0x0000000080002408(START_TEST+8): Machine 30529073 csrw   mtvec,t0
Info  mtvec 0000000000000000 -> 0000000080002440
Info 5: 'iss/cpu0', 0x000000008000240c(START_TEST+c): Machine fffff297 auipc  t0,0xfffff
Info  t0 0000000080002440 -> 000000008000140c
    
```

```

Info 6: 'iss/cpu0', 0x0000000080002410(START_TEST+10): Machine 5f428293 addi    t0,t0,1524
Info  t0 000000008000140c -> 0000000080001a00
Info 7: 'iss/cpu0', 0x0000000080002414(START_TEST+14): Machine 34029073 csrwr   mscratch,t0
Info  mscratch 0000000000000000 -> 0000000080001a00
Info 8: 'iss/cpu0', 0x0000000080002418(START_TEST+18): Machine 0200400b except24
Info (ADD_EXCEPT_TRAP) CPU 'iss/cpu0' 0x80002440 340191f3 csrrw   gp,mscratch,gp: TRAP:24 MODE:3
INTERRUPT:0
Info  mstatus 0000000a00000000 -> 0000000a00001800
Info  mepc 0000000000000000 -> 0000000080002418
Info  mcause 0000000000000000 -> 0000000000000018
Info 9: 'iss/cpu0', 0x0000000080002440(customMHandler): Machine 340191f3 csrrw   gp,mscratch,gp
Info  gp 0000000000000000 -> 0000000080001a00
Info  mscratch 0000000080001a00 -> 0000000000000000
Info 10: 'iss/cpu0', 0x0000000080002444(customMHandler+4): Machine 00alb023 sd     a0,0(gp)
Info 11: 'iss/cpu0', 0x0000000080002448(customMHandler+8): Machine 0051b423 sd     t0,8(gp)
Info 12: 'iss/cpu0', 0x000000008000244c(customMHandler+c): Machine 34102573 csrr   a0,mepc
Info  a0 0000000000000000 -> 0000000080002418
Info 13: 'iss/cpu0', 0x0000000080002450(customMHandler+10): Machine 00055503 lhu    a0,0(a0)
Info  a0 0000000080002418 -> 000000000000400b
Info 14: 'iss/cpu0', 0x0000000080002454(customMHandler+14): Machine 890d     andi    a0,a0,3
Info  a0 000000000000400b -> 0000000000000003
Info 15: 'iss/cpu0', 0x0000000080002456(customMHandler+16): Machine 1575     addi    a0,a0,-3
Info  a0 0000000000000003 -> 0000000000000000
Info 16: 'iss/cpu0', 0x0000000080002458(customMHandler+18): Machine 4289     li      t0,2
Info  t0 0000000080001a00 -> 0000000000000002
Info 17: 'iss/cpu0', 0x000000008000245a(customMHandler+1a): Machine e111     bnez   a0,8000245e
Info 18: 'iss/cpu0', 0x000000008000245c(customMHandler+1c): Machine 0289     addi    t0,t0,2
Info  t0 0000000000000002 -> 0000000000000004
Info 19: 'iss/cpu0', 0x000000008000245e(customMHandler+1e): Machine 34102573 csrr   a0,mepc
Info  a0 0000000000000000 -> 0000000080002418
Info 20: 'iss/cpu0', 0x0000000080002462(customMHandler+22): Machine 9516     add     a0,a0,t0
Info  a0 0000000080002418 -> 000000008000241c
Info 21: 'iss/cpu0', 0x0000000080002464(customMHandler+24): Machine 34151073 csrwr   mepc,a0
Info  mepc 0000000080002418 -> 000000008000241c
Info 22: 'iss/cpu0', 0x0000000080002468(customMHandler+28): Machine 0001b503 ld     a0,0(gp)
Info  a0 000000008000241c -> 0000000000000000
Info 23: 'iss/cpu0', 0x000000008000246c(customMHandler+2c): Machine 0081b283 ld     t0,8(gp)
Info  t0 0000000000000004 -> 0000000080001a00
Info 24: 'iss/cpu0', 0x0000000080002470(customMHandler+30): Machine 340191f3 csrrw   gp,mscratch,gp
Info  gp 0000000080001a00 -> 0000000000000000
Info  mscratch 0000000000000000 -> 0000000080001a00
Info 25: 'iss/cpu0', 0x0000000080002474(customMHandler+34): Machine 30200073 mret
Info  mstatus 0000000a00001800 -> 0000000a00000080
Info 26: 'iss/cpu0', 0x000000008000241c(START_TEST+1c): Machine 547d     li      s0,-1
Info  s0 0000000000000000 -> ffffffff
Info 27: 'iss/cpu0', 0x000000008000241e(START_TEST+1e): Machine 30241073 csrwr   medeleg,s0
Info  medeleg 0000000000000000 -> 00000000000b3fff
Info 28: 'iss/cpu0', 0x0000000080002422(START_TEST+22): Machine 4501     li      a0,0
Info 29: 'iss/cpu0', 0x0000000080002424(START_TEST+24): Machine custom0

```

At instruction 8, the `except24` extension instruction is executed, causing a Machine mode exception with cause 0x18 (24).

## 9 Adding Custom Local Interrupts (addLocalInterrupts)

The RISC-V architecture allows a processor to add custom interrupts, with numbers 16-31 (for RV32) and 16-63 (for RV64). The `addLocalInterrupts` extension demonstrates how to add two such local interrupts to a RISC-V model, with numbers 21 and 22.

Most behavior of this extension object is implemented in file `addLocalInterruptsExtensions.c`, but the processor configuration in file `riscvConfigList.c` of the linked processor model must also be modified to enable the local interrupt ports. Sections will be discussed in turn below.

### 9.1 Enabling Local Interrupt Ports

Local interrupt ports 21 and 22 are enabled by two lines in the configuration structure for each processor variant (in file `riscvConfigList.c` of the linked processor model):

```
static const riscvConfig configList[] = {
    {
        .name           = "RV32X",
        .arch           = ISA_U|RV32GC|ISA_X,
        .user_version   = RVUV_DEFAULT,
        .priv_version   = RVPV_DEFAULT,
        .tval_ii_code   = True,
        .ASID_bits      = 9,
        .local_int_num  = 7,           // enable local interrupts 16-22
        .unimp_int_mask = 0x1f0000, // int16-int20 absent
        .extensionConfigs = allExtensions,
    },
    {
        .name           = "RV64X",
        .arch           = ISA_U|RV64GC|ISA_X,
        .user_version   = RVUV_DEFAULT,
        .priv_version   = RVPV_DEFAULT,
        .tval_ii_code   = True,
        .ASID_bits      = 9,
        .local_int_num  = 7,           // enable local interrupts 16-22
        .unimp_int_mask = 0x1f0000, // int16-int20 absent
        .extensionConfigs = allExtensions,
    },
    {0} // null terminator
};
```

Specifying `local_int_num` of 7 indicates that local interrupts 16-22 are potentially implemented. Then, the specification of `unimp_int_mask` of `0x1f0000` indicates that local interrupts 16-20 are *not* implemented, leaving local interrupts 21 and 22 as the only implemented local interrupts. Using a combination of `local_int_num` and `unimp_int_mask` in this way allows any subset of the defined local interrupts to be specified as implemented.

### 9.2 Interrupt Codes

The new local interrupt codes re defined by the `riscvExtInt` enumeration in `addLocalInterruptsExtensions.c`:

```
typedef enum riscvExtIntE {
    EXT_I_INT21 = 21,
    EXT_I_INT22 = 22
} riscvExtInt;
```

### 9.3 Intercept Attributes

The behavior of the extension is defined using the standard `vmiosAttr` structure:

```
vmiosAttr modelAttrs = {

    //////////////////////////////////////
    // VERSION
    //////////////////////////////////////

    .versionString = VMI_VERSION,          // version string
    .modelType     = VMI_INTERCEPT_LIBRARY, // type
    .packageName  = "addCSRs",           // description
    .objectSize   = sizeof(vmiosObject), // size in bytes of OSS object

    //////////////////////////////////////
    // CONSTRUCTOR/DESTRUCTOR ROUTINES
    //////////////////////////////////////

    .constructorCB = addLocalInterruptsConstructor, // object constructor

    //////////////////////////////////////
    // ADDRESS INTERCEPT DEFINITIONS
    //////////////////////////////////////

    .intercepts    = {{0}}
};
```

In this library, there is a constructor that implements installation of the local interrupts.

### 9.4 Object Type and Constructor

The object type is defined as follows:

```
typedef struct vmiosObjectS {

    // Info for associated processor
    riscvP      riscv;

    // extension callbacks
    riscvExtCB  extCB;

} vmiosObject;
```

The constructor initializes the fields as follows:

```
static VMIOS_CONSTRUCTOR_FN(addLocalInterruptsConstructor) {

    riscvP riscv = (riscvP)processor;

    object->riscv = riscv;

    // prepare client data
    object->extCB.clientData = object;

    // install notifier for suppression of memory exceptions
    object->extCB.getInterruptPri = getInterruptPriority;

    // install notifier when trap is taken
    object->extCB.trapNotifier = takeTrap;
```



```
// register extension with base model using unique ID
riscv->cb.registerExtCB(riscv, &object->extCB, EXTID_ADDLOCALINT);
}
```

In addition to initializations that were explained in previous chapters, the constructor also initializes the `getInterruptPri` and `trapNotifier` fields.

Function `getInterruptPriority` specifies the *priority* of the new local interrupts, with respect to standard interrupts and each other. This priority determines which interrupt is taken if multiple interrupts become pending at the same time. The function is defined by the `RISCV_GET_INTERRUPT_PRI_FN` macro in `riscvModelCallbacks.h`:

```
#define RISCV_GET_INTERRUPT_PRI_FN(_NAME) riscvExceptionPriority _NAME( \
    riscvP riscv,          \
    Uns32 intNum,          \
    void *clientData      \
)
typedef RISCV_GET_INTERRUPT_PRI_FN((*riscvGetInterruptPriFn));
```

The interrupt priority callback is passed the current RISC-V processor, an interrupt number and a client data pointer (which is in fact the `vmiosObjectP` pointer for the current extension). It should return either 0 (if the interrupt is not recognized by this extension) or a priority based on the fixed priorities defined by the `riscvExceptionPriority` enumeration:

```
typedef enum riscvExceptionPriorityE {
    riscv_E_UTimerPriority    = 10,
    riscv_E_USWPriority       = 20,
    riscv_E_UExternalPriority = 30,
    riscv_E_STimerPriority    = 40,
    riscv_E_SSWPriority       = 50,
    riscv_E_SExternalPriority = 60,
    riscv_E_MTimerPriority    = 70,
    riscv_E_MSWPriority       = 80,
    riscv_E_MExternalPriority = 90,
    riscv_E_LocalPriority     = 100
} riscvExceptionPriority;
```

In this case, the extension defines that both custom interrupts are of higher priority than all standard interrupts, with interrupt 22 being the highest priority of all:

```
static RISCV_GET_INTERRUPT_PRI_FN(getInterruptPriority) {
    riscvExceptionPriority result = 0;

    if(intNum==EXT_I_INT21) {
        result = riscv_E_LocalPriority;
    } else if(intNum==EXT_I_INT22) {
        result = riscv_E_LocalPriority+1;
    }

    return result;
}
```

Note that the exact value returned by `getInterruptPriority` is not significant: what matters is the *relative order* of different interrupt priority codes. The gaps in the specified priorities of standard interrupts mean that a local interrupt can be defined to have any

intermediate priority between standard priorities: for example, a local interrupt of priority `riscv_E_MSWPriority+1` would be higher priority than a Machine Software Interrupt, but lower priority than a Machine External Interrupt.

Function `takeTrap` is called whenever the RISC-V processor takes a trap of any kind (interrupt or exception). It gives the extension object a chance to update state that is dependent on that trap. In this case, `takeTrap` simply reports whenever an interrupt is taken:

```
static RISC_V_TRAP_NOTIFIER_FN(takeTrap) {
    // vmiosObjectP object = clientData;
    Uns64 pc = getPC(riscv);
    Bool isInt = RD_CSR_FIELD(riscv, mcause, Interrupt);

    if(isInt) {
        vmiMessage("I", CPU_PREFIX "_TRAP",
            SRCREF_FMT "TRAP:%u MODE:%u INTERRUPT:%u",
            SRCREF_ARGS(riscv, pc),
            RD_CSR_FIELD(riscv, mcause, ExceptionCode),
            mode,
            RD_CSR_FIELD(riscv, mcause, Interrupt)
        );
    }
}
```

See chapter 8 for a detailed explanation of trap notifier behavior.

## 9.5 Example Execution

When local interrupts are configured, the base model automatically modifies behavior of related CSRs to reflect their presence. For example, `mie` and `mideleg` bit fields corresponding to the new local interrupt positions become writable. To demonstrate this, the following test program shows writability of these two registers and also validates the behavior of the `sie` register, which is dependent upon `mideleg`.

```
START_TEST:
    li        s0, -1

    // check sie (delegation disabled)
    csrw     sie, zero
    csrw     sie, s0

    // check mie & mideleg
    csrw     mie, s0
    csrw     mideleg, s0

    // check sie (delegation enabled)
    csrw     sie, zero
    csrw     sie, s0

EXIT_TEST
```

This can be run using the `iss.exe` simulator like this:

```
iss.exe \
--trace \
--tracechange \
--tracemode \
```

```

--traceshowcount          \
--addressbits             32          \
--processorvendor         vendor.com  \
--processorname           riscv       \
--variant                 RV64X      \
--program                 test.elf    \
--extlib                  iss/cpu0=riscv.ovpworld.org/intercept/customControl/1.0 \

```

Which produces this output:

```

Info 1: 'iss/cpu0', 0x0000000080000000(_start): Machine 4000206f j      80002400
Info 2: 'iss/cpu0', 0x0000000080002400(START_TEST): Machine 547d      li      s0,-1
Info   s0 0000000000000000 -> ffffffff
Info 3: 'iss/cpu0', 0x0000000080002402(START_TEST+2): Machine 10401073 csrw  sie,zero
Info 4: 'iss/cpu0', 0x0000000080002406(START_TEST+6): Machine 10441073 csrw  sie,s0
Info 5: 'iss/cpu0', 0x000000008000240a(START_TEST+a): Machine 30441073 csrw  mie,s0
Info   mie 0000000000000000 -> 0000000000600aaa
Info 6: 'iss/cpu0', 0x000000008000240e(START_TEST+e): Machine 30341073 csrw  mideleg,s0
Info   sie 0000000000000000 -> 0000000000600222
Info   mideleg 0000000000000000 -> 0000000000600222
Info 7: 'iss/cpu0', 0x0000000080002412(START_TEST+12): Machine 10401073 csrw  sie,zero
Info   sie 0000000000600222 -> 0000000000000000
Info   mie 0000000000600aaa -> 0000000000000888
Info 8: 'iss/cpu0', 0x0000000080002416(START_TEST+16): Machine 10441073 csrw  sie,s0
Info   sie 0000000000000000 -> 0000000000600222
Info   mie 0000000000000888 -> 0000000000600aaa
Info 9: 'iss/cpu0', 0x000000008000241a(START_TEST+1a): Machine 4501      li      a0,0
Info 10: 'iss/cpu0', 0x000000008000241c(START_TEST+1c): Machine custom0

```

Instructions 3 and 4 attempt to write all-zeros and all-ones to `sie`. This has no effect since no interrupts are by default delegated to Supervisor mode:

```

Info 3: 'iss/cpu0', 0x0000000080002402(START_TEST+2): Machine 10401073 csrw  sie,zero
Info 4: 'iss/cpu0', 0x0000000080002406(START_TEST+6): Machine 10441073 csrw  sie,s0

```

Instructions 5 and 6 attempt to write all-ones to `mie` and `mideleg`. The writes update bits corresponding to the custom local interrupt positions (as well as standard interrupts):

```

Info 5: 'iss/cpu0', 0x000000008000240a(START_TEST+a): Machine 30441073 csrw  mie,s0
Info   mie 0000000000000000 -> 0000000000600aaa
Info 6: 'iss/cpu0', 0x000000008000240e(START_TEST+e): Machine 30341073 csrw  mideleg,s0
Info   sie 0000000000000000 -> 0000000000600222
Info   mideleg 0000000000000000 -> 0000000000600222

```

Instructions 7 and 8 once more attempt to write all-zeros and all-ones to `sie`. This now has an effect for the interrupts that have been delegated to Supervisor mode:

```

Info 7: 'iss/cpu0', 0x0000000080002412(START_TEST+12): Machine 10401073 csrw  sie,zero
Info   sie 0000000000600222 -> 0000000000000000
Info   mie 0000000000600aaa -> 0000000000000888
Info 8: 'iss/cpu0', 0x0000000080002416(START_TEST+16): Machine 10441073 csrw  sie,s0
Info   sie 0000000000000000 -> 0000000000600222
Info   mie 0000000000000888 -> 0000000000600aaa

```

Although not shown by this simple example, net ports are now available for the new local interrupts so that they can be driven externally.

## 10 Adding Custom FIFOs (`fifoExtensions`)

The `fifoExtensions` extension object extends the basic RISC-V model by adding FIFO input and output ports and some new instructions to put data into a FIFO and get data from a FIFO. The instructions will block if the FIFO is full on a put or empty on a get.

All behavior of this extension object is implemented in file `fifoExtensions.c`. Sections will be discussed in turn below.

### 10.1 Intercept Attributes

The behavior of the library is defined using the standard `vmiosAttr` structure:

```
vmiosAttr modelAttrs = {
    ///////////////////////////////////////////////////////////////////
    // VERSION
    ///////////////////////////////////////////////////////////////////

    .versionString = VMI_VERSION,           // version string
    .modelType     = VMI_INTERCEPT_LIBRARY, // type
    .packageName  = "fifoExtensions",      // description
    .objectSize   = sizeof(vmiosObject),    // size in bytes of OSS object

    ///////////////////////////////////////////////////////////////////
    // CONSTRUCTOR/DESTRUCTOR ROUTINES
    ///////////////////////////////////////////////////////////////////

    .constructorCB = fifoConstructor,       // object constructor
    .docCB         = fifoDoc,              // documentation constructor

    ///////////////////////////////////////////////////////////////////
    // INSTRUCTION INTERCEPT ROUTINES
    ///////////////////////////////////////////////////////////////////

    .morphCB      = fifoMorph,             // instruction translation callback
    .disCB        = fifoDisassemble,      // disassemble instruction

    ///////////////////////////////////////////////////////////////////
    // PORT ACCESS ROUTINES
    ///////////////////////////////////////////////////////////////////

    .fifoPortSpecsCB = fifoGetPortSpec,    // callback for next fifo port

    ///////////////////////////////////////////////////////////////////
    // PARAMETER CALLBACKS
    ///////////////////////////////////////////////////////////////////

    .paramSpecsCB   = fifoParamSpecs,     // iterate parameter declarations
    .paramValueSizeCB = fifoParamTableSize, // get parameter table size

    ///////////////////////////////////////////////////////////////////
    // ADDRESS INTERCEPT DEFINITIONS
    ///////////////////////////////////////////////////////////////////

    .intercepts     = {{0}}
};
```

In this library, there is a constructor, a documentation callback, a JIT translation (morpher) function and a disassembly function, as in the previous example. In addition, there are functions to define the FIFO ports and to allow parameterization of the FIFO ports.

## 10.2 Object Type and Constructor

The object type is defined as follows:

```
typedef struct vmiosObjectS {
    // Info for associated processor
    riscvP      riscv;

    // parameters
    vmiParameterP  parameters;

    // is this extension enabled?
    Bool          enabled;

    // temporary FIFO element
    Uns64        FIFOTmp;

    // configuration (including CSR reset values)
    fifoConfig    config;

    // extension CSR info
    fifoCSRs      csr; // FIFO extension CSR values
    riscvCSRAttrs csrs[XCSR_ID(LAST)]; // modified CSR definitions

    // FIFO connections
    vmiFifoPortP  fifoPorts; // fifo port descriptions
    vmiConnInputP inputConn; // input FIFO connection
    vmiConnOutputP outputConn; // output FIFO connection

    // extended instruction decode table
    vmiDecodeTableP decode32;

    // extension callbacks
    riscvExtCB    extCB;
} vmiosObject;
```

This structure contains the `riscv`, `decode32` and `extCB` fields that are always required when using the RISC-V extension support infrastructure documented here, together with a number of other extension-specific fields. The constructor initializes the fields as follows:

```
static VMIOS_CONSTRUCTOR_FN(fifoConstructor) {
    riscvP      riscv = (riscvP)processor;
    paramValuesP params = parameterValues;

    object->riscv = riscv;

    // prepare client data
    object->extCB.clientData = object;

    // register extension with base model
    riscv->cb.registerExtCB(riscv, &object->extCB, EXTID_FIFO);

    // copy configuration from template
    object->config = *getExtConfig(riscv);

    // override parameterized values
    object->config.FIFO_bits = params->FIFO_bits;

    // initialize CSRs
    fifoCSRInit(object);

    // is extension enabled in config info?
    object->enabled = RD_FIFO_CSR_FIELD(object, fifo_cfg, fifoPresent);
}
```

```
// create fifoPorts
newFifoPorts(object);
}
```

In addition to the mandatory field setup, this constructor also defines extension specific CSRs and FIFO ports, as described in the next sections.

### 10.3 Extension CSRs

Extension `fifoExtensions` adds a single read-only CSR to the base model. The CSR is defined in file `fifoCSR.h`. An enumeration in that file first defines the set of additional CSRs:

```
typedef enum extCSRIdE {
    // FIFO configuration control and status registers
    XCSR_ID (fifo_cfg),      // 0xFF0

    // keep last (used to define size of the enumeration)
    XCSR_ID (LAST)
} extCSRId;
```

Then the fields in the `fifo_cfg` CSR are defined using a bitfield structure:

```
// -----
// fifo_cfg      (id 0xFF0)
// -----

// 32-bit view
typedef struct {
    Uns32 fifoPresent : 1;
    Uns32 _u1        : 31;
} CSR_REG_TYPE_32(fifo_cfg);

// define 32 bit type
CSR_REG_STRUCT_DECL_32(fifo_cfg);
```

A container structure is defined that holds all CSR values added by this extension:

```
typedef struct fifoCSRSS {
    CSR_REG_DECL(fifo_cfg);      // 0xFF0
} fifoCSRSS;
```

The `vmiosObject` structure contains fields that hold CSR values and describe the CSRs:

```
typedef struct vmiosObjectS {
    . . . lines omitted . . .

    // extension CSR info
    fifoCSRSS      csr;          // FIFO extension CSR values
    riscvCSRAttrS  csrs[XCSR_ID(LAST)]; // modified CSR definitions

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

In file `fifoExtensions.c`, the set of CSRs to add is defined using an array of `extCSRAttrs` structures:

```
static const extCSRAttrs csrs[XCSR_ID(LAST)] = {
    XCSR_ATTR_T_(
        // name      num      arch extension attrs      description      rCB rwCB wCB
        fifo_cfg, 0xFF0, 0,  EXT_FIFO, 0,0,0,0,  "FIFO Configuration", 0, 0, 0
    )
};
```

Type `extCSRAttrs` is a structure type containing a standard base model CSR description structure (`riscvCSRAttrs`) together with an extension-specific identifier, `extension`. The purpose of the extension-specific identifier is to allow CSRs to be conditionally included based on parameter settings or other selection controls in the extension itself:

```
typedef struct extCSRAttrsS {
    Uns32      extension;      // extension requirements
    riscvCSRAttrs baseAttrs;   // base attributes
} extCSRAttrs;
```

The macro `XCSR_ATTR_T__` is defined in file `riscvModelCallbackTypes.h`:

```
#define XCSR_ATTR_T__( \
    _ID, _NUM, _ARCH, _EXT, _ENDB, _ENDRM, _NOTR, _TVMT, _DESC, _RCB, _RWCB, _WCB \
) [XCSR_ID(_ID)] = { \
    .extension = _EXT, \
    .baseAttrs = { \
        name      : #_ID, \
        desc      : _DESC, \
        csrNum    : _NUM, \
        arch      : _ARCH, \
        wEndBlock : _ENDB, \
        wEndRM    : _ENDRM, \
        noTraceChange : _NOTR, \
        TVMT      : _TVMT, \
        readCB    : _RCB, \
        readWriteCB : _RWCB, \
        writeCB   : _WCB, \
        reg32     : XCSR_REG32_MT(_ID), \
        reg64     : XCSR_REG64_MT(_ID) \
    } \
}
```

The macro takes the following arguments:

1. The CSR *identifier*. This is used to construct both the CSR enumeration member name and the CSR name string (for reporting).
2. The CSR number, using standard RISC-V CSR numbering conventions.
3. Any architectural restrictions for the CSR, specified in the same way as architectural restrictions on instructions, discussed previously.
4. The extension identifier (see above).
5. *End-block* attribute: whether writes to the CSR should terminate a code block.
6. *End-rounding* attribute: whether writes to the CSR modify rounding mode.
7. *No-trace* attribute: whether changes to the CSR should not be reported when trace change is enabled.
8. *TVMT* attribute: whether accesses to the CSR are trapped by `mstatus.TVM`.

9. A CSR description string (used in documentation generation).
10. An optional read callback function.
11. An optional read-modify-write callback function (only for CSRs that have special behavior in this case).
12. An optional write callback function.

In this case, the CSR is implemented as a simple read-only register with a constant value. There are other macros available that allow specification of CSRs with a constant write mask (`XCSR_ATTR_TC_`), a variable write mask (`XCSR_ATTR_TV_`) and using callbacks only (`XCSR_ATTR_P_`).

Extension CSRs are initialized and registered with the base model by function `fifoCSRInit`, which is called by the constructor:

```
static void fifoCSRInit(vmiosObjectP object) {
    riscvP riscv = object->riscv;
    extCSRId id;

    // initialize CSR values that have configuration values defined
    WR_FIFO_CSR(object, fifo_cfg, object->config.csr.fifo_cfg.u64.bits);

    // register each CSR with the base model
    for(id=0; id<XCSR_ID(LAST); id++) {
        extCSRAttrsCP src = &csrs[id];
        riscvCSRAttrs *dst = &object->csrs[id];

        if(extensionPresent(object, src->extension)) {
            riscv->cb.newCSR(dst, &src->baseAttrs, riscv, object);
        }
    }
}
```

This function first sets the initial value of the `fifo_cfg` CSR from configuration defaults. It then iterates over all members of the CSR table, registering each CSR with the base model if the extension feature associated with that CSR is enabled (in fact, the FIFO feature is always enabled in this case, so `extensionPresent` always returns `True`). Registration with the base model is done by calling interface function `newCSR`, which takes these arguments:

1. A pointer to a destination `riscvCSRAttrs` object, which must be located in the current `vmiosObject` structure. This is filled with the source value passed as the second argument, augmented with a pointer back to the containing `vmiosObject` structure.
2. A source `riscvCSRAttrs` object, from the CSR table.
3. The RISC-V processor.
4. The containing `vmiosObject` structure.

Once the CSR is registered with the base model, reads and writes to it will be automatically performed by standard CSR access instructions with the relevant CSR index.



## 10.4 Extension FIFO Ports

Extension `fifoExtensions` adds two FIFO ports to the base model. The FIFO ports are defined by the `fifoPorts` array:

```
//
// Return offset of the given field in the extension object
//
#define EXTENSION_FIELD_OFFSET(F) ((void *)VMI_CPU_OFFSET(vmiosObjectP, _F))

//
// Template FIFO port list
//
static vmiFifoPort fifoPorts[] = {
    {"fifoPortIn" , vmi_FIFO_INPUT,  0, EXTENSION_FIELD_OFFSET(inputConn) },
    {"fifoPortOut", vmi_FIFO_OUTPUT, 0, EXTENSION_FIELD_OFFSET(outputConn)}
};
```

The macro `EXTENSION_FIELD_OFFSET` is used to initialize the handle field in each entry with the offset of the `inputConn` and `outputConn` fields in the `vmiosObject` structure:

```
typedef struct vmiosObjectS {
    . . . fields omitted . . .

    // FIFO connections
    vmiFifoPortP    fifoPorts;           // fifo port descriptions
    vmiConnInputP   inputConn;          // input FIFO connection
    vmiConnOutputP  outputConn;         // output FIFO connection

    . . . fields omitted . . .
} vmiosObject;
```

Extension FIFOs are created by function `newFifoPorts`, which is called by the constructor:

```
static void newFifoPorts(vmiosObjectP object) {
    Uns32 connBits = getConnBits(object);
    Uns32 i;

    object->fifoPorts = STYPE_CALLOC_N(vmiFifoPort, NUM_MEMBERS(fifoPorts));

    for(i=0; i<NUM_MEMBERS(fifoPorts); i++) {
        object->fifoPorts[i] = fifoPorts[i];

        // correct FIFO port bit size
        object->fifoPorts[i].bits = connBits;

        // correct FIFO port handle
        Uns8 *raw = (Uns8*)(object->fifoPorts[i].handle);
        object->fifoPorts[i].handle = (void **)(raw + (UnsPS)object);
    }
}
```

This function first allocates an extension-specific array of FIFO port objects. It then fills each FIFO port object from the template array, adjusting the `bits` field to correspond to the value given in a model parameter and correcting the `handle` field to point to the field location within the current `vmiosObject` structure (by adding the offset in the template

to the `vmiosObject` address). The bit size of each connection is extracted from the configuration using function `getConnBits`:

```
inline static Uns32 getConnBits(vmiosObjectP object) {
    return object->config.FIFO_bits;
}
```

A standard FIFO port iterator function, referenced in the `vmiosAttrs` structure, is used to indicate the presence of the new FIFO ports:

```
static VMIOS_FIFO_PORT_SPECS_FN(fifoGetPortSpec) {
    if (!object->enabled) {
        // Do not implement ports when not enabled
        return NULL;
    } else if (!prev) {
        // first port
        return object->fifoPorts;
    } else {
        // port other than the first
        Uns32 prevIndex = (prev-object->fifoPorts);
        Uns32 thisIndex = prevIndex+1;
        return (thisIndex<NUM_MEMBERS(fifoPorts)) ? &object->fifoPorts[thisIndex]:0;
    }
}
```

## 10.5 *Instruction Decode*

This extension object implements two new instructions, `pushb` and `popb`. These are defined by an enumeration and a table exactly as described in section 6.3:

```
typedef enum riscvExtITypeE {
    // extension instructions
    EXT_IT_PUSHB,
    EXT_IT_POPB,

    // KEEP LAST
    EXT_IT_LAST
} riscvExtIType;

const static riscvExtInstrAttrs attrsArray32[] = {
    EXT_INSTRUCTION(EXT_IT_PUSHB, "pushb", RVANY, RVIP_RD_RS1_RS2, FMT_R1,
        "|000000000000|00000|000|....|0001011|"),
    EXT_INSTRUCTION(EXT_IT_POPB, "popb", RVANY, RVIP_RD_RS1_RS2, FMT_R1,
        "|000000000000|00000|001|....|0001011|"),
};
```

In this case, the disassembly format is specified as `FMT_R1`, so that only one register (in the `Rd` position) is reported.

## 10.6 *Instruction Disassembly*

Instruction disassembly is implemented in exactly the same way as described in section 6.4.

## 10.7 *Instruction Translation*

Instruction translation uses a similar pattern to that previously described in section 6.5. In this example, the instruction translation table is specified like this:

```
const static riscvExtMorphAttr dispatchTable[] = {
    [EXT_IT_PUSHB] = {morph:emitPUSHB, variant:EXT_FIFO},
    [EXT_IT_POPB]  = {morph:emitPOPB,  variant:EXT_FIFO},
};
```

The JIT translation function is specified like this:

```
static VMIO_MORPH_FN(fifoMorph) {
    riscvP      riscv = (riscvP)processor;
    riscvExtMorphState state = {riscv:riscv, object:object};

    // get instruction and instruction type
    riscvExtIType type = decode(riscv, object, thisPC, &state.info);

    // action is only required if the instruction is implemented by this
    // extension
    if(type != EXT_IT_LAST) {
        riscvExtMorphAttrCP attrs = &dispatchTable[type];
        const char *reason = getDisableReason(object, attrs->variant);

        // fill translation attributes
        state.attrs = attrs;

        // translate instruction
        riscv->cb.morphExternal(&state, reason, opaque);
    }

    // no callback function is required
    return 0;
}
```

This is similar to the previous example, but includes an additional check for instruction validity, implemented by function `getDisableReason`:

```
static const char *getDisableReason(vmiosObjectP object, fifoVariant variant) {
    fifoVariant availableVariants = object->config.variant;
    const char *result = 0;

    // validate ext feature set
    if((availableVariants & variant) != variant) {
        result = "Unimplemented on this variant";
    }

    return result;
}
```

This function validates the feature set stated to be implemented by the extension configuration against the feature requirements of the instruction. If the instruction requires a feature set that is not implemented, the string "Unimplemented on this variant" is returned. When this is passed to the `morphExternal` interface function, code will be emitted to take an Illegal Instruction exception instead of the normal instruction behavior.

In this example, the FIFO extension is always implemented so the Illegal Instruction behavior is never triggered, but this pattern is useful for an extension object that adds multiple extra instructions in different sets, enabled by parameters or feature registers.

In this example, `pushb` and `popb` are implemented by separate instruction callbacks. The implementation of `pushb` is this:

```
static EXT_MORPH_FN(emitPUSHB) {
    vmiosObjectP object = state->object;
    riscvP riscv = state->riscv;
    riscvRegDesc rs = getRVReg(state, 0);
    vmiReg rsa = getVMIReg(riscv, rs);
    vmiReg conn = getOutputConn(object);
    Uns32 bits = getRBits(rs);
    Uns32 connBits = getConnBits(object);

    // zero-extend source value to temporary if connection is wider than GPR
    if(bits < connBits) {
        vmiReg tmp = getFIFOTmp(object);
        vmimtMoveExtendRR(connBits, tmp, bits, rsa, False);
        rsa = tmp;
    }

    // put value
    vmimtConnPutRB(connBits, conn, rsa, 0);
}
```

This function obtains a `vmiReg` for register `rd` in the same way as the previous example. It also obtains a `vmiReg` for the output connection object using utility function `getOutputConn`:

```
//
// Return VMI register for extension object field
//
inline static vmiReg getExtReg(vmiosObjectP object, void *field) {
    return vmimtGetExtReg((vmiProcessorP)(object->riscv), field);
}

//
// Return VMI register for output connection object
//
inline static vmiReg getOutputConn(vmiosObjectP object) {
    return getExtReg(object, &object->outputConn);
}
```

This uses the standard VMI Morph Time API function `vmimtGetExtReg` to get a `vmiReg` descriptor for a generic pointer.

The extension allows the size of the connection (FIFO) element in bits to be parameterized, up to `XLEN` bits in size. The next step is to get the size in bits of both the GPR and FIFO element:

```
Uns32 bits = getRBits(rs);
Uns32 connBits = getConnBits(object);
```

If the FIFO element is larger than `XLEN`, the value in `rd` is zero-extended to the full FIFO element width, using a temporary in the extension object to hold the extended value:

```
if(bits<connBits) {
    vmiReg tmp = getFIFOTmp(object);
    vmimtMoveExtendRR(connBits, tmp, bits, rsA, False);
    rsA = tmp;
}
```

Finally, the (possibly-extended) register value is written to the FIFO using a standard blocking put:

```
vmimtConnPutRB(connBits, conn, rsA, 0);
```

The implementation of `popb` is similar:

```
static EXT_MORPH_FN(emitPOPB) {
    vmiosObjectP object = state->object;
    riscvP riscv = state->riscv;
    riscvRegDesc rd = getRVReg(state, 0);
    vmiReg rdA = getVMIReg(riscv, rd);
    vmiReg conn = getInputConn(object);
    Uns32 bits = getRBits(rd);
    Uns32 connBits = getConnBits(object);
    Uns32 tmpBits = (connBits<bits) ? connBits : bits;
    vmiReg tmp = getFIFOTmp(object);

    // get value into temporary
    vmimtConnGetRB(connBits, tmp, conn, False, 0);

    // commit value, zero-extending if necessary
    vmimtMoveExtendRR(bits, rdA, tmpBits, tmp, False);
}
```

Here, a blocking get is made from the input FIFO to a temporary, and then the temporary is zero-extended into the result register.

## 11 Adding Transactional Memory (`tmExtensions`)

The `tmExtensions` extension object extends the basic RISC-V model by adding custom transactional memory and some new instructions to manage the transactional memory state.

Transactional memory is likely to be highly implementation dependent. In this example extension, the extended processor operates in two modes:

1. *Normal mode*: when no transaction is active, loads and stores are performed to memory in the usual way.
2. *Transaction mode*: when a transaction is active, stores are accumulated in a cache. Dirty data is either committed atomically at the end of the transaction or discarded if the transaction is aborted for some reason (for example, a conflicting write by another processor, or too much data for the cache). If a transaction is aborted, all processor GPR and FPR values are reset to the state they had when the transaction was started, allowing the transaction to be retried easily if required.

The base processor model supports operating in normal and transaction mode using interface function `setTMode`, described later in this section. In normal mode, the base model behavior is unchanged; in transaction mode, all loads and stores are routed to functions in the extension object, which implement a cache model (or similar structure) to hold speculative data values. The extension object is responsible for implementing the cache model, performing memory reads to populate the model, and performing memory writes to drain the cache model when required.

This extension adds four instructions:

1. `xbegin`: executed to start a new transaction;
2. `xend`: executed to end a transaction;
3. `xabort`: executed to abort an active transaction; and
4. `wfe`: a *wait for event* pseudo-instruction, used to yield control to other harts in a multicore system.

All behavior of this extension object is implemented in file `tmExtensions.c`. Sections will be discussed in turn below.

### 11.1 Intercept Attributes

The behavior of the library is defined using the standard `vmiosAttr` structure:

```
vmiosAttr modelAttrs = {
    ///////////////////////////////////////////////////////////////////
    // VERSION
    ///////////////////////////////////////////////////////////////////

    .versionString = VMI_VERSION,           // version string
    .modelType     = VMI_INTERCEPT_LIBRARY, // type
    .packageName  = "transactionalMemory", // description
}
```

```

.objectSize    = sizeof(vmiosObject),    // size in bytes of OSS object

////////////////////////////////////
// CONSTRUCTOR/DESTRUCTOR ROUTINES
////////////////////////////////////

.constructorCB    = tmConstructor,      // object constructor
.postConstructorCB = tmPostConstructor, // object post-constructor
.docCB            = tmDoc,              // documentation constructor

////////////////////////////////////
// INSTRUCTION INTERCEPT ROUTINES
////////////////////////////////////

.morphCB        = tmMorph,              // instruction morph callback
.disCB          = tmDisassemble,       // disassemble instruction

////////////////////////////////////
// PARAMETER CALLBACKS
////////////////////////////////////

.paramSpecsCB    = tmParamSpecs,       // iterate parameter declarations
.paramValueSizeCB = tmParamTableSize,  // get parameter table size

////////////////////////////////////
// ADDRESS INTERCEPT DEFINITIONS
////////////////////////////////////

.intercepts     = {{0}}
};

```

In this library, there is a constructor and post-constructor, a documentation callback, a JIT translation (morpher) function, a disassembly function, and functions allowing the extension to specify parameters.

## 11.2 Intercept Parameters

This extension is parameterized as follows:

1. A parameter `diagnosticlevel` allows the verbosity of debug messages to be specified (0, 1, 2 or 3); and
2. A bit mask parameter `variant` allows the configured extensions to be defined: if bit 0 is set, the transactional instructions are present, and if bit 1 is set the WFE instruction is present).

The parameters are defined using the standard extension parameterization interface as follows:

```

typedef struct formalValuesS {
    VMI_UN32_PARAM(diagnosticlevel);
    VMI_UN32_PARAM(variant);
} formalValues, *formalValuesP;

// Parameter table
static vmiParameter parameters[] = {
    VMI_UN32_PARAM_SPEC(formalValues, diagnosticlevel,    0, 0,    3,
    "Override the initial diagnostic level"),
    VMI_UN32_PARAM_SPEC(formalValues, variant,          EXT_ALL, 1, EXT_ALL,
    "Override the configured variant"),
    { 0 }
};

// Iterate formals

```

```

static VMIOS_PARAM_SPEC_FN(tmParamSpecs) {
    if(!prev) {
        prev = parameters;
    } else {
        prev++;
    }
    return prev->name ? prev : 0;
}

// Return size of parameter structure
static VMIOS_PARAM_TABLE_SIZE_FN(tmParamTableSize) {
    return sizeof(formalValues);
}

```

Functions `tmParamSpecs` and `tmParamTableSize` are referenced in the `vmiosAttr` structure for the extension (see section 11.1).

### 11.3 Object Type, Constructor and Post-Constructor

The object type is defined as follows:

```

typedef struct vmiosObjectS {

    // associated processor
    riscvP      riscv;

    // is this extension enabled?
    Bool        enabled;

    // configuration (including CSR reset values)
    tmConfig     config;

    // TM extension CSR registers info
    tmCSRs       csr; // TM extension CSR values
    riscvCSRAttrs csrs[XCSR_ID(LAST)]; // modified CSR definitions

    // Transaction state info
    memDomainP   physicalMem; // physical memory domain
    memRegionP   regionCache; // cached physical memory region
    tmStatusE    tmStatus; // current TM status
    Uns32        numPending; // number of lines currently pending
    cacheLine    pending[CACHE_MAX_LINES]; // current transaction cached lines

    // Abort state info
    Uns32        xbeginReg; // index of xbegin register
    Uns64        abortCode; // xabort code
    Addr         abortPC; // PC to jump to on abort
    Uns64        x[RISCV_GPR_NUM]; // GPR bank
    Uns64        f[RISCV_FPR_NUM]; // FPR bank

    // extended instruction decode table
    vmidDecodeTableP decode32;

    // Command argument values
    cmdArgValues cmdArgs;

    // extension callbacks
    riscvExtCB    extCB;

} vmiosObject;

```

This structure contains the `riscv`, `decode32` and `extCB` fields that are always required when using the RISC-V extension support infrastructure documented here, together with a number of other extension-specific fields. The constructor initializes the fields as follows:



```

static VMIO_CONSTRUCTOR_FN(tmConstructor) {
    riscvP      riscv = (riscvP)processor;
    formalValuesP params = parameterValues;

    object->riscv = riscv;

    // prepare client data
    object->extCB.clientData = object;

    // Initialize diagnostic setting to value set for parameter
    DIAG_LEVEL(object) = params->diagnosticlevel;

    // Add commands
    addCommands(object, &object->cmdArgs);

    // initialize base model callbacks
    object->extCB.switchCB      = riscvSwitch;
    object->extCB.tLoad        = riscvTLoad;
    object->extCB.tStore       = riscvTStore;
    object->extCB.trapNotifier = riscvTrapNotifier;
    object->extCB.ERETNotifier = riscvERETNotifier;

    // register extension with base model
    riscv->cb.registerExtCB(riscv, &object->extCB, EXTID_TM);

    // set status to not active to start
    object->tmStatus = TM_NOTACTIVE;

    // copy configuration from template
    object->config = *getExtConfig(riscv);

    // override configured variant
    object->config.variant = params->variant;

    // initialize CSRs
    tmCSRInit(object);

    // is extension enabled in config info?
    object->enabled = RD_XCSR_FIELD(object, tm_cfg, tmPresent);
}

```

In addition to the mandatory field setup, this constructor also defines extension specific CSRs and installs notifier functions that are called when execution context switches to or from another processor in a multicore simulation (`riscvSwitch`), when loads and stores are performed in transaction mode (`riscvTLoad` and `riscvTStore`) and when taking or resuming from exceptions (`riscvTrapNotifier` and `riscvERETNotifier`). These are all described in following sections.

This extension also defines a *post-constructor* function, `tmPostConstructor`. The post-constructor is called *after all processor model and processor extension object constructors have been called but before simulation starts*:

```

static VMIO_POST_CONSTRUCTOR_FN(tmPostConstructor) {
    // record the processor physical memory domain
    object->physicalMem = vmirtGetProcessorExternalDataDomain(processor);
}

```

In this case, the post-constructor saves the processor *external memory data domain* object for later use. This memory domain object is the target for all loads and stores performed

by this processor or others in a multicore simulation. It is required so that the extension object can implement cache line reads and writes (using VMI run time functions `vmirtReadNByteDomain` and `vmirtWriteNByteDomain`) and so that monitor callbacks can be installed on it to check for memory accesses by other processors that may invalidate an active transaction by this processor.

The call to `vmirtGetProcessorExternalDataDomain` must be placed in the *post-constructor* because it is *inspecting memory state*. The call cannot be made in the constructor because at that point there is no guarantee that all other processor and extension constructors have run, so any value returned by a VMI inspection function like this may return invalid state at that point. Any VMI function that references memory domains (such as `vmirtGetProcessorExternalDataDomain`) or processor state (such as `vmirtRegRead` or `vmirtRegWrite`) *must not be used in the constructor*.

The `diagnosticlevel` and `variant` parameters are used to modify the initial extension configuration.

## 11.4 Extension CSRs

Extension `tmExtensions` adds a single read-only CSR to the base model. The CSR is defined in file `tmCSR.h`. An enumeration in that file first defines the set of additional CSRs:

```
typedef enum extCSRIdE {
    // TM configuration control and status registers
    XCSR_ID (tm_cfg),          // 0xFD0

    // keep last (used to define size of the enumeration)
    XCSR_ID (LAST)
} extCSRId;
```

Then the fields in the `tm_cfg` CSR are defined using a bitfield structure:

```
// -----
// tm_cfg      (id 0xFD0)
// -----

// 32-bit view
typedef struct {
    Uns32 tmPresent : 1;
    Uns32 _u1       : 31;
} CSR_REG_TYPE_32(tm_cfg);

// define 32 bit type
CSR_REG_STRUCT_DECL_32(tm_cfg);
```

A container structure is defined that holds all CSR values added by this extension:

```
typedef struct tmCSRSS {
    CSR_REG_DECL(tm_cfg);          // 0xFD0
} tmCSRSS;
```

The `vmiosObject` structure contains fields that hold CSR values and describe the CSRs:

```
typedef struct vmiosObjectS {
    . . . lines omitted . . .

    // extension CSR info
    tmCSRs      csr;           // TM extension CSR values
    riscvCSRAttrs csrs[XCSR_ID(LAST)]; // modified CSR definitions

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

In file `tmExtensions.c`, the set of CSRs to add is defined using an array of `extCSRAttrs` structures:

```
static const extCSRAttrs csrs[XCSR_ID(LAST)] = {

    XCSR_ATTR_T__(
        // name      num      arch extension attrs      description      rCB rwCB wCB
        tm_cfg, 0xFD0, 0, EXT_TM, 0,0,0,0, "TM Configuration", 0, 0, 0
    )
};
```

This field is used in function `tmCSRInit` to initialize CSR descriptions, exactly as previously described in section 10.3.

## 11.5 Context Switch Monitor (*riscvSwitch*)

Extension `tmExtensions` installs an *execution context switch monitor* function, `riscvSwitch`:

```
static VMIOS_CONSTRUCTOR_FN(tmConstructor) {

    . . . lines omitted . . .

    // initialize base model callbacks
    object->extCB.switchCB = riscvSwitch;
    object->extCB.tLoad = riscvTLoad;
    object->extCB.tStore = riscvTStore;
    object->extCB.trapNotifier = riscvTrapNotifier;
    object->extCB.ERETNotifier = riscvERETNotifier;

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

In a multiprocessor simulation, each processor is executed in turn for a number of instructions (the quantum). The execution context switch monitor function is called whenever execution context switches *to* this processor (it is about to start executing) or *away from* this processor (it has finished its quantum and another processor is about to run). When modeling transactional memory, one requirement is that the model is aware when conflicting reads or writes have been made to memory addresses by other processors that would cause an active transaction on this processor to be aborted.

`riscvSwitch` is defined like this:

```
static RISCV_IASSWITCH_FN(riscvSwitch) {

    vmiosObjectP object = clientData;
```

```

. . . lines omitted . . .

if(state==RS_SUSPEND) {
    installCacheMonitor(object);
}
}

```

The function is of type `riscvIASSwitchFn`, defined in `riscvModelCallbacks.h` like this:

```

#define RISC_V_IASSWITCH_FN(_NAME) void _NAME( \
    riscvP      riscv, \
    vmiIASRunState state, \
    void        *clientData \
)
typedef RISC_V_IASSWITCH_FN((*riscvIASSwitchFn));

```

This function has effect if execution context is switching from this processor to another (state is `RS_SUSPEND`). In this case, function `installCacheMonitor` adds two kinds of memory callback to the physical memory domain cached by the post-constructor:

1. For any active line in the cache, a *write* callback is installed which is called if any other processor writes data to an address in that line;
2. For any dirty line in the cache, a *read* callback is installed which is called if any other processor reads data from an address in that line.

In both cases, a transaction abort is triggered because of a memory conflict.

## 11.6 Transactional Load and Store Functions (*riscvTLoad* and *riscvTStore*)

Extension `tmExtensions` installs transaction load and store functions, `riscvTLoad` and `riscvTStore`:

```

static VMIO_CONSTRUCTOR_FN(tmConstructor) {

. . . lines omitted . . .

// initialize base model callbacks
object->extCB.switchCB      = riscvSwitch;
object->extCB.tLoad         = riscvTLoad;
object->extCB.tStore        = riscvTStore;
object->extCB.trapNotifier  = riscvTrapNotifier;
object->extCB.ERETNotifier  = riscvERETNotifier;

. . . lines omitted . . .
}

```

When the processor is executing with transactional mode *enabled*, any load or store will cause these two functions to be executed instead of updating memory in the normal way. This allows the extension object to intervene to access data from another location (a cache model, in this case). Function `riscvTLoad` is of type `riscvTLoadFn`:

```

#define RISC_V_TLOAD_FN(_NAME) void _NAME( \
    riscvP riscv, \
    void *buffer, \
    Addr  VA, \
)

```

```

    Uns32 bytes,          \
    void *clientData     \
)
typedef RISC_V_TLOAD_FN>(*riscvTLoadFn);

```

This function must fill `buffer` with `bytes` bytes read from address `VA` to implement a load. Function `riscvTStore` is of type `riscvTStoreFn`:

```

#define RISC_V_TSTORE_FN(_NAME) void _NAME( \
    riscvP      riscv,          \
    const void *buffer,         \
    Addr        VA,             \
    Uns32       bytes,          \
    void        *clientData     \
)
typedef RISC_V_TSTORE_FN>(*riscvTStoreFn);

```

This function must take `bytes` bytes from `buffer` and save them in a data structure (in this case, representing cache lines). The implementation of the transactional memory will be implementation specific, so the details in this case will not be discussed here – refer to section 11.11 and the model source for a detailed example if required.

## 11.7 Trap and Exception Return Notifiers (*riscvTrapNotifier* and *riscvERETNotifier*)

Extension `tmExtensions` installs trap and exception return notifiers, `riscvTrapNotifier` and `riscvERETNotifier`:

```

static VMIO5_CONSTRUCTOR_FN(tmConstructor) {
    . . . lines omitted . . .

    // initialize base model callbacks
    object->extCB.switchCB = riscvSwitch;
    object->extCB.tLoad    = riscvTLoad;
    object->extCB.tStore   = riscvTStore;
    object->extCB.trapNotifier = riscvTrapNotifier;
    object->extCB.ERETNotifier = riscvERETNotifier;

    . . . lines omitted . . .
}

```

`riscvTrapNotifier` is called whenever the processor takes a trap:

```

static RISC_V_TRAP_NOTIFIER_FN(riscvTrapNotifier) {
    vmiosObjectP object = clientData;

    if(object->tmStatus == TM_NOTACTIVE) {
        // ignore exceptions outside of transactions
    } else {
        . . . lines omitted . . .

        // update status to abort transaction
        object->tmStatus |= TM_ABORT_EXCEPTION;

        // clear transaction mode during exception so memory updates will occur
        // normally during exception
        setTMode(object->riscv, False);
    }
}

```

```

    }
}

```

The notifier first detects whether the processor is operating in transaction mode. If it is, the current transaction is marked as aborted, for reason `TM_ABORT_EXCEPTION`. Then, transaction mode is disabled while the exception is handled by a call to `setTMode` (so that loads and stores in the exception routine behave normally). Function `setTMode` is a wrapper round interface function `setTMode` in the base model. This function toggles the base model between normal and transaction mode:

```

static void setTMode(riscvP riscv, Bool enable) {
    riscv->cb.setTMode(riscv, enable);
}

```

The exception return notifier is similar except that its final stage is to *re-enable transaction mode* to cause a transaction abort on next transaction activity:

```

static RISC_V_TRAP_NOTIFIER_FN(riscvERETNotifier) {

    vmiosObjectP object = clientData;

    if(object->tmStatus == TM_NOTACTIVE) {

        // ignore exception returns outside of transactions

    } else {

        . . . lines omitted . . .

        // update status to abort transaction
        object->tmStatus |= TM_ABORT_EXCEPTION;

        // restore transaction mode after exception so transaction abort will
        // occur on next activity
        setTMode(object->riscv, True);

    }
}

```

## 11.8 Instruction Decode

This extension object implements four new instructions, `xbegin`, `xend`, `xabort` and `wfe`. These are defined by an enumeration and a table exactly as described in section 6.3:

```

typedef enum riscvExtITypeE {

    // extension instructions
    EXT_IT_XBEGIN,
    EXT_IT_XEND,
    EXT_IT_XABORT,
    EXT_IT_WFE,
    // KEEP LAST
    EXT_IT_LAST

} riscvExtIType;

const static riscvExtInstrAttrs attrsArray32[] = {
    EXT_INSTRUCTION(EXT_IT_XBEGIN, "xbegin", RVANY, RVIP_RD_RS1_RS2, FMT_R1,
        "|0000000|00000|00000|011|....|0001011|"),
    EXT_INSTRUCTION(EXT_IT_XEND, "xend", RVANY, RVIP_RD_RS1_RS2, FMT_NONE,
        "|0000000|00000|00000|010|00000|0001011|"),
    EXT_INSTRUCTION(EXT_IT_XABORT, "xabort", RVANY, RVIP_RD_RS1_RS2, FMT_R1,
        "|0000000|00000|00000|100|....|0001011|"),
    EXT_INSTRUCTION(EXT_IT_WFE, "wfe", RVANY, RVIP_RD_RS1_RS2, FMT_NONE,

```

```
"|0000000|00000|00000|101|00000|0001011|")
};
```

In this case, the disassembly format is specified as `FMT_R1`, for `xbegin` and `xabort` so that only one register (in the `Rd` position) is reported, and as `FMT_NONE` for `xend` and `wfe`.

## 11.9 Instruction Disassembly

Instruction disassembly is implemented in exactly the same way as described in section 6.4.

## 11.10 Instruction Translation

Instruction translation uses a similar pattern to that previously described in section 6.5. In this example, the instruction translation table is specified like this:

```
const static riscvExtMorphAttr dispatchTable[] = {
  [EXT_IT_XBEGIN] = {morph:emitXBEGIN, variant:EXT_TM },
  [EXT_IT_XEND]   = {morph:emitXEND,   variant:EXT_TM },
  [EXT_IT_XABORT] = {morph:emitXABORT, variant:EXT_TM },
  [EXT_IT_WFE]   = {morph:emitWFE,    variant:EXT_WFE},
};
```

This extension allows the transaction instructions and the `WFE` instruction to be enabled separately (so it is possible to configure a core with `WFE` only, for example). To handle this, the instructions are given different `variant` masks.

The JIT translation function follows the same pattern as used previously for the `FIFO` extension: refer to section 10.7 for more information.

The `xbegin` instruction is executed to start a new transaction. JIT code for this is created by function `emitXBEGIN`:

```
static EXT_MORPH_FN(emitXBEGIN) {
    // get abstract register operands
    riscvRegDesc rd = getRVReg(state, 0);

    // emit call implementing XBEGIN instruction
    vmimtArgNatAddress(state->object);
    vmimtArgUns32(getRIndex(rd));
    vmimtArgSimPC(64);
    vmimtCall((vmiCallFn)xBegin);

    // transaction mode change possible so end this code block
    vmimtEndBlock();
}
```

This emits an embedded call to function `xBegin`. Because `xBegin` could change transaction mode, a call to `vmimtEndBlock` is required to terminate the current code block (the next instruction could be executed in different transaction mode state, and a single code block must not contain instructions from different transaction mode states for correct behavior). The arguments to `xBegin` are the extension object, the register index for the one register in the instruction, and the current simulated program counter:

```
static void xBegin(vmiosObjectP object, Uns32 regIdx, Uns64 thisPC) {
```

```

if(object->tmStatus != TM_NOTACTIVE) {
    // Nested transactions not supported
    object->tmStatus |= TM_ABORT_NESTED;
    doAbort(object);
} else {
    // save PC of next instruction (to be executed on abort)
    object->abortPC = thisPC+4;
    object->abortCode = 0;

    // save xbegin instruction destination register index
    object->xbeginReg = regIdx;

    // save current values of registers for abort, if necessary
    saveRegs(object);

    // start a new transaction
    object->tmStatus = TM_OK;
    setTMode(object->riscv, True);

    // set value in xbegin destination register
    setXbeginReturnValue(object, object->tmStatus);
}
}

```

If there is already an active transaction when `xBegin` is called, that transaction is aborted. Otherwise, `xBegin` does this:

1. It saves the address of the instruction *after* the `xbegin` instruction. This is the *abort address*, to which control will be transferred in the transaction fails.
2. It saves the index of the register argument to `xbegin`. This register is assigned a *status code* when the transaction succeeds or fails, so that the initiator can react appropriately.
3. It saves the value of all GPRs and FPRs into a shadow block implemented in the extension object. This allows these registers to be restored if the transaction fails.
4. It enters transaction mode by calling `setTMode`, with initial state `TM_OK`.
5. It returns the initial state to the register argument to `xbegin`, so that the initiator can react appropriately.

The `xend` instruction is executed to terminate an active transaction. JIT code for this is created by function `emitXEND`:

```

static EXT_MORPH_FN(emitXEND) {
    // XEND instruction is a NOP when not in a transaction
    if(getTMode(state->riscv)) {
        // emit call implementing XEND instruction
        vmimtArgNatAddress(state->object);
        vmimtCall((vmiCallFn)xEnd);

        // transaction mode change possible so end this code block
        vmimtEndBlock();
    }
}

```



If the processor is not in transaction mode, this instruction behaves as a NOP and no code is emitted. Otherwise, an embedded call to `xEnd` is emitted, to terminate the transaction. Once again, a call to `vmimtEndBlock` is required to terminate the current code block (the next instruction could be executed in different transaction mode state). Function `xEnd` is defined like this:

```
static void xEnd(vmiosObjectP object) {
    if(object->tmStatus != TM_OK) {
        // abort is pending
        doAbort(object);
    } else {
        // end current transaction
        deactivate(object);
    }
}
```

The either aborts the current transaction (if status is not `TM_OK`) or commits results (if status is `TM_OK`).

The `xabort` instruction is executed to abort an active transaction. JIT code for this is created by function `emitXABORT`:

```
static EXT_MORPH_FN(emitXABORT) {
    // XABORT instruction is a NOP when not in a transaction
    if(getTMode(state->riscv)) {
        // get abstract register operands
        riscvRegDesc rs = getRVReg(state, 0);

        // emit call implementing XABORT instruction
        vmimtArgNatAddress(state->object);
        vmimtArgUns32(getRIndex(rs));
        vmimtCall((vmiCallFn)xAbort);

        // transaction mode change possible so end this code block
        vmimtEndBlock();
    }
}
```

If the processor is not in transaction mode, this instruction behaves as a NOP and no code is emitted. Otherwise, an embedded call to `xAbort` is emitted, to abort the transaction. Once again, a call to `vmimtEndBlock` is required to terminate the current code block (the next instruction could be executed in different transaction mode state). Function `xAbort` is defined like this:

```
static void xAbort(vmiosObjectP object, Uns32 regIdx) {
    // get value from xabort source register
    object->abortCode = object->riscv->x[regIdx];

    // flag that this abort was from an instruction
    object->tmStatus |= TM_ABORT_INST;

    doAbort(object);
}
```

Here, the abort is done for reason `TM_ABORT_INST`. Function `doAbort` is as follows:

```
static void doAbort(vmiosObjectP object) {
    if (object->tmStatus == TM_NOTACTIVE) {
        // Not active so nothing to abort - ignore
    } else {
        vmiProcessorP proc = (vmiProcessorP)object->riscv;

        // restore values of RISC-V GPRs
        restoreRegs(object);

        // compute and set return value for xbegin
        Uns64 returnValue = (
            (object->tmStatus & 0xff) |
            ((object->abortCode & 0xff) << 8)
        );
        setXbeginReturnValue(object, returnValue);

        // clear current transaction
        deactivate(object);

        // set PC of next instruction (to be executed on abort)
        vmirtSetPC(proc, object->abortPC);
    }
}
```

To abort a transaction, the function does the following:

1. It restores GPR and FPR values to the state that was in effect *before* the transaction was started.
2. It constructs a return code by concatenating transaction status and the accumulated abort code, and then calls `setXbeginReturnValue` to assign that code to the GPR specified by the initiating `xbegin` operation.
3. It deactivates transaction mode by calling `deactivate`.
4. It uses `vmirtSetPC` to force a jump to the abort address, which is the address *after* the initiating `xbegin` instruction.

Note that in a linked model extension library it is legal to directly access fields of the base RISC-V model in cases where this can be done safely. As an example, function `restoreRegs` directly accesses the GPR and FPR values from the main model to restore them:

```
static void restoreRegs(vmiosObjectP object) {
    riscvP riscv = object->riscv;
    Uns32 i;

    for(i=1; i<RISCV_GPR_NUM; i++) {
        riscv->x[i] = object->x[i];
    }
    for(i=0; i<RISCV_FPR_NUM; i++) {
        riscv->f[i] = object->f[i];
    }
}
```

Function `deactivate` is used to transition from transaction mode to normal mode:

```
static void deactivate(vmiosObjectP object) {  
    . . . lines omitted . . .  
  
    xCommit(object);  
  
    setTMode(object->riscv, False);  
    object->tmStatus = TM_NOTACTIVE;  
    object->abortCode = 0;  
}
```

As part of the deactivation process, live data in the transaction mode cache model is drained to memory by calling function `xCommit`. Then, normal mode is enabled by calling `setTMode` with `enable` of `False`.

The `wfe` instruction is executed to suspend this processor until the end of its quantum to allow others to run (in real hardware, a similar instruction could cause a processor to stop executing and enter a low power state). JIT code for this is created by function `emitWFE`:

```
static EXT_MORPH_FN(emitWFE) {  
    vmimtIdle();  
}
```

### 11.11 **Memory Model Implementation Guidelines**

As previously stated, the transactional memory model is likely to be highly implementation dependent. This document will therefore not describe the chosen implementation in the `tmExtensions` example in detail, but instead will give some general guidelines on the approach to adopt.

1. Obtain handles to required memory domain objects in the post-constructor, as described above.
2. In the transactional load and store callback functions, use `vmirtReadNByteDomain` to read in cache line data.
3. When a transaction is being committed, use `vmirtWriteNByteDomain` to drain cache contents to physical memory.
4. When context switches *away* from the current processor, use `vmirtAddReadCallback` and `vmirtAddWriteCallback` to monitor address for which reads and writes by another processor should abort transactions on the current processor, respectively.
5. When a transaction is being committed or aborted, use `vmirtRemoveReadCallback` and `vmirtRemoveWriteCallback` to remove address monitors if required.

## 12 Appendix: Standard Instruction Patterns

This appendix describes the format of the standard instruction patterns specified by the `riscvExtInstrPattern` enumeration, including details of suitable disassembly format macros and fields set in the `riscvExtInstrInfo` structure during decode.

### 12.1 Pattern *RVIP\_RD\_RS1\_RS2*

Description: like `add x0, x1, x2`  
 Decode: 

	rs2	rs1	rd	
vvvvvvvv	.....	.....	vvv	..... vvvvvvvv

  
 Format: `FMT_R1_R2_R3` (Or `FMT_R1_R2` or `FMT_R1`)  
 Fields set: `r[0]=rd, r[1]=rs1, r[2]=rs2`

### 12.2 Pattern *RVIP\_RD\_RS1\_SI*

Description: like `addi x0, x1, imm` (immediate sign-extended to `XLEN` bits)  
 Decode: 

	imm	rs1	rd	
.....	.....	vvv	.....	vvvvvvvv

  
 Format: `FMT_R1_R2_SIMM`, `FMT_R1_R2_XIMM`  
 Fields set: `r[0]=rd, r[1]=rs1, c=imm`

### 12.3 Pattern *RVIP\_RD\_RS1\_SHIFT*

Description: like `slli x0, x1, shift`  
 Decode: 

	shift	rs1	rd	
0000000.....	.....	vvv	.....	vvvvvvvv  (RV32)
000000.....	.....	vvv	.....	vvvvvvvv  (RV64)

  
 Format: `FMT_R1_R2_SIMM`, `FMT_R1_R2_XIMM`  
 Fields set: `r[0]=rd, r[1]=rs1, c=shift`

### 12.4 Pattern *RVIP\_RD\_RS1\_RS2\_RS3*

Description: like `cmix x0, x1, x2, x3`  
 Decode: 

rs3	rs2	rs1	rd	
.....	vv	.....	vvv	..... vvvvvvvv

  
 Format: `FMT_R1_R2_R3_R4`  
 Fields set: `r[0]=rd, r[1]=rs1, r[2]=rs2, r[3]=rs3`

### 12.5 Pattern *RVIP\_RD\_RS1\_RS3\_SHIFT*

Description: like `fsri x0, x1, x2, shift`  
 Decode: 

rs3	shift	rs1	rd	
.....	v 0.....	.....	vvv	..... vvvvvvvv  (RV32)
.....	v .....	.....	vvv	..... vvvvvvvv  (RV64)

  
 Format: `FMT_R1_R2_R3_SIMM`  
 Fields set: `r[0]=rd, r[1]=rs1, r[2]=rs3, c=shift`

**12.6 Pattern RVIP\_FD\_FS1\_FS2**

Description: like `fmax.s f0, f1, f2`  
 Decode: 

	fs2	fs1		fd		
vvvvvvvW	.....	.....	vvv	.....	vvvvvvvv	

 Format: FMT\_R1\_R2\_R3 (Or FMT\_R1\_R2 Or FMT\_R1)  
 Fields set: `r[0]=fd, r[1]=fs1, r[2]=fs2`  
 Width: `W=0:s, W=1:d`

**12.7 Pattern RVIP\_FD\_FS1\_FS2\_RM**

Description: like `fadd.s f0, f1, f2, rte`  
 Decode: 

	fs2	fs1	rm	fd		
vvvvvvvW	.....	.....	...	.....	vvvvvvvv	

 Format: FMT\_R1\_R2\_R3 (Or FMT\_R1\_R2 Or FMT\_R1)  
 Fields set: `r[0]=fd, r[1]=fs1, r[2]=fs2, rm=rm`  
 Width: `W=0:s, W=1:d`

**12.8 Pattern RVIP\_FD\_FS1\_FS2\_FS3\_RM**

Description: like `fmadd.s f0, f1, f2, f3, rte`  
 Decode: 

	fs3		fs2	fs1	rm	fd		
.....	vW	.....	.....	...	.....	vvvvvvvv		

 Format: FMT\_R1\_R2\_R3\_R4  
 Fields set: `r[0]=fd, r[1]=fs1, r[2]=fs2, r[3]=fs3, rm=rm`  
 Width: `W=0:s, W=1:d`

**12.9 Pattern RVIP\_RD\_FS1\_FS2**

Description: like `feq.s x0, f1, f2`  
 Decode: 

	fs2	fs1		rd		
vvvvvvvW	.....	.....	vvv	.....	vvvvvvvv	

 Format: FMT\_R1\_R2\_R3  
 Fields set: `r[0]=rd, r[1]=fs1, r[2]=fs2`  
 Width: `W=0:s, W=1:d`

**12.10 Pattern RVIP\_VD\_VS1\_VS2\_M**

Description: like `vadd.vv v1, v2, v3, v0.m`  
 Decode: 

	m	vs1	vs2		vd		
vvvvvvv	.	.....	.....	vvv	.....	vvvvvvvv	

 Format: FMT\_R1\_R2\_R3\_RM  
 Fields set: `r[0]=vd, r[1]=vs1, r[2]=vs2, mask=m?none:v0`

**12.11 Pattern RVIP\_VD\_VS1\_SI\_M**

Description: like `vadd.vi v1, v2, imm, v0.m` (immediate sign-extended to SEW bits)  
 Decode: 

	m	vs1	imm		vd		
vvvvvvv	.	.....	.....	vvv	.....	vvvvvvvv	

 Format: FMT\_R1\_R2\_SIMM\_RM  
 Fields set: `r[0]=vd, r[1]=vs1, c=imm, mask=m?none:v0`

### 12.12 *Pattern RVIP\_VD\_VS1\_UI\_M*

Description: like `vs11.vi v1, v2, imm, v0.m` (immediate zero-extended to SEW bits)

Decode: 

	m	vs1		imm		vd		
vvvvvvv	.	.....	.....	vvv	.....	vvvvvvvv		

Format: FMT\_R1\_R2\_SIMM\_RM

Fields set: `r[0]=vd, r[1]=vs1, c=imm, mask=m?none:v0`

### 12.13 *Pattern RVIP\_VD\_VS1\_RS2\_M*

Description: like `vadd.vx v1, v2, x3, v0.m`

Decode: 

	m	vs1		rs2		vd		
vvvvvvv	.	.....	.....	vvv	.....	vvvvvvvv		

Format: FMT\_R1\_R2\_R3\_RM

Fields set: `r[0]=vd, r[1]=vs1, r[2]=rs2, mask=m?none:v0`

### 12.14 *Pattern RVIP\_VD\_VS1\_FS2\_M*

Description: like `vfadd.vf v1, v2, f3, v0.m`

Decode: 

	m	vs1		fs2		vd		
vvvvvvv	.	.....	.....	vvv	.....	vvvvvvvv		

Format: FMT\_R1\_R2\_R3\_RM

Fields set: `r[0]=vd, r[1]=vs1, r[2]=fs2, mask=m?none:v0`

### 12.15 *Pattern RVIP\_RD\_VS1\_RS2*

Description: like `vext.v r1, v2, r3`

Decode: 

	vs1		rs2		rd		
vvvvvvvv	.....	.....	vvv	.....	vvvvvvvv		

Format: FMT\_R1\_R2\_R3

Fields set: `r[0]=rd, r[1]=vs1, r[2]=rs2`

### 12.16 *Pattern RVIP\_RD\_VS1\_M*

Description: like `vpopc.m x1, v2, v0.m`

Decode: 

	m	vs1		rd		
vvvvvvv	.	.....	vvvvvvvvv	.....	vvvvvvvvv	

Format: FMT\_R1\_R2\_RM

Fields set: `r[0]=rd, r[1]=vs1, mask=m?none:v0`

### 12.17 *Pattern RVIP\_VD\_RS2*

Description: like `vmv.s.x v1, x2`

Decode: 

	rs2		vd		
vvvvvvvvvvvvvvv	.....	vvv	.....	vvvvvvvv	

Format: FMT\_R1\_R2

Fields set: `r[0]=vd, r[1]=rs2`

**12.18 Pattern RVIP\_FD\_VS1**Description: like `vfmv.f.s f1, v2`Decode: 

		vs2		fd		
	vvvvvvvv	.....	vvvvvvvv	.....	vvvvvvvv	

Format: FMT\_R1\_R2

Fields set: `r[0]=fd, r[1]=vs2`**12.19 Pattern RVIP\_VD\_FS2**Description: like `vfmv.s.f v1, f2`Decode: 

		fs2		vd		
	vvvvvvvvvvvv	.....	v	.....	vvvvvv	

Format: FMT\_R1\_R2

Fields set: `r[0]=vd, r[1]=fs2`

## 13 Appendix: Base Model Interface Service Functions

This appendix describes *interface functions* implemented by the *base model* that are available to provide services for a *linked model extension library*. All such interface functions are installed in a structure of type `riscvModelCB` accessible via the `cb` field in the RISC-V processor structure. For example, an extension library can call the `takeException` interface function (which causes an exception to be immediately taken) like this:

```
riscv->cb.takeException(riscv, EXT_E_EXCEPT24, 0);
```

The `riscvModelCB` type is defined in file `riscvModelCallbacks.h` in the base model:

```
typedef struct riscvModelCBS {
    // from riscvUtils.h
    riscvRegisterExtCBFn      registerExtCB;
    riscvGetExtClientDataFn  getExtClientData;
    riscvGetExtConfigFn      getExtConfig;
    riscvGetXlenFn           getXlenMode;
    riscvGetXlenFn           getXlenArch;
    riscvGetRegNameFn        getXRegName;
    riscvGetRegNameFn        getFRegName;
    riscvGetRegNameFn        getVRegName;
    riscvSetTModeFn          setTMode;
    riscvGetTModeFn          getTMode;
    riscvGetDataEndianFn     getDataEndian;
    riscvReadCSRNumFn        readCSR;
    riscvWriteCSRNumFn       writeCSR;
    riscvReadBaseCSRFn       readBaseCSR;
    riscvWriteBaseCSRFn      writeBaseCSR;

    // from riscvExceptions.h
    riscvHaltRestartFn        halt;
    riscvHaltRestartFn        restart;
    riscvUpdateInterruptFn    updateInterrupt;
    riscvUpdateDisableFn      updateDisable;
    riscvTestInterruptFn      testInterrupt;
    riscvIllegalInstructionFn  illegalInstruction;
    riscvIllegalVerboseFn     illegalVerbose;
    riscvIllegalInstructionFn  virtualInstruction;
    riscvIllegalVerboseFn     virtualVerbose;
    riscvTakeExceptionFn      takeException;
    riscvTakeResetFn          takeReset;

    // from riscvDecode.h
    riscvFetchInstructionFn    fetchInstruction;

    // from riscvDisassemble.h
    riscvDisassInstructionFn    disassInstruction;

    // from riscvMorph.h
    riscvInstructionEnabledFn  instructionEnabled;
    riscvMorphExternalFn      morphExternal;
    riscvMorphIllegalFn       morphIllegal;
    riscvMorphIllegalFn       morphVirtual;
    riscvGetVMIRegFn          getVMIReg;
    riscvGetVMIRegFSFn        getVMIRegFS;
    riscvWriteRegSizeFn        writeRegSize;
    riscvWriteRegFn           writeReg;
    riscvGetFPFlagsMtFn        getFPFlagsMt;
    riscvGetDataEndianMtFn     getDataEndianMt;
    riscvCheckLegalRMMtFn      checkLegalRMMt;
    riscvMorphTrapTVMFn       morphTrapTVM;
};
```



```

riscvMorphVOpFn      morphVOp;

// from riscvCSR.h
riscvNewCSRFn       newCSR;

// from riscvVM.h
riscvUpdateLdStDomainFn  updateLdStDomain;
riscvNewTLBEntryFn     newTLBEntry;
riscvFreeTLBEntryFn    freeTLBEntry;
} riscvModelCB;

```

Following subsections describe each interface function.

### 13.1 *Function registerExtCB*

```

#define RISC_V_REGISTER_EXT_CB_FN(_NAME) void _NAME( \
    riscvP      riscv, \
    riscvExtCBP extCB, \
    Uns32       id     \
)
typedef RISC_V_REGISTER_EXT_CB_FN((*riscvRegisterExtCBFn));

typedef struct riscvModelCBS {
    riscvRegisterExtCBFn  registerExtCB;
} riscvModelCB;

```

#### Description

This interface function is called from the extension object constructor to register that extension object with the base model. It requires the RISC-V processor, an object of type `riscvExtCBP` and an identifier as arguments. The identifier must be a unique index among all extensions added to the RISC-V processor.

The `riscvExtCBP` object should be pointer to a field of type `riscvExtCB` that is declared in the extension `vmiosObject` structure. The `riscvExtCB` is filled with callback functions and other information by the extension object constructor. These fields are used by the base model, primarily to notify the extension object of state changes in other events, and also to allow the extension object to modify some base model behavior.

#### Example

```

static VMIOS_CONSTRUCTOR_FN(addInstructionsConstructor) {

    riscvP riscv = (riscvP)processor;

    object->riscv = riscv;

    // prepare client data
    object->extCB.clientData = object;

    // register extension with base model using unique ID
    riscv->cb.registerExtCB(riscv, &object->extCB, EXTID_ADDINST);
}

```

#### Usage Context

Container or leaf level.

## 13.2 *Function getExtClientData*

```
#define RISC_V_GET_EXT_CLIENT_DATA_FN(_NAME) void *_NAME( \  
    riscvP riscv,          \  
    Uns32 id              \  
    ) \  
typedef RISC_V_GET_EXT_CLIENT_DATA_FN((*riscvGetExtClientDataFn)); \  
 \  
typedef struct riscvModelCBS { \  
    riscvGetExtClientDataFn  getExtClientData; \  
} riscvModelCB;
```

### Description

This interface function returns any extension object previously registered with the processor which has the given unique identifier.

### Example

```
vmiosObjectP getExtObject(riscvP riscv) { \  
    vmiosObjectP object = riscv->cb.getExtClientData(riscv, EXT_ID); \  
    return object; \  
}
```

### Usage Context

Container or leaf level.

### 13.3 *Function getExtConfig*

```
#define RISC_V_GET_EXT_CONFIG_FN(_NAME) riscvExtConfigCP _NAME( \
    riscvP riscv, \
    Uns32 id \
)
typedef RISC_V_GET_EXT_CONFIG_FN((*riscvGetExtConfigFn));

typedef struct riscvModelCBS {
    riscvGetExtConfigFn    getExtConfig;
} riscvModelCB;
```

#### Description

This interface function returns any extension configuration object for the processor, given an extension unique identifier. The extension configuration holds any variant-specific information that may modify the behavior of an extension. The extension configuration is held with the standard configuration information for a variant.

#### Example

With configuration list:

```
static riscvExtConfigCP allExtensions[] = {

    // example adding CSRs
    &(const riscvExtConfig){
        .id = EXTID_ADDCSR,
        .userData = &(const addCSRsConfig){
            .csr = {
                .custom_rol = {u32 : {bits : 0x12345678}}
            }
        },
    },
    0
};
```

In extension object:

```
static addCSRsConfigCP getExtConfig(riscvP riscv) {
    riscvExtConfigCP cfg = riscv->cb.getExtConfig(riscv, EXTID_ADDCSR);
    return cfg->userData;
}
```

See section 7 for a complete example.

#### Usage Context

Container or leaf level.

### 13.4 *Function `getXlenMode`*

```
#define RISC_V_GET_XLEN_FN(_NAME) Uns32 _NAME(riscvP riscv)
typedef RISC_V_GET_XLEN_FN(*riscvGetXlenFn);

typedef struct riscvModelCBS {
    riscvGetXlenFn      getXlenMode;
} riscvModelCB;
```

#### **Description**

This interface function returns the currently-active XLEN.

#### **Example**

```
inline static Uns32 getXlenBits(riscvP riscv) {
    return riscv->cb.getXlenMode(riscv);
}
```

#### **Usage Context**

Leaf level only.

### 13.5 *Function getXlenArch*

```
#define RISC_V_GET_XLEN_FN(_NAME) Uns32 _NAME(riscvP riscv)
typedef RISC_V_GET_XLEN_FN(*riscvGetXlenFn);

typedef struct riscvModelCBS {
    riscvGetXlenFn      getXlenArch;
} riscvModelCB;
```

#### **Description**

This interface function returns the processor architectural XLEN.

#### **Example**

```
inline static Uns32 getXLenArch(riscvP riscv) {
    return riscv->cb.getXlenArch(riscv);
}
```

#### **Usage Context**

Leaf level only.

## 13.6 *Function getXRegName*

```
#define RISC_V_GET_REG_NAME_FN(_NAME) const char *_NAME(Uns32 index)
typedef RISC_V_GET_REG_NAME_FN(*riscvGetRegNameFn);

typedef struct riscvModelCBS {
    riscvGetRegNameFn      getXRegName;
} riscvModelCB;
```

### Description

This interface function returns the name of a RISC-V GPR given its index.

### Example

```
const char *getXRegName(riscvP riscv, Uns32 index) {
    return riscv->cb.getXRegName(index);
}
```

### Usage Context

Leaf level only.

### 13.7 *Function getFRegName*

```
#define RISC_V_GET_REG_NAME_FN(_NAME) const char *_NAME(Uns32 index)
typedef RISC_V_GET_REG_NAME_FN(*riscvGetRegNameFn);

typedef struct riscvModelCBS {
    riscvGetRegNameFn    getFRegName;
} riscvModelCB;
```

#### **Description**

This interface function returns the name of a RISC-V FPR given its index.

#### **Example**

```
const char *getFRegName(riscvP riscv, Uns32 index) {
    return riscv->cb.getFRegName(index);
}
```

#### **Usage Context**

Leaf level only.

### 13.8 *Function getVRegName*

```
#define RISC_V_GET_REG_NAME_FN(_NAME) const char *_NAME(Uns32 index)
typedef RISC_V_GET_REG_NAME_FN(*riscvGetRegNameFn);

typedef struct riscvModelCBS {
    riscvGetRegNameFn    getVRegName;
} riscvModelCB;
```

#### **Description**

This interface function returns the name of a RISC-V vector register given its index.

#### **Example**

```
const char *getVRegName(riscvP riscv, Uns32 index) {
    return riscv->cb.getVRegName(index);
}
```

#### **Usage Context**

Leaf level only.



## 13.9 *Function setTMode*

```
#define RISCV_SET_TMODE_FN(_NAME) void _NAME(riscvP riscv, Bool enable)
typedef RISCV_SET_TMODE_FN(*riscvSetTModeFn);

typedef struct riscvModelCBS {
    riscvSetTModeFn      setTMode;
} riscvModelCB;
```

### Description

This interface function enables or disabled *transactional memory mode* for the processor. When processors with transactional memory are being modeled, some instructions behave differently when the mode is enabled. For example, loads and stores typically accumulate information in cache lines or other structures without committing the values to memory in transactional mode.

Enabling transactional memory mode causes JIT code translations to be stored in and used from *a separate code dictionary* while that mode is active. This means that different behaviors for the same instructions can be modeled without the inefficiency inherent in transactional memory behavior affecting non-transaction mode.

Note that implementation of transactional memory requires standard load/store instructions (and others) to be reimplemented in the extension object in the case that transactional mode is active. Contact Imperas for further advice about modeling such features.

### Example

```
static void setTMode(riscvP riscv, Bool enable) {
    riscv->cb.setTMode(riscv, enable);
}
```

### Usage Context

Leaf level only.

### 13.10 *Function getTMode*

```
#define RISC_V_GET_TMODE_FN(_NAME) Bool _NAME(riscvP riscv)
typedef RISC_V_GET_TMODE_FN(*riscvGetTModeFn);

typedef struct riscvModelCBS {
    riscvGetTModeFn    getTMode;
} riscvModelCB;
```

#### **Description**

This interface function returns a Boolean indicating whether transactional memory mode is currently active. The function may be called at either run time or morph time (within the morph callback). See section 13.9 for more information.

#### **Example**

```
static Bool getTMode(riscvP riscv) {
    return riscv->cb.getTMode(riscv);
}
```

#### **Usage Context**

Leaf level only.

### 13.11 *Function `getDataEndian`*

```
#define RISC_V_GET_DATA_ENDIAN_FN(_NAME) memEndian _NAME( \  
    riscvP riscv, \  
    riscvMode mode \  
    )  
typedef RISC_V_GET_DATA_ENDIAN_FN(*riscvGetDataEndianFn);  
  
typedef struct riscvModelCBS {  
    riscvGetDataEndianFn    getDataEndian;  
} riscvModelCB;
```

#### **Description**

This interface function returns the active endianness for loads and stores in the given processor mode. Usually, RISC-V processors used little-endian order for loads and stores, but this is configurable.

#### **Example**

```
static memEndian getDataEndian(riscvP riscv, riscvMode mode) {  
    return riscv->cb.getDataEndian(riscv, mode);  
}
```

#### **Usage Context**

Leaf level only.

### 13.12 *Function readCSR*

```
#define RISCV_READ_CSR_NUM_FN(_NAME) Uns64 _NAME( \  
    riscvP riscv, \  
    Uns32  csrNum \  
    ) \  
typedef RISCV_READ_CSR_NUM_FN((*riscvReadCSRNumFn)); \  
 \  
typedef struct riscvModelCBS { \  
    riscvReadCSRNumFn    readCSR; \  
} riscvModelCB;
```

#### **Description**

This interface function returns the current value of a CSR, given its index number. The value returned can be either from the base model or from a CSR implemented in an extension object.

#### **Example**

```
static Uns64 readCSR(riscvP riscv, Uns32 csrNum) { \  
    return riscv->cb.readCSR(riscv, csrNum); \  
}
```

#### **Usage Context**

Leaf level only.

### 13.13 *Function writeCSR*

```
#define RISCV_WRITE_CSR_NUM_FN(_NAME) Uns64 _NAME( \  
    riscvP riscv,          \  
    Uns32  csrNum,         \  
    Uns64  newValue        \  
    ) \  
typedef RISCV_WRITE_CSR_NUM_FN(*riscvWriteCSRNumFn); \  
 \  
typedef struct riscvModelCBS { \  
    riscvWriteCSRNumFn    writeCSR; \  
} riscvModelCB;
```

#### **Description**

This interface function writes a new value to a CSR, given its index number. The CSR updated can be implemented either in the base model or in an extension object.

#### **Example**

```
static void writeCSR(riscvP riscv, Uns32 csrNum, Uns64 newValue) { \  
    riscv->cb.writeCSR(riscv, csrNum, newValue); \  
}
```

#### **Usage Context**

Leaf level only.

### 13.14 *Function readBaseCSR*

```
#define RISCV_READ_BASE_CSR_FN(_NAME) Uns64 _NAME( \  
    riscvP    riscv,    \  
    riscvCSRId id      \  
    ) \  
typedef RISCV_READ_BASE_CSR_FN((*riscvReadBaseCSRFn)); \  
 \  
typedef struct riscvModelCBS { \  
    riscvReadBaseCSRFn    readBaseCSR; \  
} riscvModelCB;
```

#### **Description**

This interface function returns the current value of a CSR, given its `riscvCSRId` identifier (*not* the CSR number). The CSR read will be that in the base model, disregarding any extension object redefinition. This function is useful when an extension object wishes to add fields into a standard CSR, retaining the behavior of the standard fields. In this case, the extension can install a replacement for the standard CSR, handle the new fields itself and merge in standard fields using `readBaseCSR` and `writeBaseCSR`.

#### **Example**

```
inline static Uns64 baseR(riscvP riscv, riscvCSRId id) { \  
    return riscv->cb.readBaseCSR(riscv, id); \  
}
```

#### **Usage Context**

Leaf level only.

### 13.15 Function *writeBaseCSR*

```
#define RISCV_WRITE_BASE_CSR_FN(_NAME) Uns64 _NAME( \  
    riscvP    riscv,    \  
    riscvCSRid id,    \  
    Uns64    newValue    \  
    ) \  
typedef RISCV_WRITE_BASE_CSR_FN((*riscvWriteBaseCSRFn)); \  
 \  
typedef struct riscvModelCBS { \  
    riscvWriteBaseCSRFn    writeBaseCSR; \  
} riscvModelCB;
```

#### Description

This interface function writes the current value of a CSR, given its `riscvCSRid` identifier (*not* the CSR number). The CSR written will be that in the base model, disregarding any extension object redefinition. This function is useful when an extension object wishes to add fields into a standard CSR, retaining the behavior of the standard fields. In this case, the extension can install a replacement for the standard CSR, handle the new fields itself and merge in standard fields using `readBaseCSR` and `writeBaseCSR`.

#### Example

```
inline static void baseW(riscvP riscv, riscvCSRid id, Uns64 newValue) { \  
    riscv->cb.writeBaseCSR(riscv, id, newValue); \  
}
```

#### Usage Context

Leaf level only.

### 13.16 *Function halt*

```
#define RISCV_HALT_RESTART_FN(_NAME) void _NAME( \
    riscvP          riscv, \
    riscvDisableReason reason \
)
typedef RISCV_HALT_RESTART_FN((*riscvHaltRestartFn));

typedef struct riscvModelCBS {
    riscvHaltRestartFn    halt;
} riscvModelCB;
```

#### Description

This interface function causes the given hart to halt, for a reason given by the `riscvDisableReason` enumerated type:

```
typedef enum riscvDisableReasonE {
    RVD_ACTIVE   = 0x0, // processor running
    RVD_WFI     = 0x1, // processor halted in WFI
    RVD_RESET   = 0x2, // processor halted in reset
    RVD_DEBUG   = 0x4, // processor halted for debug
    RVD_CUSTOMI = 0x8, // processor halted for interruptible custom reason
};
```

The reason can be for WFI, in reset state or in debug mode, or for a custom reason. The hart will remain halted until a subsequent call to the `restart` function or a reset or other interrupt event.

#### Example

```
static void ceaseHart(riscvP riscv) {
    riscv->cb.halt(riscv, RVD_RESET);
}
```

#### Usage Context

Leaf level only.



### 13.17 *Function restart*

```
#define RISC_V_HALT_RESTART_FN(_NAME) void _NAME( \
    riscvP          riscv, \
    riscvDisableReason reason \
)
typedef RISC_V_HALT_RESTART_FN((*riscvHaltRestartFn));

typedef struct riscvModelCBS {
    riscvHaltRestartFn    restart;
} riscvModelCB;
```

#### Description

This interface function causes the given hart to restart, for the reason given by the `riscvDisableReason` enumerated type:

```
typedef enum riscvDisableReasonE {
    RVD_ACTIVE   = 0x0, // processor running
    RVD_WFI      = 0x1, // processor halted in WFI
    RVD_RESET    = 0x2, // processor halted in reset
    RVD_DEBUG    = 0x4, // processor halted for debug
    RVD_CUSTOMI  = 0x8, // processor halted for interruptible custom reason
};
```

The reason can be for WFI, in reset state or in debug mode, or for a custom reason. The function has no effect if the hart is not halted for the given reason.

#### Example

```
RISC_V_LRSC_ABORT_FN(custLRSCAbort) {
    if(riscv->disable & RVD_CUSTOMI) {
        riscv->cb.restart(riscv, RVD_CUSTOMI);
    }
}
```

#### Usage Context

Leaf level only.

### 13.18 *Function updateInterrupt*

```
#define RISC_V_UPDATE_INTERRUPT_FN(_NAME) void _NAME( \
    riscvP riscv, \
    Uns32 index, \
    Bool newValue \
)
typedef RISC_V_UPDATE_INTERRUPT_FN((*riscvUpdateInterruptFn));

typedef struct riscvModelCBS {
    riscvUpdateInterruptFn updateInterrupt;
} riscvModelCB;
```

#### **Description**

This interface function updates the value of the indexed interrupt, as seen in the in the mip CSR. For example, to raise the M-mode software interrupt, index would be 3 and newValue 1.

#### **Example**

```
inline static void updateStandardInterrupt(
    vmiosObjectP object,
    Uns32 index,
    Bool newValue
) {
    riscvP riscv = object->riscv;

    riscv->cb.updateInterrupt(riscv, index, newValue);
}
```

#### **Usage Context**

Leaf level only.

### 13.19 *Function updateDisable*

```
#define RISC_V_UPDATE_DISABLE_FN(_NAME) void _NAME( \  
    riscvP riscv, \  
    Uns64 disableMask \  
    ) \  
typedef RISC_V_UPDATE_DISABLE_FN(*riscvUpdateDisableFn); \  
 \  
typedef struct riscvModelCBS { \  
    riscvUpdateDisableFn    updateDisable; \  
} riscvModelCB;
```

#### **Description**

This interface function disables the indexed interrupt. For example, to disable the M-mode software interrupt, `index` would be 3 and `newValue` 1.

#### **Example**

```
inline static void disableStandardInterrupt( \  
    vmiosObjectP object, \  
    Uns32        index, \  
    Bool        newValue \  
    ) { \  
    riscvP riscv = object->riscv; \  
    riscv->cb.disableInterrupt(riscv, index, newValue); \  
}
```

#### **Usage Context**

Leaf level only.

### 13.20 *Function testInterrupt*

```
#define RISC_V_TEST_INTERRUPT_FN(_NAME) void _NAME(riscvP riscv)
typedef RISC_V_TEST_INTERRUPT_FN((*riscvTestInterruptFn));

typedef struct riscvModelCBS {
    riscvTestInterruptFn    testInterrupt;
} riscvModelCB;
```

#### **Description**

This interface function causes the base model to check for any pending interrupts. It should be called when an extension library has performed some action that might cause a pending interrupt to be activated (for example, pending it, or unmasking it when already pending).

#### **Example**

```
static void testInterrupt(riscvP riscv) {
    riscv->cb.testInterrupt(riscv);
}
```

#### **Usage Context**

Leaf level only.

### 13.21 *Function illegalInstruction*

```
#define RISC_V_ILLEGAL_INSTRUCTION_FN(_NAME) void _NAME(riscvP riscv)
typedef RISC_V_ILLEGAL_INSTRUCTION_FN(*riscvIllegalInstructionFn);

typedef struct riscvModelCBS {
    riscvIllegalInstructionFn illegalInstruction;
} riscvModelCB;
```

#### **Description**

This interface function causes the base model to take an Illegal Instruction trap immediately.

#### **Example**

```
static void illegalInstruction(riscvP riscv) {
    riscv->cb.illegalInstruction(riscv);
}
```

#### **Usage Context**

Leaf level only.

## 13.22 *Function illegalVerbose*

```
#define RISC_V_ILLEGAL_VERBOSE_FN(_NAME) void _NAME( \  
    riscvP      riscv, \  
    const char *reason \  
    ) \  
typedef RISC_V_ILLEGAL_VERBOSE_FN(*riscvIllegalVerboseFn); \  
 \  
typedef struct riscvModelCBS { \  
    riscvIllegalVerboseFn    illegalVerbose; \  
} riscvModelCB;
```

### Description

This interface function causes the base model to take an Illegal Instruction trap immediately, for a verbose reason indicated by the `reason` string.

### Example

```
static void illegalVerbose(riscvP riscv) { \  
    riscv->cb.illegalVerbose(riscv); \  
}
```

### Usage Context

Leaf level only.

### 13.23 *Function virtualInstruction*

```
#define RISC_V_ILLEGAL_INSTRUCTION_FN(_NAME) void _NAME(riscvP riscv)
typedef RISC_V_ILLEGAL_INSTRUCTION_FN(*riscvIllegalInstructionFn);

typedef struct riscvModelCBS {
    riscvIllegalInstructionFn virtualInstruction;
} riscvModelCB;
```

#### **Description**

This interface function causes the base model to take a Virtual Instruction trap immediately. It should be used only on processors that implement the Hypervisor extension.

#### **Example**

```
static void virtualInstruction(riscvP riscv) {
    riscv->cb.virtualInstruction(riscv);
}
```

#### **Usage Context**

Leaf level only.

### 13.24 *Function virtualVerbose*

```
#define RISC_V_ILLEGAL_VERBOSE_FN(_NAME) void _NAME( \  
    riscvP      riscv, \  
    const char *reason \  
    ) \  
typedef RISC_V_ILLEGAL_VERBOSE_FN(*riscvIllegalVerboseFn); \  
 \  
typedef struct riscvModelCBS { \  
    riscvIllegalVerboseFn    virtualVerbose; \  
} riscvModelCB;
```

#### **Description**

This interface function causes the base model to take a Virtual Instruction trap immediately, for a verbose reason indicated by the `reason` string. It should be used only on processors that implement the Hypervisor extension.

#### **Example**

```
static void virtualVerbose(riscvP riscv) { \  
    riscv->cb.virtualVerbose(riscv); \  
}
```

#### **Usage Context**

Leaf level only.



### 13.25 *Function takeException*

```
#define RISC_V_TAKE_EXCEPTION_FN(_NAME) void _NAME( \  
    riscvP      riscv, \  
    riscvException exception, \  
    Uns64      tval \  
    ) \  
typedef RISC_V_TAKE_EXCEPTION_FN((*riscvTakeExceptionFn)); \  
 \  
typedef struct riscvModelCBS { \  
    riscvTakeExceptionFn    takeException; \  
} riscvModelCB;
```

#### **Description**

This interface function causes the base model to take an exception immediately. The exception passed as the `exception` argument can be either a standard exception or any other index number corresponding to a custom exception. The `tval` argument provides a value that is reported in the corresponding `xtval` CSR.

#### **Example**

```
static void takeExcept24(riscvP riscv) { \  
    riscv->cb.takeException(riscv, EXT_E_EXCEPT24, 0); \  
}
```

#### **Usage Context**

Leaf level only.

### 13.26 *Function takeReset*

```
#define RISC_V_TAKE_RESET_FN(_NAME) void _NAME(riscvP riscv)
typedef RISC_V_TAKE_RESET_FN(*riscvTakeResetFn);

typedef struct riscvModelCBS {
    riscvTakeResetFn    takeReset;
} riscvModelCB;
```

#### **Description**

This interface function causes the base model to take a reset immediately.

#### **Example**

```
static void takeReset(riscvP riscv) {
    riscv->cb.takeReset(riscv);
}
```

#### **Usage Context**

Leaf level only.

### 13.27 *Function `fetchInstruction`*

```
#define RISC_V_FETCH_INSTRUCTION_FN(_NAME) Uns32 _NAME( \
    riscvP          riscv, \
    riscvAddr       thisPC, \
    riscvExtInstrInfoP info, \
    vmidDecodeTablePP tableP, \
    riscvExtInstrAttrsCP attrs, \
    Uns32          last, \
    Uns32          bits \
)
typedef RISC_V_FETCH_INSTRUCTION_FN(*riscvFetchInstructionFn);

typedef struct riscvModelCBS {
    riscvFetchInstructionFn  fetchInstruction;
} riscvModelCB;
```

#### Description

This interface function is used to decode a RISC-V instruction that conforms to a standard pattern, extracting information such as registers and constant values into the given `riscvExtMorphState` structure. See section 6 for a detailed example, and section 12 for more information about the available instruction patterns.

#### Example

```
static riscvExtIType decode(
    riscvP          riscv,
    vmiosObjectP    object,
    riscvAddr       thisPC,
    riscvExtInstrInfoP info
) {
    return riscv->cb.fetchInstruction(
        riscv, thisPC, info, &object->decode32, attrsArray32, EXT_IT_LAST, 32
    );
}
```

#### Usage Context

Leaf level only.

## 13.28 *Function `disassInstruction`*

```
#define RISC_V_DISASS_INSTRUCTION_FN(_NAME) const char *_NAME( \
    riscvP          riscv, \
    riscvExtInstrInfoP instrInfo, \
    vmiDisassAttrs  attrs \
)
typedef RISC_V_DISASS_INSTRUCTION_FN(*riscvDisassInstructionFn);

typedef struct riscvModelCBS {
    riscvDisassInstructionFn  disassInstruction;
} riscvModelCB;
```

### Description

This interface function is used to disassemble a RISC-V instruction using a standard format string, using information about registers and constant values previously extracted by interface function `fetchInstruction`. See section 6 for a detailed example, and section 12 for more information about the available instruction patterns.

### Example

```
static VMIO_DISASSEMBLE_FN(addInstructionsDisassemble) {
    riscvP          riscv = (riscvP)processor;
    const char      *result = 0;
    riscvExtInstrInfoP info;

    // action is only required if the instruction is implemented by this
    // extension
    if(decode(riscv, object, thisPC, &info) != EXT_IT_LAST) {
        result = riscv->cb.disassInstruction(riscv, &info, attrs);
    }

    return result;
}
```

### Usage Context

Leaf level only.

### 13.29 *Function instructionEnabled*

```
#define RISCV_INSTRUCTION_ENABLED_FN(_NAME) Bool _NAME( \  
    riscvP          riscv,          \  
    riscvArchitecture requiredVariant \  
    ) \  
typedef RISCV_INSTRUCTION_ENABLED_FN((*riscvInstructionEnabledFn)); \  
 \  
typedef struct riscvModelCBS { \  
    riscvInstructionEnabledFn instructionEnabled; \  
} riscvModelCB;
```

#### **Description**

*This function must be called at morph time (from within the extension object morphCB).*

This interface function is used to validate the legality of an instruction with respect to RISC-V feature letters (A-Z). The feature requirements for this instruction are passed as the `requiredVariant` argument; for example, if an instruction requires single-precision floating point to be enabled, the value `ISA_F` should be passed. Any required XLEN can also be specified; for example, `ISA_F|RV64` encodes a requirement for enabled floating point *and* XLEN of 64.

The function returns a Boolean indicating if the architectural constraint is satisfied. If it is not satisfied, the interface function emits code to cause an Illegal Instruction trap to be taken.

#### **Example**

```
static Bool instructionEnabled(riscvP riscv, riscvArchitecture riscvVariants) { \  
    return riscv->cb.instructionEnabled(riscv, riscvVariants); \  
}
```

#### **Usage Context**

Leaf level only.

### 13.30 *Function morphExternal*

```
#define RISC_V_MORPH_EXTERNAL_FN(_NAME) void _NAME( \
    riscvExtMorphStateP state, \
    const char *disableReason, \
    Bool *opaque \
);
typedef RISC_V_MORPH_EXTERNAL_FN((*riscvMorphExternalFn));

typedef struct riscvModelCBS {
    riscvMorphExternalFn morphExternal;
} riscvModelCB;
```

#### Description

*This function must be called at morph time (from within the extension object morphCB).*

This interface function is used to emit translated code for an instruction implemented by an extension object, assuming that the instruction details have been decoded into a structure of type `riscvExtMorphState` by a previous call to the `fetchInstruction` interface function. The `opaque` argument should be passed down from the `morphCB` parameter of the same name. If the `disableReason` argument is non-NULL, code to cause an Illegal Instruction trap will be emitted and this reason printed in verbose mode. If the `disableReason` argument is NULL, the function `state.attrs->morph` (defined by the extension object) will be called to emit translated code for the instruction.

See section 6 for a detailed example.

#### Example

```
static VMIOS_MORPH_FN(addInstructionsMorph) {
    riscvP riscv = (riscvP)processor;
    riscvExtMorphState state = {riscv:riscv, object:object};

    // decode instruction
    riscvExtIType type = decode(riscv, object, thisPC, &state.info);

    // action is only required if the instruction is implemented by this
    // extension
    if(type != EXT_IT_LAST) {
        // fill translation attributes
        state.attrs = &dispatchTable[type];

        // translate instruction
        riscv->cb.morphExternal(&state, 0, opaque);
    }

    // no callback function is required
    return 0;
}
```

#### Usage Context

Leaf level only.

### 13.31 *Function morphIllegal*

```
#define RISC_V_MORPH_ILLEGAL_FN(_NAME) void _NAME( \  
    riscvP    riscv, \  
    const char *reason \  
    ) \  
typedef RISC_V_MORPH_ILLEGAL_FN((*riscvMorphIllegalFn)); \  
 \  
typedef struct riscvModelCBS { \  
    riscvMorphIllegalFn    morphIllegal; \  
} riscvModelCB;
```

#### **Description**

*This function must be called at morph time (from within the extension object morphCB).*

This interface function is used to emit code to cause an Illegal Instruction exception, printing the given reason string in verbose mode.

#### **Example**

```
static void morphIllegal(riscvP riscv, const char *reason) { \  
    return riscv->cb.morphIllegal(riscv, reason); \  
}
```

#### **Usage Context**

Leaf level only.

### 13.32 *Function morphVirtual*

```
#define RISCVMORPH_ILLEGAL_FN(_NAME) void _NAME( \
    riscvP      riscv, \
    const char *reason \
)
typedef RISCVMORPH_ILLEGAL_FN((*riscvMorphIllegalFn));

typedef struct riscvModelCBS {
    riscvMorphIllegalFn      morphVirtual;
} riscvModelCB;
```

#### **Description**

*This function must be called at morph time (from within the extension object morphCB).*

This interface function is used to emit code to cause a Virtual Instruction exception, printing the given `reason` string in verbose mode. It should be used only on processors that implement the Hypervisor extension.

#### **Example**

```
static void morphVirtual(riscvP riscv, const char *reason) {
    return riscv->cb.morphVirtual(riscv, reason);
}
```

#### **Usage Context**

Leaf level only.



### 13.33 *Function getVMIReg*

```
#define RISC_V_GET_VMI_REG_FN(_NAME) vmiReg _NAME(riscvP riscv, riscvRegDesc r)
typedef RISC_V_GET_VMI_REG_FN((*riscvGetVMIRegFn));

typedef struct riscvModelCBS {
    riscvGetVMIRegFn    getVMIReg;
} riscvModelCB;
```

#### **Description**

*This function must be called at morph time (from within the extension object morphCB).*

This interface function is used to convert a RISC-V register description (of type `riscvRegDesc`) to a VMI register description. The RISC-V register description will typically be extracted from a structure of type `riscvExtMorphState` filled by a previous call to the `fetchInstruction` interface function.

The VMI register description can then be used to specify instruction functional details using the VMI Morph Time Function API. See section 6 for a detailed example.

#### **Example**

```
inline static vmiReg getVMIReg(riscvP riscv, riscvRegDesc r) {
    return riscv->cb.getVMIReg(riscv, r);
}
```

#### **Usage Context**

Leaf level only.

### 13.34 Function `getVMiRegFS`

```
#define RISC_V_GET_VMI_REG_FS_FN(_NAME) vmiReg _NAME( \
    riscvP      riscv, \
    riscvRegDesc r, \
    vmiReg      tmp    \
)
typedef RISC_V_GET_VMI_REG_FS_FN((*riscvGetVMiRegFSFn));

typedef struct riscvModelCBS {
    riscvGetVMiRegFSFn    getVMiRegFS;
} riscvModelCB;
```

#### Description

This function must be called at morph time (from within the extension object `morphCB`).

This interface function is used to convert a RISC-V register description (of type `riscvRegDesc`) to a VMI register description. The RISC-V register description will typically be extracted from a structure of type `riscvExtMorphState` filled by a previous call to the `fetchInstruction` interface function. The function must be used when the register being converted is potentially an FPR that is *narrower than FLEN* (for example, a single-precision source on a machine that supports double-precision). The function emits code to validate that the source value is correctly NaN-boxed, composing the resultant value in the `tmp` temporary, which is then returned. If the register does not require a NaN box test, a `vmiReg` object representing the register value in the processor structure is returned.

The VMI register description can then be used to specify instruction functional details using the VMI Morph Time Function API.

#### Example

```
inline static vmiReg getVMiRegFS(riscvP riscv, riscvRegDesc r, vmiReg tmp) {
    return riscv->cb.getVMiRegFS(riscv, r, tmp);
}
```

#### Usage Context

Leaf level only.

### 13.35 *Function writeRegSize*

```
#define RISC_V_WRITE_REG_SIZE_FN(_NAME) void _NAME( \  
    riscvP      riscv,      \  
    riscvRegDesc r,        \  
    Uns32      srcBits,    \  
    Bool       signExtend \  
    ) \  
typedef RISC_V_WRITE_REG_SIZE_FN((*riscvWriteRegSizeFn)); \  
 \  
typedef struct riscvModelCBS { \  
    riscvWriteRegSizeFn  writeRegSize; \  
} riscvModelCB;
```

#### **Description**

*This function must be called at morph time (from within the extension object morphCB).*

When a result of size `srcBits` has been written into the VMI register equivalent to register `r`, this function is called to handle any required extension of that value from size `srcBits` to the architectural width of register `r`. Argument `signExtend` indicates whether the value is sign-extended (if `True`) or zero-extended (if `False`). See section 6 for a detailed example.

#### **Example**

```
inline static void writeRegSize( \  
    riscvP      riscv, \  
    riscvRegDesc r, \  
    Uns32      srcBits, \  
    Bool       signExtend \  
    ) { \  
    riscv->cb.writeRegSize(riscv, r, srcBits, signExtend); \  
}
```

#### **Usage Context**

Leaf level only.

### 13.36 *Function writeReg*

```
#define RISC_V_WRITE_REG_FN(_NAME) void _NAME( \
    riscvP      riscv, \
    riscvRegDesc r, \
    Bool        signExtend \
)
typedef RISC_V_WRITE_REG_FN(*riscvWriteRegFn);

typedef struct riscvModelCBS {
    riscvWriteRegFn      writeReg;
} riscvModelCB;
```

#### Description

*This function must be called at morph time (from within the extension object morphCB).*

When a result of the bit size encoded in the description of register `r` has been written into the VMI register equivalent to that register, this function is called to handle any required extension of that value from the encoded register size to the architectural width of register `r`. Argument `signExtend` indicates whether the value is sign-extended (if `True`) or zero-extended (if `False`).

This is equivalent to:

```
riscv->cb.writeRegSize(riscv, r, getRBits(r), signExtend);
```

#### Example

```
inline static void writeReg(riscvP riscv, riscvRegDesc r) {
    riscv->cb.writeReg(riscv, r, True);
}
```

#### Usage Context

Leaf level only.

### 13.37 *Function getFPFlagsMt*

```
#define RISC_V_GET_FP_FLAGS_MT_FN(_NAME) vmiReg _NAME(riscvP riscv)
typedef RISC_V_GET_FP_FLAGS_MT_FN(*riscvGetFPFlagsMtFn);

typedef struct riscvModelCBS {
    riscvGetFPFlagsMtFn    getFPFlagsMt;
} riscvModelCB;
```

#### **Description**

*This function must be called at morph time (from within the extension object morphCB).*

When an extension object implements instructions that update floating point flag state, this function must be used to obtain a `vmiReg` descriptor for the standard RISC-V `fflags` CSR. The returned `vmiReg` register should then be used as the `flags` argument of any floating point VMI primitives used to implement that instruction.

#### **Example**

```
inline static vmiReg riscvGetFPFlagsMT(riscvP riscv) {
    return riscv->cb.getFPFlagsMt(riscv);
}
```

#### **Usage Context**

Leaf level only.

### 13.38 *Function `getDataEndianMt`*

```
#define RISC_V_GET_DATA_ENDIAN_MT_FN(_NAME) memEndian _NAME(riscvP riscv)
typedef RISC_V_GET_DATA_ENDIAN_MT_FN(*riscvGetDataEndianMtFn);

typedef struct riscvModelCBS {
    riscvGetDataEndianMtFn    getDataEndianMt;
} riscvModelCB;
```

#### **Description**

*This function must be called at morph time (from within the extension object `morphCB`).*

This interface function returns the active endianness for loads and stores when translating an instruction. The function ensures that any JIT-compiled code generated is used only when the endianness matches, meaning that the returned endianness can be assumed to be constant by the JIT translation routine in the extension object.

#### **Example**

```
inline static memEndian getDataEndian(riscvP riscv) {
    return riscv->cb.getDataEndianMt(riscv);
}
```

#### **Usage Context**

Leaf level only.

### 13.39 *Function checkLegalRMMt*

```
#define RISCV_CHECK_LEGAL_RM_MT_FN(_NAME) Bool _NAME( \  
    riscvP      riscv, \  
    riscvRMDesc rm    \  
    ) \  
typedef RISCV_CHECK_LEGAL_RM_MT_FN(*riscvCheckLegalRMMtFn); \  
 \  
typedef struct riscvModelCBS { \  
    riscvCheckLegalRMMtFn    checkLegalRMMt; \  
} riscvModelCB;
```

#### Description

*This function must be called at morph time (from within the extension object morphCB).*

Given a rounding mode of type `riscvRMDesc` (typically extracted from a structure of type `riscvExtMorphState` filled by a previous call to the `fetchInstruction` interface function), this interface function inserts code to check legality of the rounding mode and take an Illegal Instruction trap if it is invalid. The Boolean return code indicates whether the rounding mode should be assumed to be valid by the calling extension object function.

#### Example

```
inline static Bool emitCheckLegalRM(riscvP riscv, riscvRMDesc rm) { \  
    return riscv->cb.checkLegalRMMt(riscv, rm); \  
}
```

#### Usage Context

Leaf level only.

### 13.40 *Function morphTrapTVM*

```
#define RISC_V_MORPH_TRAP_TVM_FN(_NAME) void _NAME(riscvP riscv)
typedef RISC_V_MORPH_TRAP_TVM_FN(*riscvMorphTrapTVMFn);

typedef struct riscvModelCBS {
    riscvMorphTrapTVMFn    morphTrapTVM;
} riscvModelCB;
```

#### **Description**

*This function must be called at morph time (from within the extension object morphCB).*

This interface function is used to emit a trap when `mstatus.TVM=1` when executing in Supervisor mode.

#### **Example**

```
static void morphTrapTVM(riscvP riscv) {
    return riscv->cb.morphTrapTVM(riscv);
}
```

#### **Usage Context**

Leaf level only.



### 13.41 Function *morphVOp*

```
#define RISC_V_MORPH_VOP_FN(_NAME) void _NAME( \
    riscvP          riscv,          \
    Uns64           thisPC,         \
    riscvRegDesc    r0,             \
    riscvRegDesc    r1,             \
    riscvRegDesc    r2,             \
    riscvRegDesc    mask,          \
    riscvVShape     shape,          \
    riscvVExternalFn externalCB,    \
    void            *userData       \
)
typedef RISC_V_MORPH_VOP_FN((*riscvMorphVOpFn));

typedef struct riscvModelCBS {
    riscvMorphVOpFn    morphVOp;
} riscvModelCB;
```

#### Description

This function must be called at morph time (from within the extension object *morphCB*).

This interface function is used to create a *vector extension custom operation* in an extension object, according to the standard patterns implemented in the base model. The arguments are as follows:

1. *riscv*: the current processor.
2. *thisPC*: the current instruction address.
3. *r0*: first register operand.
4. *r1*: second register operand.
5. *r3*: third register operand.
6. *mask*: vector mask.
7. *shape*: vector operation shape (of type *riscvVShape*, described below).
8. *externalCB*: function of type *riscvVExternalFn*, implementing one operation.
9. *userData*: extension-specific context information.

Given an instruction previously decoded into a structure of type *riscvExtMorphState* filled by a previous call to the *fetchInstruction* interface function, most of these parameters can be extracted directly from fields in this structure, as shown in the example at the end of this section.

The *shape* argument describes the pattern of vector operation, as follows:

```
//
// Vector shape description. Each name is composed of:
// - prefix (RVVW)
// - two or three operand descriptors, each three letters:
//   1. argument type (V=vector, S=scalar, P=predicate);
//   2. VLMUL multiplier (1, 2 or 4);
//   3. element type (I=integer, F=float)
// - an optional generic suffix
//
typedef enum riscvVShapeE {
    // INTEGER ARGUMENTS
    RVVW_V1I_V1I_V1I, // SEW = SEW op SEW
```

```

RVVW_V1I_V1I_V1I_LD, // vector load operations
RVVW_V1I_V1I_V1I_ST, // vector store operations
RVVW_V1I_V1I_V1I_SAT, // saturating result
RVVW_V1I_V1I_V1I_VXRM, // uses vxrm
RVVW_V1I_V1I_V1I_SEW8, // uses SEW8
RVVW_V1I_S1I_V1I, // src1 is scalar
RVVW_S1I_V1I_V1I, // Vd is scalar
RVVW_P1I_V1I_V1I, // Vd is predicate
RVVW_S1I_V1I_S1I, // Vd and src2 are scalar
RVVW_V1I_V1I_V1I_CIN, // mask is carry-in (VADC etc)
RVVW_P1I_V1I_V1I_CIN, // Vd is predicate, mask is carry-in (VMADC etc)
RVVW_S2I_V1I_S2I, // 2*SEW = SEW op 2*SEW, Vd and src2 are scalar
RVVW_V1I_V2I_V1I, // SEW = 2*SEW op SEW
RVVW_V1I_V2I_V1I_SAT, // SEW = 2*SEW op SEW, saturating result
RVVW_V2I_V1I_V1I_IW, // 2*SEW = SEW op SEW, implicit widening
RVVW_V2I_V1I_V1I, // 2*SEW = SEW op SEW
RVVW_V2I_V1I_V1I_SAT, // 2*SEW = SEW op SEW, saturating result
RVVW_V4I_V1I_V1I, // 4*SEW = SEW op SEW
RVVW_V2I_V2I_V1I, // 2*SEW = 2*SEW op SEW
RVVW_V1I_V2I_FN, // SEW = SEW/FN

// FLOATING POINT ARGUMENTS
RVVW_V1F_V1F_V1F, // SEW = SEW op SEW
RVVW_V1F_S1F_V1F, // src1 is scalar
RVVW_S1F_V1I_V1I, // Vd is scalar
RVVW_P1I_V1F_V1F, // Vd is predicate
RVVW_S1F_V1F_S1F, // Vd and src2 are scalar
RVVW_S2F_V1F_S2F, // 2*SEW = SEW op 2*SEW, Vd and src2 are scalar
RVVW_V1F_V2F_V1F_IW, // SEW = 2*SEW op SEW, implicit widening
RVVW_V2F_V1F_V1F_IW, // 2*SEW = SEW op SEW, implicit widening
RVVW_V2F_V1F_V1F, // 2*SEW = SEW op SEW
RVVW_V2F_V2F_V1F, // 2*SEW = 2*SEW op SEW

// CONVERSIONS
RVVW_V1F_V1I, // SEW = SEW, Fd=Is
RVVW_V1I_V1F, // SEW = SEW, Id=Fs
RVVW_V2F_V1I, // 2*SEW = SEW, Fd=Is
RVVW_V2I_V1F, // 2*SEW = SEW, Id=Fs
RVVW_V1F_V2I_IW, // SEW = 2*SEW, Fd=Is, implicit widening
RVVW_V1I_V2F_IW, // SEW = 2*SEW, Id=Fs, implicit widening

// MASK ARGUMENTS
RVVW_P1I_P1I_P1I, // SEW = SEW op SEW
RVVW_V1I_P1I_P1I, // SEW = SEW op SEW

// SLIDING ARGUMENTS
RVVW_V1I_V1I_V1I_GR, // SEW, VRGATHER instructions
RVVW_V1I_V1I_V1I_UP, // SEW, VSLIDEUP instructions
RVVW_V1I_V1I_V1I_DN, // SEW, VSLIDEDOWN instructions
RVVW_V1I_V1I_V1I_CMP, // SEW, VCOMPRESS instruction

RVVW_LAST // KEEP LAST: for sizing
} riscvVShape;

```

These shapes correspond to all standard vector instructions. Most extension instructions are likely to be of types `RVVW_V1I_V1I_V1I` or `RVVW_V1F_V1F_V1F` (i.e. same-width binary integer and floating point operations, respectively).

The `morphVOp` interface function automatically handles standard Vector Extension features such as element iteration, masking, and emitting Illegal Instruction exceptions if the Vector Extension is disabled. The extension object is required only to supply a callback function of type `riscvVExternalFn` to implement the vector operation for a single element:

```

#define RISCV_VEXTERNAL_FN(_NAME) void _NAME( \
    riscvP   riscv, \
    void     *userData, \
    vmiReg   *r, \
    Uns32    SEW \
)
typedef RISCV_VEXTERNAL_FN(*riscvVExternalFn);

```

The callback function takes four arguments:

1. The current processor;
2. The `userData` pointer originally passed as the final argument to the `morphVOp` interface function;
3. An array of `vmiReg` objects for each register argument to the element operation;
4. The current selected element width (SEW).

### Example

The following code snippet shows how a simple custom widening instruction that extends a `BFLOAT16` value to a single-precision value might be implemented:

```

//
// Per-element callback for VFWCVT.S.BF16
//
static RISCV_VEXTERNAL_FN(emitVFWCVT_S_BF16CB) {
    vmiReg r0L = r[0];
    vmiReg r0H = VMI_REG_DELTA(r[0], 2);

    // move result to high part of register
    vmimtMoveRR(16, r0H, r[1]);
    vmimtMoveRC(16, r0L, 0);
}

//
// Emit VFWCVT.S.BF16
//
static void emitVFWCVT_S_BF16(
    riscvP   riscv,
    riscvExtMorphStateP state
) {
    riscv->cb.morphVOp(
        riscv,
        state->thisPC,
        state->r[0],
        state->r[1],
        state->r[2],
        state->mask,
        RVVW_V2F_V1I,
        emitVFWCVT_S_BF16CB,
        state
    );
}

```

### Usage Context

Leaf level only.

### 13.42 Function `newCSR`

```
#define RISC_V_NEW_CSR_FN(_NAME) void _NAME( \
    riscvCSRAttrsP attrs,          \
    riscvCSRAttrsCP src,           \
    riscvP riscv,                  \
    vmiosObjectP object            \
)
typedef RISC_V_NEW_CSR_FN((*riscvNewCSRFn));

typedef struct riscvModelCBS {
    riscvNewCSRFn newCSR;
} riscvModelCB;
```

#### Description

Given a template CSR, this function registers that CSR with the base model. It should always be called from the extension object constructor. See section 7 for a detailed description and extended example.

#### Example

```
static void csrInit(vmiosObjectP object) {

    riscvP riscv = object->riscv;
    extCSRId id;

    for(id=0; id<XCSR_ID(LAST); id++) {

        extCSRAttrsCP src = &csrs[id];
        riscvCSRAttrs *dst = &object->csrs[id];

        riscv->cb.newCSR(dst, &src->baseAttrs, riscv, object);
    }
}
```

#### Usage Context

Leaf level only.

### 13.43 *Function updateLdStDomain*

```
#define RISC_V_UPDATE_LD_ST_DOMAIN_FN(_NAME) void _NAME(riscvP riscv)
typedef RISC_V_UPDATE_LD_ST_DOMAIN_FN(*riscvUpdateLdStDomainFn);

typedef struct riscvModelCBS {
    riscvUpdateLdStDomainFn  updateLdStDomain;
} riscvModelCB;
```

#### **Description**

Some CSR settings affect the access mode for load and store instructions in Machine mode. For example, `mstatus.MPRV=1` can cause load and store instructions to be executed in Supervisor or User modes instead of Machine mode.

This feature of the RISC-V architecture is implemented in the model by modifying the *memory domain* to which loads and stores are routed. This function causes the current memory domain to be refreshed so that it is valid for the current CSR settings. It should be called after any CSR update that affects Machine mode loads and stores in this way.

#### **Example**

```
static void updateLdStDomain(riscvP riscv) {
    return riscv->cb.updateLdStDomain(riscv);
}
```

#### **Usage Context**

Leaf level only.

### 13.44 Function `newTLBEntry`

```
#define RISC_V_NEW_TLB_ENTRY_FN(_NAME) void _NAME( \
    riscvP      riscv, \
    riscvTLBId  tlbId, \
    riscvExtVMMapping mapping \
)
typedef RISC_V_NEW_TLB_ENTRY_FN(*riscvNewTLBEntryFn);

typedef struct riscvModelCBS {
    riscvNewTLBEntryFn    newTLBEntry;
} riscvModelCB;
```

#### Description

One RISC-V implementation choice is to implement virtual memory TLB updates using a trap to a Machine mode handler. In this case, memory mappings will typically be modified by writes to custom CSRs, or a similar mechanism.

This function is used to create a new mapping using a custom instruction. The TLB that is affected is specified by the `tlbId` enumeration:

```
typedef enum riscvTLBIdE {
    RISC_V_TLB_HS,      // HS TLB
    RISC_V_TLB_VS1,    // VS stage 1 virtual TLB
    RISC_V_TLB_VS2,    // VS stage 2 virtual TLB
} riscvTLBId
```

(`RISC_V_TLB_VS1` and `RISC_V_TLB_VS2` TLBs should only be used for processors that implement the Hypervisor extension.)

The mapping to create is described by the `mapping` parameter, which is of type `riscvExtVMMapping`:

```
typedef struct riscvExtVMMappingS {
    Uns64  lowVA;      // low VA
    Uns64  highVA;     // high VA
    Uns64  PA;        // low PA
    Uns16  entryId : 16; // custom unique identifier
    memPriv priv    : 8; // access privileges
    Bool   V        : 1; // valid bit
    Bool   U        : 1; // User-mode access
    Bool   G        : 1; // global entry
    Bool   A        : 1; // accessed
    Bool   D        : 1; // dirty
} riscvExtVMMapping
```

Most entries correspond to fields of the same name in the RISC-V Privileged Architecture description. The `entryId` field is for application use, to identify a TLB entry uniquely in a TLB.

#### Example

```
static void installTLBEntry(riscvP riscv, custTLBEntryP entry) {
```

```
if(!entry->installed) {  
    riscv->cb.newTLBEntry(  
        riscv, entry->xatp, entry->mapping  
    );  
    entry->installed = True;  
}
```

### Usage Context

Leaf level only.

### 13.45 Function *freeTLBEntry*

```
#define RISCV_FREE_TLB_ENTRY_FN(_NAME) void _NAME( \
    riscvP    riscv, \
    riscvTLBId tlbId, \
    Uns16     entryId \
)
typedef RISCV_FREE_TLB_ENTRY_FN((*riscvFreeTLBEntryFn));

typedef struct riscvModelCBS {
    riscvFreeTLBEntryFn    freeTLBEntry;
} riscvModelCB;
```

#### Description

One RISC-V implementation choice is to implement virtual memory TLB updates using a trap to a Machine mode handler. In this case, memory mappings will typically be modified by writes to custom CSRs, or a similar mechanism.

This function is used to invalidate a mapping using a custom instruction. The TLB that is affected is specified by the `tlbId` enumeration:

```
typedef enum riscvTLBIdE {
    RISCV_TLB_HS,      // HS TLB
    RISCV_TLB_VS1,    // VS stage 1 virtual TLB
    RISCV_TLB_VS2,    // VS stage 2 virtual TLB
} riscvTLBId
```

(`RISCV_TLB_VS1` and `RISCV_TLB_VS2` TLBs should only be used for processors that implement the Hypervisor extension.)

The mapping to invalidate is described by the `entryId` parameter, which is an index number corresponding to the field of the same name when the entry was created – see section 13.44 for more information.

#### Example

```
static void uninstallTLBEntry(riscvP riscv, custTLBEntryP entry) {
    if(entry->installed) {
        riscv->cb.freeTLBEntry(
            riscv, entry->xatp, entry->mapping.entryId
        );
        VMI_ASSERT(!entry->installed, "TLB entry not uninstalled");
    }
}
```

#### Usage Context

Leaf level only.



## 14 Appendix: Extension Object Interface Functions

This appendix describes *interface functions* implemented by the *extension object* that allow the extension object to provide information to modify the behavior of the base model. All such interface functions are held in a structure of type `riscvExtCBS`, defined in file `riscvModelCallbacks.h` in the base model:

```
typedef struct riscvExtCBS {
    // link pointer and id (maintained by base model)
    riscvExtCBP      next;
    Uns32           id;

    // handle back to client data
    void            *clientData;

    // exception modification
    riscvRdWrSnapFn rdSnapCB;
    riscvRdWrSnapFn wrSnapCB;

    // exception actions
    riscvSuppressMemExceptFn suppressMemExcept;
    riscvCustomNMIFn        customNMI;
    riscvTrapNotifierFn     trapNotifier;
    riscvTrapNotifierFn     trapPreNotifier;
    riscvTrapNotifierFn     ERETNotifier;
    riscvResetNotifierFn    resetNotifier;
    riscvFirstExceptionFn   firstException;
    riscvGetInterruptPriFn  getInterruptPri;

    // halt/restart actions
    riscvHRNotifierFn       haltRestartNotifier;
    riscvLRSCAbortFn       LRSCAbortFn;

    // code generation actions
    riscvDerivedMorphFn     preMorph;
    riscvDerivedMorphFn     postMorph;

    // transaction support actions
    riscvIASSwitchFn       switchCB;
    riscvTLoadFn           tLoad;
    riscvTStoreFn          tStore;

    // PMA check actions
    riscvPMAEnableFn       PMAEnable;
    riscvPMACheckFn        PMACheck;

    // virtual memory actions
    riscvVMTrapFn          VMTrap;
    riscvSetDomainNotifierFn setDomainNotifier;
    riscvFreeEntryNotifierFn freeEntryNotifier;

    // documentation
    riscvRestrictionsFn    restrictionsCB;
} riscvExtCBtype;
```

A structure of this type should be defined within the `vmiosObject` structure of the extension. Fields within the structure can either be initialized in the extension object constructor (to modify the standard base model behavior) or left as `NULL` (to use base model behavior unchanged).

Fields `next` and `id` are used by the base model to chain together multiple extensions to a single processor and should not be directly modified by the extension object. Field `clientData` should be initialized with the `vmiosObject` pointer of the extension object, as shown in all examples described in this document. Other fields may be modified by the extension object constructor and are described in following subsections.

## 14.1 Function `rdSnapCB`

```
#define RISCV_RD_WR_SNAP_FN(_NAME) Uns32 _NAME( \
    riscvP riscv, \
    Addr  address, \
    Uns32 bytes \
)
typedef RISCV_RD_WR_SNAP_FN((*riscvRdWrSnapFn));

typedef struct riscvExtCBS {
    riscvRdWrSnapFn    rdSnapCB;
} riscvExtCB;
```

### Description

This interface function is called from the base model when an unaligned *read* access is detected. The access is of size `bytes` at address `address`. The function can either allow the unaligned access to proceed normally (perhaps with data rotation) or return a code indicating that an exception should be taken. The required behavior is indicated by the `Uns32` return code, whose value is constructed using the `MEM_SNAP` macro in `vmiTypes.h`. In practice, two return codes are likely to be required:

`MEM_SNAP(1, 0)`: this indicates the unaligned access should proceed.

`MEM_SNAP(0, 0)`: this indicates the unaligned access causes an exception.

### Example

This example shows how unaligned accesses within a 64-byte cache line can be permitted, while unaligned accesses that straddle lines cause an exception:

```
//
// Get line index for address
//
inline static Uns32 getLine(Uns32 address) {
    return address/64;
}

//
// Unaligned accesses are allowed only within cache lines
//
static RISCV_RD_WR_SNAP_FN(snapCB) {

    Uns32 snap = MEM_SNAP(1, 0);

    if(getLine(address) != getLine(address+bytes-1)) {
        snap = MEM_SNAP(0, 0);
    }

    return snap;
}

static VMIOS_CONSTRUCTOR_FN(extConstructor) {
    . . . lines omitted . . .
}
```

```
object->extCB.rdsnapCB = snapCB;  
... lines omitted ...  
}
```

## 14.2 Function *wrSnapCB*

```
#define RISC_V_RD_WR_SNAP_FN(_NAME) Uns32 _NAME( \
    riscvP riscv, \
    Addr  address, \
    Uns32 bytes \
)
typedef RISC_V_RD_WR_SNAP_FN((*riscvRdWrSnapFn));

typedef struct riscvExtCBS {
    riscvRdWrSnapFn    wrSnapCB;
} riscvExtCB;
```

### Description

This interface function is called from the base model when an unaligned *write* access is detected. The access is of size *bytes* at address *address*. The function can either allow the unaligned access to proceed normally (perhaps with data rotation) or return a code indicating that an exception should be taken. The required behavior is indicated by the *Uns32* return code, whose value is constructed using the `MEM_SNAP` macro in `vmiTypes.h`. In practice, two return codes are likely to be required:

`MEM_SNAP(1, 0)`: this indicates the unaligned access should proceed.

`MEM_SNAP(0, 0)`: this indicates the unaligned access causes an exception.

### Example

This example shows how unaligned accesses within a 64-byte cache line can be permitted, while unaligned accesses that straddle lines cause an exception:

```
//
// Get line index for address
//
inline static Uns32 getLine(Uns32 address) {
    return address/64;
}

//
// Unaligned accesses are allowed only within cache lines
//
static RISC_V_RD_WR_SNAP_FN(snapCB) {

    Uns32 snap = MEM_SNAP(1, 0);

    if(getLine(address) != getLine(address+bytes-1)) {
        snap = MEM_SNAP(0, 0);
    }

    return snap;
}

static VMIO_CONSTRUCTOR_FN(extConstructor) {
    . . . lines omitted . . .
    object->extCB.wrSnapCB = snapCB;
    . . . lines omitted . . .
}
```

### 14.3 *Function suppressMemExcept*

```
#define RISCV_SUPPRESS_MEM_EXCEPT_FN(_NAME) Bool _NAME( \
    riscvP          riscv, \
    riscvException exception, \
    void            *clientData \
)
typedef RISCV_SUPPRESS_MEM_EXCEPT_FN(*riscvSuppressMemExceptFn);

typedef struct riscvExtCBS {
    riscvSuppressMemExceptFn  suppressMemExcept;
} riscvExtCB;
```

#### Description

This interface function is called from the base model when a memory exception is about to be taken. It gives the extension library an opportunity to suppress that exception if required. Argument `exception` specifies the exception that is about to be taken.

#### Example

This example shows how memory exceptions can be suppressed, and a sticky bit set instead, in a custom extension CSR `mecsr`:

```
static RISCV_SUPPRESS_MEM_EXCEPT_FN(suppressMemExcept) {

    vmiosObjectP object = clientData;
    Bool          suppress = False;

    switch(exception) {

        case riscv_E_LoadAddressMisaligned:
        case riscv_E_LoadAccessFault:
        case riscv_E_LoadPageFault:
            if(RD_XCSR_FIELD(object, mecsr, REDIS)) {
                suppress = True;
                WR_XCSR_FIELD(object, mecsr, RES, 1);
            }
            break;

        case riscv_E_StoreAMOAddressMisaligned:
        case riscv_E_StoreAMOAccessFault:
        case riscv_E_StoreAMOPageFault:
            if(RD_XCSR_FIELD(object, mecsr, WEDIS)) {
                suppress = True;
                WR_XCSR_FIELD(object, mecsr, WES, 1);
            }
            break;

        default:
            break;
    }

    return suppress;
}

static VMIOS_CONSTRUCTOR_FN(extConstructor) {

    . . . lines omitted . . .
    object->extCB.suppressMemExcept = suppressMemExcept;
    . . . lines omitted . . .
}
```

## 14.4 *Function customNMI*

```
#define RISC_V_CUSTOM_NMI_FN(_NAME) Bool _NAME( \  
    riscvP riscv, \  
    void *clientData \  
    ) \  
typedef RISC_V_CUSTOM_NMI_FN((*riscvCustomNMIFn)); \  
 \  
typedef struct riscvExtCBS { \  
    riscvCustomNMIFn      customNMI; \  
} riscvExtCB;
```

### Description

This interface function is called from the base model when an NMI is to be taken, allowing the extension to define custom behavior for that exception. The return code indicates whether special NMI behavior has been performed; if `False`, the base model will handle the NMI in the normal way.

### Example

This example shows how an extension could handle an NMI exception as a normal trap with custom cause:

```
static RISC_V_CUSTOM_NMI_FN(customNMI) { \  
    riscv->cb.takeException(riscv, riscv_E_Interrupt+EXT_NMI, 0); \  
    return True; \  
} \  
 \  
static VMIO_CONSTRUCTOR_FN(extConstructor) { \  
    . . . lines omitted . . . \  
    object->extCB.customNMI = customNMI; \  
    . . . lines omitted . . . \  
}
```

## 14.5 Function *trapNotifier*

```
#define RISC_V_TRAP_NOTIFIER_FN(_NAME) void _NAME( \
    riscvP    riscv, \
    riscvMode mode, \
    void      *clientData \
)
typedef RISC_V_TRAP_NOTIFIER_FN(*riscvTrapNotifierFn);

typedef struct riscvExtCBS {
    riscvTrapNotifierFn    trapNotifier;
} riscvExtCB;
```

### Description

This interface function is called from the base model when a trap is taken. It gives the extension library the opportunity to modify trap behavior (perhaps by recording extra information about certain trap types).

### Example

This example shows how an extension could record extra data in field `subCause` of a custom CSR `mcustcause` when a custom interrupt `EXT_INT1` is taken:

```
static RISC_V_TRAP_NOTIFIER_FN(takeTrap) {
    vmiosObjectP object = clientData;

    if(
        (mode==RISC_V_MODE_MACHINE) &&
        RD_CSR_FIELD(riscv, mcause, Interrupt) &&
        (RD_CSR_FIELD(riscv, mcause, ExceptionCode)==EXT_INT1)
    ) {
        WR_XCSR_FIELD(object, mcustcause, subCause, object->subCause);
    }
}

static VMIOS_CONSTRUCTOR_FN(extConstructor) {
    . . . lines omitted . . .
    object->extCB.trapNotifier = takeTrap;
    . . . lines omitted . . .
}
```

## 14.6 *preTrapFunction trapPreNotifier*

```
#define RISC_V_TRAP_NOTIFIER_FN(_NAME) void _NAME( \
    riscvP riscv, \
    riscvMode mode, \
    void *clientData \
)
typedef RISC_V_TRAP_NOTIFIER_FN(*riscvTrapNotifierFn);

typedef struct riscvExtCBS {
    riscvTrapNotifierFn trapPreNotifier;
} riscvExtCB;
```

### Description

This interface function is called from the base model when a trap is about to be taken. It gives the extension library the opportunity to save current model state before it is modified in the process of taking the trap.

### Example

This example shows how an extension could save the current value of `mstatus.MPP` in a custom CSR field `mcuststatus.SMPP` before a custom interrupt `EXT_INT1` is taken. This field is modified as part of the standard process of taking a trap.

```
static RISC_V_TRAP_NOTIFIER_FN(preTrap) {
    vmiosObjectP object = clientData;

    if(
        RD_CSR_FIELD(riscv, mcause, Interrupt) &&
        (RD_CSR_FIELD(riscv, mcause, ExceptionCode)==EXT_INT1)
    ) {
        Uns32 MPP = RD_CSR_FIELD_M(riscv, mstatus, MPP);
        WR_XCSR_FIELD(object, mcuststatus, SMPP, MPP);
    }
}

static VMIOS_CONSTRUCTOR_FN(extConstructor) {
    . . . lines omitted . . .
    object->extCB.trapPreNotifier = preTrap;
    . . . lines omitted . . .
}
```



## 14.7 Function *ERETNotifier*

```
#define RISC_V_TRAP_NOTIFIER_FN(_NAME) void _NAME( \
    riscvP    riscv, \
    riscvMode mode, \
    void      *clientData \
)
typedef RISC_V_TRAP_NOTIFIER_FN(*riscvTrapNotifierFn);

typedef struct riscvExtCBS {
    riscvTrapNotifierFn    ERETNotifier;
} riscvExtCB;
```

### Description

This interface function is called from the base model when a return from exception instruction is executed (MRET, SRET or URET). It gives the extension library the opportunity to modify exception return behavior (perhaps by restoring extra custom information).

### Example

This example shows how the `andes.ovpworld.org` model uses this function to restore custom fields in the `mxstatus` CSR when returning to Machine mode:

```
static RISC_V_TRAP_NOTIFIER_FN(ERETNotifier) {
    vmiosObjectP object = clientData;

    if(mode==RISC_V_MODE_MACHINE) {

        // restore mxstatus fields when MRET is executed
        COPY_FIELD(object, mxstatus, PFT_EN, PPFT_EN);
        COPY_FIELD(object, mxstatus, IME, PIME);
        COPY_FIELD(object, mxstatus, DME, PDME);
    }

    // refresh all counter objects
    refreshCounters(object);
}

void andesCSRInit(vmiosObjectP object) {
    . . . lines omitted . . .
    object->extCB.ERETNotifier = ERETNotifier;
    . . . lines omitted . . .
}
```

## 14.8 *Function resetNotifier*

```
#define RISC_V_RESET_NOTIFIER_FN(_NAME) void _NAME( \
    riscvP riscv, \
    void *clientData \
)
typedef RISC_V_RESET_NOTIFIER_FN(*riscvResetNotifierFn);

typedef struct riscvExtCBS {
    riscvResetNotifierFn    resetNotifier;
} riscvExtCB;
```

### Description

This interface function is called from the base model when a reset is executed. It gives the extension library the opportunity to perform custom reset actions and also set standard CSR fields to implementation-defined values.

### Example

This example shows how both a custom CSR and a standard CSR field can be reset:

```
static RISC_V_RESET_NOTIFIER_FN(CSRReset) {
    vmiosObjectP object = clientData;

    // reset custom CSR
    WR_XCSR(object, custom1, 0);

    // reset standard CSR fields
    WR_CSR_FIELD(riscv, mstatus, TW, 1);
}

static VMIOS_CONSTRUCTOR_FN(extConstructor) {
    . . . lines omitted . . .
    object->extCB.resetNotifier = CSRReset;
    . . . lines omitted . . .
}
```

## 14.9 Function *firstException*

```
#define RISC_V_FIRST_EXCEPTION_FN(_NAME) vmiExceptionInfoCP _NAME( \
    riscvP riscv, \
    void *clientData \
)
typedef RISC_V_FIRST_EXCEPTION_FN((*riscvFirstExceptionFn));

typedef struct riscvExtCBS {
    riscvFirstExceptionFn    firstException;
} riscvExtCB;
```

### Description

This interface function returns the first member of a NULL-terminated list of custom exceptions implemented by an extension. The base model adds all exception descriptions in the list to the visible exceptions of the derived model.

### Example

This example shows addition of a custom exception (EXCEPT24) to the base model:

```
#define EXT_EXCEPTION(_NAME, _DESC) { \
    name:#_NAME, code:EXT_E_##_NAME, description:_DESC, \
}

static const vmiExceptionInfo exceptions[] = {
    EXT_EXCEPTION (EXCEPT24, "Custom Exception 24"),
    {0}
};

static RISC_V_FIRST_EXCEPTION_FN(firstException) {
    return exceptions;
}

static VMIOS_CONSTRUCTOR_FN(extConstructor) {
    . . . lines omitted . . .
    object->extCB.firstException = firstException;
    . . . lines omitted . . .
}
```

## 14.10 Function `getInterruptPri`

```
#define RISC_V_GET_INTERRUPT_PRI_FN(_NAME) riscvExceptionPriority _NAME( \
    riscvP riscv, \
    Uns32 intNum, \
    void *clientData \
)
typedef RISC_V_GET_INTERRUPT_PRI_FN(*riscvGetInterruptPriFn);

typedef struct riscvExtCBS {
    riscvGetInterruptPriFn    getInterruptPri;
} riscvExtCB;
```

### Description

This interface function returns the relative priority of an interrupt, or 0 if the default priority for that interrupt should be used. If non-zero, priorities are expressed relative to specified priorities of standard interrupts, as define by the `riscvExceptionPriority` enumeration.

### Example

This example shows a function returning priorities for two custom interrupts, `EXT_I_INT21` and `EXT_I_INT22`:

```
static RISC_V_GET_INTERRUPT_PRI_FN(getInterruptPriority) {
    riscvExceptionPriority result = 0;

    if(intNum==EXT_I_INT21) {
        result = riscv_E_LocalPriority;
    } else if(intNum==EXT_I_INT22) {
        result = riscv_E_LocalPriority+1;
    }

    return result;
}

static VMIOS_CONSTRUCTOR_FN(extConstructor) {
    . . . lines omitted . . .
    object->extCB.getInterruptPri = getInterruptPriority;
    . . . lines omitted . . .
}
```

## 14.11 Function *haltRestartNotifier*

```
#define RISC_V_HR_NOTIFIER_FN(_NAME) void _NAME( \
    riscvP riscv, \
    void *clientData \
)
typedef RISC_V_HR_NOTIFIER_FN((*riscvHRNotifierFn));

typedef struct riscvExtCBS {
    riscvHRNotifierFn    haltRestartNotifier;
} riscvExtCB;
```

### Description

This interface function is called when a processor transitions from running to halted state, or from halted to running state. It allows the extension object to perform any state changes required at this point (typically, to components like instruction counters).

### Example

This example shows how this function is used in the `andes.ovpworld.org` model. In this model, when the processor transitions between running and halted states, counters need to be started and stopped:

```
static void refreshCounter(andesCounterP counter) {

    riscvP      riscv = object->riscv;
    andesCounterMode cmode = ACM_INACTIVE;

    // get raw counter mode
    if(counter->TYPE) {
        // no action
    } else if(counter->SEL==1) {
        cmode = ACM_CY;
    } else if(counter->SEL==2) {
        cmode = riscv->disable ? ACM_INACTIVE : ACM_IR;
    }

    . . . counter state modified based in cmode here . . .
}

static void refreshCounters(vmiosObjectP object) {

    andesCounterID id;

    for(id=0; id<AT_LAST; id++) {

        andesCounterP counter = &object->counters[id];

        if(counter->vmi) {
            refreshCounter(counter);
        }
    }
}

static RISC_V_HR_NOTIFIER_FN(haltRestartNotifier) {
    refreshCounters(clientData);
}

static VMIO_CONSTRUCTOR_FN(constructor) {

    . . . lines omitted . . .
    object->extCB.haltRestartNotifier = haltRestartNotifier;
    . . . lines omitted . . .
}
```

## 14.12 Function *LRSCAbortFn*

```
#define RISCV_LRSC_ABORT_FN(_NAME) void _NAME( \  
    riscvP riscv, \  
    void *clientData \  
    ) \  
typedef RISCV_LRSC_ABORT_FN((*riscvLRSCAbortFn)); \  
 \  
typedef struct riscvExtCBS { \  
    riscvLRSCAbortFn LRSCAbortFn; \  
} riscvExtCB;
```

### Description

This interface function is called when an active LR/SC sequence is aborted (for example, because of a conflicting store by another processor). It allows the extension object to perform any state changes required at this point.

### Example

This example shows how this function could be used to restart a processor that is halted for an implementation-defined reason when another processor in a multicore simulation causes an active LR/SC sequence to be aborted:

```
RISCV_LRSC_ABORT_FN(custLRSCAbort) { \  
    if(riscv->disable & RVD_CUSTOMI) { \  
        riscv->cb.restart(riscv, RVD_CUSTOMI); \  
    } \  
} \  
 \  
static VMIOS_CONSTRUCTOR_FN(extConstructor) { \  
    . . . lines omitted . . . \  
    object->extCB.LRSCAbortFn = custLRSCAbort; \  
    . . . lines omitted . . . \  
}
```

### 14.13 Function *preMorph*

```
#define RISCV_DERIVED_MORPH_FN(_NAME) void _NAME(riscvP riscv, void *clientData)
typedef RISCV_DERIVED_MORPH_FN(*riscvDerivedMorphFn);

typedef struct riscvExtCBS {
    riscvDerivedMorphFn    preMorph;
} riscvExtCB;
```

#### Description

This interface function is called before an instruction is translated by the base model. It allows the extension object to emit code to perform an action that should be done before any standard instruction.

#### Example

This example shows how this function is used in the `andes.ovpworld.org` model. In this model, modeling hardware stack protection (HSP) requires that the current stack pointer is saved before each instruction executes so that it can be compared with the new value after the instruction:

```
void andesRecordSP(riscvP riscv, vmiosObjectP object) {
    if(!isHSPEnabledMT(riscv, object)) {
        // no action if feature is currently disabled
    } else {
        Uns32 bits = andesGetXlenArch(riscv);
        vmiReg oldSPReg = andesObjectReg(object, RISCV_OBJ_REG(oldSP));

        // move current stack pointer to a temporary
        vmimtMoveRR(bits, oldSPReg, RISCV_GPR(RV_REG_X_SP));
    }
}

RISCV_DERIVED_MORPH_FN(andesPreMorph) {
    vmiosObjectP object = clientData;

    if(RD_ACSR_FIELD(object, mmsc_cfg, HSP)) {
        andesRecordSP(riscv, object);
    }
}

static VMIOS_CONSTRUCTOR_FN(constructor) {
    . . . lines omitted . . .
    object->extCB.preMorph = andesPreMorph;
    . . . lines omitted . . .
}
```

## 14.14 Function *postMorph*

```
#define RISC_V_DERIVED_MORPH_FN(_NAME) void _NAME(riscvP riscv, void *clientData)
typedef RISC_V_DERIVED_MORPH_FN(*riscvDerivedMorphFn);

typedef struct riscvExtCBS {
    riscvDerivedMorphFn    postMorph;
} riscvExtCB;
```

### Description

This interface function is called after an instruction is translated by the base model. It allows the extension object to emit code to perform an action that should be done after any standard instruction.

### Example

This example shows how this function is used in the `andes.ovpworld.org` model. In this model, modeling hardware stack protection (HSP) requires that the current stack pointer is checked after each instruction executes to detect illegal stack pointer changes:

```
static void doHSPCheck(riscvP riscv, vmiosObjectP object) {
    . . . check SP against base and limit, maybe take exception . . .
}

void andesCheckHSP(riscvP riscv, vmiosObjectP object) {

    if(!isHSPEnabledMT(riscv, object)) {

        // no action if feature is currently disabled

    } else if(riscv->writtenXMask & (1<<RV_REG_X_SP)) {

        // check is required only if the instruction updates SP
        vmimtArgProcessor();
        vmimtArgNatAddress(object);
        vmimtCallAttrs((vmiCallFn)doHSPCheck, VMCA_NA);
    }
}

RISC_V_DERIVED_MORPH_FN(andesPostMorph) {

    vmiosObjectP object = clientData;

    if(RD_ACSR_FIELD(object, mmsc_cfg, HSP)) {
        andesCheckHSP(riscv, object);
    }
}

static VMIOS_CONSTRUCTOR_FN(constructor) {

    . . . lines omitted . . .
    object->extCB.postMorph    = andesPostMorph;
    . . . lines omitted . . .
}
```



## 14.15 Function *switchCB*

```
#define RISCV_IASSWITCH_FN(_NAME) void _NAME( \
    riscvP      riscv, \
    vmiIASRunState state, \
    void        *clientData \
)
typedef RISCV_IASSWITCH_FN((*riscvIASSwitchFn));

typedef struct riscvExtCBS {
    riscvIASSwitchFn      switchCB;
} riscvExtCB;
```

### Description

This interface function is called when a processor has been scheduled and is about to be run, or has been descheduled and is about to stop running, in a multiprocessor simulation. It allows the extension object to make any state changes required at those points.

### Example

This example shows how this function is used in the transactional memory extension (tmExtensions) in the `vendor.com` template model. In this model, modeling transactional memory requires that memory watchpoints are placed on all active cache lines when a processor is descheduled so that conflicting writes to those lines by another processor can be detected:

```
static void installCacheMonitor(vmiosObjectP object) {

    if (object->tmStatus != TM_OK) {

        // Ignore if not in transaction or there
        // has already been an abort event

    } else {

        memDomainP domain = object->physicalMem;
        cacheLineP this;
        Uns32      i;

        for(i = 0; i < object->numPending; i++) {

            this = &object->pending[i];

            Addr lo = getLineLowPA(this);
            Addr hi = getLineHighPA(this);

            // register for a callback if any current line is written
            if (!this->readCallbackInstalled) {
                vmirtAddWriteCallback(domain, 0, lo, hi, memoryConflict, object);
                this->readCallbackInstalled = True;
            }

            // register for a callback if any dirty line is read
            if (this->dirty && !this->writeCallbackInstalled) {
                vmirtAddReadCallback(domain, 0, lo, hi, memoryConflict, object);
                this->writeCallbackInstalled = True;
            }

        }

    }

}

static RISCV_IASSWITCH_FN(riscvSwitch) {
```

```
vmiosObjectP object = clientData;

if(state==RS_SUSPEND) {
    installCacheMonitor(object);
}

static VMIOS_CONSTRUCTOR_FN(tmConstructor) {
    . . . lines omitted . . .
    object->extCB.switchCB = riscvSwitch;
    . . . lines omitted . . .
}
```

## 14.16 Function *tLoad*

```
#define RISCV_TLOAD_FN(_NAME) void _NAME( \
    riscvP riscv, \
    void *buffer, \
    Addr VA, \
    Uns32 bytes, \
    void *clientData \
)
typedef RISCV_TLOAD_FN((*riscvTLoadFn));

typedef struct riscvExtCBS {
    riscvTLoadFn tLoad;
} riscvExtCB;
```

### Description

For an extension implementing transactional memory, this function is called whenever a load is performed and transactional memory mode is enabled (base model interface function `setTMode` in the base model has been called with `enable` of `True` – see section 13.9). The function should implement a transactional load, typically by modeling active cache lines.

### Example

This example shows how this function is used in the transactional memory extension (`tmExtensions`) in the `vendor.com` template model:

```
static RISCV_TLOAD_FN(riscvTLoad) {
    vmiosObjectP object = clientData;

    if (object->tmStatus != TM_OK) {
        // abort is pending
        doAbort(object);
    } else {
        Uns8 *buffer8 = buffer;
        Addr PA;
        Uns32 thisBytes;
        Uns32 i;

        while ((thisBytes = bytes) > 0) {
            if(mapVAToPA(object, VA, &PA)) {
                // get pointer to data in cache (loads cache line if required)
                Uns8 *data = getCacheAddress(object, PA, &thisBytes, False);

                if (!data) {
                    object->tmStatus |= TM_ABORT_OVERFLOW;
                    doAbort(object);
                } else {
                    for(i=0; i<thisBytes; i++) {
                        buffer8[i] = data[i];
                    }
                }
            }
            // reduce byte count by count of bytes on this line
        }
    }
}
```

```
        bytes -= thisBytes;
        VA    += thisBytes;
    }
}

static VMIOS_CONSTRUCTOR_FN(tmConstructor) {
    . . . lines omitted . . .
    object->extCB.tLoad      = riscvTLoad;
    . . . lines omitted . . .
}
```

## 14.17 Function *tStore*

```
#define RISC_V_TSTORE_FN(_NAME) void _NAME( \
    riscvP      riscv, \
    const void *buffer, \
    Addr        VA, \
    Uns32       bytes, \
    void        *clientData \
)
typedef RISC_V_TSTORE_FN(*riscvTStoreFn);

typedef struct riscvExtCBS {
    riscvTStoreFn    tStore;
} riscvExtCB;
```

### Description

For an extension implementing transactional memory, this function is called whenever a store is performed and transactional memory mode is enabled (base model interface function `setTMode` in the base model has been called with `enable` of `True` – see section 13.9). The function should implement a transactional store, typically by modeling active cache lines.

### Example

This example shows how this function is used in the transactional memory extension (`tmExtensions`) in the `vendor.com` template model:

```
static RISC_V_TSTORE_FN(riscvTStore) {
    vmiosObjectP object = clientData;

    if (object->tmStatus != TM_OK) {
        // abort is pending
        doAbort(object);
    } else {
        const Uns8 *buffer8 = buffer;
        Addr        PA;
        Uns32       thisBytes;
        Uns32       i;

        while ((thisBytes = bytes) > 0) {
            if(mapVAToPA(object, VA, &PA)) {
                // get pointer to data in cache (loads cache line if required)
                Uns8 *data = getCacheAddress(object, PA, &thisBytes, True);

                if (!data) {
                    object->tmStatus |= TM_ABORT_OVERFLOW;
                    doAbort(object);
                } else {
                    for(i=0; i<thisBytes; i++) {
                        data[i] = buffer8[i];
                    }
                }
            }
            // reduce byte count by count of bytes on this line
        }
    }
}
```

```
        bytes -= thisBytes;
        VA    += thisBytes;
    }
}

static VMIOS_CONSTRUCTOR_FN(tmConstructor) {
    . . . lines omitted . . .
    object->extCB.tStore    = riscvTStore;
    . . . lines omitted . . .
}
```

## 14.18 Function *PMAEnable*

```
#define RISCVPMA_ENABLE_FN(_NAME) Bool _NAME( \
    riscvP riscv, \
    void *clientData \
)
typedef RISCVPMA_ENABLE_FN((*riscvPMAEnableFn));

typedef struct riscvExtCBS {
    riscvPMAEnableFn PMAEnable;
} riscvExtCB;
```

### Description

For an extension implementing physical memory attributes (PMA), this interface function allows the extension to indicate whether PMA is enabled. If so, the `PMACheck` interface function is used to perform any memory mappings needed to model PMA for a given address range.

### Example

The `andes.ovpworld.org` model uses this function to specify whether PMA is enabled:

```
RISCVPMA_ENABLE_FN(andesPMAEnable) {
    vmiosObjectP object = clientData;

    return RD_XCSR_FIELD(object, mmio_cfg, DPMA);
}

static VMIO_CONSTRUCTOR_FN(constructor) {
    . . . lines omitted . . .
    object->extCB.PMAEnable = andesPMAEnable;
    . . . lines omitted . . .
}
```

## 14.19 *Function PMACheck*

```
#define RISCVPMA_CHECK_FN(_NAME) void _NAME( \  
    riscvP    riscv,          \  
    riscvMode mode,          \  
    memPriv   requiredPriv,  \  
    Uns64     lowPA,         \  
    Uns64     highPA,        \  
    void      *clientData    \  
    ) \  
typedef RISCVPMA_CHECK_FN(*riscvPMACheckFn); \  
 \  
typedef struct riscvExtCBS { \  
    riscvPMACheckFn    PMACheck; \  
} riscvExtCB;
```

### Description

For an extension implementing physical memory attributes (PMA), this interface function allows the extension to perform any memory mappings needed to model PMA for an address range. The function is called with a processor mode, required privileges, and an address range. If the address range can be accessed legally, permissions on the memory domain objects referenced in the `pmaDomains` field of the base model should be updated to enable the access; otherwise, privileges in these domains should be left unaltered.

### Example

The `andes.ovpworld.org` model uses this function to implement PMA using a custom feature similar to standard physical memory protection (PMP). The full example is too extensive to show here: please refer to the source of that model.



## 14.20 Function *VMTrap*

```
#define RISC_V_VM_TRAP_FN(_NAME) riscvException _NAME( \
    riscvP    riscv, \
    riscvTLBId id, \
    memPriv   requiredPriv, \
    Uns64     VA, \
    void      *clientData \
)
typedef RISC_V_VM_TRAP_FN(*riscvVMTrapFn);

typedef struct riscvExtCBS {
    riscvVMTrapFn    VMTrap;
} riscvExtCB;
```

### Description

One RISC-V implementation choice is to implement virtual memory TLB updates using a trap to a Machine mode handler. In this case, memory mappings will typically be modified by writes to custom CSRs, or a similar mechanism.

When virtual memory is enabled, this notifier is called to indicate an address lookup for an address for which there is currently no valid mapping in the TLB. It gives the extension the opportunity to initiate an implementation-specific trap to handle the address. The TLB that is affected is specified by the `tlbId` enumeration:

```
typedef enum riscvTLBIdE {
    RISC_V_TLB_HS,      // HS TLB
    RISC_V_TLB_VS1,    // VS stage 1 virtual TLB
    RISC_V_TLB_VS2,    // VS stage 2 virtual TLB
} riscvTLBId
```

(`RISC_V_TLB_VS1` and `RISC_V_TLB_VS2` TLBs are only used for processors that implement the Hypervisor extension.)

### Example

This example shows how a failing address lookup can cause one of six custom exceptions:

```
typedef enum custExceptionE {
    cust_InstTLBMiss    = 24,
    cust_LoadTLBMiss    = 25,
    cust_StoreTLBMiss   = 27,
    cust_GuestInstTLBMiss = 28,
    cust_GuestLoadTLBMiss = 29,
    cust_GuestStoreTLBMiss = 31,
} custException;

RISC_V_VM_TRAP_FN(custVMTrap) {
    vmiosObjectP    object = clientData;
    Bool            S2     = (id==RISC_V_TLB_VS2);
    riscvException  result = 0;

    if(!object->config.software_table_walk) {
        // use standard hardware page table walk
    } else if(requiredPriv & MEM_PRIV_R) {
        result = S2 ? cust_GuestLoadTLBMiss : cust_LoadTLBMiss;
    }
}
```

```
    } else if(requiredPriv & MEM_PRIV_W) {
        result = S2 ? cus _GuestStoreTLBMiss : cust_StoreTLBMiss;
    } else {
        result = S2 ? cust_GuestInstTLBMiss : cust_InstTLBMiss;
    }

    return result;
}

static VMIO_CONSTRUCTOR_FN(extConstructor) {
    . . . lines omitted . . .
    object->extCB.VMTrap = custVMTrap;
    . . . lines omitted . . .
}
```

## 14.21 Function `setDomainNotifier`

```
#define RISC_V_SET_DOMAIN_NOTIFIER_FN(_NAME) void _NAME( \
    riscvP      riscv, \
    memDomainPP domainP, \
    void        *clientData \
)
typedef RISC_V_SET_DOMAIN_NOTIFIER_FN((*riscvSetDomainNotifierFn));

typedef struct riscvExtCBS {
    riscvSetDomainNotifierFn setDomainNotifier;
} riscvExtCB;
```

### Description

Some CSR settings affect the access mode for load and store instructions in Machine mode. For example, `mstatus.MPRV=1` can cause load and store instructions to be executed in Supervisor or User modes instead of Machine mode. This feature of the RISC-V architecture is implemented in the model by modifying the *memory domain* to which loads and stores are routed.

This notifier is called after the base model has selected the active load/store memory domain for the current processor mode. It gives the extension the opportunity to modify the choice of domain if required. The choice of domain is indicated by updating the `domainP` by-reference argument, typically with one of the PMP domains or the guest page table walk domain from the base model.

### Example

This example shows how the load/store domain might be affected in a processor that implements TLB updates using a Machine mode trap. In this processor, when `cstatus.MTW=1`, all loads and stores should be performed with table walk privilege:

```
RISC_V_SET_DOMAIN_NOTIFIER_FN(custSetDomain) {

    riscvMode mode = getCurrentMode5(riscv);
    vmiosObjectP object = clientData;

    if(mode!=RISC_V_MODE_M) {
        // no action unless in Machine mode
    } else if(!RD_XCSR_FIELD(object, cstatus, MTW)) {
        // Machine Table Walk not enabled
    } else if(!RD_CSR_FIELD64(riscv, mstatus, GVA)) {
        *domainP = riscv->pmpDomains[RISC_V_MODE_S][False];
    } else if(!RD_CSR_FIELD_S(riscv, hgatp, MODE)) {
        *domainP = riscv->pmpDomains[RISC_V_MODE_S][False];
    } else {
        *domainP = riscv->guestPTWDomain;
    }
}

static VMIOS_CONSTRUCTOR_FN(extConstructor) {

    . . . lines omitted . . .
    object->extCB.setDomainNotifier = custSetDomain;
    . . . lines omitted . . .
}
```

## 14.22 *Function freeEntryNotifier*

```
#define RISC_V_FREE_ENTRY_NOTIFIER_FN(_NAME) void _NAME( \
    riscvP      riscv,          \
    riscvTLBId id,             \
    Uns16      entryId,        \
    void       *clientData     \
)
typedef RISC_V_FREE_ENTRY_NOTIFIER_FN(*riscvFreeEntryNotifierFn);

typedef struct riscvExtCBS {
    riscvFreeEntryNotifierFn freeEntryNotifier;
} riscvExtCB;
```

### Description

One RISC-V implementation choice is to implement virtual memory TLB updates using a trap to a Machine mode handler. In this case, memory mappings will typically be modified by writes to custom CSRs, or a similar mechanism.

This notifier is called to indicate that the base model is deleting a cached entry in a TLB. It gives the extension the opportunity to update data structures to make them consistent with removal of that entry. The entry that is being removed is indicated by the unique `entryId` parameter, which was supplied when the TLB entry was created (see section 13.44). The TLB that is affected is specified by the `tlbId` enumeration:

```
typedef enum riscvTLBIdE {
    RISC_V_TLB_HS,          // HS TLB
    RISC_V_TLB_VS1,        // VS stage 1 virtual TLB
    RISC_V_TLB_VS2,        // VS stage 2 virtual TLB
} riscvTLBId
```

(`RISC_V_TLB_VS1` and `RISC_V_TLB_VS2` TLBs are only used for processors that implement the Hypervisor extension.)

### Example

This template shows how the notifier might be used to mark an entry in an array of implementation-specific entries as uninstalled so that it is available for reuse:

```
RISC_V_FREE_ENTRY_NOTIFIER_FN(custFreeEntry) {
    vmiosObjectP object = clientData;
    object->tlb[entryId] = False;
}

static VMIOS_CONSTRUCTOR_FN(extConstructor) {
    . . . lines omitted . . .
    object->extCB.freeEntryNotifier = custFreeEntry;
    . . . lines omitted . . .
}
```

### 14.23 *Function restrictionsCB*

```
#define RISC_V_RESTRICTIONS_FN(_NAME) void _NAME( \  
    riscvP      riscv,          \  
    vmiDocNodeP node,          \  
    void        *clientData    \  
    ) \  
typedef RISC_V_RESTRICTIONS_FN(*riscvRestrictionsFn); \  
 \  
typedef struct riscvExtCBS { \  
    riscvRestrictionsFn    restrictionsCB; \  
} riscvExtCB;
```

#### **Description**

This function is called to add documentation to a derived model listing any restrictions of that model. Documentation should be added beneath the given `node` (of type `vmiDocNodeP`) using the `vmidocAddText` function.