RISC-V Model Configuration and Custom Extension Guide

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

Author:	Imperas Software Limited
Version:	1.93
Filename:	OVP_RISCV_Model_Custom_Extension_Guide
Project:	RISC-V Model Configuration and Custom Extension Guide
Last Saved:	Thursday, 08 August 2024
Keywords:	

Copyright Notice

Right to Copy Documentation

Destination Control Statement

Disclaimer

Table of Contents

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.

Note that many aspects of standard RISC-V model behavior can be controlled using parameters. For processors within the standard envelope, it may be that no custom model is needed and that all required behavior can be specified using parameters. Section 5 of this document describes the available options in detail; the appendix in section 18 presents some of the same information from a RISC-V architectural extension perspective.

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) or the Imperas Professional products (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 `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) or the Imperas Professional products (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, 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 a user-defined output VLNV binary library directory (in this case, `/home/user1/lib/$IMPERAS_ARCH/ImperasLib`):

```
% mkdir -p /home/user1/lib/$IMPERAS_ARCH/ImperasLib
% make -f $IMPERAS_HOME/ImperasLib/buildutils/Makefile.library \
         VLNVSRC=/home/user1/LocalLib/source \
         VLNVROOT=/home/user1/lib/$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 `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 existing source code. When creating a *custom* or *extended* RISC-V processor model, some important things should be considered:

1) How easy will it be to maintain the custom model?
2) How will the custom model be adapted to changes to the specifications (either new versions of existing specifications, or entirely new standard extensions)?
3) How will the custom model be tested and verified?

The Imperas RISC-V model is designed to address these issues by allowing *linked* models to be created. Linked models use the *unmodified* RISC-V base model source and add custom features by linking in an additional custom shared object. This addresses the issues above in the following way:

1) Imperas maintain the base RISC-V model.
2) Imperas 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 possibly 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 Implementing PMA Constraints

Chapter 12 describes how custom PMA constraints can be added to a RISC-V model.

4 Linked Model Creation

4.1 Introduction

To create a custom RISC-V model, create a new vendor component VLNV library based on the `vendor.com` template in the Imperas VLNV library. The new VLNV library should use a unique vendor name (don't use `vendor.com`). Beneath the `vendor.com` directory, there are two subdirectories of significance to this process:

1. Directory `processor/riscv/1.0/model` contains files required to extend the base RISC-V model to implement the linked RISC-V model. There are three source 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. A custom model may implement either no extensions at all or only a single extension as well – the `vendor.com` model has multiple extensions to provide separate starting points for different types of extension, as described below.
    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).

    In addition to the three source files, there is a Makefile configured to automatically link to the base RISC-V model.
    The combined model is constructed by first taking all source files from the base model (the RISC-V processor model in `riscv.ovpworld.org`) and then copying in any source files from the extension, thereby replacing any files in the base model with files with the same name from the extension.
    In theory, *any* base model source file can be replaced in this way, but in practice **only `riscvConfigList.c` and `riscvInfo.c` should ever be replaced in this way** (otherwise the extended model will diverge from base model behavior).

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; some custom processors may require no extensions at all if they are simply configuring standard features implemented by the base model.

4.2 Creating a New Model Library

A new customized RISC-V linked model must be created by copying files from the `vendor.com` template model into a new vendor directory. The files that must be copied depend on whether the new model has custom extensions or not: if it has no custom extensions but simply requires configuration options to be set correctly, the process is simpler. The steps required in these two cases are described in the following sections.

4.2.1 Creating a New Model - No Custom Extensions

First copy the `vendor.com/processor` directory to a new vendor-specific library directory (called `custom.com` in this section), using commands like this (on Linux):

```
% mkdir -p /home/user1/LocalLib/source/custom.com
% cp -r $IMPERAS_HOME/ImperasLib/source/vendor.com/processor
/home/user1/LocalLib/source/custom.com
```

Once the source has been copied, update the files in the `custom.com/processor/riscv/1.0/model` directory as described below.

4.2.1.1 File `extensionConfig.h`

Delete the `extensionConfig.h` file, which is not required if there are no extensions.

4.2.1.2 File `riscvInfo.c`

This file provides generic information about a processor (not RISC-V specific). Modify it to remove all `vmiVlnvInfo` and `vmiVlnvInfoList` structures, and all references to the `mandatoryExtensions` field in `vmiProcessorInfo` definitions. Then update the `info32` and `info64` `vmiProcessorInfo` definitions to 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",

        .elfCode          = 243,
        .gdbPath          = "$IMPERAS_HOME/lib/$IMPERAS_ARCH/gdb/riscv-none-embed-gdb"
VMI_EXE_SUFFIX,
        .gdbInitCommands  = "set architecture riscv:rv32",
        .family           = "Custom",
        .QLQualified      = True
    };
```

Depending on whether the extended model supports only `XLEN` of 32 or 64, either the `info32` or `info64` entries might be deleted, and the return value of `riscvProcInfo()` adjusted accordingly.

4.2.1.3 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 standard RISC-V processor behavior to be configured, as described later in this document. 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, removing the `extensionConfigs` field which is not required if there are no extensions:

```
static const riscvConfig configList[] = {

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

    {0} // null terminator
};
```

Then refer to sections 5 and 18 in this document to specify more fields in the `riscvConfig` structure to implement the required configuration.

4.2.2 Creating a New Model – With Custom Extensions

To create a new customized RISC-V linked model with custom extensions, first copy the `vendor.com/processor` and `vendor.com/intercept` directories to a new vendor-specific library directory (called `custom.com` in this section), using commands like this (on Linux):

```
% mkdir -p /home/user1/LocalLib/source
% cp -r $IMPERAS_HOME/ImperasLib/source/vendor.com /home/user1/LocalLib/source/custom.com
```

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.

Once the source has been copied, update the files in the `custom.com/processor/riscv/1.0/model` directory as described below.

4.2.2.1 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 extended processor model.

4.2.2.2 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,
        .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.

4.2.2.3 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. 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`.

4.3 Building the linked model

The linked model is built using the standard `Makefile` system provided in a product installation, located at `$IMPERAS_HOME`. Assuming the linked model source is in a local library directory at `/home/user1/LocalLib/source/custom.com`, the following commands will build the model into an output directory `/home/user1/lib/$IMPERAS_ARCH/LocalLib`:

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

4.4 Executing the linked model

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

```
-vlnvroot lib/$IMPERAS_ARCH/LocaLib
```

The model can be used with `IMPERAS_ISS` by inclusion on the command line, using:

```
-processorvendor custom.com vlnvroot lib/$IMPERAS_ARCH/LocalLib
```

An example showing the process to create the custom processor model and to run a simple assembler test application is provided in an Imperas release at

`IMPERAS_HOME/Examples/Models/Processor/FeatureUsage/RISCV_ExtendedModel`

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 archImplicit;         // implicit feature bits (not in misa)
    riscvArchitecture archMask;             // read/write bits in architecture
    riscvArchitecture archFixed;            // fixed bits in architecture
    riscvUserVer      user_version    : 8;  // user-level ISA version
    riscvPrivVer      priv_version    : 8;  // privileged architecture version
    riscvVectVer      vect_version    : 8;  // vector architecture version
    riscvVectorSet    vect_profile    : 8;  // vector architecture profile
    riscvBitManipVer  bitmanip_version: 8;  // bitmanip architecture version
    riscvBitManipSet  bitmanip_absent :16;  // bitmanip absent extensions
    riscvCryptoVer    crypto_version  : 8;  // cryptographic architecture version
    riscvVCryptoVer   vcrypto_version : 8;  // vector cryptographic version
    riscvCryptoSet    crypto_absent   :32;  // cryptographic absent extensions
    riscvDSPVer       dsp_version     : 8;  // DSP architecture version
    riscvDSPSet       dsp_absent      : 8;  // DSP absent extensions
    riscvCompressVer  compress_version: 8;  // compressed architecture version
    riscvCompressSet  compress_present:16;  // compressed present extensions
    riscvHypVer       hyp_version     : 8;  // hypervisor architecture version
    riscvDebugVer     dbg_version     : 8;  // debugger architecture version
    riscvRNMIVer      rnmi_version    : 8;  // rnmi version
    riscvSmepmpVer    Smepmp_version  : 8;  // Smepmp version
    riscvCLICVer      CLIC_version    : 8;  // CLIC version
    riscvAIAVer       AIA_version     : 8;  // AIA version
    riscvZfinxVer     Zfinx_version   : 8;  // Zfinx version
    riscvZceaVer      Zcea_version    : 8;  // Zcea version (legacy only)
    riscvZcebVer      Zceb_version    : 8;  // Zceb version (legacy only)
    riscvZceeVer      Zcee_version    : 8;  // Zcee version (legacy only)
    riscvFP16Ver      fp16_version    : 8;  // 16-bit floating point version
    riscvFSMode       mstatus_fs_mode : 8;  // mstatus.FS update mode
    riscvDMMode       debug_mode      : 8;  // is Debug mode implemented?
    riscvDERETMode    debug_eret_mode : 8;  // debug mode MRET/SRET/DRET action
    riscvDPriority    debug_priority  : 8;  // simultaneous debug event priority
    riscvChainTVAL    chain_tval      : 8;  // chained trigger tval to select
    riscvPMPR0W1      PMP_R0W1        : 8;  // behavior when PMP R=0 W=1
    riscvPMMode       Ssnpm           : 8;  // Ssnpm implemented modes
    riscvPMMode       Smnpm           : 8;  // Smnpm implemented modes
    riscvPMMode       Smmpm           : 8;  // Smmpm implemented modes
    const char      **members;              // cluster member variants
    const char       *leaf_hart_prefix;     // prefix for hart in cluster
    const char       *mtime_counter;        // mtime counter name

    // memory constraints
    riscvMConstraint amo_constraint      : 2;   // AMO memory constraint
    riscvMConstraint lr_sc_constraint    : 2;   // LR/SC memory constraint
    riscvMConstraint push_pop_constraint : 2;   // PUSH/POP memory constraint
    riscvMConstraint vector_constraint   : 2;   // vector load/store constraint
    riscvPMARegionCP pmaStatic;                 // static PMA regions

    // unimplemented instructions
    riscvITypeCP unimplementedInstr;    // unimplemented instruction list
```

```
    // remapped CSR addresses
    const char *CSR_remap;              // "<csrName>=<number>,..."

    // configuration not visible in CSR state
    Uns64 reset_address;                // reset vector address
    Uns64 nmi_address;                  // NMI address
    Uns64 nmiexc_address;               // RNMI exception address
    Uns64 debug_address;                // debug vector address
    Uns64 dexc_address;                 // debug exception address
    Uns64 CLINT_address;                // internally-implemented CLINT address
    Flt64 mtime_Hz;                     // clock frequency of CLINT mtime
    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_edeleg;                    // non-delegated exceptions
    Uns64 ecode_mask;                   // implemented bits in xcause.ecode
    Uns64 ecode_nmi;                    // exception code for NMI
    Uns64 ecode_nmi_mask;               // implemented bits in mncause.ecode
    Uns64 Svnapot_page_mask;            // implemented Svnapot page sizes
    Uns64 miprio_mask;                  // writable entries in M-mode iprio array
    Uns64 siprio_mask;                  // writable entries in S-mode iprio array
    Uns64 hviprio_mask;                 // writable entries in VS-mode iprio array
    Uns32 counteren_mask;               // mcounteren mask
    Uns32 scounteren_zero_mask;         // zero bits in scounteren
    Uns32 hcounteren_zero_mask;         // zero bits in hcounteren
    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 regions
    Uns32 PMP_csrs;                     // number of implemented PMP CSRs
    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 EEW_index;                    // maximum index EEW (vector extension)
    Uns32 SEW_min;                      // minimum SEW (vector extension)
    Uns32 ASID_cache_size;              // ASID cache size
    Uns32 TW_time_limit;                // mstatus.TW time limit
    Uns32 STO_time_limit;               // WRS.STO time limit
    Uns32 tinfo;                        // tinfo default value (all triggers)
    Uns16 trigger_match;                // bitmask of legal trigger match values
    Uns16 cmomp_bytes;                  // cache block bytes (management/prefetch)
    Uns16 cmoz_bytes;                   // cache block bytes (zero)
    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
    Uns8  hvictl_IID_bits;              // hvictl.IID implemented bits
    Uns8  mtime_bits;                   // bit size of mtime counter
    Uns8  IPRIOLEN             : 4;     // AIA IPRIOLEN value
    Uns8  HIPRIOLEN            : 4;     // AIA HIPRIOLEN value
    Bool  isPSE                : 1;     // whether a PSE (internal use only)
    Bool  enable_expanded      : 1;     // enable expanded instructions
    memEndian endian           : 1;     // data endianness
    Bool  endianFixed          : 1;     // endianness is fixed (UBE/SBE/MBE r/o)
    Bool  use_hw_reg_names     : 1;     // use hardware names for X/F registers
    Bool  no_pseudo_inst       : 1;     // don't report pseudo-instructions
    Bool  show_c_prefix        : 1;     // show opcode compressed prefix
    Bool  ABI_d                : 1;     // ABI uses D registers for parameters
    Bool  misa_B_Zba_Zbb_Zbs   : 1;     // misa.B if Zba, Zbb and Zbs present
```

```
    Bool  agnostic_ones        : 1;     // when agnostic elements set to 1
    Bool  MXL_writable         : 1;     // writable bits in misa.MXL
    Bool  SXL_writable         : 1;     // writable bits in mstatus.SXL
    Bool  UXL_writable         : 1;     // writable bits in mstatus.UXL
    Bool  VSXL_writable        : 1;     // writable bits in mstatus.VSXL
    Bool  Smstateen            : 1;     // Smstateen implemented?
    Bool  Smcsrind             : 1;     // Smcsrind implemented?
    Bool  Sstc                 : 1;     // Sstc implemented?
    Bool  Sscofpmf             : 1;     // Sscofpmf implemented?
    Bool  Smcntrpmf            : 1;     // Smcntrpmf implemented?
    Bool  Smcdeleg             : 1;     // Smcdeleg implemented?
    Bool  Svpbmt               : 1;     // Svpbmt implemented?
    Bool  Svinval              : 1;     // Svinval implemented?
    Bool  Svadu                : 1;     // Svadu implemented?
    Bool  Ssqosid              : 1;     // Ssqosid implemented?
    Bool  Smaia                : 1;     // Smaia implemented?
    Bool  IMSIC_present        : 1;     // IMSIC present?
    Bool  noZaamo              : 1;     // Zaamo not implemented?
    Bool  noZalrsc             : 1;     // Zalrsc not implemented?
    Bool  Zacas                : 1;     // Zacas implemented?
    Bool  Zabha                : 1;     // Zabha implemented?
    Bool  Zawrs                : 1;     // Zawrs implemented?
    Bool  Zmmul                : 1;     // Zmmul implemented?
    Bool  Zfa                  : 1;     // Zfa implemented?
    Bool  Zfhmin               : 1;     // Zfhmin implemented?
    Bool  Zfbfmin              : 1;     // Zfbfmin implemented?
    Bool  Zvlsseg              : 1;     // Zvlsseg implemented?
    Bool  Zvamo                : 1;     // Zvamo implemented?
    Bool  Zvediv               : 1;     // Zvediv implemented?
    Bool  Zvqmac               : 1;     // Zvqmac implemented?
    Bool  Zvfh                 : 1;     // Zvfh implemented?
    Bool  Zvfhmin              : 1;     // Zvfhmin implemented?
    Bool  Zvfbfmin             : 1;     // Zvfbfmin implemented?
    Bool  Zvfbfwma             : 1;     // Zvfbfwma implemented?
    Bool  unitStrideOnly       : 1;     // only unit-stride operations supported
    Bool  noFaultOnlyFirst     : 1;     // fault-only-first instructions absent?
    Bool  Zihintntl            : 1;     // whether Zihintntl is present
    Bool  Zicond               : 1;     // whether Zicond is present
    Bool  Zicbom               : 1;     // whether Zicbom is present
    Bool  Zicbop               : 1;     // whether Zicbop is present
    Bool  Zicboz               : 1;     // whether Zicboz is present
    Bool  Zimop                : 1;     // whether Zimop is present
    Bool  Zcmop                : 1;     // whether Zcmop is present
    Bool  Zicfiss              : 1;     // whether Zicfiss is present
    Bool  Zicfilp              : 1;     // whether Zicfilp is present
    Bool  noZicsr              : 1;     // whether Zicsr is absent
    Bool  noZifencei           : 1;     // whether Zifencei is absent
    Bool  updatePTEA           : 1;     // hardware update of PTE A bit?
    Bool  updatePTED           : 1;     // hardware update of PTE D bit?
    Bool  unaligned_low_pri    : 1;     // low-priority unaligned exceptions
    Bool  unaligned_check_ms   : 1;     // whether memory-side checks alignment
    Bool  unaligned            : 1;     // whether unaligned accesses supported
    Bool  unalignedAMO         : 1;     // whether AMO supports unaligned
    Bool  unalignedV           : 1;     // whether vector supports unaligned
    Bool  wfi_is_nop           : 1;     // whether WFI is treated as NOP
    Bool  wfi_resume_not_trap  : 1;     // whether pending wakeup stops WFI trap
    Bool  nmi_absent           : 1;     // whether NMI is absent
    Bool  nmi_is_latched       : 1;     // whether NMI is posedge-latched
    Bool  nmi_high_priority    : 1;     // whether NMI is high priority
    Bool  nmi_update_mstatus   : 1;     // whether NMI updates mstatus mpie and mie
    Bool  nmi_update_tcontrol  : 1;     // whether NMI updates tcontrol mpte and mte
    Bool  nmi_zero_mtval       : 1;     // whether NMI zeroes mtval
    Bool  mtval_is_ro          : 1;     // whether mtval is read-only
    Bool  mtvec_is_ro          : 1;     // whether mtvec is read-only
    Bool  mtvec_sext           : 1;     // whether mtvec sign-extended
    Bool  stvec_sext           : 1;     // whether stvec sign-extended
    Bool  utvec_sext           : 1;     // whether utvec sign-extended
    Bool  mtvt_sext            : 1;     // whether mtvec sign-extended
    Bool  stvt_sext            : 1;     // whether stvec sign-extended
    Bool  utvt_sext            : 1;     // whether utvec sign-extended
    Bool  cycle_undefined      : 1;     // whether cycle CSR undefined
```

```
    Bool  mcycle_undefined     : 1;     // whether mcycle CSR undefined
    Bool  time_undefined       : 1;     // whether time CSR is undefined
    Bool  instret_undefined    : 1;     // whether instret CSR undefined
    Bool  minstret_undefined   : 1;     // whether minstret CSR undefined
    Bool  hpmcounter_undefined : 1;     // whether hpmcounter* CSRs undefined
    Bool  mhpmcounter_undefined: 1;     // whether mhpmcounter* CSRs undefined
    Bool  tdata2_undefined     : 1;     // whether tdata2 CSR is undefined
    Bool  tdata3_undefined     : 1;     // whether tdata3 CSR is undefined
    Bool  tinfo_undefined      : 1;     // whether tinfo CSR is undefined
    Bool  tcontrol_undefined   : 1;     // whether tcontrol CSR is undefined
    Bool  mcontext_undefined   : 1;     // whether mcontext CSR is undefined
    Bool  scontext_undefined   : 1;     // whether scontext CSR is undefined
    Bool  mscontext_undefined  : 1;     // whether mscontext CSR is undefined
    Bool  scontext_0x5A8       : 1;     // whether scontext explicitly at 0x5A8
    Bool  hcontext_undefined   : 1;     // whether hcontext CSR is undefined
    Bool  mnoise_undefined     : 1;     // whether mnoise CSR is undefined
    Bool  dscratch0_undefined  : 1;     // whether dscratch0 CSR is undefined
    Bool  dscratch1_undefined  : 1;     // whether dscratch1 CSR is undefined
    Bool  amo_trigger          : 1;     // whether triggers used with AMO
    Bool  amo_aborts_lr_sc     : 1;     // whether AMO aborts active LR/SC
    Bool  lr_sc_match_size     : 1;     // whether LR/SC size must match
    Bool  ignore_non_leaf_DAU  : 1;     // whether ignore non-leaf PTE D, A, U
    Bool  no_hit               : 1;     // whether tdata1.hit* unimplemented
    Bool  no_sselect_2         : 1;     // whether textra.sselect=2 is illegal
    Bool  d_requires_f         : 1;     // whether misa D requires F to be set
    Bool  enable_fflags_i      : 1;     // whether fflags_i register present
    Bool  enable_DAZ           : 1;     // whether floating point DAZ mode
    Bool  enable_FZ            : 1;     // whether floating point FZ mode
    Bool  mstatus_FS_zero      : 1;     // whether mstatus.FS hardwired to zero
    Bool  trap_preserves_lr    : 1;     // whether trap preserves active LR/SC
    Bool  xret_preserves_lr    : 1;     // whether xret preserves active LR/SC
    Bool  fence_g_preserves_vs : 1;     // whether G-stage fence preserves VS-stage
    Bool  vstart0_non_ld_st    : 1;     // require vstart 0 for non-load-store?
    Bool  vstart0_ld_st        : 1;     // require vstart 0 for load-store?
    Bool  align_whole          : 1;     // whole register load aligned to hint?
    Bool  vill_trap            : 1;     // trap instead of setting vill?
    Bool  enable_CSR_bus       : 1;     // enable CSR implementation bus
    Bool  mcounteren_present   : 1;     // force mcounteren to be present
    Bool  PMP_decompose        : 1;     // decompose unaligned PMP accesses
    Bool  PMP_undefined        : 1;     // unimplemented PMP CSR accesses cause
                                        // exceptions
    Bool  PMP_maskparams       : 1;     // enable parameters to change PMP CSR
                                        // read-only masks
    Bool  PMP_active_inM       : 1;     // PMP is enforced in M-Mode
    Bool  PMP_initialparams    : 1;     // enable parameters to change PMP CSR
                                        // reset values
    Bool  external_int_id      : 1;     // enable external interrupt ID ports
    Bool  tval_zero            : 1;     // whether [smu]tval are always zero
    Bool  tval_zero_ebreak     : 1;     // whether [smu]tval always zero on ebreak
    Bool  tval_ii_code         : 1;     // instruction bits in [smu]tval for
                                        // illegal instruction exception?
    Bool  debug_ctrlt_illegal  : 1;     // whether control transfer illegal in
                                        // debug mode
    Bool  debug_auipc_illegal  : 1;     // whether instructions using PC illegal
                                        // in debug mode
    Bool  no_resethaltreq      : 1;     // resethaltreq gives haltreq reason
    Bool  defer_step_bug       : 1;     // defer step breakpoint for one
                                        // instruction when interrupt on return
                                        // from debug mode (hardware bug)
    Bool  mask_tselect         : 1;     // writes to tselect are masked based on
                                        // trigger_num setting

    // CLIC configuration
    Uns64 mclicbase;                    // base of M-mode CLIC region
    Uns64 sclicbase;                    // base of S-mode CLIC region
    Uns64 uclicbase;                    // base of U-mode CLIC region (N extension)
    Bool  CLICANDBASIC         : 1;     // whether implements basic mode also
    Bool  CLICSELHVEC          : 1;     // selective hardware vectoring?
    Bool  CLICXNXTI            : 1;     // *nxti CSRs implemented?
    Bool  CLICXCSW             : 1;     // *scratchcs* CSRs implemented?
    Bool  externalCLIC         : 1;     // is CLIC externally implemented?
    Bool  tvt_undefined        : 1;     // whether *tvt CSRs are undefined
```

```
    Bool  intthresh_undefined  : 1;     // whether *intthresh CSRs undefined
    Bool  mclicbase_undefined  : 1;     // whether mclicbase CSR is undefined
    Bool  CSIP_present         : 1;     // whether CSIP interrupt is present
    Bool  no_clic_tv_align     : 1;     // no CLIC mode alignment restriction
                                        //  for xtvec/xtvt
    Bool  posedge_other        : 1;     // fixed int[64:N] positive edge
    Bool  poslevel_other       : 1;     // fixed int[64:N] positive level
    Uns64 posedge_0_63;                 // fixed int[63:0] positive edge
    Uns64 poslevel_0_63;                // fixed int[63:0] positive level
    Uns32 CLICLEVELS;                   // number of CLIC interrupt levels
    Uns16 nlbits_valid;                 // mask of valid cliccfg.nlbits values
    Uns8  CLICVERSION;                  // CLIC version
    Uns8  CLICINTCTLBITS;               // bits implemented in clicintctl[i]
    Uns8  CLICCFGMBITS;                 // bits implemented for cliccfg.nmbits
    Uns8  CLICCFGLBITS;                 // bits implemented for cliccfg.nlbits
    Uns8  INTTHRESHBITS;                // bits implemented in xintthresh CSRs

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

    // CSR configurable reset 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 (mconfigptr);      // mconfigptr value
        CSR_REG_DECL (mtvec);           // mtvec value
        CSR_REG_DECL (mstatus);         // mstatus reset value
        CSR_REG_DECL (mseccfg);         // mseccfg value
        CSR_REG_DECL_0_15(pmpcfg);      // pmpcfg values
        CSR_REG_DECL_0_63(pmpaddr);     // pmpaddr values
    } csr;

    // CSR configurable 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);               // mtvt mask
        CSR_REG_DECL (stvt);               // stvt mask
        CSR_REG_DECL (utvt);               // utvt mask
        CSR_REG_DECL (jvt);                // jvt 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
        CSR_REG_DECL (mvien);              // mvien mask
        CSR_REG_DECL (mvip);               // mvip mask
        CSR_REG_DECL (hvien);              // hvien mask
        CSR_REG_DECL (hvip);               // hvip mask
        CSR_REG_DECL (envcfg);             // envcfg mask
        CSR_REG_DECL_0_15(romask_pmpcfg);  // pmpcfg read-only bit masks
        CSR_REG_DECL_0_63(romask_pmpaddr); // pmpaddr read-only bit masks
    } 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 variant 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 riscvVariant in the base model. The default value is typically formed by bitwise-or of values from this file, for example, the value: ISA_U|RV32GC|ISA_X specifies an RV32 processor with I, M, A, C, F, D, U and custom extensions (X). The value should include B and K if the Bit Manipulation or Cryptographic extensions are present, *even though those do not appear in the misa CSR*. The value in a processor instance can be indirectly modified by parameters misa_MXL, misa_Extensions, add_Extensions and sub_Extensions.
archImplicit	riscvArchitecture	This specifies architecture bits for extension features that are implemented and always enabled, but not visible in the misa CSR. Standard extensions that originally were misa-controlled but are now regarded as fundamental (e.g. the B and K extensions) could be specified here, but the model will also set this automatically from equivalent arch bits, based on the specified versions of those extensions, so this should rarely be required to be explicitly set in an extension. The value in a processor instance can be indirectly modified by parameters add_implicit_Extensions and sub_implicit_Extensions.
archMask	riscvArchitecture	This specifies writable architecture bits in the misa CSR. Non-writable bits will be fixed to 1 or 0 depending on the value in the arch field. The value in a processor instance can be indirectly modified by parameters misa_MXL_mask, misa_Extensions_mask, add_Extensions_mask and sub_Extensions_mask.
archFixed	riscvArchitecture	This specifies bits in the misa CSR that may *never* be overridden by parameters. For example, a value of ISA_V means that the vector extension may never be configured or deconfigured. It is usually easier to specify as a bitwise-negation of features that *may* 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 riscvUserVer enumeration:
		`typedef enum riscvUserVerE {`
		`  RVUV_2_2,  // 2.2`
		`  RVUV_2_3,  // 2.3`
		`  RVUV_20190305,  // 20190305`
		`  RVUV_20191213,  // 20191213`
		`  RVUV_DEFAULT = RVUV_20191213,`
		`} riscvUserVer;`

priv_version	riscvPrivVer	This specifies the implemented Privileged Architecture version, defined by the riscvPrivVer enumeration:
		`typedef enum riscvPrivVerE {`
		`  RVPV_1_10,  // 1.10`
		`  RVPV_1_11,  // 1.11`

		RVPV_20190405, // 20190405 RVPV_20190608, // 20190608 RVPV_20211203, // 20211203 RVPV_1_12, // 1.12 RVPV_MASTER, RVPV_DEFAULT = RVPV_1_12 } riscvPrivVer;
endian	memEndian	This specifies the initial endianness as reported in mstatus.MBE and equivalent fields for other privilege modes.
endianFixed	Bool	This specifies that MBE, SBE and UBE fields in mstatus are read-only (data endianness is fixed). This parameter only has effect for privileged version RVPC_1_12 and later.
noZicsr	Bool	This specifies whether Zicsr is *absent* (the negation of parameter Zicsr). If the field is True, then no CSRs or CSR access instructions are defined, and an alternative privileged scheme must be implemented in the extension.
enable_expanded	Bool	The *RISC-V Unprivileged ISA* specification outlines a scheme to support instructions greater that 32 bits in length. By default, such instructions are Illegal Instructions in the base model, but this field can be set to True to enable support for 48-bit and 64-bit instructions when queried by the vmicxtFetch decoder function. This is considered so fundamental that the field may only be set directly in a processor configuration structure (there is no parameter to override this). Instruction lengths longer that 64 bits are not currently supported.

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 unless otherwise specified.

Field	Type	Description

rnmi_version	riscvRNMIVer	This specifies the implemented Resumable NMI extension version, defined by the riscvRNMIVer enumeration:
		`typedef enum riscvRNMIVerE {`
		`  RNMI_NONE,  // RNMI not implemented`
		`  RNMI_0_2_1,  // RNMI 0.2.1`
		`  RNMI_0_4_NMIE1, // RNMI version 0.4, but nmie=1`
		`  // at reset`
		`  RNMI_0_4,  // RNMI 0.4`
		`} riscvRNMIVer;`
		Use RNMI_NONE if this extension is not implemented.

reset_address	Uns64	This specifies the address to jump to on reset.
nmi_absent	Bool	This specifies that the core does not implement NMI ports or exceptions.
nmi_address	Uns64	This specifies the address to jump to on NMI. This is ignored if nmi_absent is True.
nmiexc_address	Uns64	If the Resumable NMI extension is implemented, this specifies the address to jump to to handle an exception when an NMI is active.
nmi_is_latched	Bool	This indicates whether the NMI input is posedge-latched (if True) or level-sensitive (if False). If the NMI is latched, the latch is cleared when the NMI is taken. This is ignored if nmi_absent is True.
nmi_high_priority	Bool	This indicates whether the NMI input is higher priority than Debug and Trigger Module events (if True) or lower priority (if False). This is ignored if nmi_absent is True.
nmi_update_mstatus	Bool	If the Resumable NMI extension is NOT implemented, setting this to True means that when an NMI exception is taken the mstatus CSR mpie field is set to the value of mie and mie is set to 0. When false, mstatus is not updated upon taking an NMI exception.
nmi_update_tcontrol	Bool	If the Resumable NMI extension is NOT implemented, setting this to True means that when an NMI exception is taken the tcontrol CSR mpte field is set to the value of mte and mte is set to 0. When false, tcontrol is not updated upon taking an NMI exception.
nmi_zero_mtval	Bool	If the Resumable NMI extension is NOT implemented, and this is True, upon an NMI exception the mtval CSR is set to 0. When false, mtval is not updated upon taking an NMI exception.
unimp_int_mask	Uns64	This is a bitmask specifying interrupts that are *not* 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 *x*cause.ecode.
ecode_nmi	Uns64	This defines the cause reported by an NMI interrupt. This is ignored if nmi_absent is True.
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 *x*tvec CSRs in *non-direct* mode (when the least-significant two bits of the *x*tvec CSR are not 00). For example, a value of 64 implies that mtvec is masked to enforce 64-byte alignment. It has no effect in direct mode (when the least-significant two bits of the *x*tvec CSR are 00).
mtval_is_ro	Bool	This specifies whether mtval is a read-only register (note that when tval_zero is true this does not need to be set, since mtval is hardwired to 0 in that case.)
mtvec_is_ro	Bool	This specifies whether mtvec is a read-only register.
trap_preserves_lr	Bool	This specifies whether a trap preserves any active LR/SC transaction state (if False, the LR/SC is aborted).
xret_preserves_lr	Bool	This specifies whether an *x*ret instruction preserves any active LR/SC transaction state (if False, the LR/SC is aborted).
tval_zero	Bool	This specifies whether *x*tval registers are always zero.
tval_ii_code	Bool	This specifies whether *x*tval registers are filled with the instruction value for Illegal Instruction exceptions. If False, *x*tval registers are zeroed instead.
csrMask.mtvec	Uns64	This specifies writable bits in the *mtvec* CSR. Use parameter mtvec_mask to override this value.
csrMask.stvec	Uns64	This specifies writable bits in the *stvec* CSR (when Supervisor mode is implemented). Use parameter stvec_mask to override this value.
csrMask.utvec	Uns64	This specifies writable bits in the *utvec* CSR (when N extension is implemented). Use parameter utvec_mask to override this value.
csrMask.mip	Uns64	mip CSR write mask. Use parameter mip_mask to override this value.
csrMask.sip	Uns64	sip CSR write mask (when Supervisor mode is implemented). Use parameter sip_mask to override this value.
csrMask.uip	Uns64	uip CSR write mask (when N extension is implemented). Use parameter uip_mask to override this value.
csrMask.hip	Uns64	hip CSR write mask (when H extension is implemented). Use parameter hip_mask to override this value.
mtvec_sext	Bool	This specifies whether the *mtvec* CSR is sign-extended from the most significant writable bit.
stvec_sext	Bool	This specifies whether the *stvec* CSR is sign-extended from the most significant writable bit (when Supervisor mode is implemented).
utvec_sext	Bool	This specifies whether the *utvec* CSR is sign-extended from the most significant writable bit (when N extension is implemented).

5.2.1 Reset

The RISC-V Privileged Architecture Manual specifies a small state set that must be updated on a reset event. The base RISC-V model will automatically perform these actions at reset:

1. The hart will be restarted if it is in a halted or WFI state.
2. The hart will be forced into Machine mode.
3. If PMP registers are present, these will be reset to the state specified in the configuration (see section 5.9).
4. If Smstateen CSRs are present, these will be reset as described in that extension.

5. The `mcause` register will be zeroed.
6. If the vector extension is present, the `vtype` and `vl` CSRs will be reset.
7. The `misa` CSR will be reset to indicate all configured extensions are enabled.
8. If the Trigger Module is implemented, all triggers are reset to their initial state.
9. If Debug mode is implemented, the `dcsr` CSR is reset to its initial state.
10. Any LR/SC exclusive access is cleared.
11. If the AIA extension is present, all `miprio`, `siprio` and `vsiprio` state is reset.
12. If a CLINT is present and internally implemented, the `msip` CSR will be cleared.
13. If a CLIC is present and implemented by the model, all interrupts will be configured as M-mode with priority 255 and `mintstatus` will be cleared.
14. If implemented, timers will be zeroed and pending timer interrupts cleared.
15. The hart will start executing at the configured reset address (`reset_address`).

No other state changes are made at reset. Extensions may specify additional behavior using reset notifiers (see sections 15.11 and 15.12).

5.3 Timers and Timer Interrupts

Fields here define model aspects related to timers and timer interrupts. All field defaults can be overridden using a model parameter of the same name.

Field	Type	Description
time_undefined	Bool	This specifies whether the time CSR is undefined (its behavior must be emulated by a trap).
mtime_Hz	Flt64	If the time CSR is implemented (time_undefined is False), this specifies the frequency of the timer (in Hertz). This parameter is also used if the internal CLINT model is implemented (see section 5.5.4).
Sstc	Bool	This specifies whether the stimecmp/stimecmph CSRs are implemented and, if Hypervisor mode is present, whether the vstimecmp/vstimecmph CSRs are implemented.

5.4 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 unless otherwise specified.

Field	Type	Description
wfi_is_nop	Bool	This specifies whether the WFI instruction is treated as a NOP (if False, it will cause execution of the current core to suspend waiting for an interrupt).
noZaamo	Bool	When the atomic extension (A) is implemented, this specifies whether the Zaamo instruction subset is *absent*. This is the negation of the Zaamo parameter value.
noZalrsc	Bool	When the atomic extension (A) is implemented, this specifies whether the Zalrsc instruction subset is *absent*. This is the negation of the Zalrsc parameter value.
Zacas	Bool	When the atomic extension (A) is implemented, this specifies whether the Zacas instruction subset is implemented.
Zabha	Bool	When the atomic extension (A) is implemented, this specifies whether the Zabha instruction subset is implemented.
Zawrs	Bool	This specifies whether the Zawrs extension is implemented.
Zimop	Bool	This specifies whether the Zimop extension is implemented.
Zcmop	Bool	This specifies whether the Zcmop extension is implemented.
Zicfiss	Bool	This specifies whether the Zicfiss extension is implemented.
Zicfilp	Bool	This specifies whether the Zicfilp extension is implemented.
Zmmul	Bool	If the M extension is present, this specifies that only multiply instructions are implemented (not divide or remainder).
Zihintntl	Bool	This specifies whether Zihintntl instructions are decoded (treated as NOPs by the model)
Zicond	Bool	This specifies whether Zicond is present.
noZifencei	Bool	This specifies whether Zifencei is *absent* (the negation of parameter Zifencei). If the field is True, instruction fence.i is undefined.
Smstateen	Bool	This specifies whether *x*stateen state enable CSRs are implemented.
Smcsrind	Bool	This specifies whether Smcsrind is present (and also Sscsrind if Supervisor mode is implemented).
Smcdeleg	Bool	This specifies whether the Smcdeleg extension is implemented (and also Ssccfg).
Sscofpmf	Bool	This specifies whether the Sscofpmf extension is implemented.
Smcntrpmf	Bool	This specifies whether the Smcntrpmf extension is implemented.
Ssqosid	Bool	This specifies whether the Ssqosid extension is implemented.
TW_time_limit	Uns32	If wfi_is_nop is False or Zawrs is True, this specifies the number of simulated cycles that will pass when a hart is stalled by WFI or WRS.NTO before an Illegal Instruction or Virtual Instruction trap is taken.
wfi_resume_not_trap	Bool	If wfi_is_nop is True or TW_time_limit is zero and a WFI restart event is pending when the WFI is executed, this specifies that the WFI should be treated as a NOP if it would otherwise trap because mstatus.TW=1.
STO_time_limit	Uns32	If Zawrs is True, this specifies the number of simulated cycles that will pass when a hart is stalled by WRS.STO before the hart resumes execution.
cycle_undefined	Bool	This specifies whether the cycle CSR is undefined (its behavior must be emulated by a trap).
mcycle_undefined	Bool	This specifies whether the mcycle CSR is undefined (its behavior must be emulated by a trap).
time_undefined	Bool	This specifies whether the time CSR is undefined (its behavior must be emulated by a trap). If present, the frequency of the timer (in Hertz) is given by mtime_Hz (see section 5.3).

instret_undefined	Bool	This specifies whether the instret CSR is undefined (its behavior must be emulated by a trap).
minstret_undefined	Bool	This specifies whether the minstret CSR is undefined (its behavior must be emulated by a trap).
hpmcounter_undefined	Bool	This specifies whether the hpmcounter* CSRs are undefined (their behavior must be emulated by a trap).
mhpmcounter_undefined	Bool	This specifies whether the mhpmcounter* and mhpmevent* CSRs are undefined (their behavior must be emulated by a trap).
counteren_mask	Uns32	This is a bitmask specifying writable bits in mcounteren, scounteren and hcounteren registers. Note that writable bits in scounteren and hcounteren may be further modified by scounteren_zero_mask and hcounteren_zero_mask (see below).
scounteren_zero_mask	Uns32	This is a bitmask specifying bits in scounteren that are always zero even if they are indicated as writable in mcounteren by counteren_mask (see above).
hcounteren_zero_mask	Uns32	This is a bitmask specifying bits in hcounteren that are always zero even if they are indicated as writable in mcounteren by counteren_mask (see above).
noinhibit_mask	Uns32	This is a bitmask specifying counters that may *not* be inhibited using mcountinhibit (in other words, 1 bits in this mask are always 0 in mcountinhibit).

5.5 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 if the field is applicable – see the description below of when each field is used.

The CLIC is implemented if field `CLICLEVELS` is non-zero. When implemented, the behavior can either be modeled by an *internal* memory-mapped component or *externally*, depending on the value of field `externalCLIC`. Depending on the settings of these two fields, more fields are used and parameters made available to further configure the CLIC behavior; these are described in separate tables below.

5.5.1 CLIC Fundamental Fields

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: 1. `irq_id_i`: input port written with highest-priority pending interrupt; 2. `irq_lev_i`: input port written with level of highest-priority pending interrupt; 3. `irq_sec_i`: input port written with execution level of highest-priority pending interrupt; 4. `irq_shv_i`: input port written with indication of whether the highest-priority pending interrupt uses selective hardware vectoring; 5. `irq_i`: active-high input indicating that an external CLIC interrupt is pending.

5.5.2 CLIC Common Fields

These fields are used when the CLIC is implemented (`CLICLEVELS!=0`), whether internally or externally. All field defaults can be overridden using a model parameter of the same name unless otherwise specified.

Field	Type	Description
`CLICVERSION`	`Uns32`	This specifies the CLIC version.
`CLICXNXTI`	`Bool`	This specifies whether *x*`nxti` CSRs are implemented.
`CLICXCSW`	`Bool`	This specifies whether `xscratchcs`*x* CSRs are implemented.
`tvt_undefined`	`Bool`	This specifies whether *x*`tvt` CSRs are implemented – if `False`, then *x*`tvec` registers are used instead.
`intthresh_undefined`	`Bool`	This specifies whether *x*`intthresh` CSRs are undefined.
`INTTHRESHBITS`	`Uns8`	This specifies the number of writable bits in *x*`intthresh` CSRs (if defined).
`mclicbase_undefined`	`Bool`	This specifies whether the *`mclicbase`* CSR is undefined.
`csrMask.mtvt`	`Uns64`	This specifies writable bits in the *`mtvt`* CSR. Use parameter `mtvt_mask` to override this value.
`csrMask.stvt`	`Uns64`	This specifies writable bits in the *`stvt`* CSR (if Supervisor mode is implemented). Use parameter `stvt_mask` to override this value.
`csrMask.utvt`	`Uns64`	This specifies writable bits in the *`utvt`* CSR (if the N extension is implemented). Use parameter `utvt_mask` to override this value.
`mtvt_sext`	`Bool`	This specifies whether the *`mtvt`* CSR is sign-extended from the most significant writable bit.
`stvt_sext`	`Bool`	This specifies whether the *`stvt`* CSR is sign-extended from the most significant writable bit (if Supervisor mode is implemented).
`utvt_sext`	`Bool`	This specifies whether the *`utvt`* CSR is sign-extended from the most significant writable bit (if the N extension is implemented).

5.5.3 CLIC Internal Fields

These fields are used when the CLIC is implemented *internally* (`CLICLEVELS!=0` and `externalCLIC=False`).

Field	Type	Description

`CLIC_version`	`riscvCLICVer`	This specifies the implemented CLIC version, defined by the `riscvCLICVer` enumeration:
		`typedef enum riscvCLICVerE {`
		`  RVCLC_20180831,  // 20180831`
		`  RVCLC_0_9_20191208, // 0.9-draft-20191208`
		`  RVCLC_0_9_20220315, // 0.9-draft-20221315`
		`  RVCLC_0_9_20221108, // 0.9-draft-20221108`
		`  RVCLC_0_9_20230801, // 0.9-draft-20230801`
		`  RVCLC_MASTER,`
		`  RVCLC_DEFAULT = RVCLC_0_9_20230801`
		`} riscvCLICVer;`

`CLICANDBASIC`	`Bool`	This specifies whether basic interrupt control is also implemented.
`CLICINTCTLBITS`	`Uns32`	This specifies the number of bits implemented in `clicintctl[i]`.
`CLICCFGMBITS`	`Uns32`	This specifies the maximum value to which `cliccfg.nmbits` may be set. Attempts to set larger values are clamped to this maximum.
`CLICCFGLBITS`	`Uns32`	This specifies the maximum value to which `cliccfg.nlbits` may be set. Attempts to set larger values are clamped to this maximum.
`mclicbase`	`Uns64`	`mclicbase` value
`sclicbase`	`Uns64`	`sclicbase` value (if Supervisor mode is implemented)
`uclicbase`	`Uns64`	`uclicbase` value (if the N extension is implemented)
`nlbits_valid`	`Uns16`	This is a bitmask of valid values for `cliccfg.nlbits`. For example, `cliccfg.nlbits` may only hold the value 8 if bit 8 is set in `nlbits_valid`. Attempts to set `cliccfg.nlbits` to values not selected by this bitmask will be rejected, and the previous value of the field will be retained.
`CLICSELHVEC`	`Bool`	This specifies whether *selective hardware vectoring* is supported.
`CSIP_present`	`Bool`	This specifies whether the optional positive-edge-triggered `CSIP` interrupt is present. Depending on the CLIC version selected, this may be either ID 12 or ID 16.
`no_clic_tv_align`	`Bool`	This specifies whether the alignment rules for *x*tvec.base and *x*tvt required by the sxclic extensions are implemented. When False the compliant behavior of forcing at least 64 byte alignment when CLIC mode is active for *x*tvec.base and *x*tvt is implemented.
`posedge_0_63`	`Uns64`	This mask specifies interrupts in the range 0 to 63 that have fixed positive edge-sensitive configuration.
`poslevel_0_63`	`Uns64`	This mask specifies interrupts in the range 0 to 63 that have fixed positive level-sensitive configuration.
`posedge_other`	`Bool`	This specifies whether all interrupts with index 64 and higher have fixed positive edge-sensitive configuration.
`poslevel_other`	`Bool`	This specifies whether all interrupts with index 64 and higher have fixed positive level-sensitive configuration.

5.5.4 Custom registers in the CLIC MMIO custom region

Behavior for registers in the CLIC MMIO custom region may be implemented by an extension by registering a notifier function in the constructor with the *clicCustomRd*/*clicCustomWr* member of the riscvExtCB struct. These notifiers will be

called when the custom region of the CLIC MMIO address space, at offset 0x800 to 0xFFF, is read or written.

See Appendix sections 15.39 and 15.40 for details.

5.6 CLINT

Fields here define the *Core Local Interruptor* (CLINT) configuration. This block models a legacy SiFive-specific component and is not intended for general use. All field defaults can be overridden using a model parameter of the same name.

Field	Type	Description
`CLINT_address`	`Uns64`	If zero, this specifies that the CLINT is unimplemented, otherwise it specifies the address of the CLINT block.
`mtime_Hz`	`Flt64`	If the CLINT is implemented (`CLINT_address!=0`), this specifies the frequency of the CLINT `mtime` counter.

5.7 AIA

Fields here define the *Advanced Interrupt Architecture* (AIA) configuration. All field defaults can be overridden using a model parameter of the same name unless otherwise specified.

Note that the model does not implement an interrupt controller component (e.g. APLIC or IMSIC): if such components are required, they must be implemented in the platform and integrated using the interface as described below.

Field	Type	Description
Smaia	Bool	This specifies that the AIA is implemented. Subsequent fields in this table are ignored unless Smaia is True.
IMSIC_present	Bool	This indicates whether an *Incoming MSI Controller* (IMSIC) is implemented in the platform. If True, then *xstopei* CSRs are enabled and an IMSIC bus port is present which must be connected to the externally-implemented IMSIC. See below for information about connecting such a model.

AIA_version	riscvAIAVer	This specifies the implemented Smaia extension version, defined by the riscvAIAVerVer enumeration:
		`typedef enum riscvAIAVerE {`
		`  RVAIA_1_0_RC1,  // 1.0-RC1`
		`  RVAIA_1_0_RC3,  // 1.0-RC3`
		`  RVAIA_1_0_RC5,  // 1.0-RC5`
		`  RVAIA_1_0,  // 1.0 ratified version`
		`  RVAIA_MASTER,`
		`  RVAIA_DEFAULT = RVAIA_1_0`
		`} riscvAIAVer;`

miprio_mask	Uns64	This specifies which M-mode IPRIO array entries are writable. An IPRIO entry with index i is writable only if bit i of miprio_mask is set.
siprio_mask	Uns64	If Supervisor mode is present, this specifies which S-mode IPRIO array entries are writable. An IPRIO entry with index i is writable only if bit i of siprio_mask is set.
hviprio_mask	Uns64	If the Hypervisor extension is present, this specifies which entries in hviprio CSRs are writable. An hviprio CSR entry corresponding to interrupt i is writable only if bit i of hviprio_mask is set.
IPRIOLEN	Uns8	This specifies the number of priority bits implemented for external interrupts (in the range 1 to 8).
HIPRIOLEN	Uns8	If the Hypervisor extension is present, this specifies the number of priority bits implemented for virtual interrupts in hviprio registers (in the range 6 to 8).
hvictl_IID_bits	Uns8	If the Hypervisor extension is present, this specifies the number of bits implemented in the hvictl.IID field (in the range 6 to 12).
csrMask.mvip	Uns64	This specifies writable bits in the mvip CSR. Bits 12:0 are ignored and always zero. Use parameter mvip_mask to override this value.
csrMask.mvien	Uns64	This specifies writable bits in the mvien CSR. Bits 12:0 are ignored and always zero. Bits that are set in mvip_mask but clear in mvien_mask will be hard-wired to 1 in mvien. Use parameter mvien_mask to override this value.
csrMask.hvip	Uns64	hvip CSR write mask (only bits 63:13 used). Bits 12:0 are ignored and always zero. Use parameter hvip_mask to override this value.
csrMask.hvien	Uns64	This specifies writable bits in the hvien CSR. Bits 12:0 are ignored and always zero. Bits that are set in hvip_mask but clear in hvien_mask will be hard-wired to 1 in hvien. Use parameter hvien_mask to override this value.

5.7.1 AIA Net Port Interface

When the AIA is implemented, net ports are present allowing the external interrupt controller model to supply the highest-priority pending interrupt. These are:

`miprio`: this input should be written with the id of the highest-priority pending M-mode interrupt from the external interrupt controller model.

`siprio`: this input is present if Supervisor mode is implemented. It should be written with the id of the highest-priority pending S-mode interrupt from the external interrupt controller model.

`vsiprio`: this input is present if the Hypervisor extension is implemented. It should be written with the id of the highest-priority pending VS-mode interrupt from the external interrupt controller model.

5.7.2 IMSIC Interrupt Controller Integration

This model does not implement an interrupt controller to drive the AIA interface: this must be implemented as a platform component and connected to the processor model. For an APLIC, use the standard hart interrupt net ports plus the priority ports described in the previous section. When an IMSIC is implemented (`IMSIC_present` is `True`), additional work is required, as follows:

1. Connect a 16-bit bus to the artifact `CSR` bus of the hart (this bus port is present when `IMSIC_present` is `True`). This will be used to observe and react to changes to IMSIC-related CSRs.
2. Connect a second 16-bit bus to the artifact `IMSIC` bus of the hart (also present when `IMSIC_present` is `True`). This bus is used to implement IMSIC registers.
3. Install watchpoints on the `CSR` bus to observe writes to `mtopei`, `stopei` and `vstopei` CSRs. Writes made by the hart must cause the highest-priority pending interrupt of the indicated type to be acknowledged by the IMSIC. These CSRs have indices `0x35C`, `0x15C` and `0x25C` respectively, meaning write callbacks must be installed at addresses `0x35C0`, `0x15C0` and `0x25C0` on the CSR artifact bus.
4. If the Hypervisor extension is implemented, also install a watchpoint on the `CSR` bus to observe writes to the `hstatus` CSR (`0x600`). The value written to `hstatus.VGIEN` selects the guest interrupt file that should be reported by the IMSIC via the `vsiprio` input and also the IMSIC registers mapped using `vsiselect` and `vsireg` CSRs.
5. Expose IMSIC indirectly-accessed interrupt-file registers to the hart by installing read and write callbacks on the `IMSIC` artifact bus. An IMSIC register with index `0xAB` is mapped on the bus at address `0xMAB0`, where `M` indicates the register mode (S=1, VS=2, M=3); as a concrete example, implementing Machine-mode IMSIC register `0x72` requires installation of 4-byte or 8-byte callbacks at address `0x3720` on the `IMSIC` artifact bus.

5.8 Memory Subsystem

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

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).
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 always active. See also parameter Svadu below.
updatePTED	Bool	This indicates whether hardware update of page table entry D bit is implemented. See also parameter Svadu below.
Svadu	Bool	This indicates whether the Svadu extension is implemented (this extension adds CSR controls for hardware update of PTE A/D bits). If Svadu is True, updatePTEA and updatePTED are both forced to be False.
ignore_non_leaf_DAU	Bool	This indicates whether the value of D, A and U bits in non-leaf page table entries should be ignored. If False, any of these bits being non-zero will cause a Page Fault trap.
Svnapot_page_mask	Uns64	If non-zero, this indicates that the Svnapot extension is implemented, with contiguous pages of sizes specified by the mask. For example, a value of 0x10000 indicates that 64kB NAPOT contiguous pages are supported.
Svpbmt	Bool	This indicates whether the Svpbmt extension is implemented (page-based memory types, specified by bits 62:61 of a page table entry). Note that except for their effect on Page Faults, the encoded memory types do not alter the behavior of this model, which always implements strongly-ordered non-cacheable semantics.
Svinval	Bool	This indicates whether the Svinval extension is implemented (fine-grained address-translation cache invalidation instructions sinval.vma, sfence.w.inval and sfence.inval.ir, together with hinval.vvma and hinval.gvma if Hypervisor mode is also present).

Ssnpm	riscvPMMode	This indicates whether the Ssnpm extension is implemented (Supervisor mode next-level pointer masking). Supported options are defined by the riscvPMMode enumeration:
		`typedef enum riscvPMModeE {`
		`  RVPMMD_NONE,  // not implemented`
		`  RVPMMD_48,  // XLEN-48`
		`  RVPMMD_57,  // XLEN-57`
		`  RVPMMD_48_57, // XLEN-48 and XLEN-57`
		`} riscvPMMode;`

Smnpm	riscvPMMode	This indicates whether the Smnpm extension is implemented (Machine mode next-level pointer masking). Supported options are defined by the riscvPMMode enumeration (see above).
Smmpm	riscvPMMode	This indicates whether the Smmpm extension is implemented (Machine mode pointer masking). Supported options are defined by the riscvPMMode enumeration (see above).
unaligned	Bool	This indicates whether unaligned accesses are supported.
unaligned_low_pri	Bool	If an access is both unaligned and would also cause a page fault or access fault, this specifies that the unaligned address fault has higher priority (if False) or lower priority (if

		True) than the page fault or access fault.
unaligned_check_ms	Bool	For processors that normally always permit unaligned accesses, this specifies that load address misaligned and store/AMO address misaligned exceptions can still be generated by memory components.
unalignedAMO	Bool	This indicates whether unaligned accesses are supported for AMO operations. Use parameter Zam to modify this value.
amo_aborts_lr_sc	Bool	This indicates whether an AMO operation aborts any active lr/sc pair.
lr_sc_match_size	Bool	This indicates lr/sc data size must match for sc instruction to succeed (RV64 only).
Zicbom	Bool	This indicates whether the Zicbom extension is implemented (instructions cbo.clean, cbo.flush and cbo.inval). The instructions may cause traps if used illegally but otherwise are NOPs in this model.
cmomp_bytes	Uns16	If the Zicbom extension is implemented, this specifies the cache block size for those instructions.
Zicbop	Bool	This indicates whether the Zicbop extension is implemented (instructions prefetch.i, prefetch.r and prefetch.w). If implemented, the instructions behave as NOPs in this model.
Zicboz	Bool	This indicates whether the Zicboz extension is implemented (instruction cbo.zero).
cmoz_bytes	Uns16	If the Zicboz extension is implemented, this specifies the cache block size for cbo.zero.

amo_constraint	riscvMConstraint	This specifies constraints on memory accesses performed by AMO operations, defined by the riscvMConstraint enumeration (see below for more information about these constraints).
		`typedef enum riscvMConstraintE {`
		`RVMC_NONE,`
		`RVMC_USER1,`
		`RVMC_USER2,`
		`} riscvMConstraint;`

lr_sc_constraint	riscvMConstraint	This specifies constraints on memory accesses performed by LR/SC operations, defined by the riscvMConstraint enumeration (see below for more information about these constraints).
		`typedef enum riscvMConstraintE {`
		`RVMC_NONE,`
		`RVMC_USER1,`
		`RVMC_USER2,`
		`} riscvMConstraint;`

push_pop_constraint	riscvMConstraint	This specifies constraints on memory accesses performed by PUSH/POP operations, defined by the riscvMConstraint enumeration (see below for more information about these constraints).
		`typedef enum riscvMConstraintE {`
		`RVMC_NONE,`
		`RVMC_USER1,`
		`RVMC_USER2,`
		`} riscvMConstraint;`

5.8.1 Memory Access Constraints

Memory access constraints specified by amo_constraint, lr_sc_constraint and push_pop_constraint fields allow access for AMO, LR/SC and PUSH/POP instructions to be denied to certain memory regions, respectively. The enumeration members have the following meanings:

1. RVMC_NONE: no access restriction.
2. RVMC_USER1: deny access to memory regions including permission MEM_PRIV_USER1.
3. RVMC_USER2: deny access to memory regions including permission MEM_PRIV_USER2.

Region privileges MEM_PRIV_USER1 and MEM_PRIV_USER2 will typically be defined on certain regions by custom PMA code in a processor extension. For example, in the Andes processor, custom PMA code disallows access by AMO instructions if the PMA region is configured as a NAMO (no AMO) region. Function refinePMARegionRange specifies the required privilege as follows:

```
        // refine privilege
        if(e.MTYP==PMAMT_Black_Hole) {
            *privP = MEM_PRIV_NONE;
        } else if(!e.NAMO) {
            *privP = MEM_PRIV_RWX;
        } else {
            *privP = MEM_PRIV_RWX|MEM_PRIV_USER1;
        }
```

This privilege is then used in function setDomainPriv to enforce PMA region permissions:

```
            vmirtProtectMemory(dataDomain, low, high, priv, MEM_PRIV_SET);
```

In Andes processor configurations, values of amo_constraint, lr_sc_constraint and push_pop_constraint fields are all set to RVMC_USER1, meaning access to NAMO regions will be denied for all AMO, LR, SC, PUSH and POP instructions.

5.9 PMP Configuration

Fields here define PMP features. All field defaults can be overridden using a model parameter. Most parameters have the same name as the field, with the exception of the `romask_...` fields, which are overridden with a `mask_...` parameter.

Field	Type	Description
`PMP_registers`	`Uns32`	This defines the number of PMP regions that are implemented (maximum of 64). A value of 0 indicates the PMP feature is absent.
`PMP_crs`	`Uns32`	This defines the number of PMP address registers (and implies the number of PMP_cfg registers) that are present, including those that are RAZ/WI (maximum of 64). A value of 0 indicates the number of registers defaults to 64 (for priv_version 1.12 and later, and 16 for earlier versions.
`PMP_decompose`	`Bool`	This indicates whether unaligned PMP accesses are decomposed into individual aligned accesses (thus allowing accesses that straddle region boundaries).
`PMP_undefined`	`Bool`	This indicates whether accesses to unimplemented PMP registers cause Illegal Instruction traps. If `False`, then such accesses are ignored.
`PMP_grain`	`Uns32`	This defines the minimum granularity for PMP regions in the standard format (0: 4 bytes, 1: 8 bytes, etc).

`PMP_R0W1`	`riscvPMPR0W1`	This defines the behavior of `pmpcfg R/W/X` fields when an illegal value with `R=0` and `W=1` is written. Choices are described by the `riscvPMPR0W1` enumeration:
		`typedef enum riscvPMPR0W1E {`
		`  RVPMPI_RWX_00X, // set R=0, W=0, use new X`
		`  RVPMPI_RWX_11X, // set R=1, W=1, use new X`
		`  RVPMPI_RWX_PPX, // revert R and W, use new X`
		`  RVPMPI_RWX_PPP, // revert R, W and X`
		`  RVPMPI_RWX_000, // set RWX=000`
		`} riscvPMPR0W1;`

`PMP_active_inM`	`Bool`	When true all PMP checks are enforced in M mode when any PMP region is active. Normally this is only true for regions with the lock bit set.
`pmpaddr0..63`	`XLEN`	This defines the reset value for the `pmpaddr` registers. There are 64 separate fields.
`pmpcfg0..15`	`Uns64 (even)` `Uns32 (odd)`	This defines the reset value for the `pmpcfg` registers. There are 15 separate fields. The odd numbered fields are ignored when XLEN is 64. The high order word of even numbered fields is ignored when XLEN is 32.
`romask_pmpaddr0..63`	`XLEN`	This defines the read-only bit masks for the `pmpaddr` registers. These masks are applied along with other architecturally defined masks (they do not override other masks.)
`romask_pmpcfg0..15`	`XLEN`	This defines the read-only bit masks for the `pmpcfg` registers. These masks are applied along with other architecturally defined masks (they do not override other masks.)

`Smepmp_version`	`riscvSmepmpVer`	This specifies the implemented `Smepmp` extension version, defined by the `riscvSmepmpVer` enumeration:
		`typedef enum riscvSmepmpVerE {`
		`  RVSP_NONE,  // Smepmp not implemented`
		`  RVSP_0_9_5,  // 0.9.5`
		`  RVSP_1_0,  // 1.0`
		`  RVSP_DEFAULT = RVSP_1_0`
		`} riscvSmepmpVer;`

The PMP_registers field defines the number of PMP regions implemented. Fields for specifying reset and mask values for unimplemented region indexes are ignored.

By default, the maximum number of PMP regions is 64 when priv_version is 1.12 or later, and 16 when priv_version is earlier than 1.12 and it is a fatal error if the value of PMP_registers exceeds this. This can be overridden by the value of PMP_crs. When PMP_crs is not zero it species the number of PMP address registers that are implemented, which may be any value between 1 and 64, and the value of PMP_registers may be set to any value between 0 and PMP_crs.

The pmpcfg and romask_pmpcfg fields behave differently depending on the XLEN that is implemented:

- Fields corresponding to even numbered registers are 64 bits. When XLEN is 32 the upper word is ignored.
- Fields corresponding to odd numbered registers are 32 bits. When XLEN is 64 the corresponding register does not exist and the field is ignored.

The parameters to override the pmpaddr, pmpcfg, romask_pmpaddr and romask_pmpcfg fields are not available by default. The parameters to enable them are:

- **PMP_initialparams**
  Enables the *reset* value related parameters pmpaddr<x> and pmpcfg<x>
- **PMP_maskparams**
  Enables the *mask* related parameters mask_pmpaddr<x> and mask_pmpcfg<x>.

Note the parameters to override the romask_pmpaddr and romask_pmpcfg fields have a slightly different name because they specify a *write* mask, not a read-only mask like the field values. The parameter values are negated before overriding the configuration field. (The field values must be read-only masks to maintain backwards compatibility with existing models that have a default values of 0 for this field.)

5.9.1 Mask Values Available in Integration Registers

The values of the write masks cannot be determined by the usual technique of saving a regsiter's value, writing all 0's followed by all 1's, then reading and restoring the register because the pmpcfg entries become locked if the L field is written with a 1.

Therefore the mask values have been made available in integration support registers named mask_pmpcfg0..15 and mask_pmpaddr0..63.

Note that these are write mask values, so are inverted versions of the romask config values.

Note that these are simulator-only, not architecturally-defined, registers, and can be accessed only through the debug interface, not from a program executing on the model. But they can be useful for a harness or intercept library to discover which PMP register bits are configured as not writable.

5.10 Floating Point

Fields here are applicable when floating point is implemented (D or F extensions indicated as implemented by the processor `arch` field). All field defaults can be overridden using a model parameter of the same name. Field `fp16_version` may also be set to `RVFP16_IEEE754` by setting parameter `Zfh` to `True` (if parameter `fp16_version` is not also set).

Field	Type	Description

`mstatus_fs_mode`	`riscvFSMode`	This field specifies how the implementation-dependent `mstatus.FS` field is updated by the model. The possible behaviors are specified by the `riscvFSMode` enumeration (see section 5.10.2 below):
		`typedef enum riscvFSModeE {`
		`  RVFS_WRITE_NZ,`
		`  RVFS_WRITE_ANY,`
		`  RVFS_EXECUTE_NOT_S,`
		`  RVFS_EXECUTE_ANY,`
		`  RVFS_ALWAYS_DIRTY,`
		`  RVFS_FORCE_DIRTY`
		`} riscvFSMode;`

`d_requires_f`	`Bool`	This field specifies whether `misa.D` can only be set if `misa.F` is also set.

`Zfinx_version`	`riscvZfinxVer`	This field specifies whether, the `Zfinx` option is specified, and if so with what version. The possible values are specified by the `riscvZfinxVer` enumeration:
		`typedef enum riscvZfinxVerE {`
		`  RVZFINX_NA,  // Zfinx not implemented`
		`  RVZFINX_0_4,  // Zfinx 0.4`
		`  RVZFINX_0_41,  // Zfinx 0.41`
		`  RVZFINX_1_0,  // Zfinx 1.0`
		`  RVZFINX_DEFAULT = RVZFINX_1_0,`
		`} riscvZfinxVer;`

`fp16_version`	`riscvFP16Ver`	This field allows a format to be selected for 16-bit floating point. Values are defined by the `riscvFP16Ver` enumeration:
		`typedef enum riscvFP16VerE {`
		`  RVFP16_NA,  // 16-bit FP not supported`
		`  RVFP16_IEEE754, // IEEE half-precision format`
		`  RVFP16_BFLOAT16,// BFLOAT16 format`
		`  RVFP16_DYNAMIC  // dynamic (IEEE 754 or BF16)`
		`} riscvFP16Ver;`
		If `RVFP16_DYNAMIC` is selected, the format to be used for 16-bit floating point can be dynamically changed at runtime using artifact register `fp16Format`.

`Zfa`	`Bool`	This field specifies whether additional floating point instructions in the `Zfa` extension are implemented.
`Zfhmin`	`Bool`	If half-precision floating point is present (with format indicated by `fp16_version`), this field indicates whether only a minimal half-precision subset is implemented. If `False`, half-precision instructions are fully supported.
`Zfbfmin`	`Bool`	This field indicates whether the `Zfbfmin` extension is supported (minimal `BFLOAT16` support)
`mstatus_FS_zero`	`Bool`	If floating point is *not* implemented but *Supervisor mode is present*, this field specifies whether `mstatus.FS` is forced to zero (normally, it is writable to enable floating point emulation).
`enable_fflags_i`	`Bool`	If `True`, this field causes an 8-bit artifact register, `fflags_i`, to be present. This register reports floating point flags updated by each instruction (unlike the standard `fflags` CSR, which reports cumulative flags).
`enable_DAZ`	`Bool`	If `True`, this field enables the floating point *Denormals-Are-Zero* mode where subnormal results are flushed to +/- zero. See 5.10.6 for

		more detailed information. *Note that such behavior is not compliant with the RISC-V or IEEE 2008 specifications.*
enable_FZ	Bool	If True, this field enables the floating point *Flush-to-Zero* mode where subnormal input values to certain floating point instruction are flushed to +/- zero before . See 5.10.6 for more detailed information. *Note that such behavior is not compliant with the RISC-V or IEEE 2008 specifications.*

5.10.1 misa CSR D and F Bits

When both single and double precision floating point are implemented, D and F bits in the misa CSR often have implementation-specific dependencies. The model allows four different behaviors to be specified:

1. **default**: D and F bits can be set independently;
2. **d_requires_f=1**: D and F bits can be set independently, but D bit cannot be set to 1 if F bit is 0;
3. **archMask bit 5 (F) is 1 and bit 3 (D) is 0**: only bit 3 can be set in misa, and bit 5 is a read-only shadow of this.
4. **archMask bit 5 (F) is 0 and bit 3 (D) is 1**: only bit 5 can be set in misa, and bit 3 is a read-only shadow of this.

In both cases 3 and 4, single and double precision floating point cannot be enabled separately.

5.10.2 mstatus.FS Update Modes

The RISC-V Privileged Specification allows the behavior of the mstatus.FS field to be implementation defined. The model allows the behavior of this CSR field to be configured using the mstatus_fs_mode configuration option as follows:

1. **RVFS_WRITE_NZ**: mstatus.FS will be set to *dirty* (3) state whenever an FPR is written or a floating point operation signals an exception. Floating point operations that do not write an FPR or cause an exception will not set mstatus.FS. As an example, floating point comparison operations with Normal operands will *not* set mstatus.FS because these do not signal an exception.
2. **RVFS_WRITE_ANY**: mstatus.FS will be set whenever an FPR is written or a floating point operation could *potentially* signal an exception. As an example, floating point comparison operations with Normal operands *will* set mstatus.FS, because those operations could signal an exception with some inputs (e.g. SNaN operands).
3. **RVFS_EXECUTE_NOT_S**: mstatus.FS will be set to *dirty* (3) state when any floating point instruction retires, except for floating point stores, which do not modify mstatus.FS.
4. **RVFS_EXECUTE_ANY**: mstatus.FS will be set to *dirty* (3) state when any floating point instruction retires.
5. **RVFS_ALWAYS_DIRTY**: mstatus.FS can only hold either *off* (0) or *dirty* (3) states.
6. **RVFS_FORCE_DIRTY**: mstatus.FS is forced to *dirty* (3) state.

5.10.3 Half-Precision Support

Scalar half-precision support may be configured as follows:

1. **No half-precision support**
   Set fp16_version=RVFP16_NA and Zfhmin=False in the configuration structure or set parameter Zfh=False and parameter Zfhmin=False at simulation time.
2. **Standard Zfh extension**
   Set fp16_version=RVFP16_IEEE754 and Zfhmin=False in the configuration structure or set parameter Zfh=True at simulation time (in which case parameter Zfhmin is ignored).
3. **Standard Zfhmin extension**
   Set fp16_version=RVFP16_IEEE754 and Zfhmin=True in the configuration structure or set parameter Zfh=False and parameter Zfhmin=True at simulation time.
4. **Non-Standard half-precision support**
   Set fp16_version accordingly

5.10.4 Zfinx Support

The Zfinx extension replaces some standard floating point behavior. The options defined in this specification are configured as follows:

1. **Zfinx**
   Enable F extension and set Zfinx_version to a non-zero value.
2. **Zdinx**
   Enable F and D extensions and set Zfinx_version to a non-zero value.
3. **Zhinx**
   Enable F extension, set Zfh=True and set Zfinx_version to a non-zero value.
4. **Zhinxmin**
   Enable F extension, set Zfhmin=True and set Zfinx_version to a non-zero value.

5.10.5 Unimplemented Instructions

If a variant implements only a *subset* of the floating-point extension (for example, all instructions except DIV/SQRT/FMAC) then this can be specified using the unimplementedInstr field, described in section 5.23).

5.10.6 Flushing Subnormal Values to Zero

The configuration parameters enable_DAZ and enable_FZ may be set to True to cause subnormal results or inputs, respectively, to be flushed to zero for certain instructions. *Note that this behavior is not compliant with the RISC-V arc*

Enabling DAZ/FZ in the RiscV model only affects "real" floating point operations and not operations that are may be implemented by simple bit manipulation. This means these parameters have no effect on these RISC-V floating point instructions:

```
fabs.[sd]
fneg.[sd]
fsgnjn.[sd]
fsgnjx.[sd]
fclass.[sd]
```

They also won't affect moves to and from floating point registers, or conversions from integral to floating point types (note that these can never generate subnormal results).

The `enable_DAZ` and `enable_FZ` settings will affect these instructions:

```
fsqrt.[sd]
fadd.[sd]
fdiv.[sd]
fmax.[sd]
fmin.[sd]
fmul.[sd]
fsub.[sd]
fmadd.[sd]
fmsub.[sd]
fnmadd.[sd]
fnmsub.[sd]
fcvt.l.[sd]
fcvt.lu.[sd]
fcvt.w.[sd]
fcvt.wu.[sd]
feq.[sd]
fle.[sd]
flt.[sd]
```

In all these cases, floating-point inputs and results will be flushed to correctly-signed zeros, without setting any floating point flags as a consequence. Floating point flags may be set by the operation itself with its freshly flushed-to-zero inputs, of course.

Additional information may be found in section **10 Floating Point Operations** in the document *OVP_VMI_Morph_Time_Function_Reference.pdf*.

5.11 Vector Extension

Fields here are applicable when the Vector Extension (V) is indicated as implemented by the processor `arch` field. All field defaults can be overridden using a model parameter of the same name, unless otherwise indicated.

Field	Type	Description

`vect_version`	`riscvVectVer`	This specifies the implemented Vector Extension version, defined by the `riscvVectVer` enumeration:
		`typedef enum riscvVectVerE {`
		`  RVVV_0_7_1,  // 0.7.1-draft-20190605`
		`  RVVV_0_7_1_P,  // 0.7.1+`
		`  RVVV_0_8_20190906, // 0.8-draft-20190906`
		`  RVVV_0_8_20191004, // 0.8-draft-20191004`
		`  RVVV_0_8_20191117, // 0.8-draft-20191117`
		`  RVVV_0_8_20191118, // 0.8-draft-20191118`
		`  RVVV_0_8,  // 0.8`
		`  RVVV_0_9,  // 0.9`
		`  RVVV_1_0_20210130, // 1.0-draft-20210130`
		`  RVVV_1_0_20210608, // 1.0-draft-20210608`
		`  RVVV_1_0,  // 1.0`
		`  RVVV_MASTER,`
		`  RVVV_DEFAULT = RVVV_1_0,`
		`} riscvVectVer;`
		Version `RVVV_1_0_20210608` corresponds to the 1.0-rc1-20210608 release candidate.

`vect_profile`	`riscvVectorSet`	If non-zero, this specifies an applicable embedded profile, and should be one of:
		`  RVVS_Application  (zero value)`
		`  RVVS_Zve32x`
		`  RVVS_Zve32f`
		`  RVVS_Zve64x`
		`  RVVS_Zve64f`
		`  RVVS_Zve64d`
		Any one of these values can be specified using parameters `Zve32x`, `Zve32f`, `Zve64x`, `Zve64f` or `Zve64d` when the model is instantiated, if required.

`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 `VLEN`.
`VLEN`	`Uns32`	This specifies the vector size, in bits (a power of two in the range 32 to 65536).
`EEW_index`	`Uns32`	If non-zero, this specifies the maximum EEW that may be used for offset elements in indexed load and store instructions. If offsets larger than this are used, an Illegal Instruction exception is raised.
`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 `Zvlsseg` extension is implemented.
`Zvamo`	`Bool`	This specifies whether the `Zvamo` extension is implemented.
`Zvediv`	`Bool`	This specifies whether the `Zvediv` extension is implemented. *Note: this must currently always be False because the base model does not implement this extension.*
`Zvqmac`	`Bool`	This specifies whether the `Zvqmac` extension is implemented.
`Zvfh`	`Bool`	This specifies whether the `Zvfh` extension is implemented.
`Zvfhmin`	`Bool`	This specifies whether the `Zvfhmin` extension is implemented.
`Zvfbfmin`	`Bool`	This specifies whether the `Zvfbfmin` extension is implemented.
`Zvfbfwma`	`Bool`	This specifies whether the `Zvfbfwma` extension is implemented.
`unitStrideOnly`	`Bool`	This specifies whether only unit-stride vector loads and stores are supported. If `True`, then any other vector load/store instruction will

		be reported as an Illegal Instruction. *Note that such behavior is not compliant with the Vector Extension specification.*
noFaultOnlyFirst	Bool	This specifies whether *fault-only-first* vector loads are unimplemented. If True, then any vector fault-only-first instruction will be reported as an Illegal Instruction. *Note that such behavior is not compliant with the Vector Extension specification.*
vstart0_non_ld_st	Bool	This specifies whether vector instructions that that are not load/store instructions require the vstart CSR to be zero. If True, then attempting to execute such instructions with vstart!=0 will cause an Illegal Instruction trap.
vstart0_ld_st	Bool	This specifies whether vector load/store instructions require the vstart CSR to be zero. If True, then attempting to execute such instructions with vstart!=0 will cause an Illegal Instruction trap.
align_whole	Bool	This specifies whether whole-register load addresses must be aligned using the encoded EEW.
vill_trap	Bool	This specifies whether illegal vtype values cause a trap.
agnostic_ones	Bool	When tail/mask agnostic behavior is implemented, this specifies whether agnostic elements are filled with ones (if True) or preserved (if False).
unalignedV	Bool	This specifies whether vector load/store instructions support unaligned memory accesses.

5.11.1 Half-Precision Support

Vector half-precision floating point requires either Zfh or Zfhmin to be also configured (see section 5.10.3). Given this prerequisite, vector half-precision support may then be configured in these three ways:

1. **Zvfh=False, Zvfhmin=False**
   Set Zvfh=False and Zfhmin=False in the configuration structure or set parameter Zvfh=False and parameter Zvfhmin=False at simulation time.
2. **Zvfh=True, Zvfhmin=True**
   Set Zvfh=True in the configuration structure or set parameter Zvfh=True at simulation time (Zvfhmin is ignored because it is implied by Zvfh).
3. **Zvfh=False, Zvfhmin=True**
   Set Zvfh=False and Zfhmin=True in the configuration structure or set parameter Zvfh=False and parameter Zvfhmin=True at simulation time.

5.11.2 Unimplemented Instructions

If a variant implements only a *subset* of the vector extension then this can be specified using the unimplementedInstr field, described in section 5.23).

5.12 Bit Manipulation Extension

Fields here are applicable when the Bit Manipulation Extension (B) is indicated as implemented by the processor `arch` field[1]. All field defaults can be overridden using a model parameter of the same name, unless otherwise indicated.

Field	Type	Description

`bitmanip_version`	`riscvBitManipVer`	This specifies the implemented Bit Manipulation Extension version, defined by the `riscvBitManipVer` enumeration:
		`typedef enum riscvBitManipVerE {`
		`  RVBV_0_90,  // 0.90`
		`  RVBV_0_91,  // 0.91`
		`  RVBV_0_92,  // 0.92`
		`  RVBV_0_93_DRAFT, // 0.93 (intermediate)`
		`  RVBV_0_93,  // 0.93`
		`  RVBV_0_94,  // 0.94`
		`  RVBV_1_0_0,  // 1.0.0`
		`  RVBV_MASTER,`
		`  RVBV_DEFAULT = RVBV_0_92,`
		`} riscvBitManipVer;`

`bitmanip_absent`	`riscvBitManipSet`	By default, all Bit Manipulation Extension instruction subsets are implemented. This field is a bitmask allowing *unimplemented* subsets to be specified:
		`typedef enum riscvBitManipSetE {`
		`  RVBS_Zba  = (1<<0),`
		`  RVBS_Zbb  = (1<<1),`
		`  RVBS_Zbc  = (1<<2),`
		`  RVBS_Zbe  = (1<<3),`
		`  RVBS_Zbf  = (1<<4),`
		`  RVBS_Zbm  = (1<<5),`
		`  RVBS_Zbp  = (1<<6),`
		`  RVBS_Zbr  = (1<<7),`
		`  RVBS_Zbs  = (1<<8),`
		`  RVBS_Zbt  = (1<<9),`
		`} riscvBitManipSet;`
		The presence or absence of each of these subsets can be individually specified using parameters `Zba`, `Zbb`, `Zbc`, `Zbe`, `Zbf`, `Zbm`, `Zbp`, `Zbr`, `Zbs` and `Zbt`.

`misa_B_Zba_Zbb_Zbs`	`Bool`	For bit manipulation extension versions 1.0.0 and later, this specifies that when all of the `Zba`, `Zbb` and `Zbs` sub-extensions are present then `misa.B` is set. If `False`, then `misa.B` is always unset for these extension versions.

[1] The B bit must be specified in the `arch` field *even when not visible in the `misa` register* for this extension to be active.

5.13 Cryptographic Extension

Fields here are applicable when the Cryptographic Extension (K) is indicated as implemented by the processor `arch` field[2], except for `vcrypto_version`, which requires both the Cryptographic Extension *and* the Vector Extension (V). All field defaults can be overridden using a model parameter of the same name, unless otherwise indicated.

Field	Type	Description

`crypto_version`	`riscvCryptoVer`	This specifies the implemented Scalar Cryptographic Extension version, defined by the `riscvCryptoVer` enumeration:
		`typedef enum riscvCryptoVerE {`
		`  RVKV_0_7_2,  // 0.7.2`
		`  RVKV_0_8_1,  // 0.8.1`
		`  RVKV_0_9_0,  // 0.9.0`
		`  RVKV_0_9_2,  // 0.9.2`
		`  RVKV_1_0_0_RC1,  // 1.0.0-RC1`
		`  RVKV_1_0_0_RC5,  // 1.0.0-RC5`
		`  RVKV_DEFAULT = RVKV_1_0_0_RC5`
		`} riscvCryptoVer;`

`vcrypto_version`	`riscvVCryptoVer`	This specifies the implemented Vector Cryptographic Extension version, defined by the `riscvVCryptoVer` enumeration:
		`typedef enum riscvVCryptoVerE {`
		`  RVKVV_0_3_0,  // version 0.3.0`
		`  RVKVV_0_5_2,  // version 0.5.2`
		`  RVKVV_1_0_0_RC1, // version 1.0.0-rc1`
		`  RVKVV_MASTER,  // version master`
		`  RVKVV_DEFAULT = RVKVV_1_0_0_RC1,`
		`} riscvVCryptoVer;`

`crypto_absent`	`riscvCryptoSet`	By default, all Cryptographic Extension instruction subsets are implemented. This field is a bitmask allowing *unimplemented* subsets to be specified:
		`typedef enum riscvCryptoSetE {`
		`  // SCALAR SUBSETS`
		`  RVKS_Zbkb  = (1<<0),`
		`  RVKS_Zbkc  = (1<<1),`
		`  RVKS_Zbkx  = (1<<2),`
		`  RVKS_Zkr  = (1<<3),`
		`  RVKS_Zknd  = (1<<4),`
		`  RVKS_Zkne  = (1<<5),`
		`  RVKS_Zknh  = (1<<6),`
		`  RVKS_Zksed  = (1<<7),`
		`  RVKS_Zksh  = (1<<8),`
		`  RVKS_Zkb  = RVKS_Zbkb,`
		`  RVKS_Zkg  = RVKS_Zbkc,`
		`  // VECTOR SUBSETS`
		`  RVKS_Zvbb  = (1<<9),`
		`  RVKS_Zvbc  = (1<<10),`
		`  RVKS_Zvkg  = (1<<11),`
		`  RVKS_Zvknha = (1<<12),`
		`  RVKS_Zvknhb = (1<<13),`
		`  RVKS_Zvkned = (1<<14),`
		`  RVKS_Zvksed = (1<<15),`
		`  RVKS_Zvksh  = (1<<16),`
		`  RVKS_Zvkb  = RVKS_Zvbb`
		`} riscvCryptoSet;`
		The presence or absence of each *scalar* subset can be individually specified using parameters `Zbkb`, `Zbkc`, `Zbkx`, `Zkr`, `Zknd`, `Zkne`, `Zknh`, `Zksed` and `Zksh`. Deprecated parameters `Zkb` and `Zkg` are equivalent to `Zbkb` and `Zbkc`, respectively.

[2] The K bit must be specified in the `arch` field *even when not visible in the `misa` register* for this extension to be active

		The presence or absence of each *vector* subset (when V extension is present) can be individually specified using parameters `Zvbb`, `Zvbc`, `Zvkg`, `Zvknha`, `Zvknhb`, `Zvkned`, `Zvksed` and `Zvksh`. Parameter `Zvkb` is a deprecated alias for `Zvkb`.
`mnoise_undefined`	`Bool`	This specifies that the `mnoise` CSR is undefined, and that accesses to it will trap to Machine mode. If defined, it has address `0x7A9`.

5.14 Hypervisor Extension

Fields here are applicable when the Hypervisor Extension (H) is indicated as implemented by the processor `arch` field. All field defaults can be overridden using a model parameter of the same name unless otherwise indicated.

Field	Type	Description

`hyp_version`	`riscvHypVer`	This specifies the implemented Hypervisor Extension version, defined by the `riscvHypVer` enumeration:
		`typedef enum riscvHypVerE {`
		`  RVHV_0_6_1,  // 0.6.1`
		`  RVHV_1_0,  // 1.0`
		`  RVHV_DEFAULT = RVHV_1_0,`
		`} riscvHypVer;`

`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 `True`, this specifies that only pseudo-instructions are reported by `htinst`/`mtinst`; if `False`, then true instructions can be reported as well.
`fence_g_preserves_vs`	`Bool`	If `True`, this specifies that `HFENCE.GVMA` preserves cached VS-stage address translations; if `False`, then `HFENCE.GVMA` will invalidate all cached VS-stage address translations.
`csrMask.hvip`	`Uns64`	`hvip` CSR write mask (only bits 63:13 used). Use parameter `hvip_mask` to override this value.

5.15 DSP Extension

Fields here are applicable when the DSP Extension (P) is indicated as implemented by the processor `arch` field. All field defaults can be overridden using a model parameter of the same name, unless otherwise indicated.

Field	Type	Description

`dsp_version`	`riscvDSPVer`	This specifies the implemented DSP Extension version, defined by the `riscvDSPVer` enumeration:
		`typedef enum riscvDSPVerE {`
		`  RVDSPV_0_5_2,  // 0.5.2`
		`  RVDSPV_0_9_6,  // 0.9.6`
		`  RVDSPV_DEFAULT = RVDSPV_0_5_2,`
		`} riscvDSPVer;`

`dsp_absent`	`riscvDSPSet`	By default, all DSP Extension instruction subsets are implemented. This field is a bitmask allowing *unimplemented* subsets to be specified:
		`typedef enum riscvDSPSetE {`
		`  RVPS_Zpsfoperand = (1<<0),`
		`} riscvDSPSet;`
		The single entry `Zpsfoperand` is valid for RV32 only, and indicates whether instructions operating on register *pairs* are implemented. Its presence can by modified using a parameter of the same name.

5.16 Code Size Reduction Extension

Fields here are applicable when the Code Size Reduction Extension is implemented. All field defaults can be overridden using a model parameter of the same name, unless otherwise indicated. The parameters available for configuration of this extension depend on whether a *legacy* or *recent* specification version is configured.

Field	Type	Description

compress_version	riscvCompressVer	This specifies the implemented Code Size Reduction version, defined by the riscvCompressVer enumeration:
		`typedef enum riscvCompressVerE {`
		`  RVCV_NA_LEGACY,`
		`  RVCV_0_70_1,  // 0.70.1`
		`  RVCV_0_70_5,  // 0.70.5`
		`  RVCV_1_0_0_RC57, // 1.0.0-RC5.7`
		`  RVCV_1_0,  // 1.0 (ratified)`
		`  RVCV_DEFAULT = RVCV_1_0,`
		`} riscvCompressVer;`
		A value of RVCV_NA_LEGACY indicates that either the Code Size Reduction extension is absent or that a legacy version is in use, in which case subsets are separately-versioned (see below). Other versions do not not have separately-versioned subsets.

Zcea_version	riscvZceaVer	This specifies the implemented legacy Code Size Reduction Zcea subset Extension version, defined by the riscvZceaVer enumeration:
		`typedef enum riscvZceaVerE {`
		`  RVZCEA_NA,`
		`  RVZCEA_0_50_1,  // 0.50.1`
		`  RVZCEA_DEFAULT = RVZCEA_0_50_1,`
		`} riscvZceaVer;`
		This is ignored unless the legacy Code Size Reduction extension is configured.

Zceb_version	riscvZcebVer	This specifies the implemented legacy Code Size Reduction Zceb subset Extension version, defined by the riscvZcebVer enumeration:
		`typedef enum riscvZcebVerE {`
		`  RVZCEB_NA,`
		`  RVZCEB_0_50_1,  // 0.50.1`
		`  RVZCEB_DEFAULT = RVZCEB_0_50_1,`
		`} riscvZcebVer;`
		This is ignored unless the legacy Code Size Reduction extension is configured.

Zcee_version	riscvZceeVer	This specifies the implemented legacy Code Size Reduction Zcee subset Extension version, defined by the riscvZceeVer enumeration:
		`typedef enum riscvZceeVerE {`
		`  RVZCEE_NA,`
		`  RVZCEE_1_0_0_RC, // 1.0.0-RC`
		`  RVZCEE_DEFAULT = RVZCEE_1_0_0_RC,`
		`} riscvZceeVer;`
		This is ignored unless the legacy Code Size Reduction extension is configured.

compress_present	riscvCompressSet	This field is a bitmask allowing *implemented* Code Size Reduction Extension subsets to be specified:
		`typedef enum riscvCompressSetE {`
		`  // legacy values`
		`  RVCS_Zcea  = (1<<0),`
		`  RVCS_Zceb  = (1<<1),`
		`  RVCS_Zcee  = (1<<2),`
		`  // new values`
		`  RVCS_Zca  = (1<<3),`
		`  RVCS_Zcb  = (1<<4),`
		`  RVCS_Zcd  = (1<<5),`

		RVCS_Zcf = (1<<6), RVCS_Zcmb = (1<<7), RVCS_Zcmp = (1<<8), RVCS_Zcmpe = (1<<9), RVCS_Zcmt = (1<<10), } riscvCompressSet; Values for subsets Zcea, Zceb and Zcee are derived from the legacy version parameters above, if the legacy Code Size Reduction Extension is configured. Subset Zcd is an artifact used to indicate presence of compressed extension double-precision load/store instructions; its value is automatically maintained by the model. For other subsets, presence or absence can be individually specified using parameters Zca, Zcb, Zcf, Zcmb, Zcmp, Zcmpe and Zcmt, when the legacy Code Size Reduction extension is not configured.
csrMask.jvt	Uns64	jvt CSR write mask (Zcmt extension). Use parameter jvt_mask to override this value.

5.17 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 unless otherwise specified.

Field	Type	Description

debug_version	riscvDebugVer	This specifies the implemented Debug version, defined by the riscvDebugVer enumeration:
		`typedef enum riscvDebugVerE {`
		`  RVDBG_0_13_2,  // 0.13.2-DRAFT`
		`  RVDBG_0_14_0,  // 0.14.0-DRAFT`
		`  RVDBG_1_0_0,  // 1.0.0-STABLE`
		`  RVDBG_1_0,  // 1.0-STABLE`
		`  RVDBG_DEFAULT = RVDBG_1_0`
		`} riscvDebugVer;`
		Versions RVDBG_1_0_0 and RVDBG_1_0 correspond to two versions both called *1.0-STABLE*; They differ because in version RVDBG_1_0, the nmi field has moved to the itrigger view of tdata1, and behavior of that field has changed.

debug_mode	riscvDMMode	This field specifies how Debug mode is implemented. The possible behaviors are specified by the riscvDMMode enumeration:
		`typedef enum riscvDMModeE {`
		`  RVDM_NONE,`
		`  RVDM_VECTOR,`
		`  RVDM_INTERRUPT,`
		`  RVDM_HALT,`
		`  RVDM_INJECT`
		`} riscvDMMode;`
		See below for a detailed description of how this affects behavior.

debug_address	Uns64	This specifies the address to jump to on a debug event (when debug_mode is RVDM_VECTOR).
dexc_address	Uns64	This specifies the address to jump to on a debug exception (when debug_mode is RVDM_VECTOR).

debug_priority	riscvDPriority	This specifies relative priority of simultaneous trigger-after, step, execute-address, resethaltreq and haltreq breakpoint events, defined by the riscvDPriority enumeration:
		`typedef enum riscvDPriorityE {`
		`  RVDP_A_S_X_H,`
		`  RVDP_A_S_H_X,`
		`  RVDP_A_H_S_X,`
		`  RVDP_H_A_S_X,`
		`  RVDP_DEFAULT = RVDP_H_A_S_X,`
		`} riscvDPriority;`
		Value RVDP_A_S_X_H indicates the priority order: trigger-after, step, execute-address, resethaltreq, haltreq (in highest to lowest order).
		Value RVDP_A_S_H_X indicates the priority order: trigger-after, step, resethaltreq, haltreq, execute-address.
		Value RVDP_A_H_S_X indicates the priority order: trigger-after, resethaltreq, haltreq, step, execute-address.
		Value RVDP_H_A_S_X indicates the priority order: resethaltreq, haltreq, trigger-after, step, execute-address.
		See section 5.17.4 for more information.

debug_eret_mode	riscvDERETMode	This specifies the required behavior when MRET, SRET or URET instructions are executed in Debug mode. The possible behaviors are specified by the riscvDERETMode enumeration:
		`typedef enum riscvDERETModeE {`
		`  RVDRM_NOP,  // treat as NOP`
		`  RVDRM_JUMP, // jump to dexc_address`
		`  RVDRM_TRAP, // trap to dexc_address`

		} riscvDERETMode For RVDRM_JUMP, the instruction is considered to have retired. For RVDRM_TRAP, the instruction is *not* considered to have retired.
debug_ctrlt_illegal	Bool	This specifies that control transfer instructions will cause Illegal Instruction traps in Debug mode. If False, the instructions are executed normally.
debug_auipc_illegal	Bool	This specifies that auipc instructions will cause Illegal Instruction traps in Debug mode. If False, the instructions are executed normally.
dscratch0_undefined	Bool	This specifies that the dscratch0 register is undefined, and that accesses to it will trap to Machine mode.
dscratch1_undefined	Bool	This specifies that the dscratch1 register is undefined, and that accesses to it will trap to Machine mode.
no_resethaltreq	Bool	If False, report code 5 in dcsr when resethaltreq is applied at reset, otherwise (if True) report code 3.
defer_step_bug	Bool	The 0.13.2 Debug specification was ambiguous about required behavior when a debug single step is attempted while there is a pending exception that will be taken before the non-debug-mode instruction can execute. By default, the model will take the step breakpoint *before* the first trap handler instruction is executed in this case, which was the intention of the specification and explicitly stated in the 0.14.0 draft. Setting this configuration option to True enables an alternative behavior where the step breakpoint is instead taken *after the first trap handler instruction has executed*. This may be required for compatibility with legacy hardware where the specification was misinterpreted. There is no parameter to override this configuration option and it should be explicitly set in the configuration of any variant that requires it.

5.17.1 Debug Mode Behaviors

Field debug_mode can be used to specify four different simulation behaviors, as follows:

1. If set to value RVDM_VECTOR, then operations that would cause entry to Debug mode result in the processor jumping to the address specified by the debug_address field. It will execute at this address, in Debug mode, until a dret instruction causes return to non-Debug mode. Any exception generated during this execution will cause a jump to the address specified by the dexc_address field.
2. If set to value RVDM_INTERRUPT, then operations that would cause entry to Debug mode result in the processor simulation call (e.g. opProcessorSimulate) returning, with a stop reason of OP_SR_INTERRUPT. In this usage scenario, the Debug Module is implemented in the simulation harness.
3. If set to value RVDM_HALT, then operations that would cause entry to Debug mode result in the processor halting. Depending on the simulation environment, this might cause a return from the simulation call with a stop reason of OP_SR_HALT, or debug mode might be implemented by another platform component which then restarts the debugged processor again.
4. If set to value RVDM_INJECT, then operations that would cause entry to Debug mode result in the processor continuing to execute from the current address in Debug mode. The harness should detect that Debug mode has been entered by monitoring the DM integration support register, and inject Debug-mode

instructions one at a time using function `opProcessorSimulateInstruction`. Debug mode is exited by either an explicit write of `False` to the `DM` register or by execution of an injected `dret` instruction, as described in section 5.17.3 below.

5.17.2 Debug State Entry

The specification does not define how Debug mode is implemented. In this model, Debug mode is enabled by a Boolean pseudo-register, `DM`. When `DM` is `True`, the processor is in Debug mode. When `DM` is `False`, mode is defined by `mstatus` in the usual way. Entry to Debug mode can be performed in any of these ways:

1. By writing `True` to register `DM` (e.g. using `opProcessorRegWrite`) followed by simulation of at least one cycle (e.g. using `opProcessorSimulate`) - in this case, `dcsr.cause` will report a cause of `trigger` (2);
2. By writing a 1 then 0 to net `haltreq` (using `opNetWrite`) followed by simulation of at least one cycle (e.g. using `opProcessorSimulate`) - in this case, `dcsr.cause` will report a cause of `haltreq` (3);
3. By writing a 1 to net `resethaltreq` (using `opNetWrite`) while the `reset` signal undergoes a negative edge transition, followed by simulation of at least one cycle (e.g. using `opProcessorSimulate`) - in this case, `dcsr.cause` will report a cause of `resethaltreq` (5) or `haltreq` (3), depending on the value of field `no_resethaltreq`;
4. By executing an `ebreak` instruction when Debug mode entry for the current processor mode is enabled by `dcsr.ebreakm`, `dcsr.ebreaks` or `dcsr.ebreaku` - in this case, `dcsr.cause` will report a cause of `ebreak` (1);
5. By executing a single instruction when Debug mode entry for the current processor mode is enabled by `dcsr.step` - in this case, `dcsr.cause` will report a cause of `step` (4);
6. By a Trigger Module trigger, when that trigger is configured to enter Debug mode - in this case, `dcsr.cause` will report a cause of `trigger` (2).

In all cases, the processor will save required state in `dpc` and `dcsr` and then perform actions described above, depending in the value of the `debug_mode` field.

5.17.3 Debug State Exit

Exit from Debug mode can be performed in either of these ways:

1. By writing `False` to register `DM` (e.g. using `opProcessorRegWrite`) followed by simulation of at least one cycle (e.g. using `opProcessorSimulate`);
2. By executing a `dret` instruction when Debug mode.

In both cases, the processor will perform the steps described in section 4.6 (Resume) of the Debug Specification.

5.17.4 `haltreq` Signal and `debug_priority` Field

There are various events that can cause entry into Debug mode:

1. The hart might execute a special instruction, `EBREAK`, which implements a software breakpoint;
2. The hart might enter Debug mode because of a Trigger Module event;
3. The processor might be forced into Debug mode by a special external signal, `haltreq`.
4. The hart might be configured to perform a single step by Debug mode. When single-stepping, the hart returns from Debug mode, executes a single instruction in non-debug mode, and then re-enters Debug mode automatically.

Apart from the `haltreq` signal, these events are all synchronous. The type of event causing Debug mode entry is reported to Debug mode.

The model prioritizes *synchronous* events in this order:

1. Any Trigger Module pre-trigger is handled first (this will typically be an execute address trigger);
2. Explicit entry by `EBREAK` has next highest priority;
3. Any Trigger Module post-trigger is handled next;
4. If the instruction was single-stepped, that is handled last.

This means, for example, that if single-stepping an instruction which also causes a Trigger Module post-trigger, it is the post-trigger event that is reported to Debug mode, not the step event.

The Debug Specification is not clear about the priority of the asynchronous `haltreq` signal with respect to the other priorities above; for example, it doesn't define what happens if (for example) the `haltreq` signal is raised while single-stepping an instruction which also causes a Trigger Module post-trigger. The `debug_priority` field allows the model to be configured so that the `haltreq` event priority can be programmed with any one of four priorities:

`RVDP_A_S_X_H`
`haltreq` lowest priority.

`RVDP_A_S_H_X`
`haltreq` priority above execute address pre-trigger on the next instruction, but lower priority than post-triggers and single-step.

`RVDP_A_H_S_X`
`haltreq` lower priority than post-triggers, but higher priority than execute address pre-trigger on the next instruction and single-step.

`RVDP_H_A_S_X`
`haltreq` highest priority.

5.18 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 unless otherwise indicated.

Field	Type	Description
trigger_num	Uns32	This field specifies how many trigger registers are implemented. If trigger_num is 0, the trigger module is not configured.
mask_tselect	Bool	When true, writes to tselect will be masked based on the value of trigger_num, which determines the number of bits implemented for the tselect register. Attempts to write values greater than the trigger_num setting will be ignored, regardless of whether they have been masked.
tinfo	Uns32	This specifies the implemented trigger types (bits 15:0) and Sdtrig version (bits 31:24). For example, if trigger types 5 and 6 are supported and Sdtrig version is 1, the value should be 0x01000060.
trigger_match	Uns32	This is bitmask specifying legal values for the match field for address triggers (types 2 and 6). For example, if match values 0, 2 and 3 are supported, trigger_match should be specified as 0x000d.
mcontext_bits	Uns32	This specifies the number of implemented bits in the mcontext register.
scontext_bits	Uns32	This specifies the number of implemented bits in the scontext register.
mvalue_bits	Uns32	This specifies the number of implemented bits in the textra32/textra64 mvalue field. If zero, the mselect 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 textra32/textra64 svalue field. If zero, the sselect field in the same register is tied to zero.
mcontrol_maskmax	Uns32	This specifies the value of the mcontrol.maskmax field.
tdata2_undefined	Bool	This specifies that the tdata2 register is undefined, and that accesses to it will trap to Machine mode.
tdata3_undefined	Bool	This specifies that the tdata3 register is undefined, and that accesses to it will trap to Machine mode.
tinfo_undefined	Bool	This specifies that the tinfo register is undefined, and that accesses to it will trap to Machine mode.
tcontrol_undefined	Bool	This specifies that the tcontrol register is undefined, and that accesses to it will trap to Machine mode.
mcontext_undefined	Bool	This specifies that the mcontext register is undefined, and that accesses to it will trap to Machine mode.
scontext_undefined	Bool	This specifies that the scontext register is undefined, and that accesses to it will trap to Machine mode.
mscontext_undefined	Bool	This specifies that the mscontext 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 hcontext 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 hit bits in tdata1 are unimplemented.
no_sselect_2	Bool	If Supervisor mode is present, this specifies whether the

		sselect field in textra32/textra64 registers is unable to hold value 2 (indicating match by ASID is not allowed).
csrMask.tdata1	Uns64	tdata1 CSR write mask. Use parameter tdata1_mask to override this value.
chain_tval	riscvChainTVAL	When chained triggers are implemented, the Debug Specification does not state which trigger supplies the value reported in xtval when the trigger causes a breakpoint exception. This field allows one of four possible behaviors to be specified: RVCT_FIRST, RVCT_LAST, RVCT_FIRST_NON_EPC and RVCT_LAST_NON_EPC. These have the following meanings: RVCT_FIRST: report value from the first trigger in the chain that matches. RVCT_LAST: report value from the last trigger in the chain that matches. RVCT_FIRST_NON_EPC: report value from the first trigger in the chain that matches, but prefer values that are not the current PC (because this would be redundant with xepc). RVCT_LAST_NON_EPC: report value from the last trigger in the chain that matches, but prefer values that are not the current PC (because this would be redundant with xepc).

5.19 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
`numHarts`	`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.
`members`	`const char **`	For a multicore variant (`numHarts!=0`), a null value for this parameter specifies that the processor is an SMP processor. If `numHarts!=0` and `members` is non-null, then `members` must be a list of `numHarts` strings, each string specifying the variant name of a *cluster member*. 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 nth hart is of the type corresponding to the nth element of `members`.

5.20 CSR Index Numbers

CSR numbers are defined by the Privileged Specification and other extension specifications. Sometimes, CSR numbers are reallocated as specifications evolve; for example, in the RNMI specification (which is not yet ratified), CSR `mnstatus` was originally assigned number `0x353`, but in more recent versions of the specification it has been allocated number `0x744` instead. These changes in CSR numbers are automatically handled by the base model if the correct extension version number is selected.

Occasionally, variants are created that use *non-standard* CSR numbers. For example, some SiFive variants with interim RNMI implementations use number `0x743` for `mnstatus`, which does not correspond with any specification version. To implement this, the `CSR_remap` field in the configuration can be used. If non-`NULL`, this field is a string containing comma-separated entries of the form:

```
<csrName>=<number>
```

For SiFive cores implementing an interim RNMI extension, this field is set to:

```
"mnstatus=0x743"
```

This indicates that number `0x743` must be used for `mnstatus` instead of the default number (`0x744`). CSR number remapping can be overridden using string parameter `CSR_remap` if required.

5.21 CSR Initial 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
`csr.mvendorid`	`Uns64`	`mvendorid` CSR value.
`csr.marchid`	`Uns64`	`marchid` CSR value.
`csr.mimpid`	`Uns64`	`mimpid` CSR value.
`csr.mhartid`	`Uns64`	`mhartid` CSR value.
`csr.mconfigptr`	`Uns64`	`mconfigptr` CSR value. Not used for Privileged Architecture versions 1.11 and earlier.
`csr.mtvec`	`Uns64`	`mtvec` CSR initial value.
`csr.mstatus`	`Uns64`	`mstatus` CSR initial value. Fields `mstatus.FS` and `mstatus.VS` can be overridden using parameters `mstatus_FS` and `mstatus_VS`, respectively.

5.22 CSR Masks

The `csrMask` field defines write masks for some CSRs. If the fields are zero, default values are used as described in the *Default if Zero* column below. All field defaults can be overridden using parameters with `_mask` suffix; for example use parameter `mtvec_mask` to override the value of `csrMask.mtvec`.

Field	Type	Description	Default if Zero
`csrMask.mtvec`	`Uns64`	`mtvec` CSR write mask.	all bits writable except `mtvec[1:0]`, which depend on basic IC/CLIC presence
`csrMask.stvec`	`Uns64`	`stvec` CSR write mask.	all bits writable except `stvec[1:0]`, which depend on basic IC/CLIC presence
`csrMask.utvec`	`Uns64`	`utvec` CSR write mask (N extension).	all bits writable except `utvec[1:0]`, which depend on basic IC/CLIC presence
`csrMask.mtvt`	`Uns64`	`mtvt` CSR write mask (CLIC).	all bits writable except `mtvt[5:0]`
`csrMask.stvt`	`Uns64`	`stvt` CSR write mask (CLIC).	all bits writable except `stvt[5:0]`
`csrMask.utvt`	`Uns64`	`utvt` CSR write mask (CLIC and N extension).	all bits writable except `utvt[5:0]`
`csrMask.jvt`	`Uns64`	`jvt` CSR write mask (`Zcmt` extension).	all bits writable except `jvt[5:0]`
`csrMask.tdata1`	`Uns64`	`tdata1` CSR write mask (Trigger Module).	all bits writable
`csrMask.mip`	`Uns64`	`mip` CSR write mask	`0x337`
`csrMask.sip`	`Uns64`	`sip` CSR write mask	`0x103`
`csrMask.uip`	`Uns64`	`uip` CSR write mask (N extension)	`0x001`
`csrMask.hip`	`Uns64`	`hip` CSR write mask (H extension)	`0x004`
`csrMask.mvien`	`Uns64`	`mvien` CSR write mask (`Smaia` extension, only bits 63:13 used)	`0x0`
`csrMask.mvip`	`Uns64`	`mvip` CSR write mask (`Smaia` extension, only bits 63:13 used)	`0x0`
`csrMask.hvien`	`Uns64`	`hvien` CSR write mask (`Smaia` extension with H extension, only bits 63:13 used)	`0x0`
`csrMask.hvip`	`Uns64`	`hvip` CSR write mask (H extension, only bits 63:13 used)	`0x0`
`csrMask.envcfg`	`Uns64`	`menvcfg/henvcfg/senvcfg` write mask. Not used for Privileged Architecture versions 1.11 and earlier.	`0x80000000000000f1`

5.23 Unimplemented Instructions

Sometimes a RISC-V processor variant will implement only a *subset* of the instructions in a standard extension. For example, a processor might implement all floating-point instructions except divide instructions, or only the LR/SC instructions from the atomic extension. This behavior could be implemented in an extended model by decoding the unimplemented instructions and reimplementing them as Illegal Instructions (see section 6). An easier alternative is to use the `unimplementedInstr` field in the configuration structure.

If this field is non-NULL, it must be a null-terminated list of `riscvIType` enumeration members for instructions that are unimplemented. This enumeration is declared in file `riscvDecodeTypes.h` and has one entry for each fundamental instruction type recognized by the instruction decoder. As an example, if a variant implements the atomic (A) extension but only the LR/SC instructions from that, and the floating-point extensions (F and D) but not DIV/SQRT/FMAC instructions, then this could be specified as follows:

```
//
// Definition of unimplemented instructions
//
static const riscvIType unimplementedInstructions[] = {

    // Unimplemented atomics
    RV_IT_AMOADD_R,
    RV_IT_AMOAND_R,
    RV_IT_AMOMAX_R,
    RV_IT_AMOMAXU_R,
    RV_IT_AMOMIN_R,
    RV_IT_AMOMINU_R,
    RV_IT_AMOOR_R,
    RV_IT_AMOSWAP_R,
    RV_IT_AMOXOR_R,

    // Unimplemented floating-point instructions
    RV_IT_FDIV_R,
    RV_IT_FSQRT_R,
    RV_IT_FMADD_R4,
    RV_IT_FMSUB_R4,
    RV_IT_FNMADD_R4,
    RV_IT_FNMSUB_R4,

    // list terminator
    0
};

static const riscvConfig configList[] = {

    {
        .name               = "RV32X",
        .arch               = ISA_U|RV32GC|ISA_X,
        . . . fields omitted . . .
        .unimplementedInstr = &unimplementedInstructions[0]
    },

    {0} // null terminator
};
```

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, // shared object type
    .interceptType = VMI_IT_PROC_EXTENSION, // intercept 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        (R-Type)
    RVIP_RD_RS1_SI,         // op   xd, xs1, imm        (I-Type)
    RVIP_RD_RS1_SHIFT,      // op   xd, xs1, shift      (I-Type - 5 or 6 bit shift)
    RVIP_BASE_RS2_OFFSET,   // op   base, xs2, offset   (S-Type)
    RVIP_RS1_RS2_OFFSET,    // op   xs1, xs2, offset    (B-Type)
    RVIP_RD_SI,             // op   xd, imm             (U-Type)
    RVIP_RD_OFFSET,         // op   xd, offset          (J-Type)
    RVIP_RD_RS1_RS2_RS3,    // op   xd, xs1, xs2, xs3   (R4-Type)
    RVIP_RD_RS1_RS3_SHIFT,  // op   xd, xs1, xs2, shift (Non-Standard)

                            // 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 13 for a detailed description of each of these.

5. 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.
6. 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:

1. The RISC-V processor object;
2. 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 the 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 feature 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 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

This example can be found in

```
Examples/Models/Processor/FeatureUsage/RISCV_ExtendedProcessor
```

Using the assembler test program

```
asmtest/test_ex_ch6.S
```

The code below shows the main part of the simple assembler program that can be used to exercise the extension:

```
//
// 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 -> 00000000000ab000
Info 5: 'iss/cpu0', 0x000000008000240c(START_TEST+c): Machine bbd4849b addiw   s1,s1,-
1091
Info   s1 00000000000ab000 -> 00000000000aabbd
Info 6: 'iss/cpu0', 0x0000000080002410(START_TEST+10): Machine 00c49493 slli    s1,s1,0xc
Info   s1 00000000000aabbd -> 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 -> fffffffff9aa5b88
Info 10: 'iss/cpu0', 0x0000000080002420(START_TEST+20): Machine 0294290b chacha20qr3
s2,s0,s1
Info   s2 fffffffff9aa5b88 -> 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 writable). 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 writable). 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_ro1);
    } 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
    .interceptType = VMI_IT_PROC_EXTENSION, // intercept 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 `addCSRsCSR.h` like this:

```
typedef struct addCSRsCSRsS {
    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_ro1);       // 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 riscvCSRAttrsS {

    const char        *name;            // register name
    const char        *desc;            // register description
    vmiosObjectP       object;          // custom extension
    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
    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
    vmiReg             reg;             // register
    vmiReg             writeMaskV;      // configuration-dependent write mask
    Uns32              writeMaskC32;    // constant 32-bit write mask
    Uns64              writeMaskC64;    // constant 64-bit write mask

    riscvCSRStateenBit Smstateen    :8; // whether xstateen-controlled access
    riscvCSRTrap       trap         :2; // whether trapped
    riscvCSRTrace      noTraceChange:2; // trace mode
    Bool               wEndBlock    :1; // whether write terminates this block
    Bool               wEndRM       :1; // whether write invalidates RM assumption
    Bool               noSaveRestore:1; // whether to exclude from save/restore
    Bool               writeRd      :1; // whether write updates Rd
    Bool               aliasV       :1; // whether CSR has virtual alias
    Bool               undefined    :1; // whether CSR is undefined
    Bool               forceRO      :1; // type is RO even if write CB defined

} riscvCSRAttrs;
```

Usage of this type is described in detail in Appendix 0; this chapter describes some common use cases.

Field `csrs` is an array of these structures, of size `XCSR_ID(LAST)`, which is a member of the `extCSRId` enumeration in `addCSRsExtensions.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(addCSRsConstructor) {

    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
    addCSRsCSRInit(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. This callback is called *after* all base model reset behavior has been performed, so it can update standard CSR values to reflect custom behavior if required. In this case, the notifier writes reset values to various custom CSRs:

```
static RISCV_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_ro1, object->config.csr.custom_ro1.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_ro1` using the *default* value from the configuration:

```
    WR_XCSR(object, custom_ro1, object->config.csr.custom_ro1.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_ro1,    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,_TRAP, _DESC, _RCB, _RWCB, _WCB \
) [XCSR_ID(_ID)] = { \
    .extension = _EXT,                                \
    .baseAttrs = {                                    \
        name          : #_ID,                         \
        desc          : _DESC,                        \
        csrNum        : _NUM,                         \
        arch          : _ARCH,                        \
        wEndBlock     : _ENDB,                        \
        wEndRM        : _ENDRM,                       \
        noTraceChange : _NOTR,                        \
        trap          : _TRAP,                        \
        readCB        : _RCB,                         \
        readWriteCB   : _RWCB,                        \
        writeCB       : _WCB,                         \
        reg           : XCSR_REG_MT(_ID),             \
        writeMaskC32  : WM32_##_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: see Appendix 0 for full details. 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: specifies whether changes to the CSR are reported when trace change is enabled (`RCSRT_YES` indicates always reported, `RCSRT_NO` indicates never reported, `RCSRT_VOLATILE` indicates only reported if `traceVolatile` parameter is set in the base model).
8. A *trap* attribute: whether accesses to the CSR are trapped by `mstatus.TVM=1` (if *trap* is `CSRT_TVM`) or by `hvictl.VTI=1` (if *trap* is `CSRT_VTI`).

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 such as `mip` 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_ro1` are all implemented as plain registers with defined write masks (see section 7.1).

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 RISCV_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,_TRAP, _DESC, _RCB, _RWCB, _WCB \
) [XCSR_ID(_ID)] = { \
    .extension = _EXT,                              \
    .baseAttrs = {                                  \
        name          : #_ID,                       \
        desc          : _DESC,                      \
        csrNum        : _NUM,                       \
        arch          : _ARCH,                      \
        wEndBlock     : _ENDB,                      \
        wEndRM        : _ENDRM,                     \
        noTraceChange : _NOTR,                      \
        trap          : _TRAP,                      \
        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 to hold the value 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 RISCV_WRITE_BASE_CSR_FN(_NAME) Uns64 _NAME( \
    riscvP     riscv,           \
    riscvCSRId id,              \
    Uns64      newValue         \
)
typedef RISCV_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

This example can be found in

`Examples/Models/Processor/FeatureUsage/RISCV_ExtendedProcessor`

Using the assembler test program

`asmtest/test_ex_ch7.S`

The code below shows the main part of the simple assembler program that can be used to exercise the extension:

```
// 64-bit processor load mnemonics
#define SX sd
#define LX ld


.macro SETUP_M_HANDLER _BASE=defaultMHandler, _SCRATCH=defaultMScratch

        la          t0, \_BASE
        csrw        mtvec, t0
        la          t0, \_SCRATCH
        csrw        mscratch, t0
.endm

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
        csrw    0xBC0, s1
        csrw    0xBC1, s1
        csrw    0xBC2, s1
        csrw    0xBC3, s1
        csrw    0xFC0, s1

        // test mstatus with custom field at bit 30
        csrr    s1, mstatus
        li      s1, -1
        csrw    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
        csrw        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           \
    --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 csrw    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 csrw
mscratch,t0
Info   mscratch 0000000000000000 -> 0000000080001a00
Info 8: 'iss/cpu0', 0x0000000080002418(START_TEST+18): Machine bc0024f3 csrr
s1,custom_rw1_32
Info 9: 'iss/cpu0', 0x000000008000241c(START_TEST+1c): Machine bc1024f3 csrr
s1,custom_rw2_64
Info   s1 0000000000000000 -> 0000000000000034
Info 10: 'iss/cpu0', 0x0000000080002420(START_TEST+20): Machine bc2024f3 csrr
s1,custom_rw3_32
Info   s1 0000000000000034 -> 0000000000000000
```

```
Info 11: 'iss/cpu0', 0x0000000080002424(START_TEST+24): Machine bc3024f3 csrr
s1,custom_rw4_64
Info 12: 'iss/cpu0', 0x0000000080002428(START_TEST+28): Machine fc0024f3 csrr
s1,custom_ro1
Info   s1 0000000000000000 -> 0000000012345678
Info 13: 'iss/cpu0', 0x000000008000242c(START_TEST+2c): Machine 54fd     li      s1,-1
Info   s1 0000000012345678 -> ffffffffffffffff
Info 14: 'iss/cpu0', 0x000000008000242e(START_TEST+2e): Machine bc049073 csrw
custom_rw1_32,s1
Info   custom_rw1_32 0000000000000000 -> 00000000ff00ff
Info 15: 'iss/cpu0', 0x0000000080002432(START_TEST+32): Machine bc149073 csrw
custom_rw2_64,s1
Info   custom_rw2_64 0000000000000034 -> ff00ff0000ff00ff
Info 16: 'iss/cpu0', 0x0000000080002436(START_TEST+36): Machine bc249073 csrw
custom_rw3_32,s1
Info   custom_rw3_32 0000000000000000 -> ffffffffffffffff
Info 17: 'iss/cpu0', 0x000000008000243a(START_TEST+3a): Machine bc349073 csrw
custom_rw4_64,s1
Info   custom_rw4_64 0000000000000000 -> ffffffffffffffff
Info 18: 'iss/cpu0', 0x000000008000243e(START_TEST+3e): Machine fc049073 csrw
custom_ro1,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 csrrw
gp,mscratch,gp
Info   gp 0000000000000000 -> 0000000080001a00
Info   mscratch 0000000080001a00 -> 0000000000000000
Info 20: 'iss/cpu0', 0x0000000080002484(customMHandler+4): Machine 00a1b023 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 csrr
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 csrr
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 csrw
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 csrrw
gp,mscratch,gp
Info   gp 0000000080001a00 -> 0000000000000000
Info   mscratch 0000000000000000 -> 0000000080001a00
```

```
Info 35: 'iss/cpu0', 0x00000000800024b4(customMHandler+34): Machine 30200073 mret
Info   mstatus 0000000200001800 -> 0000000200000080
Info 36: 'iss/cpu0', 0x0000000080002442(START_TEST+42): Machine 300024f3 csrr
s1,mstatus
Info   s1 ffffffffffffffff -> 0000000200000080
Info 37: 'iss/cpu0', 0x0000000080002446(START_TEST+46): Machine 54fd     li      s1,-1
Info   s1 0000000200000080 -> ffffffffffffffff
Info 38: 'iss/cpu0', 0x0000000080002448(START_TEST+48): Machine 30049073 csrw
mstatus,s1
Info   mstatus 0000000200000080 -> 8000000240227888
Info 39: 'iss/cpu0', 0x000000008000244c(START_TEST+4c): Machine 300024f3 csrr
s1,mstatus
Info   s1 ffffffffffffffff -> 8000000240227888
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).

7.6 General CSR Reset Template Code

Section 7.4 showed how the reset notifier can be used to reset extension model CSRs. This notifier can also be used to reset *standard* CSRs if required, using one of two base model interface service functions, depending on requirements:

`writeCSR`
This will write a CSR using the extension model view (if there is one) or the base model view (if not). The CSR is identified by *number*.

`writeBaseCSR`
This will write a CSR using the base model view, bypassing any extension model view. The CSR is identified by `riscvCSRId` value (defined in file `riscvCSR.h` in the base model).

Both these functions apply write masking and any CSR write callbacks, so they won't normally modify fields that wouldn't be writable by a CSR update instruction. Some CSRs have fields that can only be modified by hardware and are *read-only* for normal CSR access; these CSR fields can be unlocked by setting the `riscv->artifactAccess` Boolean to `True` while performing the write.

This example is a template showing how the standard Trigger Module CSRs can be reset in the reset notifier:

```
static RISCV_RESET_NOTIFIER_FN(CSRReset) {

    Uns32 trigger_num = riscv->configInfo.trigger_num;
    Uns32 i;
```

```
    // enable full write access to CSRs
    riscv->artifactAccess = True;

    // reset trigger module CSRs
    for(i=0; i<trigger_num; i++) {
        riscv->cb.writeBaseCSR(riscv, CSR_ID(tselect), i);
        riscv->cb.writeBaseCSR(riscv, CSR_ID(tdata1),  0);
        riscv->cb.writeBaseCSR(riscv, CSR_ID(tdata2),  0);
        riscv->cb.writeBaseCSR(riscv, CSR_ID(tdata3),  0);
    }

    // reset tselect and tcontrol (not banked)
    riscv->cb.writeBaseCSR(riscv, CSR_ID(tselect),  0);
    riscv->cb.writeBaseCSR(riscv, CSR_ID(tcontrol), 0);

    // disable full write access to CSRs
    riscv->artifactAccess = False;
}
```

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
    .interceptType = VMI_IT_PROC_EXTENSION, // intercept 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 RISCV_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 RISCV_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 `RISCV_TRAP_NOTIFIER_FN` macro defined in `riscvModelCallbacks.h` in the base model:

```
#define RISCV_TRAP_NOTIFIER_FN(_NAME) void _NAME( \
    riscvP    riscv,            \
    riscvMode mode,             \
    void     *clientData        \
)
typedef RISCV_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

This example can be found in
`Examples/Models/Processor/FeatureUsage/RISCV_ExtendedProcessor`

Using the assembler test program
`asmtest/test_ex_ch8.S`

The code below shows the main part of the simple assembler program that can be used to exercise the extension:

```
#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 csrw
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 00a1b023 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 csrw
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 -> ffffffffffffffff
Info 27: 'iss/cpu0', 0x000000008000241e(START_TEST+1e): Machine 30241073 csrw
medeleg,s0
Info   medeleg 0000000000000000 -> 000000000000b3ff
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
    .interceptType = VMI_IT_PROC_EXTENSION, // intercept 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` is 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 {

    // this is the lowest architectural priority
    riscv_E_MinPriority        = -120,

    // low-priority local interrupt default priorities when AIA present
    riscv_E_Local32Priority    = riscv_E_MinPriority,
    riscv_E_Local16Priority    = -110,
    riscv_E_Local33Priority    = -100,
    riscv_E_Local34Priority    = -90,
    riscv_E_Local17Priority    = -80,
    riscv_E_Local35Priority    = -70,
    riscv_E_Local36Priority    = -60,
    riscv_E_Local18Priority    = -50,
    riscv_E_Local37Priority    = -40,
    riscv_E_Local38Priority    = -30,
    riscv_E_Local19Priority    = -20,
    riscv_E_Local39Priority    = -10,

    // standard major interrupts (0-15)
    riscv_E_UTimerPriority     = 10,
    riscv_E_USWPriority        = 20,
    riscv_E_UExternalPriority  = 30,
    riscv_E_VSTimerPriority    = 40,
    riscv_E_VSSWPriority       = 50,
    riscv_E_VSExternalPriority = 60,
    riscv_E_SGEIPriority       = 70,
    riscv_E_STimerPriority     = 80,
    riscv_E_SSWPriority        = 90,
    riscv_E_SExternalPriority  = 100,
    riscv_E_MTimerPriority     = 110,
    riscv_E_MSWPriority        = 120,
    riscv_E_MExternalPriority  = 130,

    // high-priority local interrupt default priorities when AIA present
```

```
    riscv_E_Local40Priority    = 140,
    riscv_E_Local20Priority    = 150,
    riscv_E_Local41Priority    = 160,
    riscv_E_Local42Priority    = 170,
    riscv_E_Local21Priority    = 180,
    riscv_E_Local43Priority    = 190,
    riscv_E_Local44Priority    = 200,
    riscv_E_Local22Priority    = 210,
    riscv_E_Local45Priority    = 220,
    riscv_E_Local46Priority    = 230,
    riscv_E_Local23Priority    = 240,
    riscv_E_Local47Priority    = 250,

    // local interrupt default priorities when AIA absent
    riscv_E_LocalPriority      = 260

} 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 that 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 RISCV_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`.

This example can be found in
`Examples/Models/Processor/FeatureUsage/RISCV_ExtendedProcessor`

Using the assembler test program
`asmtest/test_ex_ch9.S`

The code below shows the main part of the simple assembler program that can be used to exercise the extension:

```
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                         \
    --traceshowicount                   \
    --addressbits       32              \
    --processorvendor   vendor.com      \
    --processorname     riscv           \
    --variant           RV64X           \
    --override          iss/cpu0/add_Extensions="S" \
    --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 -> ffffffffffffffff
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
    .interceptType = VMI_IT_PROC_EXTENSION, // intercept 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
    vmidDecodeTableP 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
} fifoCSRs;
```

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

```
typedef struct vmiosObjectS {

    . . . lines omitted . . .

    // extension CSR info
    fifoCSRs         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 withing 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 VMIOS_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 was 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
    .interceptType = VMI_IT_PROC_EXTENSION, // intercept 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_UNS32_PARAM(diagnosticlevel);
    VMI_UNS32_PARAM(features);
} formalValues, *formalValuesP;

// Parameter table
static vmiParameter parameters[] = {
    VMI_UNS32_PARAM_SPEC(formalValues, diagnosticlevel,       0, 0,       3, "Override
the initial diagnostic level"),
    VMI_UNS32_PARAM_SPEC(formalValues, features,         EXT_ALL, 1, EXT_ALL, "Override
the configured variant features"),
    { 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 VMIOS_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);

    // add extension registers
    addExtRegs(object);

    // 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->features;

    // 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 VMIOS_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 `features` parameters are used to modify the initial extension configuration.

11.4 Extension Registers

Extension `tmExtensions` adds a single read-only extension register to the base model. This register is not a CSR: see the next section for information about adding CSRs.

The register added is called `TM` and holds the current value of the transaction mode state for the processor. The register is added by function `addExtRegs`, called in the constructor:

```
static void addExtRegs(vmiosObjectP object) {

    riscvP riscv = object->riscv;

    // create description of read-only transaction mode register
    vmiRegInfo tmReg = {
        name        : "TM",
        description : "Transaction mode state",
        bits        : 8,
        gdbIndex    : 0,
        access      : vmi_RA_R,
        raw         : vmimtGetExtReg((vmiProcessorP)riscv, &object->tmStatus)
    };

    // add register description
    riscv->cb.newExtReg(riscv, &tmReg);
}
```

The function first defines a `vmiRegInfo` object describing register `TM`. In this case, the register value is simply the value of field `tmStatus` in the extension object. More complex registers may have read and write callbacks to form and assign their value: see the *OVP Processor Modeling Guide* for more information about the use of `vmiRegInfo` objects to describe registers.

The register is added to the processor external register view using the `newExtReg` interface function:

```
    riscv->cb.newExtReg(riscv, &tmReg);
```

This function can be called as many times as needed to add extension registers. Each added register should be given a unique name and index (`gdbIndex`). When added to the processor, the given index number will be modified using bitwise-or with `0x80000000` to ensure extension register indices do not conflict with base model register indices.

11.5 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
} tmCSRs;
```

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.

11.6 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 RISCV_IASSWITCH_FN(_NAME) void _NAME( \
    riscvP         riscv,       \
    vmiIASRunState state,       \
    void          *clientData   \
)
typedef RISCV_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.

11.7 Transactional Load and Store Functions (`riscvTLoad` and `riscvTStore`)

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

```
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 . . .
}
```

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 RISCV_TLOAD_FN(_NAME) void _NAME( \
    riscvP riscv,               \
    void  *buffer,              \
    Addr   VA,                  \
    Uns32  bytes,               \
    void  *clientData           \
)
typedef RISCV_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 RISCV_TSTORE_FN(_NAME) void _NAME( \
    riscvP      riscv,          \
```

```
    const void *buffer,         \
    Addr        VA,             \
    Uns32       bytes,          \
    void       *clientData      \
)
typedef RISCV_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 11.12 and the model source for a detailed example if required.

11.8 Trap and Exception Return Notifiers (`riscvTrapNotifier` and `riscvERETNotifier`)

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

```
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 . . .
}
```

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

```
static RISCV_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 RISCV_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);
    }
}
```

11.9 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`.

11.10 Instruction Disassembly

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

11.11 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 RISCV 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();
}
```

11.12 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 Implementing Custom PMA Behavior

The RISC-V architecture allows accesses to memory to be constrained by *physical memory attributes* (PMA). These PMA restrictions are not specified by the architecture but are machine-dependent. Typically, restrictions are either *statically defined* (e.g. by memory regions specified at build time) or *dynamic* (e.g. using custom CSRs in a similar way to the standard PMP CSRs).

The RISC-V model allows PMA constraints to be specified in several ways:

1. By static region definitions in the `riscvConfig` structure;
2. By commands callable when the simulation runs;
3. By extension model callbacks.

These three methods are each described below. The first two are appropriate for static PMA mappings, and the third for dynamic mappings.

12.1 Memory Domain Hierarchy

The RISC-V model uses OVP *memory domains* to efficiently implement access controls. A memory domain is essentially an address space with specified access permissions for subregions of that address space. Memory domains can implement physical memories, or peripherals (via callbacks on certain regions). They can also contain regions that are aliases of regions other memory domains, and so be used to model bus interconnect or MMU mappings very efficiently.

The RISC-V base model implements a static memory domain hierarchy, with a number of internally-created domain objects aliased to each other:

```
typedef struct riscvS {
    . . . lines omitted . . .
    memDomainP          extDomains[2];   // external domains (including CLIC)
    memDomainP          pmaDomains[2];   // PMA domains
    memDomainP          pmpDomains[2][2];// PMP domains (M-mode and other modes)
    . . . lines omitted . . .
} riscv;
```

Furthest from the processor, there are the *external* domains (`extDomains`). These are provided by the instruction and data busses connected to the processor at instantiation time. In the `extDomains` array, index 0 holds the external *data* domain and index 1 the external *code* domain; often, these are identical.

Nearer the processor, there are the *PMA* domains (`pmaDomains`). These domains are aliases of the external domains. By default, these aliases are made with full RWX permissions and impose no other access constraints. If the external data and code domains are different objects, there will be distinct `pmaDomains` as well, otherwise they will be the same domain.

Nearer again to the processor, there are the *PMP* domains (`pmpDomains`), with separate versions for Machine mode and other modes, as well as for data and code accesses. These domains are aliases of the PMA domains, with additional constraints imposed by the

architectural *physical memory protection* (PMP) unit, if implemented. If PMA data and code domains are different objects, there will be distinct `pmpDomains` as well. If the PMA data and code domains are the same object, then whether `pmpDomains` code/data domains are also the same object is controlled by the `distinctPhysMem` extension object interface function (see section 15.30). Distinct domains are required if either address space view contains physical components that are not visible in the other (for example, closely-coupled memories).

If virtual memory is implemented, there are additional memory domains (not described in detail here), used to model the MMU. Regions in these domains are dynamically mapped to regions in the PMP domains to reflect MMU mappings as simulation runs.

Depending on the operating mode of the processor, the current data and code domains may either be MMU domains (if address translation is enabled) or PMP domains (if address translation is disabled).

The legality of an access is constrained by permission restrictions at *all levels* of the active domain hierarchy: a domain level can remove permissions for a type of access but not add permission. In order to model PMA, access permissions and other attributes need to be updated on the PMA domain objects.

12.2 Static PMA Region Definitions

The simplest way to model PMA constraints is by static PMA region definitions in the `riscvConfig` structure. This is done as follows for the defined configurations in the in the `vendor.com` template model:

```
//
// Static PMA mappings
//
static const riscvPMARegion pmaStaticRegions[] = {
    {.lo=0x0000000000, .hi=0x0000001FFF, .attrs="r-xa 1248"}, // BOOTROM
    {.lo=0x0000002000, .hi=0x0000FFFFFF, .attrs="rwxa 1248"}, // RAM1
    {.lo=0x0080000000, .hi=0x00FFFFFFFF, .attrs="rwx- 1248"}, // RAM2
    {0} // KEEP LAST: terminator
};

//
// Defined configurations
//
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,
        .pmaStatic        = &pmaStaticRegions[0]
    },

    {
        .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,
        .pmaStatic        = &pmaStaticRegions[0]
    },

    {0} // null terminator
};
```

The `pmaStaticRegions` array is a NULL-terminated array of PMA region descriptions, each described by the following structure:

```
typedef struct riscvPMARegionS {
    Uns64       lo;                         // low region bound
    Uns64       hi;                         // high region bound
    const char *attrs;                      // region attributes
} riscvPMARegion;
```

The bounds of a PMA region are specified by the `lo` and `hi` fields. Access constraints are specified by the `attrs` string, which can contain characters as follows:

r: read access allowed
w: write access allowed
x: execute access allowed
a: unaligned accesses disallowed
A: atomic instruction access disallowed (if `amo_constraint` specified)
L: `lr`/`sc` instruction access disallowed (if `lr_sc_constraint` specified)
P: `push`/`pop` instruction access disallowed (if `push_pop_constraint` specified)
V: vector instruction access disallowed (if `vector_constraint` specified)
M: `Zicbom` instruction access disallowed
Z: `Zicboz` instruction access disallowed
1: 1-byte accesses permitted
2: 2-byte accesses permitted
4: 4-byte accesses permitted
8: 8-byte accesses permitted
space: ignored, use for formatting if desired
-: ignored, use for formatting if desired

As an example, the string `"r-xa 1248"` allows 1, 2, 4 and 8 bytes read and fetch accesses, but not write accesses. Misaligned accesses are also disallowed. Any disallowed access that is not a misaligned access will cause an Access Fault exception of appropriate type. Misaligned accesses will by default cause Address Misaligned exceptions of appropriate type; this behavior can be modified using the `rdFaultCB` and `wrFaultCB` extension model interface functions (see sections 15.1 and 15.2). Instruction-type-specific accesses are controlled using instruction constraints, as described in section 5.8.1.

When a static PMA region definition is used, those definitions are applied once, when memory domains are created and initialized before simulation starts. Any areas of

memory without corresponding PMA region definitions will have all access denied. PMA region definitions are applied in sequence, replacing any previous definitions for the specified range: this means that it is possible to describe general permissions with a large region, and then restrict permissions for subranges of that region with subsequent region definitions if required.

12.3 PMA Region Definitions using Model Commands

PMA region definitions can also be specified by the `setPMA` command. For example, this command line applies some static PMA region definitions when the base RISC-V model is run using `iss.exe`:

```
iss.exe \
    --processorvendor riscv.ovpworld.org \
    --processorname riscv \
    --variant RV64GC \
    --program test.elf \
    --override iss/cpu0/simulateexceptions=T \
    --callcommand iss/cpu0/setPMA -lo 0x90000000 -hi 0x90000fff -attributes "r--" \
    --callcommand iss/cpu0/setPMA -lo 0x90001000 -hi 0x90001fff -attributes "-w-" \
    --callcommand iss/cpu0/setPMA -lo 0x90002000 -hi 0x90002fff -attributes "--x"
```

As for static region definitions, the `setPMA` commands are applied in sequence, so later calls can refine definitions specified by earlier calls. Unlike static definitions described in section 12.2, regions of the memory space not covered by `setPMA` commands have no PMA access constraints imposed.

It is possible to call `setPMA` commands at run time to modify PMA access constraints dynamically, in which case the specified region access constraints will replace any existing access constraints for the specified region.

12.4 PMA Control by Extension Models

Some RISC-V implementations allow PMA constraints to be configured dynamically, for example by using a custom CSR interface similar to the standard PMP interface. If such an interface is required, there are two places in which it can be activated:

1. When there are explicit updates to PMA control registers. For example, a CSR write that modifies range or access constraints of a programmable PMA region must modify memory domains in the `pmaDomains` array to reflect that change.
2. When there are accesses to memory (e.g. by a processor load or store) for which PMA access constraints are in effect. This is handled using the `PMACheck` extension interface function, which also results in modifications to memory domains in the `pmaDomains` array.

To implement PMA efficiently, it is best to use an approach in which PMA privileges are applied in a lazy fashion. Using this approach, any explicit updates to PMA control registers should remove all PMA permissions from a region range implied by the *original* value of the PMA control register as well as any region range implied by the *new* value of the PMA control register. The effect of this is that any subsequent memory access in

either range will cause the `PMACheck` extension interface function to be activated, which can update PMA privileges to either permit or deny the access.

12.4.1 Example PMA Control Implementation

The OpenHardware CV32E40X model implements PMA using regions specified as parameters, using the `PMACheck` extension interface function. It therefore provides a good template for modelling of lazy PMA mapping.

The `PMACheck` extension interface function is defined as:

```
static RISCV_PMA_CHECK_FN(openhwPMACheck) {

    vmiosObjectP object = clientData;

    // apply PMA privileges
    mapPMA(riscv, object, requiredPriv, lowPA, highPA);
}
```

This function is passed the physical address range of a memory access using `lowPA` and `highPA` parameters. Function `mapPMA` then maps underlying PMA regions controlling the address range (there could be more than one region in the case of an access that straddles a PMA region boundary):

```
static void mapPMA(
    riscvP       riscv,
    vmiosObjectP object,
    memPriv      requiredPriv,
    Uns64        lowPA,
    Uns64        highPA
) {
    Uns64 mask    = getAddressMask(riscv->extBits);
    Uns64 highMap = lowPA-1;
    Uns64 lowMap;

    // iterate while unprocessed regions remain
    do {

        // get next region bounds to try
        lowMap  = highMap+1;
        highMap = mask;

        // attempt PMA privilege change for regions in implemented range
        if(lowMap<=highMap) {
            mapPMAInt(riscv, object, requiredPriv, lowPA, highPA, &lowMap, &highMap);
        } else {
            highMap = highPA;
        }

    } while(highMap<highPA);
}
```

Function `mapPMAInt` is responsible for mapping a single region:

```
static void mapPMAInt(
    riscvP       riscv,
    vmiosObjectP object,
    memPriv      requiredPriv,
    Uns64        lowPA,
    Uns64        highPA,
    Uns64       *lowMapP,
    Uns64       *highMapP
```

```
) {
    Uns64   PA   = *lowMapP;
    memPriv priv = getDefaultPriv();
    Int32 i;

    // set widest possible range initially
    *lowMapP = 0;

    // handle all regions in lowest-to-highest priority order
    for(i=object->PMA_NUM_REGIONS-1; i>=0; i--) {
        refinePMARegionRange(object, lowMapP, highMapP, PA, i, &priv);
    }

    // get PMA region range
    Uns64 lowMap  = *lowMapP;
    Uns64 highMap = *highMapP;

    // clamp physical range to maximum page size
    riscvClampPage(riscv, lowPA, highPA, &lowMap, &highMap);

    // update PMA privileges
    setPMAPriv(riscv, lowMap, highMap, priv, True);
}
```

This function traverses the set of configured PMA regions in lowest-to-highest priority order. After the traversal, it returns the bounds of the highest-priority region containing the accessed range and the access privileges required for that region. The returned bounds are possibly further restricted by the configured PMP/PMA maximum region size (see section 12.5) and then privileges on that range are updated by function `setPMAPriv`:

```
static void setDomainPriv(
    riscvP      riscv,
    memDomainPP domains,
    Uns64       low,
    Uns64       high,
    memPriv     priv,
    const char *name,
    Bool        verbose
) {
    memDomainP dataDomain = domains[0];
    memDomainP codeDomain = domains[1];

    if(dataDomain==codeDomain) {

        // set permissions in unified domain
        vmirtProtectMemory(dataDomain, low, high, priv, MEM_PRIV_SET);

    } else {

        // get privileges for data and code domains
        memPriv privRW = priv&MEM_PRIV_RW ? priv&~MEM_PRIV_X  : MEM_PRIV_NONE;
        memPriv privX  = priv&MEM_PRIV_X  ? priv&~MEM_PRIV_RW : MEM_PRIV_NONE;

        // set permissions in data domain
        vmirtProtectMemory(dataDomain, low, high, privRW, MEM_PRIV_SET);

        // set permissions in code domain
        vmirtProtectMemory(codeDomain, low, high, privX, MEM_PRIV_SET);
    }
}

static void setPMAPriv(
    riscvP  riscv,
    Uns64   low,
    Uns64   high,
    memPriv priv,
    Bool    verbose
```

```
) {
    setDomainPriv(riscv, &riscv->pmaDomains[0], low, high, priv, "PMA", verbose);
}
```

If the privileges set are insufficient for the access, the processor configuration then causes an appropriate Access Fault to be taken: this does not have to be explicitly handled by the `PMACheck` extension interface function.

12.5 PMA Mapped Page Size Restriction

By default, the example PMA control implementation in the previous section updates access permissions for the largest possible address range implied by the PMA settings; this could potentially be the entire address space range. This could cause a performance issue if the range being updated is highly fragmented, for example as a result of being divided into many separate subregions because of page descriptions implied by an active MMU.

To handle this potential performance issue, configuration parameter `PMP_max_page` can be used to limit the maximum size of PMP or PMA regions for which permission updates are made. If non-zero, the parameter specifies a power-of-two maximum page size that should be used for a single PMA or PMP mapping. The effect of this is to restrict the number of regions that are scanned when a large PMA region is updated.

The effect of `PMP_max_page` is applied by base model function `riscvClampPage`:

```
    // get PMA region range
    Uns64 lowMap  = *lowMapP;
    Uns64 highMap = *highMapP;

    // clamp physical range to maximum page size
    riscvClampPage(riscv, lowPA, highPA, &lowMap, &highMap);
```

This function is passed the address range of the memory access (`lowPA` and `highPA`) and the potentially large PMA region bounds (`lowMap` and `highMap`). It modifies `lowMap` and `highMap` to restrict them to page boundaries implied by `PMP_max_page` that include the required physical address range.

13 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.

13.1 Pattern RVIP_RD_RS1_RS2 (R-Type)

Description: like **`add x1, x2, x3`**

Decode:
```
|       | rs2 | rs1 |   | rd  |       |
|vvvvvvv|.....|.....|vvv|.....|vvvvvvv|
```

Format: `FMT_R1_R2_R3` (or `FMT_R1_R2` or `FMT_R1`)

Fields set: `r[0]=rd`, `r[1]=rs1`, `r[2]=rs2`

13.2 Pattern RVIP_RD_RS1_SI (I-Type)

Description: like **`addi x1, x2, imm`** (immediate sign-extended to `XLEN` bits)

Decode:
```
|  imm11:0   | rs1 |   | rd  |       |
|............|.....|vvv|.....|vvvvvvv|
```

Format: `FMT_R1_R2_SIMM`, `FMT_R1_R2_XIMM`

Fields set: `r[0]=rd`, `r[1]=rs1`, `c=sext(imm11:0)`

13.3 Pattern RVIP_RD_RS1_SHIFT (I-Type - 5 or 6 bit shift)

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

Decode:
```
|      shift | rs1 |   | rd  |       |
|0000000.....|.....|vvv|.....|vvvvvvv| (RV32)
|000000......|.....|vvv|.....|vvvvvvv| (RV64)
```

Format: `FMT_R1_R2_SIMM`, `FMT_R1_R2_XIMM`

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

13.4 Pattern RVIP_BASE_RS2_OFFSET (S-Type)

Description: like **`sw x2, offset(x1)`**

Decode:
```
|imm11:5|  rs2|  rs1|   |imm4:0|      |
|.......|.....|.....|vvv|.....|vvvvvvv|
```

Format: `FMT_R1_OFF_R2`

Fields set: `r[0]=rs2`, `r[1]=rs1`, `c=sext(imm11:0)`

13.5 Pattern RVIP_RS1_RS2_OFFSET (B-Type)

Description: like **`beq x1, x2, offset`**

Decode:
```
 imm                      imm
|12,10:5|  rs2|  rs1|   |4:1,11|      |
|.......|.....|.....|vvv|.....|vvvvvvv|
```

Format: `FMT_R1_R2_TGT`

Fields set: `r[0]=rs1`, `r[1]=rs2`, `tgt=PC + sext(imm12:1 << 1)`

13.6 Pattern RVIP_RD_SI (U-Type)

Description: like **lui x1, imm**

Decode:
```
|      imm31:12      | rd  |       |
|....................|.....|vvvvvvv|
```

Format: FMT_R1_UI

Fields set: r[0]=rd, c=sext(imm31:12 << 12)

13.7 Pattern RVIP_RD_OFFSET (J-Type)

Description: like **jal x1, offset**

Decode:
```
| imm20,10:1,11,19:12| rd  |       |
|....................|.....|vvvvvvv|
```

Format: FMT_R1_TGT

Fields set: r[0]=rd, tgt=PC + sext(imm20:1 << 1)

13.8 Pattern RVIP_RD_RS1_RS2_RS3 (R4-Type)

Description: like **cmix x0, x1, x2, x3**

Decode:
```
| rs3 |  | rs2 | rs1 |   | rd  |       |
|.....|vv|.....|.....|vvv|.....|vvvvvvv|
```

Format: FMT_R1_R2_R3_R4

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

13.9 Pattern RVIP_RD_RS1_RS3_SHIFT (Non-Standard)

Description: like **fsri x0, x1, x2, shift**

Decode:
```
| rs3 | | shift| rs1 |   | rd  |       |
|.....|v|0.....|.....|vvv|.....|vvvvvvv| (RV32)
|.....|v|......|.....|vvv|.....|vvvvvvv| (RV64)
```

Format: FMT_R1_R2_R3_SIMM

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

13.10 Pattern RVIP_FD_FS1_FS2

Description: like **fmax.s f0, f1, f2**

Decode:
```
|       | fs2 | fs1 |   | fd  |       |
|vvvvvvW|.....|.....|vvv|.....|vvvvvvv|
```

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

13.11 Pattern RVIP_FD_FS1_FS2_RM

Description: like **fadd.s f0, f1, f2, rte**

Decode:
```
|       | fs2 | fs1 | rm| fd  |       |
|vvvvvvW|.....|.....|...|.....|vvvvvvv|
```

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

13.12 Pattern RVIP_FD_FS1_FS2_FS3_RM

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

Decode:
```
| fs3 |  | fs2 | fs1 | rm| fd  |       |
|.....|vW|.....|.....|...|.....|vvvvvvv|
```

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

13.13 Pattern RVIP_RD_FS1_FS2

Description: like **feq.s x0, f1, f2**

Decode:
```
|       | fs2 | fs1 |   | rd  |       |
|vvvvvvW|.....|.....|vvv|.....|vvvvvvv|
```

Format: FMT_R1_R2_R3

Fields set: r[0]=rd, r[1]=fs1, r[2]=fs2

Width: *W=0:s*, W=1:d

13.14 Pattern RVIP_VD_VS1_VS2_M

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

Decode:
```
|      |m| vs1 | vs2 |   | vd  |       |
|vvvvvv|.|.....|.....|vvv|.....|vvvvvvv|
```

Format: FMT_R1_R2_R3_RM

Fields set: r[0]=vd, r[1]=vs1, r[2]=vs2, mask=m?none:v0

13.15 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  |       |
|vvvvvv|.|.....|.....|vvv|.....|vvvvvvv|
```

Format: FMT_R1_R2_SIMM_RM

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

13.16 Pattern RVIP_VD_VS1_UI_M

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

Decode:
```
|      |m| vs1 | imm |   | vd  |       |
|vvvvvv|.|.....|.....|vvv|.....|vvvvvvv|
```

Format: FMT_R1_R2_SIMM_RM

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

13.17 Pattern RVIP_VD_VS1_RS2_M

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

Decode:
```
|      |m| vs1 | rs2 |   | vd  |       |
|vvvvvv|.|.....|.....|vvv|.....|vvvvvvv|
```

Format: FMT_R1_R2_R3_RM

Fields set: r[0]=vd, r[1]=vs1, r[2]=rs2, mask=m?none:v0

13.18 Pattern RVIP_VD_VS1_FS2_M

Description: like **vfadd.vf v1, v2, f3, v0.m**

Decode:
```
|      |m| vs1 | fs2 |   | vd  |       |
|vvvvvv|.|.....|.....|vvv|.....|vvvvvvv|
```

Format: FMT_R1_R2_R3_RM

Fields set: r[0]=vd, r[1]=vs1, r[2]=fs2, mask=m?none:v0

13.19 Pattern RVIP_RD_VS1_RS2

Description: like **vext.v r1, v2, r3**

Decode:
```
|       | vs1 | rs2 |   | rd  |       |
|vvvvvvv|.....|.....|vvv|.....|vvvvvvv|
```

Format: FMT_R1_R2_R3

Fields set: r[0]=rd, r[1]=vs1, r[2]=rs2

13.20 Pattern RVIP_RD_VS1_M

Description: like **vpopc.m x1, v2, v0.m**

Decode:
```
|      |m| vs1 |        | rd  |       |
|vvvvvv|.|.....|vvvvvvvv|.....|vvvvvvv|
```

Format: FMT_R1_R2_RM

Fields set: r[0]=rd, r[1]=vs1, mask=m?none:v0

13.21 Pattern RVIP_VD_RS2

Description: like **vmv.s.x v1, x2**

Decode:
```
|            | rs2 |   | vd  |       |
|vvvvvvvvvvvv|.....|vvv|.....|vvvvvvv|
```

Format: FMT_R1_R2

Fields set: r[0]=vd, r[1]=rs2

13.22 Pattern RVIP_FD_VS1

Description: like **vfmv.f.s f1, v2**

Decode:
```
|       | vs2 |        | fd  |       |
|vvvvvvv|.....|vvvvvvvv|.....|vvvvvvv|
```

Format: FMT_R1_R2

Fields set: r[0]=fd, r[1]=vs2

13.23 Pattern RVIP_VD_FS2

Description: like **vfmv.s.f v1, f2**

Decode:
```
|            | fs2 |   | vd  |       |
|vvvvvvvvvvvv|.....|vvv|.....|vvvvvvv|
```

Format: FMT_R1_R2

Fields set: r[0]=vd, r[1]=fs2

14 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        block;
    riscvHaltRestartFn        restart;
    riscvUpdateInterruptFn    updateInterrupt;
    riscvUpdateDisableFn      updateDisable;
    riscvUpdateDisableNMIFn   updateDisableNMI;
    riscvTestInterruptFn      testInterrupt;
    riscvResumeFromWFIFn      resumeFromWFI;
    riscvIllegalInstructionFn illegalInstruction;
    riscvIllegalVerboseFn     illegalVerbose;
    riscvIllegalInstructionFn virtualInstruction;
    riscvIllegalVerboseFn     virtualVerbose;
    riscvIllegalCustomFn      illegalCustom;
    riscvTakeExceptionFn      takeException;
    riscvPendFetchExceptionFn pendFetchException;
    riscvTakeResetFn          takeReset;
    riscvAcknowledgeCLICIntFn acknowledgeCLICInt;

    // 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;
    riscvLoadMtFn             loadMt;
    riscvStoreMtFn            storeMt;
    riscvRequireModeMtFn      requireModeMt;
    riscvRequireNotVMtFn      requireNotVMt;
    riscvCheckLegalRMMtFn     checkLegalRMMt;
    riscvMorphTrapTVMFn       morphTrapTVM;
    riscvMorphVOpFn           morphVOp;

    // from riscvCSR.h
    riscvNewCSRFn             newCSR;
    riscvHPMAccessValidFn     hpmAccessValid;

    // from riscvVM.h
    riscvMapAddressFn         mapAddress;
    riscvUnmapPMPRegionFn     unmapPMPRegion;
    riscvUpdateLdStDomainFn   updateLdStDomain;
    riscvNewTLBEntryFn        newTLBEntry;
    riscvFreeTLBEntryFn       freeTLBEntry;

    // from riscvDebug.h
    riscvNewExtRegFn          newExtReg;

} riscvModelCB;
```

Following subsections describe each interface function.

14.1 Function registerExtCB

```
#define RISCV_REGISTER_EXT_CB_FN(_NAME) void _NAME( \
    riscvP      riscv,  \
    riscvExtCBP extCB,  \
    Uns32       id      \
)
typedef RISCV_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.

14.2 Function `getExtClientData`

```
#define RISCV_GET_EXT_CLIENT_DATA_FN(_NAME) void *_NAME( \
    riscvP riscv,      \
    Uns32  id          \
)
typedef RISCV_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.

14.3 Function `getExtConfig`

```
#define RISCV_GET_EXT_CONFIG_FN(_NAME) riscvExtConfigCP _NAME( \
    riscvP riscv,   \
    Uns32  id       \
)
typedef RISCV_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_ro1 = {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.

14.4 Function `getXlenMode`

```
#define RISCV_GET_XLEN_FN(_NAME) Uns32 _NAME(riscvP riscv)
typedef RISCV_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.

14.5 Function `getXlenArch`

```
#define RISCV_GET_XLEN_FN(_NAME) Uns32 _NAME(riscvP riscv)
typedef RISCV_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.

14.6 Function getXRegName

```
#define RISCV_GET_REG_NAME_FN(_NAME) const char *_NAME(riscvP riscv, Uns32 index)
typedef RISCV_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(riscv, index);
}
```

Usage Context
Leaf level only.

14.7 Function `getFRegName`

```
#define RISCV_GET_REG_NAME_FN(_NAME) const char *_NAME(riscvP riscv, Uns32 index)
typedef RISCV_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(riscv, index);
}
```

Usage Context
Leaf level only.

14.8 Function getVRegName

```
#define RISCV_GET_REG_NAME_FN(_NAME) const char *_NAME(riscvP riscv, Uns32 index)
typedef RISCV_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(riscv, index);
}
```

Usage Context
Leaf level only.

14.9 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.

14.10 Function getTMode

```
#define RISCV_GET_TMODE_FN(_NAME) Bool _NAME(riscvP riscv)
typedef RISCV_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 14.9 for more information.

Example

```
static Bool getTMode(riscvP riscv) {
    return riscv->cb.getTMode(riscv);
}
```

Usage Context
Leaf level only.

14.11 Function `getDataEndian`

```
#define RISCV_GET_DATA_ENDIAN_FN(_NAME) memEndian _NAME( \
    riscvP    riscv,    \
    riscvMode mode      \
)
typedef RISCV_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 use 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.

14.12 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.

14.13 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.

14.14 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.

14.15 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.

14.16 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     = 0x00,  // running
    RVD_WFI        = 0x01,  // waiting in WFI
    RVD_RESET      = 0x02,  // waiting in reset
    RVD_DEBUG      = 0x04,  // waiting for debug
    RVD_WRS        = 0x08,  // waiting for reservation set (wrs.*)
    RVD_STO        = 0x10,  // waiting for reservation set (wrs.sto)
    RVD_CUSTOM_WFI = 0x20,  // waiting for custom reason with WFI semantics
    RVD_CUSTOM_NMI = 0x40,  // waiting for custom reason with NMI semantics
};
```

Reason `RVD_WFI` is used by the base model to indicate the hart is stalled in a `wfi` instruction.

Reason `RVD_RESET` is used by the base model to indicate the hart is stalled because the reset signal is high.

Reason `RVD_DEBUG` is used only when debug mode is configured to halt the processor (`debug_mode` is `RVDM_HALT`) and indicates the hart is halted in debug state.

Reason `RVD_WRS` is used by the base model to indicate the hart is stalled in a `wrs.nto` or `wrs.sto` instruction (see the `Zawrs` extension).

Reason `RVD_STO` is used by the base model to indicate the hart is stalled in a `wrs.sto` instruction (see the `Zawrs` extension).

Reason `RVD_CUSTOM_WFI` is available for use in a custom extension. It indicates the processor is stalled for a reason with the same implicit wake-up semantics as the `wfi` instruction (for example, a locally-enabled interrupt becomes pending, even if globally disabled).

Reason `RVD_CUSTOM_NMI` is also available for use in a custom extension. It indicates the processor is stalled and should awaken on reset or if a fully-enabled interrupt is pending.

The hart will remain halted until a subsequent call to the `restart` function or a reset or other interrupt event, depending on the halt reason semantics.

Example
This example shows implementation of a wait-for-event (`wfe`) instruction with the same semantics as `wfi` for a processor with M-mode and U-mode.

```
static void doWFE(riscvP riscv) {

    riscvMode mode = getCurrentMode5(riscv);

    if((mode!=RISCV_MODE_M) && RD_CSR_FIELDC(riscv, mstatus, TW)) {

        // WFE not available when mstatus.TW=1
        riscv->cb.illegalVerbose(riscv, "mstatus.TW=1");

    } else if(!riscv->cb.resumeFromWFI(riscv)) {

        // stall for custom reason (WFE) with WFI wake semantics
        riscv->cb.halt(riscv, RVD_CUSTOM_WFI);
    }
}
```

Usage Context
Leaf level only.

14.17 Function block

```
#define RISCV_HALT_RESTART_FN(_NAME) void _NAME( \
    riscvP             riscv,   \
    riscvDisableReason reason   \
)
typedef RISCV_HALT_RESTART_FN((*riscvHaltRestartFn));

typedef struct riscvModelCBS {
    riscvHaltRestartFn       block;
} riscvModelCB;
```

Description
This interface function causes the given hart to block during execution of an instruction, for a reason given by the `riscvDisableReason` enumerated type:

```
typedef enum riscvDisableReasonE {
    RVD_ACTIVE     = 0x00,  // running
    RVD_WFI        = 0x01,  // waiting in WFI
    RVD_RESET      = 0x02,  // waiting in reset
    RVD_DEBUG      = 0x04,  // waiting for debug
    RVD_WRS        = 0x08,  // waiting for reservation set (wrs.*)
    RVD_STO        = 0x10,  // waiting for reservation set (wrs.sto)
    RVD_CUSTOM_WFI = 0x20,  // waiting for custom reason with WFI semantics
    RVD_CUSTOM_NMI = 0x40,  // waiting for custom reason with NMI semantics
};
```

See section 14.16 for a description of the `riscvDisableReason` enumerated type.

The hart will remain blocked until a subsequent call to the `restart` function or a reset or other interrupt event, depending on the halt reason semantics. Upon restart, the currently-executing instruction will be *re-executed*, provided that the reason for restart was not to take a trap.

Typically, this function is used to allow simulation of load or store instructions with blocking semantics. The custom extension should add a net port allowing a signal to be raised while the load or store is active, to block execution if required. An external component that drives this signal needs to implement logic to determine whether a particular load or store should be blocked, taking into account the fact that the load or store will be restarted when the hard resumes execution from a blocked state.

Example
This example shows an example implementation of a block signal.

```
static VMI_NET_CHANGE_FN(blockCB) {

    vmiosObjectP object  = userData;
    riscvP       riscv   = object->riscv;
    Bool         disable = newValue & 1;

    if(disable) {
        riscv->cb.block(riscv, RVD_CUSTOM_NMI);
    } else {
        riscv->cb.restart(riscv, RVD_CUSTOM_NMI);
    }
}
```

Usage Context
Leaf level only.

14.18 Function `restart`

```
#define RISCV_HALT_RESTART_FN(_NAME) void _NAME( \
    riscvP             riscv,   \
    riscvDisableReason reason   \
)
typedef RISCV_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     = 0x00,  // running
    RVD_WFI        = 0x01,  // waiting in WFI
    RVD_RESET      = 0x02,  // waiting in reset
    RVD_DEBUG      = 0x04,  // waiting for debug
    RVD_WRS        = 0x08,  // waiting for reservation set (wrs.*)
    RVD_STO        = 0x10,  // waiting for reservation set (wrs.sto)
    RVD_CUSTOM_WFI = 0x20,  // waiting for custom reason with WFI semantics
    RVD_CUSTOM_NMI = 0x40,  // waiting for custom reason with NMI semantics
};
```

See section 14.16 for a description of the `riscvDisableReason` enumerated type.

The function has no effect if the hart is not halted for the given reason.

Example
This example shows implementation of a wait-for-event (`wfe`) instruction with the same semantics as `wfi`.

```
static VMI_NET_CHANGE_FN(wakeWFE) {

    vmiosObjectP object = userData;
    riscvP       riscv  = object->riscv;
    Bool         old    = object->wakeWFE;
    Bool         new    = newValue&1;

    // detect rising edge
    if(!old && new) {
        riscv->cb.restart(riscv, RVD_CUSTOM_WFI);
    }

    object->wakeWFE = new;
}
```

Usage Context
Leaf level only.

14.19 Function updateInterrupt

```
#define RISCV_UPDATE_INTERRUPT_FN(_NAME) void _NAME( \
    riscvP riscv,               \
    Uns32  index,               \
    Bool   newValue             \
)
typedef RISCV_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.

14.20 Function updateDisable

```
#define RISCV_UPDATE_DISABLE_FN(_NAME) void _NAME( \
    riscvP riscv,               \
    Uns64  disableMask          \
)
typedef RISCV_UPDATE_DISABLE_FN((*riscvUpdateDisableFn));

typedef struct riscvModelCBS {
    riscvUpdateDisableFn      updateDisable;
} riscvModelCB;
```

Description

This interface function allows a supplementary mask of disabled interrupts to be applied by a custom extension: any standard interrupt with an index number corresponding to a bit that is set in this mask will be disabled, overriding standard behavior specified by the `mie` CSR.

Example

```
static Uns64 ecsrW(vmiosObjectP object, Uns64 newValue, Uns64 mask) {

    riscvP riscv    = object->riscv;
    Uns64  oldValue = RD_XCSR(object, mecsr);

    // update mecsr value
    WR_XCSR(object, mecsr, ((newValue & mask) | (oldValue & ~mask)));

    // get mask of externally-disabled interrupts
    Uns64 disableMask = RD_XCSR_FIELD(object, mecsr, WEDIS) ? (1<<CUST_MERROR) : 0;

    // update mask of externally-disabled interrupts
    riscv->cb.updateDisable(riscv, disableMask);

    // return composed value
    return RD_XCSR(object, mecsr) & mask;
}
```

Usage Context

Leaf level only.

14.21 Function updateDisableNMI

```
#define RISCV_UPDATE_DISABLE_NMI_FN(_NAME) void _NAME( \
    riscvP riscv,               \
    Bool   disable              \
)
typedef RISCV_UPDATE_DISABLE_NMI_FN((*riscvUpdateDisableNMIFn));

typedef struct riscvModelCBS {
    riscvUpdateDisableNMIFn   updateDisableNMI;
} riscvModelCB;
```

Description

This interface allows NMI interrupts to be enabled or disabled for a particular hart. This allows extensions to control which harts in a multi-hart implementation react to external NMI inputs. This is typically required in implementations that broadcast a single NMI to multiple harts.

Example

```
static void updateNMIPDEL(riscvP thread, Bool delegate) {
    thread->cb.updateDisableNMI(thread, !delegate);
}
```

Usage Context

Leaf level only.

14.22 Function testInterrupt

```
#define RISCV_TEST_INTERRUPT_FN(_NAME) void _NAME(riscvP riscv)
typedef RISCV_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);
}

static RISCV_CSR_WRITEFN(slieW) {

    vmiosObjectP object   = attrs->object;
    Uns32        oldValue = RD_XCSR(object, slie);
    Uns32        mask     = slixMask(object);

    // update writeable bits
    newValue = (newValue & mask) | (oldValue & ~mask);

    // handle any change to interrupt state
    if(newValue != oldValue) {
        WR_XCSR(object, slie, newValue);
        testInterrupt(riscv);
    }

    return newValue;
}
```

Usage Context
Leaf level only.

14.23 Function `resumeFromWFI`

```
#define RISCV_RESUME_FROM_WFI_FN(_NAME) Bool _NAME(riscvP riscv)
typedef RISCV_RESUME_FROM_WFI_FN((*riscvResumeFromWFIFn));

typedef struct riscvModelCBS {
    riscvResumeFromWFIFn      resumeFromWFI;
} riscvModelCB;
```

Description

This interface function returns `True` if the base model is in a state where it should resume from a stalled state because of a `wfi` instruction (or an instruction with similar semantics). This is the case if there are pending interrupts that are locally enabled even if they are globally disabled.

Example

This example shows implementation of a wait-for-event (`wfe`) instruction with the same semantics as `wfi` for a processor with M-mode and U-mode.

```
static void doWFE(riscvP riscv) {

    riscvMode mode = getCurrentMode5(riscv);

    if((mode!=RISCV_MODE_M) && RD_CSR_FIELDC(riscv, mstatus, TW)) {

        // WFE not available when mstatus.TW=1
        riscv->cb.illegalVerbose(riscv, "mstatus.TW=1");

    } else if(!riscv->cb.resumeFromWFI(riscv)) {

        // stall for custom reason (WFE) with WFI wake semantics
        riscv->cb.halt(riscv, RVD_CUSTOM_WFI);
    }
}
```

Usage Context

Leaf level only.

14.24 Function `illegalInstruction`

```
#define RISCV_ILLEGAL_INSTRUCTION_FN(_NAME) void _NAME(riscvP riscv)
typedef RISCV_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);
}

static Bool accessUserCCTL(vmiosObjectP object, riscvP riscv) {

    if(riscv->artifactAccess) {

        // all artifact accesses are allowed
        return True;

    } else if(getCurrentMode3(riscv)==RISCV_MODE_MACHINE) {

        // access always possible in Machine mode
        return True;

    } else if(RD_XCSR_FIELD(object, mcache_ctl, CCTL_SUEN)) {

        // access possible in Supervisor and User mode if mcache_ctl.CCTL_SUEN=1
        return True;

    } else {

        // take Illegal Instruction exception
        illegalInstruction(riscv);
        return False;
    }
}
```

Usage Context
Leaf level only.

14.25 Function `illegalVerbose`

```
#define RISCV_ILLEGAL_VERBOSE_FN(_NAME) void _NAME( \
    riscvP      riscv,          \
    const char *reason          \
)
typedef RISCV_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 doWFE(riscvP riscv) {

    riscvMode mode = getCurrentMode5(riscv);

    if((mode!=RISCV_MODE_M) && RD_CSR_FIELDC(riscv, mstatus, TW)) {

        // WFE not available when mstatus.TW=1
        riscv->cb.illegalVerbose(riscv, "mstatus.TW=1");

    } else if(!riscv->cb.resumeFromWFI(riscv)) {

        // stall for custom reason (WFE) with WFI wake semantics
        riscv->cb.halt(riscv, RVD_CUSTOM_WFI);
    }
}
```

Usage Context
Leaf level only.

14.26 Function `virtualInstruction`

```
#define RISCV_ILLEGAL_INSTRUCTION_FN(_NAME) void _NAME(riscvP riscv)
typedef RISCV_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.

14.27 Function `virtualVerbose`

```
#define RISCV_ILLEGAL_VERBOSE_FN(_NAME) void _NAME( \
    riscvP      riscv,          \
    const char *reason          \
)
typedef RISCV_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.

14.28 Function `illegalCustom`

```
#define RISCV_ILLEGAL_CUSTOM_FN(_NAME) void _NAME( \
    riscvP         riscv,       \
    riscvException exception,   \
    const char    *reason       \
)
typedef RISCV_ILLEGAL_CUSTOM_FN((*riscvIllegalCustomFn));

typedef struct riscvModelCBS {
    riscvIllegalCustomFn      illegalCustom;
} riscvModelCB;
```

Description

This interface function causes the base model to take a custom trap immediately, for a verbose reason indicated by the `reason` string. The exception passed as the `exception` argument can be either a standard exception or any other index number corresponding to a custom exception. The *`x`*`tval` CSR is updated in the same way as for a standard Illegal Instruction exception; that is, it will either be set to zero or to the opcode of the failing instruction, depending on the value of the `tval_ii_code` and `tval_zero` configuration options.

Example

```
static void illegalCustom(
    vmiosObjectP   object,
    riscvException exception,
    const char    *reason
) {
    riscvP riscv = object->riscv;
    riscv->cb.illegalCustom(riscv, exception, reason);
}
```

Usage Context

Leaf level only.

14.29 Function takeException

```
#define RISCV_TAKE_EXCEPTION_FN(_NAME) void _NAME( \
    riscvP         riscv,       \
    riscvException exception,   \
    Uns64          tval         \
)
typedef RISCV_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 *`x`*`tval` CSR.

Example

```
void andesTakeException(riscvP riscv, andesException exception, Uns64 tval) {
    riscv->cb.takeException(riscv, exception, tval);
}

static void doHSPException(
    riscvP         riscv,
    vmiosObjectP   object,
    andesException exception
) {
    // revert SP to its value prior to instruction execution
    riscv->x[RV_REG_X_SP] = object->oldSP;

    // clear stack protection enables
    WR_XCSR_FIELD(object, mhsp_ctl, OVF_EN, 0);
    WR_XCSR_FIELD(object, mhsp_ctl, UDF_EN, 0);

    // do standard exception actions
    andesTakeException(riscv, exception, 0);
}
```

Usage Context

Leaf level only.

14.30 Function pendFetchException

```
#define RISCV_PEND_FETCH_EXCEPTION_FN(_NAME) void _NAME( \
    riscvP         riscv,       \
    riscvException exception    \
)
typedef RISCV_PEND_FETCH_EXCEPTION_FN((*riscvPendFetchExceptionFn));

typedef struct riscvModelCBS {
    riscvPendFetchExceptionFn pendFetchException;
} riscvModelCB;
```

Description

This interface function causes the base model to schedule a pending exception that will be taken the next time the model attempts an instruction fetch. The exception passed as the `exception` argument will typically be an index number corresponding to a custom exception. The *`xtval`* CSR will be filled with the program counter corresponding to the fetch address.

If the exception is scheduled before an attempt to execute an instruction, then the exception will report that instruction address as the faulting address. If the exception is scheduled before an interrupt that is taken in CLIC mode, then the exception will report the CLIC handler address as the faulting address, and hart state will be updated to indicate a hardware vector table fetch fault.

Example

```
static VMI_NET_CHANGE_FN(instructionBusFault) {

    vmiosObjectP object = userData;
    riscvP       riscv  = object->riscv;
    Bool         old    = object->instructionBusFault;
    Bool         new    = newValue&1;

    // detect rising edge
    if(!old && new) {
        riscv->cb.pendFetchException(riscv, OHW_E_InstructionBusFault);
    }

    object->instructionBusFault = new;
}
```

Usage Context

Leaf level only.

14.31 Function takeReset

```
#define RISCV_TAKE_RESET_FN(_NAME) void _NAME(riscvP riscv)
typedef RISCV_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.

14.32 Function acknowledgeCLICInt

```
#define RISCV_ACKNOWLEDGE_CLIC_INT_FN(_NAME) void _NAME( \
    riscvP hart,       \
    Uns32  intIndex    \
)
typedef RISCV_ACKNOWLEDGE_CLIC_INT_FN((*riscvAcknowledgeCLICIntFn));
```

Description

This callback clears the pending bit of an edge triggered interrupt in an internally implemented CLIC, and updates the pending-and-enabled state of the CLIC. The CLIC specification requires that the pending bit is cleared when taking an edge triggered interrupt configured to use selective hardware vectoring (`shv`).

When an extension implements a custom trap location using the `getHandlerPC` extension callback it must call this function before returning `True` from the `getHandlerPC` function, when taking a `shv`-configured interrupt. This is required because the call to this function is bypassed in the base model when a call to `getHandlerPC` returns `True`.

Example

```
//
// Use alternate base for CLIC vectored interrupts
//
static RISCV_GET_HANDLER_PC_FN(vendorGetHandlerPC) {
    vmiosObjectP object = clientData;
    Bool         custom = False;

    if (!isInterrupt(exception)) {
        // No custom behavior unless exception is an interrupt
    } else if (!(mode & riscv_int_CLIC)) {
        // No custom behavior unless in CLIC mode
    } else if (!CLICInternal(riscv)) {
        // No custom behavior if internal CLIC not being modeled
    } else if (!riscv->clic.sel.shv) {
        // No custom behavior if not a vectored interrupt
    } else {
        Addr           tableAddr = object->local_mtvt + (ecode*4);
        memEndian      endian    = riscv->cb.getDataEndian(riscv, mode);
        memDomainP     domain    = getCodeDomain(riscv);
        memAccessAttrs memAttrs  = MEM_AA_TRUE|MEM_AA_FETCH;

        // read 4-byte table entry
        *handlerPCP = vmirtRead4ByteDomain(domain, tableAddr, endian, memAttrs);

        // SHV interrupts must be acknowledged when interrupt taken
        riscv->cb.acknowledgeCLICInt(riscv, ecode);

        // Use custom handler PC
        custom = True;
    }

    return custom;
}
```

Usage Context
Leaf level only.

14.33 Function `fetchInstruction`

```
#define RISCV_FETCH_INSTRUCTION_FN(_NAME) Uns32 _NAME( \
    riscvP               riscv,     \
    riscvAddr            thisPC,    \
    riscvExtInstrInfoP   info,      \
    vmidDecodeTablePP    tableP,    \
    riscvExtInstrAttrsCP attrs,     \
    Uns32                last,      \
    Uns32                bits       \
)
typedef RISCV_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 13 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.

14.34 Function `disassInstruction`

```
#define RISCV_DISASS_INSTRUCTION_FN(_NAME) const char *_NAME( \
    riscvP             riscv,       \
    riscvExtInstrInfoP instrInfo,   \
    vmiDisassAttrs     attrs        \
)
typedef RISCV_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 13 for more information about the available instruction patterns.

Example

```
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;
}
```

Usage Context
Leaf level only.

14.35 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, morph-time.

14.36 Function morphExternal

```
#define RISCV_MORPH_EXTERNAL_FN(_NAME) void _NAME( \
    riscvExtMorphStateP state,          \
    const char         *disableReason,  \
    Bool               *opaque          \
);
typedef RISCV_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, morph-time.

14.37 Function `morphIllegal`

```
#define RISCV_MORPH_ILLEGAL_FN(_NAME) void _NAME( \
    riscvP      riscv,  \
    const char *reason  \
)
typedef RISCV_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, morph-time.

14.38 Function `morphVirtual`

```
#define RISCV_MORPH_ILLEGAL_FN(_NAME) void _NAME( \
    riscvP      riscv,  \
    const char *reason  \
)
typedef RISCV_MORPH_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, morph-time.

14.39 Function getVMIReg

```
#define RISCV_GET_VMI_REG_FN(_NAME) vmiReg _NAME(riscvP riscv, riscvRegDesc r)
typedef RISCV_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, morph-time.

14.40 Function getVMIRegFS

```
#define RISCV_GET_VMI_REG_FS_FN(_NAME) vmiReg _NAME( \
    riscvP       riscv,     \
    riscvRegDesc r,         \
    vmiReg       tmp        \
)
typedef RISCV_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, morph-time.

14.41 Function writeRegSize

```
#define RISCV_WRITE_REG_SIZE_FN(_NAME) void _NAME( \
    riscvP       riscv,     \
    riscvRegDesc r,         \
    Uns32        srcBits,   \
    Bool         signExtend \
)
typedef RISCV_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, morph-time.

14.42 Function writeReg

```
#define RISCV_WRITE_REG_FN(_NAME) void _NAME( \
    riscvP       riscv,     \
    riscvRegDesc r,         \
    Bool         signExtend \
)
typedef RISCV_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, morph-time.

14.43 Function `getFPFlagsMt`

```
#define RISCV_GET_FP_FLAGS_MT_FN(_NAME) vmiReg _NAME(riscvP riscv)
typedef RISCV_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, morph-time.

14.44 Function `getDataEndianMt`

```
#define RISCV_GET_DATA_ENDIAN_MT_FN(_NAME) memEndian _NAME(riscvP riscv)
typedef RISCV_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, morph-time.

14.45 Function loadMT

```
#define RISCV_LOAD_MT_FN(_NAME) void _NAME( \
    riscvP            riscv,    \
    vmiReg            rd,       \
    Uns32             rdBits,   \
    vmiReg            ra,       \
    Uns32             memBits,  \
    Uns64             offset,   \
    riscvExtLdStAttrs attrs     \
)
typedef RISCV_LOAD_MT_FN((*riscvLoadMtFn))

typedef struct riscvModelCBS {
    riscvLoadMtFn             loadMt;
} riscvModelCB;
```

Description
This function must be called at morph time (from within the extension object `morphCB`).

This interface function emits JIT code to perform a load of `memBits` wide data from the address `ra+offset`. The loaded value is extended to `rdBits` wide and written to register `rd`. Other attributes of the load are defined by the `attrs` parameter, which is defined in `riscvModelCallbackTypes.h` as follows:

```
typedef struct riscvExtLdStAttrsS {
    memConstraint constraint : 4;   // access constraints
    Bool          sExtend    : 1;   // sign-extension (load only)
    Bool          isVirtual  : 1;   // whether HLV/HLVX/HSV
    Bool          isCode     : 1;   // whether load as if fetch (HLVX)
} riscvExtLdStAttrs;
```

Fields in this structure are as follows:

`constraint`: this bitfield enumeration specifies constraints for the memory access. For the RISC-V model, these values are significant:

`MEM_CONSTRAINT_ALIGNED`: whether the access must be aligned;
`MEM_CONSTRAINT_USER1`: whether the access is atomic.

`sExtend`: this Boolean value indicates whether the loaded value must be sign-extended to `rdBits` width, if it is smaller. If `False`, the value is zero-extended.

`isVirtual`: this Boolean value indicates whether the load is a result of an instruction like `HLV` or `HLVX`, requiring access to guest virtual address space (only ever `True` when Hypervisor mode is implemented).

`isCode`: this Boolean value indicates whether the load is a result of an instruction like `HLVX`, where the load should be treated as if it was a fetch.

When this function is used, any Trigger Module triggers sensitive to the specified access will fire if required, and the load will also comply with any transactional memory model installed by the extension - this will not be the case if more fundamental VMI primitives such as `vmimtLoadRRO` are used instead.

Example

```
static void emitLoadCommon(
    andesMorphStateP state,
    vmiReg           rd,
    Uns32            rdBits,
    vmiReg           ra,
    memConstraint    constraint
) {
    riscvP riscv   = state->riscv;
    Uns32  memBits = state->info.memBits;
    Uns64  offset  = state->info.c1;

    riscvExtLdStAttrs attrs = {
        constraint : constraint,
        sExtend    : !state->info.unsExt
    };

    riscv->cb.loadMt(riscv, rd, rdBits, ra, memBits, offset, attrs);
}
```

Usage Context

Leaf level only, morph-time.

14.46 Function storeMT

```
#define RISCV_STORE_MT_FN(_NAME) void _NAME( \
    riscvP            riscv,    \
    vmiReg            rs,       \
    vmiReg            ra,       \
    Uns32             memBits,  \
    Uns64             offset,   \
    riscvExtLdStAttrs attrs     \
)
typedef RISCV_STORE_MT_FN((*riscvStoreMtFn))

typedef struct riscvModelCBS {
    riscvStoreMtFn            storeMt;
} riscvModelCB;
```

Description

This function must be called at morph time (from within the extension object `morphCB`).

This interface function emits JIT code to perform a store of `memBits` wide data to address `ra+offset`. The stored value is `memBits` wide and sourced from register `rs`. Other attributes of the load are defined by the `attrs` parameter, which is defined in `riscvModelCallbackTypes.h` as follows:

```
typedef struct riscvExtLdStAttrsS {
    memConstraint constraint : 4;   // access constraints
    Bool          sExtend    : 1;   // sign-extension (load only)
    Bool          isVirtual  : 1;   // whether HLV/HLVX/HSV
    Bool          isCode     : 1;   // whether load as if fetch (HLVX)
} riscvExtLdStAttrs;
```

Relevant fields in this structure for a store are as follows:

`constraint`: this bitfield enumeration specifies constraints for the memory access. For the RISC-V model, these values are significant:

`MEM_CONSTRAINT_ALIGNED`: whether the access must be aligned;
`MEM_CONSTRAINT_USER1`: whether the access is atomic.

`isVirtual`: this Boolean value indicates whether the store is a result of an instruction like `HSV`, requiring access to guest virtual address space (only ever `True` when Hypervisor mode is implemented).

When this function is used, any Trigger Module triggers sensitive to the specified access will fire if required, and the store will also comply with any transactional memory model installed by the extension - this will not be the case if more fundamental VMI primitives such as `vmimtStoreRRO` are used instead.

Example

```
static void emitStoreCommon(
    andesMorphStateP state,
    vmiReg           rs,
    vmiReg           ra,
    memConstraint    constraint
```

```
) {
    riscvP riscv   = state->riscv;
    Uns32  memBits = state->info.memBits;
    Uns64  offset  = state->info.c1;

    riscvExtLdStAttrs attrs = {constraint : constraint};

    riscv->cb.storeMt(riscv, rs, ra, memBits, offset, attrs);
}
```

Usage Context
Leaf level only, morph-time.

14.47 Function checkLoadMT

```
#define RISCV_CHECK_MEM_MT_FN(_NAME) void _NAME( \
    riscvP riscv,               \
    vmiReg base,                \
    Int32  offset,              \
    Uns32  bits,                \
    void  *clientData           \
)
typedef RISCV_CHECK_MEM_MT_FN((*riscvCheckMemMtFn));

typedef struct riscvModelCBS {
    riscvCheckMemMtFn         checkLoadMt;
} riscvModelCB;
```

Description

This interface function is called by the base model when an instruction that loads from a location in memory is being translated. It gives the extended model the opportunity to emit JIT code to apply extra checks before the load is performed. This function can be used to implement extended features such as hardware stack protection where memory accesses using the stack pointer register (`x2`) as a base are bounds-checked.

The callback is passed the `vmiReg` object corresponding to the *base* register being used for the load and the offset from the base that must be added to form the full load address. It is also given the data width of the load, in bits. It will typically emit an embedded call to perform a run-time check on address validity.

Example

The following example shows a template for a hardware stack protection check where any access using the stack pointer (`x2`) is range-checked.

```
//
// Implement hardware stack protection check
//
static void doHSPCheckLimit(
    riscvP         riscv,
    vmiosObjectP   object,
    Int32          offset,
    Uns32          bytes,
    Uns64          loLimit,
    riscvException exception
) {
    Uns64 lo = riscv->x[RV_REG_X_SP] + offset;
    Uns64 hi = lo + bytes - 1;

    if(
        // highest byte address must be lower than the stack base address
        (hi < RD_XCSR(object, mspcba)) &&
        // lowest byte address equal to of higher than low limit
        (lo >= loLimit)
    ) {

        // valid stack access

    } else {

        // do standard exception actions
        takeException(riscv, exception, lo);
    }
}
```

```
//
// Implement hardware stack protection check for load (low limit is SP)
//
static void doHSPCheckLoad(
    riscvP       riscv,
    vmiosObjectP object,
    Int32        offset,
    Uns32        bytes
) {
    doHSPCheckLimit(
        riscv,
        object,
        offset,
        bytes,
        riscv->x[RV_REG_X_SP],
        riscv_E_LoadAccessFault
    );
}

//
// Should HSP check be applied to the current instruction?
//
static Bool applyHSPMT(riscvP riscv, vmiosObjectP object, vmiReg base) {

    return (
        // HSP feature must be enabled
        isHSPEnabledMT(object) &&
        // base register must be the stack pointer
        VMI_REG_EQUAL(base, RISCV_SP)
    );
}

//
// Emit code to check hardware stack protection on load if required
//
static RISCV_CHECK_MEM_MT_FN(emitCheckLoadHSP) {

    vmiosObjectP object = clientData;

    if(applyHSPMT(riscv, object, base)) {
        vmimtArgProcessor();
        vmimtArgNatAddress(object);
        vmimtArgUns32(offset);
        vmimtArgUns32(BITS_TO_BYTES(bits));
        vmimtCallAttrs((vmiCallFn)doHSPCheckLoad, VMCA_NA);
    }
}
```

Usage Context
Leaf level only.

14.48 Function checkStoreMT

```
#define RISCV_CHECK_MEM_MT_FN(_NAME) void _NAME( \
    riscvP riscv,               \
    vmiReg base,                \
    Int32  offset,              \
    Uns32  bits,                \
    void  *clientData           \
)
typedef RISCV_CHECK_MEM_MT_FN((*riscvCheckMemMtFn));

typedef struct riscvModelCBS {
    riscvCheckMemMtFn         checkStoreMt;
} riscvModelCB;
```

Description

This interface function is called by the base model when an instruction that stores to a location in memory is being translated. It gives the extended model the opportunity to emit JIT code to apply extra checks before the store is performed. This function can be used to implement extended features such as hardware stack protection where memory accesses using the stack pointer register (`x2`) as a base are bounds-checked.

The callback is passed the `vmiReg` object corresponding to the *base* register being used for the store and the offset from the base that must be added to form the full store address. It is also given the data width of the store, in bits. It will typically emit an embedded call to perform a run-time check on address validity.

Example

The following example shows a template for a hardware stack protection check where any access using the stack pointer (`x2`) is range-checked.

```
//
// Implement hardware stack protection check
//
static void doHSPCheckLimit(
    riscvP         riscv,
    vmiosObjectP   object,
    Int32          offset,
    Uns32          bytes,
    Uns64          loLimit,
    riscvException exception
) {
    Uns64 lo = riscv->x[RV_REG_X_SP] + offset;
    Uns64 hi = lo + bytes - 1;

    if(
        // highest byte address must be lower than the stack base address
        (hi < RD_XCSR(object, mspcba)) &&
        // lowest byte address equal to of higher than low limit
        (lo >= loLimit)
    ) {

        // valid stack access

    } else {

        // do standard exception actions
        takeException(riscv, exception, lo);
    }
}
```

```
//
// Implement hardware stack protection check for store (low limit is mspcta)
//
static void doHSPCheckStore(
    riscvP       riscv,
    vmiosObjectP object,
    Int32        offset,
    Uns32        bytes
) {
    doHSPCheckLimit(
        riscv,
        object,
        offset,
        bytes,
        RD_XCSR(object, mspcta),
        riscv_E_StoreAMOAccessFault
    );
}

//
// Should HSP check be applied to the current instruction?
//
static Bool applyHSPMT(riscvP riscv, vmiosObjectP object, vmiReg base) {

    return (
        // HSP feature must be enabled
        isHSPEnabledMT(object) &&
        // base register must be the stack pointer
        VMI_REG_EQUAL(base, RISCV_SP)
    );
}

//
// Emit code to check hardware stack protection on store if required
//
static RISCV_CHECK_MEM_MT_FN(emitCheckStoreHSP) {

    vmiosObjectP object = clientData;

    if(applyHSPMT(riscv, object, base)) {
        vmimtArgProcessor();
        vmimtArgNatAddress(object);
        vmimtArgUns32(offset);
        vmimtArgUns32(BITS_TO_BYTES(bits));
        vmimtCallAttrs((vmiCallFn)doHSPCheckStore, VMCA_NA);
    }
}
```

Usage Context
Leaf level only.

14.49 Function `requireModeMt`

```
#define RISCV_REQUIRE_MODE_MT_FN(_NAME) Bool _NAME( \
    riscvP    riscv,    \
    riscvMode mode      \
)
typedef RISCV_REQUIRE_MODE_MT_FN((*riscvRequireModeMtFn));
```

Description
This function must be called at morph time (from within the extension object `morphCB`).

This interface function validates that the current processor mode is at least the given mode and emits code to generate either an Illegal Instruction or Virtual Instruction exception if not. If the required mode is Machine mode, or the processor is not currently in Virtual Supervisor or Virtual User mode, then an Illegal Instruction exception is taken; otherwise, a Virtual Instruction exception is taken.

The return value is `True` if the processor is executing in a sufficiently high privilege mode and `False` if not.

Example

```
inline static Bool requireModeMT(riscvP riscv, riscvMode required) {
    return riscv->cb.requireModeMt(riscv, required);
}
```

Usage Context
Leaf level only, morph-time.

14.50 Function `requireNotVMt`

```
#define RISCV_REQUIRE_NOT_V_MT_FN(_NAME) Bool _NAME(riscvP riscv)
typedef RISCV_REQUIRE_NOT_V_MT_FN((*riscvRequireNotVMtFn));
```

Description

This function must be called at morph time (from within the extension object `morphCB`).

This interface function validates that the current processor mode is not a virtual mode; if it is, code to take a Virtual Instruction exception is emitted.

The return value is `True` if the processor is executing in a non-virtual mode and `False` if not.

Example

```
inline static Bool requireNonVirtual(riscvP riscv) {
    return riscv->cb.requireNotVMt(riscv);
}
```

Usage Context

Leaf level only, morph-time.

14.51 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, morph-time.

14.52 Function `morphTrapTVM`

```
#define RISCV_MORPH_TRAP_TVM_FN(_NAME) void _NAME(riscvP riscv)
typedef RISCV_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, morph-time.

14.53 Function morphVOp

```
#define RISCV_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 RISCV_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_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
    RVVW_V1F_V1F_V1I_UP,    // SEW, VFSLIDEUP instructions
    RVVW_V1F_V1F_V1I_DN,    // SEW, VFSLIDEDOWN instructions

                            // 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_P1I_P1I_P1I_VMSF,  // SEW = SEW op SEW, VMSBF/VMSOF/VMSIF instructions
    RVVW_V1I_P1I_P1I_IOTA,  // SEW = SEW op SEW, VIOTA instruction
    RVVW_V1I_P1I_P1I_ID,    // SEW = SEW op SEW, VID instruction

                            // 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, morph-time.

14.54 Function newCSR

```
#define RISCV_NEW_CSR_FN(_NAME) void _NAME( \
    riscvCSRAttrsP  attrs,          \
    riscvCSRAttrsCP src,            \
    riscvP          riscv,          \
    vmiosObjectP    object          \
)
typedef RISCV_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, in constructor.

14.55 Function hpmAccessValid

```
#define HPM_ACCESS_VALID_FN(_NAME) Bool _NAME( \
    riscvCSRAttrsCP attrs,          \
    riscvP          riscv           \
)
typedef HPM_ACCESS_VALID_FN((*riscvHPMAccessValidFn));

typedef struct riscvModelCBS {
    riscvHPMAccessValidFn     hpmAccessValid;
} riscvModelCB;
```

Description

Access to performance counter CRSs is controlled by number of other registers (`mcounteren`, `scounteren` and, if Hypervisor mode is implemented, `hcounteren`). This function implements the logic of that control and returns a Boolean indicating whether access to a performance counter register defined by the `attrs` argument is legal in the current processor mode. It is useful when implementing custom performance counter CSRs in a derived model.

Example

```
static RISCV_CSR_READFN(mtimeR) {

    vmiosObjectP object = attrs->object;
    Uns64        result = 0;

    if(riscv->artifactAccess) {
        // no action
    } else if(!riscv->cb.hpmAccessValid(attrs, riscv)) {
        // invalid access, standard exception
    } else {
        customTimeException(object);
    }

    return result;
}
```

Usage Context

Leaf level only.

14.56 Function `mapAddress`

```
#define RISCV_MAP_ADDRESS_FN(_NAME) Bool _NAME( \
    riscvP         riscv,           \
    memDomainP     domain,          \
    memPriv        requiredPriv,    \
    Uns64          address,         \
    Uns32          bytes,           \
    memAccessAttrs attrs            \
)
typedef RISCV_MAP_ADDRESS_FN((*riscvMapAddressFn));

typedef struct riscvModelCBS {
    riscvMapAddressFn         mapAddress;
} riscvModelCB;
```

Description

This function can be called by a derived model to attempt to establish a memory mapping for the given address and access size (`bytes`) in the given memory domain object, which must be one of the domains corresponding to an address space for the processor. The function returns `True` if the mapping was *unsuccessful* because a virtual memory address mapping failure and `False` otherwise. The function also establishes any permission restrictions implied by PMP/PMA regions (if implemented). The `attrs` parameter controls whether a mapping failure causes the processor to take an exception.

This function is usually called from within `rdSnapCB` or `wrSnapCB` callbacks to implement alignment checks.

Example

```
static memPriv getDomainPrivileges(
    riscvP     riscv,
    memDomainP domain,
    Uns64      address,
    memPriv    priv
) {
    memPriv mappedPriv = vmirtGetDomainPrivileges(domain, address);

    // if no privilege is set, try mapping memory and get privilege again
    if(!mappedPriv) {
        riscv->cb.mapAddress(riscv, domain, priv, address, 1, MEM_AA_FALSE);
        mappedPriv = vmirtGetDomainPrivileges(domain, address);
    }

    return mappedPriv;
}
```

Usage Context

Leaf level only.

14.57 Function unmapPMPRegion

```
#define RISCV_UNMAP_PMP_REGION_FN(_NAME) void _NAME( \
    riscvP riscv,                   \
    Uns32  regionIndex              \
)
typedef RISCV_UNMAP_PMP_REGION_FN((*riscvUnmapPMPRegionFn));

typedef struct riscvModelCBS {
    riscvUnmapPMPRegionFn     unmapPMPRegion;
} riscvModelCB;
```

Description
This function can be called by a derived model to force the indexed PMP region to be unmapped by the base model. This may be required if the derived model enhances the default permissions applied by the base model (for example, with supplementary custom CSRs).

Example

```
static void unmapPMPRegion (riscvP riscv, Uns32 regionIndex) {
    riscv->cb.unmapPMPRegion(riscv, regionIndex);
}
```

Usage Context
Leaf level only.

14.58 Function updateLdStDomain

```
#define RISCV_UPDATE_LD_ST_DOMAIN_FN(_NAME) void _NAME(riscvP riscv)
typedef RISCV_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) {
    riscv->cb.updateLdStDomain(riscv);
}
```

Usage Context
Leaf level only.

14.59 Function newTLBEntry

```
#define RISCV_NEW_TLB_ENTRY_FN(_NAME) void _NAME( \
    riscvP            riscv,        \
    riscvTLBId        tlbId,        \
    riscvExtVMMapping mapping       \
)
typedef RISCV_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 {

    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 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.

14.60 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 14.59 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.

14.61 Function newExtReg

```
#define RISCV_NEW_EXT_REG_FN(_NAME) void _NAME( \
    riscvP       riscv,             \
    vmiRegInfoCP extReg             \
)
typedef RISCV_NEW_EXT_REG_FN((*riscvNewExtRegFn));

typedef struct riscvModelCBS {
    riscvNewExtRegFn          newExtReg;
} riscvModelCB;
```

Description
Given a template generic extension register description, this function registers that extension register with the base model. It should always be called from the extension object constructor. See section 11.4 for a detailed description.

Example

```
static void addExtRegs(vmiosObjectP object) {

    riscvP riscv = object->riscv;

    // create description of read-only transaction mode register
    vmiRegInfo tmReg = {
        name        : "TM",
        description : "Transaction mode state",
        bits        : 8,
        gdbIndex    : 0,
        access      : vmi_RA_R,
        raw         : vmimtGetExtReg((vmiProcessorP)riscv, &object->tmStatus)
    };

    // add register description
    riscv->cb.newExtReg(riscv, &tmReg);
}
```

Usage Context
Leaf level only, in constructor.

15 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
    riscvRdWrFaultFn          rdFaultCB;
    riscvRdWrFaultFn          wrFaultCB;
    riscvRdWrSnapFn           rdSnapCB;
    riscvRdWrSnapFn           wrSnapCB;

    // exception actions
    riscvSuppressMemExceptFn  suppressMemExcept;
    riscvCustomNMIFn          customNMI;
    riscvCustomIAssignFn      customIAssign;
    riscvTrapNotifierFn       trapNotifier;
    riscvTrapNotifierFn       trapPreNotifier;
    riscvTrapNotifierFn       ERETNotifier;
    riscvResetNotifierFn      preResetNotifier;
    riscvResetNotifierFn      resetNotifier;
    riscvFirstExceptionFn     firstException;
    riscvGetInterruptPriFn    getInterruptPri;
    riscvGetHandlerPCFn       getHandlerPC;
    riscvIntUpdateFn          intUpdate;

    // halt/restart actions
    riscvHRNotifierFn         haltRestartNotifier;
    riscvLRSCAbortFn          LRSCAbortFn;

    // code generation actions
    riscvDerivedMorphFn       preMorph;
    riscvDerivedMorphFn       postMorph;
    riscvDerivedMorphFn       AMOCheck;
    riscvDerivedMorphFn       AMOMorph;
    riscvEmitCSRCheckFn       emitCSRCheck;
    riscvUnitStrideCheckFn    unitStrideCheck;
    riscvVFREDSUMMorphFn      emitVFREDUSUM;
    riscvVFREDSUMMorphFn      emitVFWREDUSUM;

    // transaction support actions
    riscvIASSwitchFn          switchCB;
    riscvTLoadFn              tLoad;
    riscvTStoreFn             tStore;

    // memory access logging support actions
    riscvCheckMemMtFn         checkLoadMt;
    riscvCheckMemMtFn         checkStoreMt;

    // physical memory actions
    riscvDistinctPhysMemFn    distinctPhysMem;
    riscvPhysMemFn            installPhysMem;

    // PMP support actions
    riscvPMPPrivFn            PMPPriv;

    // PMA check actions
```

```
    riscvPMAEnableFn          PMAEnable;
    riscvPMACheckFn           PMACheck;

    // virtual memory actions
    riscvVMTrapFn             VMTrap;
    riscvValidPTEFn           validPTE;
    riscvSetDomainNotifierFn  setDomainNotifier;
    riscvFreeEntryNotifierFn  freeEntryNotifier;

    // CLIC actions
    riscvClicCRdFn            clicCustomRd;
    riscvClicCWrFn            clicCustomWr;
    riscvClicUpdatedFn        clicUpdated;

    // 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.

15.1 Function `rdFaultCB`

```
#define RISCV_RD_WR_FAULT_FN(_NAME) Bool _NAME( \
    riscvP     riscv,           \
    memDomainP domain,          \
    Addr       address,         \
    Uns32      bytes,           \
    void      *clientData       \
)
typedef RISCV_RD_WR_FAULT_FN((*riscvRdWrFaultFn))

typedef struct riscvExtCBS {
    riscvRdWrFaultFn          rdFaultCB;
} riscvExtCB;
```

Description

This interface function is called from the base model when an unaligned *load* access is detected. The access is of size `bytes` at address `address`. The function should return `True` if the access should cause a Load Access Fault exception (code 5) and `False` if the access should cause a Load Address Misaligned exception (code 4).

Example

This example shows how to force all misaligned loads to be reported as access faults:

```
static RISCV_RD_WR_FAULT_FN(misalignedAccess) {
    return True;
}

static VMIOS_CONSTRUCTOR_FN(extConstructor) {

    . . . lines omitted . . .
    object->extCB.rdFaultCB = misalignedAccess;
    . . . lines omitted . . .
}
```

15.2 Function `wrFaultCB`

```
#define RISCV_RD_WR_FAULT_FN(_NAME) Bool _NAME( \
    riscvP     riscv,           \
    memDomainP domain,          \
    Addr       address,         \
    Uns32      bytes,           \
    void      *clientData       \
)
typedef RISCV_RD_WR_FAULT_FN((*riscvRdWrFaultFn))

typedef struct riscvExtCBS {
    riscvRdWrFaultFn          wrFaultCB;
} riscvExtCB;
```

Description

This interface function is called from the base model when an unaligned *store* access is detected. The access is of size `bytes` at address `address`. The function should return `True` if the access should cause a Store/AMO Access Fault exception (code 7) and `False` if the access should cause a Store/AMO Address Misaligned exception (code 6).

Example

This example shows how to force all misaligned stores to be reported as access faults:

```
static RISCV_RD_WR_FAULT_FN(misalignedAccess) {
    return True;
}

static VMIOS_CONSTRUCTOR_FN(extConstructor) {

    . . . lines omitted . . .
    object->extCB.wrFaultCB = misalignedAccess;
    . . . lines omitted . . .
}
```

15.3 Function rdSnapCB

```
#define RISCV_RD_WR_SNAP_FN(_NAME) Uns32 _NAME( \
    riscvP     riscv,           \
    memDomainP domain,          \
    Addr       address,         \
    Uns32      bytes,           \
    atomicCode atomic,          \
    void      *clientData       \
)
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 *load* 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 . . .
}
```

15.4 Function wrSnapCB

```
#define RISCV_RD_WR_SNAP_FN(_NAME) Uns32 _NAME( \
    riscvP     riscv,           \
    memDomainP domain,          \
    Addr       address,         \
    Uns32      bytes,           \
    atomicCode atomic,          \
    void      *clientData       \
)
typedef RISCV_RD_WR_SNAP_FN((*riscvRdWrSnapFn));

typedef struct riscvExtCBS {
    riscvRdWrSnapFn           wrSnapCB;
} riscvExtCB;
```

Description

This interface function is called from the base model when an unaligned *store* 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.wrSnapCB = snapCB;
    . . . lines omitted . . .
}
```

15.5 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 . . .
}
```

15.6 Function customNMI

```
#define RISCV_CUSTOM_NMI_FN(_NAME) Bool _NAME( \
    riscvP riscv,               \
    void  *clientData           \
)
typedef RISCV_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.

If the callback returns `False`, then depending on whether the RISC-V RNMI extension is configured, *the callback must set the full value of either `mcause` or `mncause` before returning*. The base model will then perform all other actions to indicate an M-mode exception has been taken, either as a standard exception or as a standard RNMI trap (depending on whether RNMI is configured). As part of this, any exception code specified by the `ecode_nmi` parameter or driven using the `nmi_cause` input signal will be combined with the value of `mcause` or `mncause` set by the extension callback using bitwise-or.

If the callback returns `True`, then no standard actions will be performed by the base model to handle the NMI: all required state changes must be made by the extension.

Example
This example shows how an extension could handle an NMI exception as a normal trap with custom cause:

```
static RISCV_CUSTOM_NMI_FN(customNMI) {

    riscv->cb.takeException(riscv, riscv_E_Interrupt+EXT_NMI, 0);

    return True;
}

static VMIOS_CONSTRUCTOR_FN(extConstructor) {

    . . . lines omitted . . .
    object->extCB.customNMI = customNMI;
    . . . lines omitted . . .
}
```

15.7 Function customIAssign

```
#define RISCV_CUSTOM_IASSIGN_FN(_NAME) Uns64 _NAME( \
    riscvP              riscv,      \
    Uns64               ip,         \
    riscvBasicIntStateP intState,   \
    void               *clientData  \
)
typedef RISCV_CUSTOM_IASSIGN_FN((*riscvCustomIAssignFn));

typedef struct riscvExtCBS {
    riscvCustomIAssignFn      customIAssign;
} riscvExtCB;
```

Description
This interface function allows an extension to modify the allocation of standard interrupts to privilege levels, or to inject supplementary interrupts that are not visible using the standard `mip` CSR. It is called when the base model has determined the appropriate privilege levels for enabled standard interrupts but before the highest-priority interrupt has been determined.

The function is passed a pointer to an object of type `riscvBasicIntState`:

```
typedef struct riscvBasicIntStateS {
    Uns64 ip[RISCV_MODE_LAST];  // pending and enabled interrupts per mode
    Bool  hvictl;               // whether hvictl-injected interrupt
} riscvBasicIntState;
```

In this structure, the `ip` array holds bitmasks of pending and enabled interrupts that have been delegated to each privilege mode. The function can modify the values here to add or remove pending interrupts or change their priority assignment.

The function is also passed an `ip` parameter, which is the bitwise-or of pending and enabled interrupts at all privilege levels. It should return a value consistent with any changes it makes to interrupt assignments.

Example
Andes processors implement custom `slip`, `slie` and `mslideleg` CSRs allowing performance monitor interrupts to be delivered to S-mode even when those interrupts are disabled at M-mode. This is handled in the following custom interrupt assignment function:

```
RISCV_CUSTOM_IASSIGN_FN(andesCustomIAssign) {

    vmiosObjectP object = clientData;
    Uns32        slip   = RD_XCSR(object, slip) & RD_XCSR(object, slie);

    if(slip) {

        Uns32 mslideleg = RD_XCSR(object, mslideleg);

        // indicate slip interrupts are pending and enabled
        ip |= slip;

        // route interrupts to either M-mode or S-mode
        intState->ip[RISCV_MODE_M] |= (slip & ~mslideleg);
```

```
        intState->ip[RISCV_MODE_S] |= (slip &  mslideleg);
    }

    return ip;
}

static VMIOS_CONSTRUCTOR_FN(constructor) {

    . . . lines omitted . . .
          object->extCB.customIAssign = andesCustomIAssign;
    . . . lines omitted . . .
}
```

15.8 Function `trapNotifier`

```
#define RISCV_TRAP_NOTIFIER_FN(_NAME) void _NAME( \
    riscvP        riscv,        \
    riscvMode     mode,         \
    riscvERETType eretType,     \
    void         *clientData    \
)
typedef RISCV_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). The notifier is called when all hart state has been modified to handle the trap: section 15.9 describes an alternative notifier that is called *before* hart state is modified.

The `mode` argument specifies the mode to which the trap is taken.

The `eretType` argument is significant only when the callback is used in the context of an exception return notifier (see section 15.10). Here, it is always passed the value `ERT_NA`.

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 RISCV_TRAP_NOTIFIER_FN(takeTrap) {

    vmiosObjectP object = clientData;

    if(
        (mode==RISCV_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 . . .
}
```

15.9 Function `trapPreNotifier`

```
#define RISCV_TRAP_NOTIFIER_FN(_NAME) void _NAME( \
    riscvP        riscv,        \
    riscvMode     mode,         \
    riscvERETType eretType,     \
    void         *clientData    \
)
typedef RISCV_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. Section 15.8 describes an alternative notifier that is called *after* hart state is modified.

The `mode` argument specifies the mode to which the trap is taken.

The `eretType` argument is significant only when the callback is used in the context of an exception return notifier (see section 15.10). Here, it is always passed the value `ERT_NA`.

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 RISCV_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 . . .
}
```

15.10 Function ERETNotifier

```
#define RISCV_TRAP_NOTIFIER_FN(_NAME) void _NAME( \
    riscvP        riscv,        \
    riscvMode     mode,         \
    riscvERETType eretType,     \
    void         *clientData    \
)
typedef RISCV_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. It gives the extension library the opportunity to modify exception return behavior (perhaps by restoring extra custom information).

The `mode` argument specifies the mode from which the trap return is being made.

The `eretType` argument indicates the exact type of exception return instruction that has been executed, described by the `riscvERETType` enumeration:

```
typedef enum riscvERETTypeE {
    ERT_NA,     // not an exception return instruction
    ERT_M,      // mret instruction
    ERT_D,      // dret instruction
    ERT_MN,     // mnret instruction (from resumable NMI)
    ERT_S,      // sret (from S/HS mode)
    ERT_VS,     // sret (from VS mode)
    ERT_U,      // uret (from U/HU mode)
    ERT_VU,     // uret (from VU mode)
} riscvERETType;
```

Example

This example shows how the `andes.ovpworld.org` model uses this function to implement the Andes-specific `CRASHSAVE` extension and restore custom fields in the `mxstatus` CSR when executing an `MRET` instruction:

```
static RISCV_TRAP_NOTIFIER_FN(ERETNotifier) {

    vmiosObjectP object = clientData;

    // restore CRASHSAVE extension information if required
    if(RD_CSR_FIELD(object->riscv, mcause, Interrupt)) {
        // no action if last cause was an interrupt
    } else if(eretType!=ERT_M) {
        // no action unless an mret instruction
    } else if(!RD_XCSR_FIELD(object, mmsc_cfg, CRASHSAVE)) {
        // CRASHSAVE feature is absent
    } else {
        WR_CSR( object->riscv, mepc,    RD_XCSR(object, msaveepc1));
        WR_CSR( object->riscv, mcause,  RD_XCSR(object, msavecause1));
        WR_XCSR(object,        mdcause, RD_XCSR(object, msavedcause1));
    }

    // restore mxstatus fields when MRET is executed
    if(eretType==ERT_M) {
        COPY_FIELD(object, mxstatus, PFT_EN, PPFT_EN);
```

```
        COPY_FIELD(object, mxstatus, IME,    PIME);
        COPY_FIELD(object, mxstatus, DME,    PDME);
        COPY_FIELD(object, mxstatus, TYP,    PTYP);
    }

    // refresh all counter objects
    refreshCounters(object);
}

void andesCSRInit(vmiosObjectP object) {

    . . . lines omitted . . .
    object->extCB.ERETNotifier = ERETNotifier;
    . . . lines omitted . . .
}
```

15.11 Function preResetNotifier

```
#define RISCV_RESET_NOTIFIER_FN(_NAME) void _NAME( \
    riscvP riscv,               \
    void  *clientData           \
)
typedef RISCV_RESET_NOTIFIER_FN((*riscvResetNotifierFn));

typedef struct riscvExtCBS {
    riscvResetNotifierFn      preResetNotifier;
} riscvExtCB;
```

Description

This interface function is called from the base model when a reset is executed, *before* any standard reset actions have been performed. It allows the extension to implement custom features such as crash state logging, which require saving current processor state before it is modified as part of the reset process.

Example

This example shows a template for crash state logging based on a feature defined for Andes RISC-V cores:

```
//
// Fill a field in mcrash_statesave
//
#define FILL_STATESAVE(_O, _F, _V) WR_XCSR_FIELD(_O, mcrash_statesave, _F, _V)

//
// Perform CSR changes prior to reset
//
static RISCV_RESET_NOTIFIER_FN(CSRPreReset) {

    vmiosObjectP object = clientData;
    riscvMode    mode   = getCurrentMode3(riscv);

    // fill mcrash_statesave
    FILL_STATESAVE(object, MIE,     RD_CSR_FIELD(riscv, mstatus, MIE));
    FILL_STATESAVE(object, CP,      mode);
    FILL_STATESAVE(object, PPFT_EN, RD_XCSR_FIELD(object, mxstatus, PPFT_EN));
    FILL_STATESAVE(object, PIME,    RD_XCSR_FIELD(object, mxstatus, PIME));
    FILL_STATESAVE(object, PDME,    RD_XCSR_FIELD(object, mxstatus, PDME));
    FILL_STATESAVE(object, PTYP,    RD_XCSR_FIELD(object, mxstatus, PTYP));

    . . . lines omitted . . .
}

static VMIOS_CONSTRUCTOR_FN(extConstructor) {

    . . . lines omitted . . .
    object->extCB.preResetNotifier = CSRPreReset;
    . . . lines omitted . . .
}
```

15.12 Function `resetNotifier`

```
#define RISCV_RESET_NOTIFIER_FN(_NAME) void _NAME( \
    riscvP riscv,               \
    void  *clientData           \
)
typedef RISCV_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, *after* all standard reset actions have been performed. It gives the extension 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 RISCV_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 . . .
}
```

15.13 Function `firstException`

```
#define RISCV_FIRST_EXCEPTION_FN(_NAME) vmiExceptionInfoCP _NAME( \
    riscvP riscv,               \
    void  *clientData           \
)
typedef RISCV_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 RISCV_FIRST_EXCEPTION_FN(firstException) {
    return exceptions;
}

static VMIOS_CONSTRUCTOR_FN(extConstructor) {

    . . . lines omitted . . .
    object->extCB.firstException = firstException;
    . . . lines omitted . . .
}
```

15.14 Function `getInterruptPri`

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

static VMIOS_CONSTRUCTOR_FN(extConstructor) {

    . . . lines omitted . . .
    object->extCB.getInterruptPri = getInterruptPriority;
    . . . lines omitted . . .
}
```

15.15 Function `getHandlerPC`

```
#define RISCV_GET_HANDLER_PC_FN(_NAME) Bool _NAME( \
    riscvP         riscv,       \
    Uns64          tvec,        \
    riscvException exception,   \
    Uns32          ecode,       \
    Uns64         *handlerPCP,  \
    void          *clientData   \
)
typedef RISCV_GET_HANDLER_PC_FN((*riscvGetHandlerPCFn));

typedef struct riscvExtCBS {
    riscvGetHandlerPCFn       getHandlerPC;
} riscvExtCB;
```

Description

This interface function allows an extension model to return a custom interrupt or exception handler address. It is called when a trap occurs and is passed the *xtvec* address for the target mode, the exception that is being taken, the exception code (usually derived from the exception, but can differ for interrupts when the external interrupt ID signals are driven) and a pointer to an `Uns64` handler PC to be filled with the desired handler address if required.

If the custom handler address function is active, it should fill the `handlerPCP` result and return `True`; otherwise, it should return `False`.

Example

This example shows how this function is used in the `andes.ovpworld.org` model. In this model, a custom vectored trap mode is enabled when `mmisc_ctl.VEC_PLIC=1`:

```
RISCV_GET_HANDLER_PC_FN(andesGetHandlerPC) {

    vmiosObjectP object = clientData;
    Bool         custom = RD_XCSR_FIELD(object, mmisc_ctl, VEC_PLIC);

    if(custom) {

        riscvMode      mode     = getCurrentMode5(riscv);
        memEndian      endian   = riscv->cb.getDataEndian(riscv, mode);
        memDomainP     domain   = getDataDomain(riscv);
        memAccessAttrs memAttrs = MEM_AA_TRUE;
        Uns32          offset   = 0;
        Uns64          handlerPC;

        // get table offset for this exception type
        switch(exception) {

            case riscv_E_UExternalInterrupt:
            case riscv_E_SExternalInterrupt:
            case riscv_E_MExternalInterrupt:
                offset = ecode*4;
                break;

            default:
                break;
        }

        // read 4-byte table entry
        handlerPC = vmirtRead4ByteDomain(domain, tvec+offset, endian, memAttrs);
```

```
        // mask off LSB from result
        *handlerPCP = handlerPC &= -2;
    }

    return custom;
}

static VMIOS_CONSTRUCTOR_FN(extConstructor) {

    . . . lines omitted . . .
        object->extCB.getHandlerPC   = andesGetHandlerPC;
    . . . lines omitted . . .
}
```

15.16 Function intUpdate

```
#define RISCV_INTERRUPT_UPDATE_FN(_NAME) Bool _NAME( \
    riscvP           riscv,      \
    Uns32            intNum,     \
    Uns64            *newValue,  \
    void             *clientData \
)
typedef RISCV_INTERRUPT_UPDATE_FN((*riscvIntUpdateFn));
```

Description
This notifier function informs the extension when a write has occurred on an interrupt input pin of a processor.

intNum is the interrupt number where the write occurred.

*newValue is a pass by reference of the value that was written. The value may be changed in the notifier.

If the function returns true then a re-evaluation of the interrupt state will be forced, regardless of the value that was written (otherwise, re-evaluation will occur only if the new value causes a change in the interrupt state.) This should be done if the interrupt state is modified in the callback,

Example
This example shows how an extension can keep state info on the values on the first 32 interrupt inputs:

```
static RISCV_INTERRUPT_UPDATE_FN(intUpdate) {
    vmiosObjectP object = clientData;

    if (intNum < 32) {
        Uns32 mask      = 1 << intNum;
        Bool  intIn     = *newValue != 0;

        if (intIn) {
            object->interruptLevels |= mask;
        } else {
            object->interruptLevels &= ~mask;
        }
    }

    return False;
}
```

15.17 Function haltRestartNotifier

```
#define RISCV_HR_NOTIFIER_FN(_NAME) void _NAME( \
    riscvP riscv,               \
    void  *clientData           \
)
typedef RISCV_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 RISCV_HR_NOTIFIER_FN(haltRestartNotifier) {
    refreshCounters(clientData);
}

static VMIOS_CONSTRUCTOR_FN(constructor) {

    . . . lines omitted . . .
    object->extCB.haltRestartNotifier = haltRestartNotifier;
    . . . lines omitted . . .
}
```

15.18 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_CUSTOM_WFI) {
        riscv->cb.restart(riscv, RVD_CUSTOM_WFI);
    }
}

static VMIOS_CONSTRUCTOR_FN(extConstructor) {

    . . . lines omitted . . .
    object->extCB.LRSCAbortFn = custLRSCAbort;
    . . . lines omitted . . .
}
```

15.19 Function `preMorph`

```
#define RISCV_DERIVED_MORPH_FN(_NAME) void _NAME( \
    riscvP          riscv,      \
    void           *clientData, \
    riscvInstrInfoP instrInfo,  \
    Uns64           thisPC      \
)
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. The function is passed the address of the instruction (`thisPC`) and a pointer to a structure giving information about the decoded instruction (`instrInfo`); see section 16 for more information about this structure.

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. The callback is also used to specify the alignment constraints for any load/store instructions (there is a custom register that enables or disables unaligned access for some instruction types).

```
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;
    andesMorphState state;

    // handle hardware stack protection if required
    if(RD_XCSR_FIELD(object, mmsc_cfg, HSP)) {
        andesRecordSP(riscv, object);
    }

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

    if(state.info.type == AN_IT_MSA_UNA) {

        // base model instructions controlled by mmisc_ctl.MSA_UNA
```

```
        riscv->configInfo.unaligned = RD_XCSR_FIELD(object, mmisc_ctl, MSA_UNA);

    } else if(state.info.type == AN_IT_LAST) {

        // other base model instructions always disallow unaligned access
        riscv->configInfo.unaligned = False;
    }
}

static VMIOS_CONSTRUCTOR_FN(constructor) {

    . . . lines omitted . . .
    object->extCB.preMorph       = andesPreMorph;
    . . . lines omitted . . .
}
```

15.20 Function postMorph

```
#define RISCV_DERIVED_MORPH_FN(_NAME) void _NAME( \
    riscvP          riscv,      \
    void           *clientData, \
    riscvInstrInfoP instrInfo,  \
    Uns64           thisPC      \
)
typedef RISCV_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. The function is passed the address of the instruction (`thisPC`) and a pointer to a structure giving information about the decoded instruction (`instrInfo`); see section 16 for more information about this structure.

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. There is also a requirement to revert unaligned access behavior to a default state:

```
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);
    }
}

RISCV_DERIVED_MORPH_FN(andesPostMorph) {

    vmiosObjectP object = clientData;

    // handle hardware stack protection if required
    if(RD_ACSR_FIELD(object, mmsc_cfg, HSP)) {
        andesCheckHSP(riscv, object);
    }

    // reset default unaligned access behavior (required for PMP updates to
    // be correctly handled)
    riscv->configInfo.unaligned = True;
}

static VMIOS_CONSTRUCTOR_FN(constructor) {
```

```
    . . . lines omitted . . .
    object->extCB.postMorph      = andesPostMorph;
    . . . lines omitted . . .
}
```

15.21 Function AMOCheck

```
#define RISCV_DERIVED_MORPH_FN(_NAME) void _NAME( \
    riscvP          riscv,      \
    void           *clientData, \
    riscvInstrInfoP instrInfo,  \
    Uns64           thisPC      \
)
typedef RISCV_DERIVED_MORPH_FN((*riscvDerivedMorphFn));

typedef struct riscvExtCBS {
    riscvDerivedMorphFn       AMOCheck;
} riscvExtCB;
```

Description

This interface function is called *before* any code is emitted for an atomic memory access (AMO) instruction. It allows the extension object to emit code to validate the legality of the AMO instruction in a custom manner if required. The custom behavior will be performed before any other behavior of that instruction. The function is passed the address of the instruction (`thisPC`) and a pointer to a structure giving information about the decoded instruction (`instrInfo`); see section 16 for more information about this structure.

See also interface function `AMOMorph`, which allows custom behavior to be specified at a later stage in execution of AMO instructions.

Example

This example shows how this function can be used to cause all AMO instructions to take a custom exception if a custom CSR enable bit `custctl.AMOEN` is zero.

```
static void illegalCustom(
    vmiosObjectP   object,
    riscvException exception,
    const char    *reason
) {
    riscvP riscv = object->riscv;
    riscv->cb.illegalCustom(riscv, exception, reason);
}

void customAMOCheck(vmiosObjectP object) {
    if(!RD_XCSR_FIELD(object, custctl, AMOEN)) {
        illegalCustom(object, custom_E_IllegalInstruction, "Custom AMO disable");
    }
}

RISCV_DERIVED_MORPH_FN(customAMOCheck) {

    vmimtArgNatAddress(clientData);
    vmimtCallAttrs((vmiCallFn)customAMOCheck, VMCA_NA);
}

static VMIOS_CONSTRUCTOR_FN(constructor) {

    . . . lines omitted . . .
    object->extCB.AMOCheck       = customAMOCheck;
    . . . lines omitted . . .
}
```

15.22 Function AMOMorph

```
#define RISCV_DERIVED_MORPH_FN(_NAME) void _NAME( \
    riscvP          riscv,      \
    void           *clientData, \
    riscvInstrInfoP instrInfo,  \
    Uns64           thisPC      \
)
typedef RISCV_DERIVED_MORPH_FN((*riscvDerivedMorphFn));

typedef struct riscvExtCBS {
    riscvDerivedMorphFn       AMOMorph;
} riscvExtCB;
```

Description

This interface function allows an extension object to emit code that is executed *after* permission checks have been performed for an atomic memory access (AMO) instruction, but *before* any of the accesses specified for that instruction have been performed. It allows the extension object to modify the behavior of the AMO instruction in a custom manner if required. The function is passed the address of the instruction (`thisPC`) and a pointer to a structure giving information about the decoded instruction (`instrInfo`); see section 16 for more information about this structure.

See also interface function `AMOCheck`, which allows custom behavior to be specified at an earlier stage in execution of AMO instructions.

Example

This example shows how this function can be used to cause all AMO instructions to take a custom exception if the permission checks succeed. The effect will be that AMO instructions preferentially take any access exceptions implied by their behavior, and then, if there are no exceptions generated as a result, a custom trap is taken.

```
static void illegalCustom(
    vmiosObjectP   object,
    riscvException exception,
    const char    *reason
) {
    riscvP riscv = object->riscv;
    riscv->cb.illegalCustom(riscv, exception, reason);
}

void customAMOException(vmiosObjectP object) {
    illegalCustom(object, custom_E_IllegalInstruction, "Custom AMO emulation");
}

RISCV_DERIVED_MORPH_FN(customAMOMorph) {

    vmimtArgNatAddress(clientData);
    vmimtCallAttrs((vmiCallFn)customAMOException, VMCA_EXCEPTION);
}

static VMIOS_CONSTRUCTOR_FN(constructor) {

    . . . lines omitted . . .
    object->extCB.AMOMorph       = customAMOMorph;
    . . . lines omitted . . .
}
```

15.23 Function `emitCSRCheck`

```
#define RISCV_EMIT_CSR_CHECK_FN(_NAME) void _NAME( \
    riscvP          riscv,      \
    void           *clientData, \
    riscvInstrInfoP instrInfo,  \
    Bool            isRead,     \
    Bool            isWrite,    \
    Bool            useRS1      \
)
typedef RISCV_EMIT_CSR_CHECK_FN((*riscvEmitCSRCheckFn));

typedef struct riscvExtCBS {
    riscvEmitCSRCheckFn       emitCSRCheck;
} riscvExtCB;
```

Description

Usually, CSR access constraints can be cleanly handled when a CSR is defined (as shown in section 7). Sometimes, however, CSRs have custom access constraints that cannot be described in the standard definition. For example, a CSR may permit access using `csrrw` instructions but deny access using `csrrs` or `csrrc` instructions. In such cases, the custom *CSR check* interface function can be used. This function is called before any behavioral code is emitted to implement the CSR and allows the extension to emit higher-priority checks that cause an exception to be taken if access constraints are not satisfied.

The function is passed the following arguments, in addition to the `riscv` and `clientData`:

1. `instrInfo`: this is a pointer to a structure giving information about the decoded CSR access instruction - see section 16 for more information;
2. `isRead`: this indicates whether the instruction is reading the CSR.
3. `isWrite`: this indicates whether the instruction is writing the CSR.
4. `useRS1`: if `True`, this indicates the instruction is writing the CSR and taking the value to write from the GPR specified in the `rs1` field of the instruction.

CSR-related fields from the `instrInfo` structure can be extracted for use in this function as follows:

```
    // CSR index number
    Uns32 csr = instrInfo->csr;

    // CSR update semantics:
    //    RV_CSR_RW : read/write
    //    RV_CSR_RS : read/set
    //    RV_CSR_RC : read/clear
    riscvCSRUDesc csrUpdate = instrInfo->csrUpdate;

    // whether constant is used (CSRRI)
    Bool useC = (instrInfo->type==RV_IT_CSRRI_I);

    // constant value (if CSRRI)
    Uns32 c = instrInfo->c;
```

To clarify parameter and `instrInfo` field use with different classes of instruction, the following table shows values that will be used for different types of CSR access instruction.

Disassembly	Alias	csrUpdate	isRead	isWrite	useRS1	useC
csrrw x0,satp,x0	csrw satp,x0	RV_CSR_RW	False	True	False	False
csrrw x1,satp,x0		RV_CSR_RW	True	True	False	False
csrrw x0,satp,x1	csrw satp,x1	RV_CSR_RW	False	True	True	False
csrrw x1,satp,x2		RV_CSR_RW	True	True	True	False
csrrs x0,satp,x0	csrr x0,satp	RV_CSR_RS	True	False	False	False
csrrs x1,satp,x0	csrr x1,satp	RV_CSR_RS	True	False	False	False
csrrs x0,satp,x1	csrs satp,x1	RV_CSR_RS	True	True	True	False
csrrs x1,satp,x2		RV_CSR_RS	True	True	True	False
csrrc x0,satp,x0	csrc satp,x0	RV_CSR_RC	True	False	False	False
csrrc x1,satp,x0		RV_CSR_RC	True	False	False	False
csrrc x0,satp,x1	csrc satp,x1	RV_CSR_RC	True	True	True	False
csrrc x1,satp,x2		RV_CSR_RC	True	True	True	False
csrrwi x0,satp,0	csrwi satp,0	RV_CSR_RW	False	True	False	True
csrrwi x1,satp,0		RV_CSR_RW	True	True	False	True
csrrwi x0,satp,1	csrwi satp,1	RV_CSR_RW	False	True	False	True
csrrwi x1,satp,1		RV_CSR_RW	True	True	False	True
csrrsi x0,satp,0	csrsi satp,0	RV_CSR_RS	True	False	False	True
csrrsi x1,satp,0		RV_CSR_RS	True	False	False	True
csrrsi x0,satp,1	csrsi satp,1	RV_CSR_RS	True	True	False	True
csrrsi x1,satp,1		RV_CSR_RS	True	True	False	True
csrrci x0,satp,0	csrci satp,0	RV_CSR_RC	True	False	False	True
csrrci x1,satp,0		RV_CSR_RC	True	False	False	True
csrrci x0,satp,1	csrci satp,1	RV_CSR_RC	True	True	False	True
csrrci x1,satp,1		RV_CSR_RC	True	True	False	True

Example

This example shows how this function can be used to deny all accesses to a set of CSRs unless those accesses use the `csrrw` instruction variant with `rs1` that is not `x0`.

```
static RISCV_EMIT_CSR_CHECK_FN(emitCSRCheck) {

    Uns32         csr       = instrInfo->csr;
    riscvCSRUDesc csrUpdate = instrInfo->csrUpdate;

    if(!((csr==0xBF9) || (csr==0xBFA) || (csr==0xBFC))) {
        // no action
    } else if(isWrite && useRS1 && (csrUpdate==RV_CSR_RW)) {
        // valid csrrw write access using RS1
    } else {
        riscv->cb.morphIllegal(riscv, "illegal access");
    }
}

static VMIOS_CONSTRUCTOR_FN(constructor) {

    . . . lines omitted . . .
    object->extCB.emitCSRCheck = emitCSRCheck;
    . . . lines omitted . . .
}
```

15.24 Function `unitStrideCheck`

```
#define RISCV_UNIT_STRIDE_CHECK_FN(_NAME) void _NAME( \
    riscvP          riscv,      \
    void           *clientData, \
    riscvInstrInfoP instrInfo,  \
    vmiReg          baseAddr,   \
    Uns32           memBits,    \
    Bool            isLoad      \
)
typedef RISCV_UNIT_STRIDE_CHECK_FN((*riscvUnitStrideCheckFn));

typedef struct riscvExtCBS {
    riscvUnitStrideCheckFn    unitStrideCheck;
} riscvExtCB;
```

Description

This interface function is called for Vector extension unit-stride load/store instructions. It allows the extension object to emit code that is executed *after* the validity of the load/store instruction has been verified, but *before* any of the accesses specified for that instruction have been performed. It allows the extension to insert extra custom checks on unit-stride load/store instructions if required.

Argument `instrInfo` gives information about the decoded vector instruction (see section 16 for more information about this structure), argument `baseAddr` is the base address of the load/store, argument `memBits` indicates the element size, and argument `isLoad` is `True` if the instruction is a load and `False` if it is a store.

Example

This example shows how this function can be used to perform an additional custom check that the entire range potentially accessed by a unit-stride load/store is accessible before any accesses are performed.

```
//
// Check vector unit-stride load
//
static void checkVLd(vmiosObjectP object, Uns64 addr, Uns32 memBits) {

    riscvP riscv = object->riscv;
    Uns32  vl    = RD_CSR32(riscv, vl);

    if(vl) {

        memDomainP domain = vmirtGetProcessorDataDomain((vmiProcessorP)riscv);
        Uns32      bytes  = (vl*memBits)/8;

        // do try-load of required address
        vmirtReadNByteDomain(
            domain, addr, 0, bytes, &object->rcache, MEM_AA_TRUE
        );
    }
}

//
// Check vector unit-stride store
//
static void checkVSt(vmiosObjectP object, Uns64 addr, Uns32 memBits) {

    riscvP riscv = object->riscv;
    Uns32  vl    = RD_CSR32(riscv, vl);
```

```
    if(vl) {

        memDomainP domain = vmirtGetProcessorDataDomain((vmiProcessorP)riscv);
        Uns32      bytes  = (vl*memBits)/8;

        // do try-store of required address
        vmirtWriteNByteDomain(
            domain, addr, 0, bytes, &object->rcache, MEM_AA_TRUE
        );
    }
}

static RISCV_UNIT_STRIDE_CHECK_FN(customUnitStrideCheck) {

    vmimtArgNatAddress(clientData);
    vmimtArgReg(VPRRAT_64, baseAddr);
    vmimtArgUns32(memBits);

    if(isLoad) {
        vmimtCall((vmiCallFn)checkVLd);
    } else {
        vmimtCall((vmiCallFn)checkVSt);
    }
}

static VMIOS_CONSTRUCTOR_FN(constructor) {

    . . . lines omitted . . .
          object->extCB.unitStrideCheck = customUnitStrideCheck;
    . . . lines omitted . . .
}
```

15.25 Function `emitVFREDUSUM`

```
#define RISCV_VFREDSUM_MORPH_FN(_NAME) void _NAME( \
    riscvP riscv,       \
    void  *clientData,  \
    vmiReg vdAcc,       \
    void  *vs2,         \
    void  *vm,          \
    Uns32  SEW          \
)
typedef RISCV_VFREDSUM_MORPH_FN((*riscvVFREDSUMMorphFn));

typedef struct riscvExtCBS {
    riscvVFREDSUMMorphFn      emitVFREDUSUM;
} riscvExtCB;
```

Description

This interface function is called for Vector extension floating point single-width reduction instructions (`vfredusum.vs`). The specification does not define precisely how these are implemented, so this enables a custom implementation to be provided by code emitted by an extended model. The base model performs all instruction validity checks, so these do not have to be done by the custom implementation. If no custom implementation is provided, the default behavior matches `vfredosum.vs`.

When the instruction is executed, argument `vdAcc` contains the scalar input value initially and should be written with the reduction result on completion. `vs2` is a pointer to the vector input argument and `vm` is a pointer to the mask argument, or `NULL` if the operation is unmasked. `SEW` indicates the operation width.

Example

This example shows how this function could be used to reimplement the default single-width floating point reduction operation using floating point emulation functions provided by the *VMI Run Time Function* API. Note that this is an example only, as this behavior is the default if the interface function is omitted.

```
//
// Return current rounding mode
//
static vmiFPRC getRC(riscvP riscv) {

    static const vmiFPRC map[8] = {
        [0] = vmi_FPR_NEAREST,
        [1] = vmi_FPR_ZERO,
        [2] = vmi_FPR_NEG_INF,
        [3] = vmi_FPR_POS_INF,
        [4] = vmi_FPR_AWAY,
    };

    return map[RD_CSR_FIELDC(riscv, fcsr, frm)];
}

//
// Return floating point type for the given bits
//
static vmiFType getFType(Uns32 fBits) {

    vmiFType result = 0;

    if(fBits==16) {
```

```
        result = vmi_FT_16_IEEE_754;
    } else if(fBits==32) {
        result = vmi_FT_32_IEEE_754;
    } else if(fBits==64) {
        result = vmi_FT_64_IEEE_754;
    }

    return result;
}

//
// If the operation is masked, indicate whether the indexed element is enabled
//
static Bool elementEnabled(Uns8 *vm, Uns32 i) {

    Bool enabled = True;

    if(vm) {

        Uns32 byte = i/8;
        Uns32 mask = 1<<(i&7);

        enabled = vm[byte] & mask;
    }

    return enabled;
}

//
// Get floating point operation descriptor for add of the passed width
//
static vmiFPBinopDescCP getFAddDesc(riscvP riscv, Uns32 fBits, Uns32 num) {

    vmiProcessorP processor = (vmiProcessorP)riscv;
    vmiFType      fType     = getFType(fBits);
    vmiFPRC       rc        = getRC(riscv);

    return vmirtGetFBinopRRRDesc(processor, fType, num, vmi_FADD, 0, rc, False);
}

//
// Execute floating point binary operation
//
static void doFBinop(
    riscvP           riscv,
    vmiFPBinopDescCP opDesc,
    void            *rd,
    void            *rs1,
    void            *rs2
) {
    Uns8 flags = 0;

    vmirtFBinopSimdRRR((vmiProcessorP)riscv, opDesc, rd, rs1, rs2, &flags);

    riscv->fpFlagsMT |= flags;
}

//
// Custom VFREDUSUM implementation
//
static Uns64 doVFREDUSUM(
    vmiosObjectP object,
    Uns64        vdAcc,
    Uns8        *vs2,
    Uns8        *vm,
    Uns32        SEW,
    Uns32        SEWMul
) {
    riscvP riscv  = object->riscv;
    Uns32  vl     = RD_CSRC(riscv, vl);
    Uns32  fBits  = SEW*SEWMul;
```

```
    Uns32  fBytes = fBits/8;
    Uns32  i;

    // get operation description for single floating point add
    vmiFPBinopDescCP opDescX1 = getFAddDesc(riscv, fBits, 1);

    // handle all enabled elements
    for(i=0; i<vl; i++) {

        if(elementEnabled(vm, i)) {

            Uns8 *vs2Elem = vs2+(i*fBytes);

            doFBinop(riscv, opDescX1, &vdAcc, &vdAcc, vs2Elem);
        }
    }

    return vdAcc;
}

//
// Common routine to emit callback for VFREDUSUM and VFWREDUSUM
//
static void emitVFREDUSUMCommon(
    vmiosObjectP object,
    vmiCallFn    opCB,
    vmiReg       vdAcc,
    void        *vs2,
    void        *vm,
    Uns32        SEW,
    Uns32        SEWMul
) {
    // extend temporary accumulator to 64 bits
    vmimtMoveExtendRR(64, vdAcc, SEW*SEWMul, vdAcc, False);

    // call custom implementation
    vmimtArgNatAddress(object);
    vmimtArgReg(64, vdAcc);
    vmimtArgNatAddress(vs2);
    vmimtArgNatAddress(vm);
    vmimtArgUns32(SEW);
    vmimtArgUns32(SEWMul);
    vmimtCallResultAttrs(opCB, 64, vdAcc, VMCA_NO_INVALIDATE);
}

static RISCV_VFREDSUM_MORPH_FN(customVFREDUSUM) {

    vmiCallFn opCB = (vmiCallFn)doVFREDUSUM;

    emitVFREDUSUMCommon(clientData, opCB, vdAcc, vs2, vm, SEW, 1);
}

static VMIOS_CONSTRUCTOR_FN(constructor) {

    . . . lines omitted . . .
    object->extCB.emitVFREDUSUM = customVFREDUSUM;
    . . . lines omitted . . .
}
```

15.26 Function `emitVFWREDUSUM`

```
#define RISCV_VFREDSUM_MORPH_FN(_NAME) void _NAME( \
    riscvP riscv,       \
    void  *clientData,  \
    vmiReg vdAcc,       \
    void  *vs2,         \
    void  *vm,          \
    Uns32  SEW          \
)
typedef RISCV_VFREDSUM_MORPH_FN((*riscvVFREDSUMMorphFn));

typedef struct riscvExtCBS {
    riscvVFREDSUMMorphFn      emitVFWREDUSUM;
} riscvExtCB;
```

Description

This interface function is called for Vector extension floating point widening reduction instructions (`vfwredusum.vs`). The specification does not define precisely how these are implemented, so this enables a custom implementation to be provided by code emitted by an extended model. The base model performs all instruction validity checks, so these do not have to be done by the custom implementation. If no custom implementation is provided, the default behavior matches `vfwredosum.vs`.

When the instruction is executed, argument `vdAcc` contains the scalar input value initially and should be written with the reduction result on completion. `vs2` is a pointer to the *unwidened* vector input argument and `vm` is a pointer to the mask argument, or `NULL` if the operation is unmasked. `SEW` indicates the operation width.

Example

This example shows how this function could be used to reimplement the default widening floating point reduction operation using floating point emulation functions provided by the *VMI Run Time Function* API. Note that this is an example only, as this behavior is the default if the interface function is omitted. Some of the implementation is shared with the `vfredusum.vs` example (section 15.25) so common functions are omitted here.

```
//
// Get floating point operation descriptor for widening conversion
//
static vmiFPConvertDescCP getFCvtDesc(
    riscvP riscv,
    Uns32  srcBits,
    Uns32  dstBits
) {
    vmiProcessorP processor = (vmiProcessorP)riscv;
    vmiFType      srcType   = getFType(srcBits);
    vmiFType      dstType   = getFType(dstBits);
    vmiFPRC       rc        = vmi_FPR_NEAREST;

    return vmirtGetFConvertRRDesc(processor, 1, dstType, srcType, 0, rc, False);
}

//
// Execute floating point widening conversion
//
static void doFConvert(
    riscvP             riscv,
    vmiFPConvertDescCP opDesc,
```

```
    void              *rd,
    void              *rs
) {
    Uns8 flags = 0;

    vmirtFConvertSimdRR((vmiProcessorP)riscv, opDesc, rd, rs, &flags);

    riscv->fpFlagsMT |= flags;
}

//
// Custom VFWREDUSUM implementation
//
static Uns64 doVFWREDUSUM(
    vmiosObjectP object,
    Uns64        vdAcc,
    Uns8        *vs2,
    Uns8        *vm,
    Uns32        SEW,
    Uns32        SEWMul
) {
    riscvP riscv    = object->riscv;
    Uns32  vl       = RD_CSRC(riscv, vl);
    Uns32  srcBits  = SEW;
    Uns32  dstBits  = SEW*SEWMul;
    Uns32  srcBytes = srcBits/8;
    Uns32  dstBytes = dstBits/8;
    Uns8   vs2Tmp[vl*dstBytes];
    Uns32  i;

    // get operation description for single widening conversion
    vmiFPConvertDescCP opDescX1 = getFCvtDesc(riscv, srcBits, dstBits);

    // widen all enabled elements
    for(i=0; i<vl; i++) {

        if(elementEnabled(vm, i)) {

            Uns8 *src = vs2+(i*srcBytes);
            Uns8 *dst = &vs2Tmp[i*dstBytes];

            doFConvert(riscv, opDescX1, dst, src);
        }
    }

    // use common algorithm with widened source
    return doVFREDUSUM(object, vdAcc, vs2Tmp, vm, SEW, SEWMul);
}

static RISCV_VFREDSUM_MORPH_FN(customVFWREDUSUM) {

    vmiCallFn opCB = (vmiCallFn)doVWFREDUSUM;

    emitVFREDUSUMCommon(clientData, opCB, vdAcc, vs2, vm, SEW, 2);
}

static VMIOS_CONSTRUCTOR_FN(constructor) {

    . . . lines omitted . . .
    object->extCB.emitVFWREDUSUM = customVFWREDUSUM;
    . . . lines omitted . . .
}
```

15.27 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 . . .
}
```

15.28 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 14.9). 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 . . .
}
```

15.29 Function tStore

```
#define RISCV_TSTORE_FN(_NAME) void _NAME( \
    riscvP      riscv,          \
    const void *buffer,         \
    Addr        VA,             \
    Uns32       bytes,          \
    void       *clientData      \
)
typedef RISCV_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 14.9). 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 RISCV_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 . . .
}
```

15.30 Function `distinctPhysMem`

```
#define RISCV_DISTINCT_PHYS_MEM_FN(_NAME) Bool _NAME( \
    riscvP riscv,               \
    void  *clientData           \
)
typedef RISCV_DISTINCT_PHYS_MEM_FN((*riscvDistinctPhysMemFn));

typedef struct riscvExtCBS {
    riscvDistinctPhysMemFn    distinctPhysMem;
} riscvExtCB;
```

Description

This interface function is required when an extension implements physical memory map modifications (for example, by installing custom local memories into the physical memory hierarchy – see section 15.31 for more details). The function should return a Boolean indicating whether, as a result of these custom physical memory changes, code and data domain views of memory are different. For example, the function should return `True` if a model implements Code and Data local memories where the Data local memory is not visible in the instruction address space.

Example

The Andes extended model implements Instruction and Data local memories in the physical address space, and the Data local memory is not visible in the instruction address space. The function below is used to indicate that separate code and data domain views are required if either local memory is present.

```
//
// Are separate code and data physical memory view required?
//
RISCV_DISTINCT_PHYS_MEM_FN(andesDistinctPhysicalMem) {

    vmiosObjectP object = clientData;

    // determine whether either local memory is present
    Uns32 ILMSize = createLocalMemory(object, object->csr.micm_cfg, True);
    Uns32 DLMSize = createLocalMemory(object, object->csr.mdcm_cfg, False);

    // separate view is required if either LM is present
    return ILMSize || DLMSize;
}
```

15.31 Function `installPhysMem`

```
#define RISCV_PHYS_MEM_FN(_NAME) void _NAME( \
    riscvP riscv,               \
    void  *clientData           \
)
typedef RISCV_PHYS_MEM_FN((*riscvPhysMemFn));

typedef struct riscvExtCBS {
    riscvPhysMemFn           installPhysMem;
} riscvExtCB;
```

Description
This interface function allows an extension to modify the physical memory map of a processor. It is called once the default physical memory constructor has been called. Typically, the function will create new memory domain (`memDomainP`) objects and replace the default physical memory domains with those (see lines in bold in the following example 1). This is typically required if, for example, the extended model has components such as local memories that reside at fixed locations in the physical memory map.

If a processor implements custom physical memory attributes (PMA), this function can also be used to specify any initial PMA constraints (the default is that no constraints are imposed initially).

Example 1
The Andes extended model implements Instruction and Data local memories in the physical address space. The details of the operation of these are quite complex; code below shows how the standard physical memory domains are replaced with Andes-specific domains by the `installPhysMem` callback. Refer to the source of the Andes model for more information on the implementation of local memories.

```
//
// Create local memory physical alias domain for the given mode and access type
//
static void newLMDomain(vmiosObjectP object, riscvMode mode, Bool isCode) {

    riscvP     riscv = object->riscv;
    memDomainP base  = riscv->physDomains[mode][isCode];
    Uns32      bits  = vmirtGetDomainAddressBits(base);

    // create local memory domain
    memDomainP result = createLMDomain(mode, bits, isCode);

    // initially make it an alias of the physical domain
    vmirtAliasMemory(base, result, 0, getAddressMask(bits), 0, 0);

    // replace physical domain
    riscv->physDomains[mode][isCode] = result;
}

//
// Install local memory domains in the domain hierarchy
//
static void installLocalMemoryDomains(vmiosObjectP object) {

    riscvP    riscv = object->riscv;
    riscvMode mode;
```

```
    // create physical domain aliases for all non-User modes
    for(mode=RISCV_MODE_S; mode<=RISCV_MODE_M; mode++) {

        memDomainP dataDomain = riscv->physDomains[mode][0];
        memDomainP codeDomain = riscv->physDomains[mode][1];

        // save current physical domain to allow aliasing to it later
        object->physDomains[mode][0] = dataDomain;
        object->physDomains[mode][1] = codeDomain;

        if(dataDomain) {
            newLMDomain(object, mode, False);
            newLMDomain(object, mode, True);
        }
    }

    // use Supervisor aliases for User mode
    riscv->physDomains[RISCV_MODE_U][0] = riscv->physDomains[RISCV_MODE_S][0];
    riscv->physDomains[RISCV_MODE_U][1] = riscv->physDomains[RISCV_MODE_S][1];
}

//
// Install ILM/DLM domains if required
//
RISCV_PHYS_MEM_FN(andesInstallPhysicalMem) {

    vmiosObjectP object = clientData;

    // create local memories if required
    Uns32 ILMSize = createLocalMemory(object, object->csr.micm_cfg, True);
    Uns32 DLMSize = createLocalMemory(object, object->csr.mdcm_cfg, False);

    // if either local memory is present, insert ILM domains into the memory
    // domain hierarchy
    if(ILMSize || DLMSize) {
        installLocalMemoryDomains(object);
    }

    // enable ILM if required
    if(RD_XCSR_FIELD(object, milmb, EN)) {
        updateLocalMemory(object, True, True);
    }

    // enable DLM if required
    if(RD_XCSR_FIELD(object, mdlmb, EN)) {
        updateLocalMemory(object, False, True);
    }
}
```

Example 2

The OpenHardware CV32E40X extended model requires that all PMA access is initially denied (the model is structured so that PMA mappings are created on demand using configured PMA range definitions). This is implemented as follows:

```
//
// Set privileges in the given domain set
//
static void setDomainPriv(
    riscvP      riscv,
    memDomainPP domains,
    Uns64       low,
    Uns64       high,
    memPriv     priv,
    const char *name,
    Bool        verbose
) {
    memDomainP dataDomain = domains[0];
```

```
    memDomainP codeDomain = domains[1];

    if(dataDomain==codeDomain) {

        // set permissions in unified domain
        vmirtProtectMemory(dataDomain, low, high, priv, MEM_PRIV_SET);

    } else {

        // get privileges for data and code domains
        memPriv privRW = priv&MEM_PRIV_RW ? priv&~MEM_PRIV_X  : MEM_PRIV_NONE;
        memPriv privX  = priv&MEM_PRIV_X  ? priv&~MEM_PRIV_RW : MEM_PRIV_NONE;

        // set permissions in data domain
        vmirtProtectMemory(dataDomain, low, high, privRW, MEM_PRIV_SET);

        // set permissions in code domain
        vmirtProtectMemory(codeDomain, low, high, privX, MEM_PRIV_SET);
    }
}

//
// Set privileges in PMA domain
//
static void setPMAPriv(
    riscvP  riscv,
    Uns64   low,
    Uns64   high,
    memPriv priv,
    Bool    verbose
) {
    setDomainPriv(riscv, &riscv->pmaDomains[0], low, high, priv, "PMA", verbose);
}

//
// Handle PMA initialization
//
static RISCV_PHYS_MEM_FN(openhwInstallPhysicalMem) {

    // remove all PMA permissions
    setPMAPriv(riscv, 0, -1, MEM_PRIV_NONE, False);
}
```

15.32 Function PMPPriv

```
#define RISCV_PMP_PRIV_FN(_NAME) memPriv _NAME( \
    riscvP  riscv,              \
    memPriv priv,               \
    Uns32   regionIndex,        \
    void   *clientData          \
)
typedef RISCV_PMP_PRIV_FN((*riscvPMPPrivFn));

typedef struct riscvExtCBS {
    riscvPMPPrivFn            PMPPriv;
} riscvExtCB;
```

Description

This interface function allows an extension to modify the memory privileges of a PMP region in a custom manner. When a new PMP mapping is being established, this function is called with the region index and default access permissions; it returns the final access permissions after applying custom behavior.

Example

This example shows how alignment might be forced using extension CSRs with a one-to-one mapping to the base model PMP CSRs:

```
static RISCV_PMP_PRIV_FN(updatePMAPriv) {

    vmiosObjectP object = clientData;
    Uns32        CCA    = object->pmacfg.u8[regionIndex] & 7;

    // if CCA!=0, accesses in this region must be aligned
    if(CCA) {
        priv |= MEM_PRIV_ALIGN;
    }

    return priv;
}

static VMIOS_CONSTRUCTOR_FN(constructor) {

    . . . lines omitted . . .
    object->extCB.PMAPriv = updatePMAPriv;
    . . . lines omitted . . .
}
```

15.33 Function PMAEnable

```
#define RISCV_PMA_ENABLE_FN(_NAME) Bool _NAME( \
    riscvP riscv,               \
    void  *clientData           \
)
typedef RISCV_PMA_ENABLE_FN((*riscvPMAEnableFn));

typedef struct riscvExtCBS {
    riscvPMAEnableFn          PMAEnable;
} riscvExtCB;
```

Description
For an extension implementing physical memory attributes (PMA) when maximum PMA page size is restricted by configuration parameter `PMP_max_page`, 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:

```
RISCV_PMA_ENABLE_FN(andesPMAEnable) {

    vmiosObjectP object = clientData;

    return RD_XCSR_FIELD(object, mmsc_cfg, DPMA);
}

static VMIOS_CONSTRUCTOR_FN(constructor) {

    . . . lines omitted . . .
    object->extCB.PMAEnable      = andesPMAEnable;
    . . . lines omitted . . .
}
```

See section 12 for further details about this function.

15.34 Function PMACheck

```
#define RISCV_PMA_CHECK_FN(_NAME) void _NAME( \
    riscvP  riscv,              \
    memPriv requiredPriv,       \
    Uns64   lowPA,              \
    Uns64   highPA,             \
    void   *clientData          \
)
typedef RISCV_PMA_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 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

See section 12 for a detailed example using this function.

15.35 Function validPTE

```
#define RISCV_VALID_PTE_FN(_NAME) Bool _NAME( \
    riscvP      riscv,          \
    riscvTLBId  id,             \
    riscvVAMode vaMode,         \
    Uns64       PTE,            \
    void       *clientData      \
)
typedef RISCV_VALID_PTE_FN((*riscvValidPTEFn));

typedef struct riscvExtCBS {
    riscvValidPTEFn           validPTE;
} riscvExtCB;
```

Description

When virtual memory is active, this function allows an extension to perform custom checks for PTE validity. It should return `True` if the given page table entry is valid and `False` otherwise. This validity check is performed in addition to the standard checks performed by the base model and described in the *Privileged Architecture Specification*.

15.36 Function VMTrap

```
#define RISCV_VM_TRAP_FN(_NAME) riscvException _NAME( \
    riscvP     riscv,           \
    riscvTLBId id,              \
    memPriv    requiredPriv,    \
    Uns64      VA,              \
    void      *clientData       \
)
typedef RISCV_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 {

    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 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;

RISCV_VM_TRAP_FN(custVMTrap) {

    vmiosObjectP   object = clientData;
    Bool           S2     = (id==RISCV_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 VMIOS_CONSTRUCTOR_FN(extConstructor) {

    . . . lines omitted . . .
    object->extCB.VMTrap            = custVMTrap;
    . . . lines omitted . . .
}
```

15.37 Function `setDomainNotifier`

```
#define RISCV_SET_DOMAIN_NOTIFIER_FN(_NAME) void _NAME( \
    riscvP      riscv,          \
    memDomainPP domainP,        \
    void       *clientData      \
)
typedef RISCV_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:

```
RISCV_SET_DOMAIN_NOTIFIER_FN(custSetDomain) {

    riscvMode    mode   = getCurrentMode5(riscv);
    vmiosObjectP object = clientData;

    if(mode!=RISCV_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[RISCV_MODE_S][False];
    } else if(!RD_CSR_FIELD_S(riscv, hgatp, MODE)) {
        *domainP = riscv->pmpDomains[RISCV_MODE_S][False];
    } else {
        *domainP = riscv->guestPTWDomain;
    }
}

static VMIOS_CONSTRUCTOR_FN(extConstructor) {

    . . . lines omitted . . .
    object->extCB.setDomainNotifier = custSetDomain;
    . . . lines omitted . . .
}
```

15.38 Function freeEntryNotifier

```
#define RISCV_FREE_ENTRY_NOTIFIER_FN(_NAME) void _NAME( \
    riscvP     riscv,           \
    riscvTLBId id,              \
    Uns16      entryId,         \
    void      *clientData       \
)
typedef RISCV_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 14.59). 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 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:

```
RISCV_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 . . .
}
```

15.39 Function clicCustomRd

```
#define RISCV_CLIC_CUSTOM_READ_FN(_NAME) Bool _NAME( \
    riscvP riscv,               \
    Uns32  offset,              \
    Uns8  *val,                 \
    void  *clientData           \
)
typedef RISCV_CLIC_CUSTOM_READ_FN((*riscvClicCRdFn));
```

Description
This notifier is called when the custom region of the CLIC MMIO has been read. Reads are handled one byte at a time, so a 4 byte read will result in 4 separate calls to the notifier.

offset is the offset from the *mclicbase* location.

If the extension implements a register at the offset location it should copy the value to be read for that byte in **val* and return *True* indicating the register is present.

If there is no register implemented by the extension at the offset then it should return *False*.

Example

```
RISCV_CLIC_CUSTOM_READ_FN(customClicRd) {
    vmiosObjectP object = clientData;

    if ((offset & ~0x3) == 0x800) { // Custom register: 4 bytes at 0x800
        Uns32 byte = offset & 0x3;

        *val = (object->custom >> (byte*8)) & 0xff;
        return True;
    }

    return False
)
```

Notes
In a multi hart processor *riscv* is the root processor of the cluster

Once any callback has returned *True* no additional callbacks will be done.

15.40 Function clicCustomWr

```
#define RISCV_CLIC_CUSTOM_WRITE_FN(_NAME) Bool _NAME( \
    riscvP riscv,               \
    Uns32  offset,              \
    Uns8   val,                 \
    void  *clientData           \
)
typedef RISCV_CLIC_CUSTOM_WRITE_FN((*riscvClicCWrFn));
```

Description
This notifier is called when the custom region of the CLIC MMIO has been written. Writes are handled one byte at a time, so a 4 byte write will result in 4 separate calls to the notifier.

offset is the offset from the *mclicbase* location.

If the extension implements a register at the offset location it should copy *val* to the appropriate byte of the register and return *True* indicating the register is present.

If there is no register implemented by the extension at the offset then it should return *False*.

Example

```
RISCV_CLIC_CUSTOM_WRITE_FN(customClicWr) {
    vmiosObjectP object = clientData;

    if ((offset & ~0x3) == 0x800) { // Custom register: 4 bytes at 0x800
        Uns32 byte     = offset & 0x3;
        Uns32 newByte  = ((Uns32)val) << (byte*8);
        Uns32 byteMask = 0xff << (byte*8);

        object->custom &= ~byteMask;
        object->custom |= newByte;

        return True;
    }

    return False;
)
```

Notes
In a multi hart processor *riscv* is the root processor of the cluster

Once any callback has returned *True* no additional callbacks will be done.

15.41 Function clicUpdated

```
# define RISCV_CLIC_UPDATED_FN(_NAME) Bool _NAME( \
    riscvP riscv,               \
    void  *clientData           \
)
typedef RISCV_CLIC_UPDATED_FN((*riscvClicUpdatedFn));
```

Description
This notifier is called when the CLIC configuration has been updated by software through a write to a CLIC MMIO configuration register.

It gives an extension an opportunity to update any CLIC related state after a configuration change. If the CLIC state is updated in the callback then the function should return True, which will force a re-evaluation of the CLIC status by the base model.

Example

```
static RISCV_CLIC_UPDATED_FN(cirrusCLICUpdateNotify) {
    vmiosObjectP object = clientData;

    return updateExtensionClicState(riscv, object);
}
```

Notes
In a multi hart processor *riscv* is the root processor of the cluster

15.42 Function `restrictionsCB`

```
#define RISCV_RESTRICTIONS_FN(_NAME) void _NAME( \
    riscvP      riscv,          \
    vmiDocNodeP node,           \
    void       *clientData      \
)
typedef RISCV_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.

16 Appendix: `riscvInstrInfo` Structure

Some extension object interface functions described in section 15 are passed a pointer to a structure of type `riscvInstrInfo`. This structure gives comprehensive information obtained by decoding the instruction that is currently being translated, which can be useful when generating custom behavior. For example, the structure can be used to implement custom behavior to cause Illegal Instructions to be generated for some instruction types or argument combinations. This section describes this structure and indicates when particular fields within it may be useful.

The structure is defined in file `riscvDecodeTypes.h` as follows:

```
typedef struct riscvInstrInfoS {

    // COMMON FIELDS FOR ALL INSTRUCTION TYPES
    riscvIType        type;             // instruction type
    riscvAddr         thisPC;           // instruction address
    Uns64             instruction;      // instruction word
    Uns8              bytes;            // instruction size in bytes
    riscvRegDesc      r[RV_MAX_AREGS];  // argument registers
    riscvAddr         tgt;              // constant target address
    Uns64             c;                // constant value
    Int32             memBits;          // load/store size
    Bool              unsExt;           // whether to extend unsigned

    // CSR UPDATE INSTRUCTIONS
    riscvCSRUDesc     csrUpdate;        // CSR update semantics
    Uns32             csr;              // CSR index

    // FLOATING POINT INSTRUCTIONS
    riscvRMDesc       rm;               // rounding mode
    Bool              useF;             // whether F registers used

    // VECTOR EXTENSION INSTRUCTIONS
    riscvRegDesc      mask;             // vector mask register
    riscvWholeDesc    isWhole;          // is this a whole-register instruction?
    riscvVType        vtype;            // vector type information
    Uns32             eew;              // explicit EEW encoding
    Uns8              eewDiv;           // explicit EEW divisor
    Uns8              eewIndex;         // number of EEW index operand
    Uns8              nf;               // nf value
    Bool              isFF;             // is this a first-fault instruction?

    // CODE SIZE REDUCTION INSTRUCTIONS
    Uns32             rlist;            // register list (Zcea push/pop)
    Uns32             alist;            // argument register list (Zcea push)
    riscvRetValDesc   retval;           // return value (Zcea pop)
    riscvCompressSet  Zc;               // compressed extension
    Bool              doRet;            // do return (Zcea pop)
    Bool              embedded;         // whether embedded modifier required

    // INTEGER DIVISION INSTRUCTIONS
    Bool              Zmmul;            // whether affected by Zmmul

    // SIMD EXTENSION INSTRUCTIONS
    riscvCrossOpDesc  crossOp;          // cross operation
    riscvHalfDesc     half;             // top/bottom half operation
    riscvPackDesc     pack;             // byte packing
    Uns8              elemSize;         // element size
    Bool              doDouble;         // whether additional doubling step
    Bool              round;            // whether additional rounding step

    // INSTRUCTION SUBSET VALIDATION
    riscvArchitecture arch;             // architecture requirements
```

```
    // INSTRUCTION DISASSEMBLY
    const char       *opcode;           // opcode name
    const char       *format;           // disassembly format string
    riscvVIType       VIType;           // vector instruction type
    riscvAQRLDesc     aqrl;             // acquire/release specifier
    riscvFenceDesc    pred;             // predecessor fence
    riscvFenceDesc    succ;             // successor fence
    riscvXPERMDesc    xperm;            // XPERM descriptor
    riscvCSufDesc     cSuffix;          // compressed suffix
    Uns8              explicitType;     // whether types are explicit in opcode
    Bool              explicitW;        // whether 'w' explicit in opcode
    Bool              explicitRM;       // whether rounding explicit in opcode
    Bool              explicitDot;      // whether explicit type has dot
    Bool              unsPfx;           // show unsigned as z/s prefix not u suffix
    Bool              csrInOp;          // whether to emit CSR as part of opcode
    Bool              cPrefix;          // whether compressed prefix enabled
    Uns8              shN;              // shN prefix

} riscvInstrInfo;
```

Fields in the structure are grouped according to their applicability. These groups are described below.

16.1 Common Fields

These fields will have useful values for a wide range of instruction types:

```
    riscvIType        type;             // instruction type
    riscvAddr         thisPC;           // instruction address
    Uns64             instruction;      // instruction word
    Uns8              bytes;            // instruction size in bytes
    riscvRegDesc      r[RV_MAX_AREGS];  // argument registers
    riscvAddr         tgt;              // constant target address
    Uns64             c;                // constant value
    Int32             memBits;          // load/store size
    Bool              unsExt;           // whether to extend unsigned
```

16.1.1 `riscvIType type`

This field is an enumeration member describing the abstract type of the decoded instruction; there are a very large number of these. The value is typically useful in contexts where particular instructions have custom behavior or are unimplemented and should cause an Illegal Instruction trap. For example, in an atomic instruction context, the particular type of atomic instruction that has been encountered can be found by comparing against these values:

```
//
// This enumerates generic instructions
//
typedef enum riscvITypeE {

    . . .values omitted . . .

    // A-extension R-type instructions
    RV_IT_AMOADD_R,
    RV_IT_AMOAND_R,
    RV_IT_AMOMAX_R,
    RV_IT_AMOMAXU_R,
    RV_IT_AMOMIN_R,
    RV_IT_AMOMINU_R,
    RV_IT_AMOOR_R,
    RV_IT_AMOSWAP_R,
    RV_IT_AMOXOR_R,
```

```
    RV_IT_LR_R,
    RV_IT_SC_R

    . . .values omitted . . .
};
```

16.1.2 riscvAddr thisPC

This field gives the simulated address of the current instruction.

16.1.3 Uns64 instruction

This field gives the byte pattern of the current instruction.

16.1.4 Uns8 bytes

This field gives the size in bytes of the current instruction.

16.1.5 riscvRegDesc r[RV_MAX_AREGS]

This array field gives the registers used by the current instruction in left-to-right order in an encoded form, using the `riscvRegDesc` type, defined in file `riscvRegisterTypes.h`:

```
typedef enum riscvRegDescE {

    RV_RD_NA         = 0x0,
                                        // REGISTER INDEX
    RV_RD_INDEX_MASK = 0x01f,           // mask to select register index

                                        // REGISTER SIZE (in bits)
    RV_RD_4          = RV_RD_INDEX_MASK+1,  // 4-bit
    RV_RD_8          = RV_RD_4 *2,      // 8-bit
    RV_RD_16         = RV_RD_8 *2,      // 16-bit
    RV_RD_32         = RV_RD_16*2,      // 32-bit
    RV_RD_64         = RV_RD_32*2,      // 64-bit
    RV_RD_128        = RV_RD_64*2,      // 128-bit
    RV_RD_BITS_MASK  = (RV_RD_4|RV_RD_8|RV_RD_16|RV_RD_32|RV_RD_64|RV_RD_128),

                                        // REGISTER TYPE
    RV_RD_X          = RV_RD_128*2,     // integer (X) register
    RV_RD_F          = RV_RD_X*2,       // floating point register
    RV_RD_V          = RV_RD_F*2,       // vector register
    RV_RD_TYPE_MASK  = (RV_RD_X|RV_RD_F|RV_RD_V|RV_RD_BITS_MASK),

                                        // REGISTER TYPE MODIFIERS
    RV_RD_ZFINX      = RV_RD_V*2,       // Zfinx-modified register
    RV_RD_BF16       = RV_RD_ZFINX*2,   // explicit BFLOAT16 register

                                        // DISASSEMBLY CONTROL
    RV_RD_Q          = RV_RD_BF16*2,    // quiet (don't show type)
    RV_RD_WL         = RV_RD_Q*2,       // explicit w/l type name
    RV_RD_FX         = RV_RD_WL*2,      // explicit x type name
    RV_RD_U          = RV_RD_FX*2,      // explicit u type name

} riscvRegDesc;
```

File `riscvRegisterTypes.h` also implements a set of inline helper functions that can be used to extract information from a `riscvRegDesc` value. The most useful of these are summarized below.

```
// Is the register an X register?
inline static Bool isXReg(riscvRegDesc r);
```

```
// Is the register an F register?
inline static Bool isFReg(riscvRegDesc r);

// Is the register a V register?
inline static Bool isVReg(riscvRegDesc r);

// Is the register a BF16 register?
inline static Bool isBF16Reg(riscvRegDesc r);

// Is the register an X register holding a floating point value (Zfinx)?
inline static Bool isZfinxReg(riscvRegDesc r);

// Return register index
inline static Uns32 getRIndex(riscvRegDesc r);

// Return register size in bits
inline static Uns32 getRBits(riscvRegDesc r);
```

16.1.6 riscvAddr tgt

For direct branch instructions, this field gives the target address.

16.1.7 Uns64 c

For instructions with encoded constants, this gives the constant value, sign-extended to 64 bits if required.

16.1.8 Int32 memBits

For instructions that access memory, this gives the width of the memory access in bits. The special value `-1` is used for polymorphic vector instructions that are accessing memory using the current `SEW`.

16.1.9 Int32 Bool unsExt

For instructions that extend narrow values to wider ones (either memory reads or register-to-register operations), this indicates whether the extension is unsigned.

16.2 CSR Instruction Fields

These fields will have useful values only for CSR access instructions (types `RV_IT_CSRR_I` or `RV_IT_CSRRI_I`):

```
    riscvCSRUDesc     csrUpdate;        // CSR update semantics
    Uns32             csr;              // CSR index
```

16.2.1 riscvCSRUDesc csrUpdate

This field indicates the update semantics of the CSR access instruction. Type `riscvCSRUDesc` is defined in file `riscvTypes.h` as follows:

```
typedef enum riscvCSRUDescE {
    RV_CSR_NA,      // no update semantics
    RV_CSR_RW,      // read/write
    RV_CSR_RS,      // read/set
    RV_CSR_RC,      // read/clear
} riscvCSRUDesc;
```

16.2.2 Uns32 csr

This field gives the index number of the CSR being accessed:

16.3 Floating Point Instruction Fields

These fields provide information about floating point instructions:

```
    riscvRMDesc       rm;               // rounding mode
    Bool              useF;             // whether F registers used
```

16.3.1 riscvRMDesc rm

This field indicates the rounding mode encoded in the instruction. Type `riscvRMDesc` is defined in file `riscvTypes.h` as follows:

```
typedef enum riscvRMDescE {
    RV_RM_NONE,     // no rounding mode
    RV_RM_CURRENT,  // round using current rounding mode
    RV_RM_RNE,      // round to nearest, ties to even
    RV_RM_RTZ,      // round towards zero
    RV_RM_RDN,      // round towards -infinity
    RV_RM_RUP,      // round towards +infinity
    RV_RM_RMM,      // round to nearest, ties away
    RV_RM_ROD,      // round to odd (jamming)
    RV_RM_BAD5,     // illegal rounding mode 5
    RV_RM_BAD6,     // illegal rounding mode 6
} riscvRMDesc;
```

16.3.2 Bool useF

This field indicates whether the current instruction accesses any `FPR`:

16.4 Vector Extension Instruction Fields

These fields provide information about vector extension instructions:

```
    riscvRegDesc      mask;             // vector mask register
    riscvWholeDesc    isWhole;          // is this a whole-register instruction?
    riscvVType        vtype;            // vector type information
    Uns32             eew;              // explicit EEW encoding
    Uns8              eewDiv;           // explicit EEW divisor
    Uns8              eewIndex;         // number of EEW index operand
    Uns8              nf;               // nf value
    Bool              isFF;             // is this a first-fault instruction?
```

16.4.1 riscvRegDesc mask

This field specifies the mask register to use with the current vector instruction; if zero, the instruction is unmasked. For vector extension versions up to 1.0, only register v0 may be used as a mask.

16.4.2 riscvWholeDesc isWhole

For whole-register vector instructions, this specifies the class of instruction, as follows:

```
typedef enum riscvWholeDescE {
    RV_WD_NA,       // not a whole register instruction
    RV_WD_LD_ST,    // whole register load/store
    RV_WD_MV,       // whole register move
} riscvWholeDesc;
```

16.4.3 riscvVType vtype

This field is used to categorize a vector instruction shape. It is mainly useful for disassembly:

```
typedef enum riscvVITypeE {
    RV_VIT_NA,      // not a vector instruction
    RV_VIT_V,       // instruction type .v
    RV_VIT_W,       // instruction type .w
    RV_VIT_VV,      // instruction type .vv
    RV_VIT_VI,      // instruction type .vi
    RV_VIT_VX,      // instruction type .vx
    RV_VIT_WV,      // instruction type .wv
    RV_VIT_WI,      // instruction type .wi
    RV_VIT_WX,      // instruction type .wx
    RV_VIT_VF,      // instruction type .vf
    RV_VIT_WF,      // instruction type .wf
    RV_VIT_VS,      // instruction type .vs
    RV_VIT_M,       // instruction type .m
    RV_VIT_MM,      // instruction type .mm
    RV_VIT_VM,      // instruction type .vm
    RV_VIT_VVM,     // instruction type .vvm
    RV_VIT_VXM,     // instruction type .vxm
    RV_VIT_VIM,     // instruction type .vim
    RV_VIT_VFM,     // instruction type .vfm
    RV_VIT_VN,      // instruction type .v/.w (version-dependent)
    RV_VIT_VVN,     // instruction type .vv/.wv (version-dependent)
    RV_VIT_VIN,     // instruction type .vi/.wi (version-dependent)
    RV_VIT_VXN,     // instruction type .vx/.wx (version-dependent)
    RV_VIT_V_V,     // instruction type .v.v
    RV_VIT_LAST     // KEEP LAST: for sizing
} riscvVIType;
```

16.4.4 Uns32 eew

This field gives any `EEW` value explicitly encoded in the instruction. If zero, there is no explicit `EEW` encoding.

16.4.5 Uns8 eewDiv

This field gives any `EEW` divisor explicitly encoded in the instruction. If zero, there is no explicit `EEW` divisor.

16.4.6 Uns8 eewIndex

For instructions that have `EEW` specified in a register operand, this gives the index number of that operand.

16.4.7 Uns8 nf

This field gives any `nf` field number explicitly encoded in the instruction. If zero, there is no explicit `nf` field number.

16.4.8 Bool isFF

This field indicates whether the current instruction has *first fault* behavior.

16.5 Code Size Reduction Extension Instruction Fields

These fields provide information about code size reduction extension instructions:

```
    Uns32             rlist;          // register list (Zcea push/pop)
    Uns32             alist;          // argument register list (Zcea push)
    riscvRetValDesc   retval;         // return value (Zcea pop)
    riscvCompressSet  Zc;             // compressed extension
    Bool              doRet;          // do return (Zcea pop)
    Bool              embedded;       // whether embedded modifier required
```

16.5.1 Uns32 rlist

This field is a bitmask of registers read or written by Zcea subset push/pop instructions. Register xN is read or written if bit N is set in the mask.

16.5.2 Uns32 alist

This field is a bitmask of argument registers used by Zcea subset push instructions. Register xN is an argument if bit N is set in the mask.

16.5.3 riscvRetValDesc retval

This field indicates the return value for the Zcea subset pop instruction:

```
typedef enum riscvRetValDescE {
    RV_RV_NA,       // {}
    RV_RV_0,        // {0}
    RV_RV_P1,       // {1}
    RV_RV_M1,       // {-1}
    RV_RV_Z,        // explicit Z in opcode
    RV_RV_LAST      // KEEP LAST: for sizing
} riscvRetValDesc;
```

16.5.4 riscvCompressSet Zc

This field indicates any extension subset requirements for the instruction. Depending on the extension version, either legacy or new values are indicated. The value is a bitfield; if more than one bit is set, then the instruction is configured if any of the subsets is present:

```
typedef enum riscvCompressSetE {
    // legacy values
    RVCS_Zcea      = (1<<0),          // Zcea subset
    RVCS_Zceb      = (1<<1),          // Zceb subset
    RVCS_Zcee      = (1<<2),          // Zcee subset
    // new values
    RVCS_Zca       = (1<<3),          // Zca subset
    RVCS_Zcb       = (1<<4),          // Zcb subset
    RVCS_Zcd       = (1<<5),          // Zcd subset (implicit)
    RVCS_Zcf       = (1<<6),          // Zcf subset
    RVCS_Zcmb      = (1<<7),          // Zcmb subset
    RVCS_Zcmp      = (1<<8),          // Zcmp subset
    RVCS_Zcmpe     = (1<<9),          // Zcmpe subset
    RVCS_Zcmt      = (1<<10),         // Zcmt subset
} riscvCompressSet;
```

16.5.5 Bool doRet

For the Zcea subset pop instruction, this indicates whether the instruction does a return.

16.5.6 Bool embedded

This indicates whether the embedded extension (E) is active. If it is, the registers stored by the Zcea subset push instruction are modified.

16.6 Zmmul Instruction Fields

One field is relevant for the Zmmul extension, which describes a subset of the standard M extension:

```
    Bool              Zmmul;            // whether affected by Zmmul
```

16.6.1 Bool Zmmul

This field indicates whether the current instruction is absent if extension Zmmul is active (currently, this is integer divide and modulus instructions only).

16.7 Packed SIMD Extension Instruction Fields

These fields provide information about packed SIMD extension instructions:

```
    riscvCrossOpDesc  crossOp;          // cross operation
    riscvHalfDesc     half;             // top/bottom half operation
    riscvPackDesc     pack;             // byte packing
    Uns8              elemSize;         // element size
    Bool              doDouble;         // whether additional doubling step
    Bool              round;            // whether additional rounding step
```

This extension has not been ratified and the fields are not discussed further here.

16.8 Other Fields

Remaining fields in the structure are concerned with standard extension subset validation and disassembly. They are unlikely to be useful in intended models and not discussed further here.

17 Appendix: Custom CSR Description

This appendix provides detailed information about the `riscvCSRAttrs` type, used to define custom CSRs in an extended processor model. It should be read in conjunction with chapter 7, which describes common use cases.

17.1 `riscvCSRAttrs` Fields

The `riscvCSRAttrs` type is defined as follows:

```
typedef struct riscvCSRAttrsS {

    const char        *name;            // register name
    const char        *desc;            // register description
    vmiosObjectP       object;          // custom extension
    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
    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
    vmiReg             reg;             // register
    vmiReg             writeMaskV;      // configuration-dependent write mask
    Uns32              writeMaskC32;    // constant 32-bit write mask
    Uns64              writeMaskC64;    // constant 64-bit write mask

    riscvCSRStateenBit Smstateen    :8; // whether xstateen-controlled access
    riscvCSRTrap       trap         :2; // whether trapped
    riscvCSRTrace      noTraceChange:2; // trace mode
    Bool               wEndBlock    :1; // whether write terminates this block
    Bool               wEndRM       :1; // whether write invalidates RM assumption
    Bool               noSaveRestore:1; // whether to exclude from save/restore
    Bool               writeRd      :1; // whether write updates Rd
    Bool               aliasV       :1; // whether CSR has virtual alias
    Bool               undefined    :1; // whether CSR is undefined
    Bool               forceRO      :1; // type is RO even if write CB defined

} riscvCSRAttrs;
```

The fields in this structure are as follows:

17.1.1 Field `name`

This is the name of the CSR, used in tracing and for named access from a harness.

17.1.2 Field `desc`

This is a CSR description, used by documentation generation.

17.1.3 Field `object`

This field should be zeroed in the CSR template structure. It is initialized by the RISC-V model with a pointer to the extension CSR object.

17.1.4 Field `csrNum`

This field should be set to the CSR number, in the range 0-4095.

17.1.5 Field `arch`

This field gives special architectural constraints which determine whether the CSR is *present* or not. For example, if `arch` is set to `ISA_S`, then the CSR will only be present for variants that implement *Supervisor* mode. If the field is zero, then whether the CSR is present is determined by the standard RISC-V rules applying to the CSR number, specified by the `csrNum` field (see the RISC-V Privileged Architecture Specification).

17.1.6 Field `access`

This field gives special architectural constraints which determine whether the CSR is *accessible* or not. For example, if `arch` is set to `ISA_S`, then the CSR will only be accessible if the hart is operating in *Supervisor* mode (or any higher privilege level, Machine mode in this case). If the field is zero, then whether the CSR is accessible is determined by the standard RISC-V rules applying to the CSR number, specified by the `csrNum` field (see the RISC-V Privileged Architecture Specification).

17.1.7 Field `version`

This field indicates the minimum Privileged Architecture version for which the CSR is supported. In extension objects this is likely always to be zero, implying to restriction.

17.1.8 Field `presentCB`

This field specifies an optional function that implements custom rules to determine whether a CSR is present. The function is of type `riscvCSRPresentFn`, defined as follows:

```
#define RISCV_CSR_PRESENTFN(_NAME) Bool _NAME( \
    riscvCSRAttrsCP attrs,      \
    riscvP          riscv       \
)
typedef RISCV_CSR_PRESENTFN((*riscvCSRPresentFn));
```

The function should return `True` if the CSR is present and `False` otherwise; typically, it will examine some state within the extension object to determine this. For example:

```
static RISCV_CSR_PRESENTFN(csrxPresent) {
    return attrs->object->csrxPresent; // set by an extension parameter?
}
```

Note that the `object` field is used here to obtain a pointer to the extension object.

17.1.9 Field `readCB`

This field specifies an optional function that is called implement a read of the CSR. The function is of type `riscvCSRReadFn`, defined as follows:

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

The function should return the CSR value as an `Uns64`. For example:

```
static RISCV_CSR_READFN(custom_rw3_32R) {

    vmiosObjectP object = attrs->object;
    Int32        result = RD_XCSR(object, custom_rw3_32);

    return result;
}
```

Note that the `object` field is used here to obtain a pointer to the extension object.

If the `readCB` callback is not specified, then a read of the CSR will return the raw value from any register specified by the `reg` field. If that field is `VMI_NOREG`, then a read of the register will return zero.

17.1.10 Field `readWriteCB`

This field specifies an optional function that is called implement a read of the CSR in a read-write context (i.e. using instruction `csrrw` or an alias of it). A few standard CSRs such as `mip` have special behavior in such cases. If the `readWriteCB` is not specified, then the `readCB` is used for CSR reads in such cases as well.

17.1.11 Field `writeCB`

This field specifies an optional function that is called implement a write of the CSR. The function is of type `riscvCSRWriteFn`, defined as follows:

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

The function takes the value to be written and should return the resulting CSR value as an `Uns64` (this may be different to the given value if CSR bits are read-only or if special WARL restrictions are implemented). For example:

```
static RISCV_CSR_WRITEFN(custom_rw3_32W) {

    vmiosObjectP object = attrs->object;

    WR_XCSR(object, custom_rw3_32, newValue);

    return newValue;
}
```

Note that the `object` field is used here to obtain a pointer to the extension object.

If the `writeCB` callback is not specified, then whether the CSR is writable is determined by `csrNum[11:10]` using standard RISC-V rules (see the RISC-V Privileged Architecture Specification). However, if the `writeCB` callback is specified for a CSR with a number that would *not* normally be writable, that CSR is writable (overriding the standard rules).

If the writeCB callback is absent for a CSR that is defined to be writable by standard RISC-V CSR number rules, then a write of the CSR will update the raw value in any register specified by the reg field, using the masking constraints described for that field below. If the reg field is VMI_NOREG, then writes of the register will be ignored.

17.1.12 Field wstateCB

This field specifies an optional function that is called update code generator state after a write of the CSR. It is used in the RISC-V base model but is not intended for use in extensions.

17.1.13 Field reg

This vmiReg field specifies an optional description of the register value in the extension object. If the register value is not held as a field in the extension object, reg should be set to VMI_NOREG; otherwise, it should be initialized using the XCSR_REG_MT macro. For example, given this extension object definition:

```
typedef struct addCSRsCSRsS {
    CSR_REG_DECL (custom_rw1_32);
} addCSRsCSRs;

typedef struct vmiosObjectS {
    addCSRsCSRs csr;
} vmiosObject;
```

Then the custom_rw1_32 CSR value in the extension object can be referenced like this:

```
XCSR_REG_MT(custom_rw1_32)
```

If read callbacks are not specified for the CSR, then the value indicated by the reg field will be returned; if this is VMI_NOREG, then zero will be returned.

If a write callback is not specified for the CSR and the CSR number indicates it has write access, then the value indicated by the reg field will be written; if this is VMI_NOREG, then the write will be ignored. The written value will be masked according to these rules:

1. If the writeMaskV field is not VMI_NOREG, then this specifies a configurable mask register in the extension object. Only bits that are *non-zero* in this mask will be updated in the written register; bits that are *zero* in the mask will be preserved in the target register.
2. If the writeMaskV field is VMI_NOREG and a *32-bit* CSR is written, then field writeMaskC32 specifies a constant 32-bit mask of writable bits in the target register. Only bits that are *non-zero* in this mask will be updated in the written register; bits that are *zero* in the mask will be preserved in the target register. Set this field to all-ones to allow unrestricted access to the target register.
3. If the writeMaskV field is VMI_NOREG and a *64-bit* CSR is written, then field writeMaskC64 specifies a constant 64-bit mask of writable bits in the target register. Only bits that are *non-zero* in this mask will be updated in the written

register; bits that are *zero* in the mask will be preserved in the target register. Set this field to all-ones to allow unrestricted access to the target register.

17.1.14 Field writeMaskV

This field specifies a configurable write mask for the CSR. See section 17.1.13.

17.1.15 Field writeMaskC32

This field specifies a constant 32-bit write mask for the CSR. See section 17.1.13.

17.1.16 Field writeMaskC64

This field specifies a constant 64-bit write mask for the CSR. See section 17.1.13.

17.1.17 Field Smstateen

This field specifies any `Smstateen` bit that controls access to this CSR. If it is non-zero then `Smstateen` control register state will be added to control access to the register using the indicated bit. Some basic control bit aliases are defined by the `riscvCSRStateenBit` enumeration:

```
typedef enum riscvCSRStateenBitE {
    bit_stateen_NA       = 0,
    bit_stateen_Zfinx    = 1,
    bit_stateen_Zcmt     = 2,
    bit_stateen_xcse     = 57,
    bit_stateen_IMSIC    = 58,
    bit_stateen_AIA      = 59,
    bit_stateen_sireg    = 60,
    bit_stateen_xenvcfg  = 62,
    bit_stateen_xstateen = 63,
} riscvCSRStateenBit;
```

Extensions may specify these values or any other values in the range 1-255.

17.1.18 Field trap

This field specifies whether CSR accesses should be trapped by either `mstatus.TVM` or `hvictl.VTI`, according to the `riscvCSRTrap` enumeration:

```
typedef enum riscvCSRTrapE {
    CSRT_NA,            // CSR not trapped
    CSRT_TVM = 1<<0,    // trapped by mstatus.TVM=1 (e.g. satp register)
    CSRT_VTI = 1<<1,    // trapped by hvictl.VTI=1 (e.g. sip, sie registers)
} riscvCSRTrap;
```

17.1.19 Field noTraceChange

This field specifies whether the CSR value should be shown during simulation when register tracing is enabled. Normally register values should be traced, but for some volatile registers, like instruction counters, this can produce a lot of noise. Trace options are defined by the `riscvCSRTrace` enumeration:

```
typedef enum riscvCSRTraceE {
    RCSRT_YES,          // always trace CSR
    RCSRT_NO,           // never trace CSR
    RCSRT_VOLATILE,     // trace only in volatile mode
```

```
} riscvCSRTrace;
```

Value `RCSRT_YES` indicates the CSR value should always be shown in the trace.
Value `RCSRT_NO` indicates the CSR value should never be shown in the trace.
Value `RCSRT_VOLATILE` indicates the CSR value should be shown in the trace only when parameter `traceVolatile` on the RISC-V model is `True`.

17.1.20 Field `wEndBlock`

This field specifies whether, when the CSR is written by a CSR update instruction, that instruction must be the last in a JIT-translated code block. This is necessary if writes to the CSR can modify constraints enforced using VMI block masks or similar constructs. See the *OVP Processor Modeling Guide* for more information about the use of block masks in processor models.

17.1.21 Field `wEndRM`

This field specifies whether, when the CSR is written by a CSR update instruction, any assumptions about floating point rounding mode must be invalidated. This is intended for use in the base model and not likely to be required in an extension.

17.1.22 Field `noSaveRestore`

The simulation environment supports save and restore of processor model state. To implement save and restore, CSRs are usually simply read and written using the defined read/write callbacks for that CSR, or using the raw register value if no read/write callbacks are specified. This field indicates that the CSR should not be saved and restored in this way.

17.1.23 Field `writeRd`

This field indicates that, when a CSR is written and has a write callback, the value returned by the write callback should be assigned to the `rd` field in the `csrrs`/`csrrc` instruction that accesses the CSR. This is not standard CSR behavior, but is required to implement CSRs like `mnxti` in the CLIC extension.

17.1.24 Field `aliasV`

This field indicates that a Supervisor mode CSR has a virtual alias, with index `csrNum+0x100`. When a hart is executing in VS mode, apparent references to the CSR will instead access the virtual alias CSR.

17.1.25 Field `undefined`

This field indicates that a CSR is undefined, and that accesses to it should always generate an Illegal Instruction trap. This is useful for extensions that remove standard CSRs for any reason.

17.1.26 Field `forceRO`

This field indicates that a CSR is read-only, regardless of:

- whether a write callback is defined for it, which would normally make it be considered read-write even when the CSR index is in the read-only region.
- the index is in a CSR region defined as a read-write region

This may be used when defining a write callback that supports artifact writes for a read-only CSR.

17.2 CSR Definition Macros

CSRs can be defined using C structure declarations using named-field syntax if required. However, there are several macros defined in file `riscvModelCallbackTypes.h` that cover the common types of declaration in a more concise form. These are described below.

17.2.1 Macro XCSR_ATTR_UIP

This macro declares an undefined CSR. This can be used where an extension model does not implement a standard CSR for some reason. It is defined as:

```
#define XCSR_ATTR_UIP(_ID, _NUM, _ARCH, _EXT) [XCSR_ID(_ID)] = { \
    .extension = _EXT,                                  \
    .baseAttrs = {                                      \
        name          : #_ID,                           \
        csrNum        : _NUM,                           \
        arch          : _ARCH,                          \
        undefined     : True                            \
    }                                                   \
}
```

17.2.2 Macro XCSR_ATTR_NIP

This macro declares a defined but unimplemented CSR. This can be used where an extension model is under development and a temporary placeholder is needed for future implementation. It is defined as:

```
#define XCSR_ATTR_NIP( \
    _ID, _NUM, _ARCH, _EXT, _ENDB,_ENDRM,_NOTR,_TRAP, _DESC \
) [XCSR_ID(_ID)] = { \
    .extension = _EXT,                                  \
    .baseAttrs = {                                      \
        name          : #_ID,                           \
        desc          : _DESC" (not implemented)",      \
        csrNum        : _NUM,                           \
        arch          : _ARCH,                          \
        wEndBlock     : _ENDB,                          \
        wEndRM        : _ENDRM,                         \
        noTraceChange : _NOTR,                          \
        trap          : _TRAP,                          \
    }                                                   \
}
```

17.2.3 Macro XCSR_ATTR_T__

This macro declares a CSR that has a value implemented by a field in the extension object with no write mask constraints. There can optionally be read or write callbacks. It is defined as:

```
#define XCSR_ATTR_T__( \
    _ID, _NUM, _ARCH, _EXT, _ENDB,_ENDRM,_NOTR,_TRAP, _DESC, _RCB, _RWCB, _WCB \
) [XCSR_ID(_ID)] = { \
    .extension = _EXT,                                  \
    .baseAttrs = {                                      \
        name          : #_ID,                           \
        desc          : _DESC,                          \
        csrNum        : _NUM,                           \
        arch          : _ARCH,                          \
        wEndBlock     : _ENDB,                          \
```

```
        wEndRM        : _ENDRM,                     \
        noTraceChange : _NOTR,                      \
        trap          : _TRAP,                      \
        readCB        : _RCB,                       \
        readWriteCB   : _RWCB,                      \
        writeCB       : _WCB,                       \
        reg           : XCSR_REG_MT(_ID),           \
        writeMaskC32  : -1,                         \
        writeMaskC64  : -1                          \
    }                                               \
}
```

17.2.4 Macro XCSR_ATTR_TC_

This macro declares a CSR that has a value implemented by a field in the extension object and *constant* write masks. There can optionally be read or write callbacks. It is defined as:

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

17.2.5 Macro XCSR_ATTR_TV_

This macro declares a CSR that has a value implemented by a field in the extension object and *configurable* write mask. There can optionally be read or write callbacks. It is defined as:

```
#define XCSR_ATTR_TV_( \
    _ID, _NUM, _ARCH, _EXT, _ENDB,_ENDRM,_NOTR,_TRAP, _DESC, _RCB, _RWCB, _WCB \
) [XCSR_ID(_ID)] = { \
    .extension = _EXT,                              \
    .baseAttrs = {                                  \
        name          : #_ID,                       \
        desc          : _DESC,                      \
        csrNum        : _NUM,                       \
        arch          : _ARCH,                      \
        wEndBlock     : _ENDB,                      \
        wEndRM        : _ENDRM,                     \
        noTraceChange : _NOTR,                      \
        trap          : _TRAP,                      \
        readCB        : _RCB,                       \
        readWriteCB   : _RWCB,                      \
        writeCB       : _WCB,                       \
        reg           : XCSR_REG_MT(_ID),           \
        writeMaskV    : XCSR_MASK_MT(_ID)           \
    }                                               \
}
```

17.2.6 Macro XCSR_ATTR_P__

This macro declares a CSR that is implemented by callbacks only and has no corresponding value field in the extension object. It is defined as:

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

If both read and write callbacks are absent, then this specifies a CSR that reads as zero and ignores writes.

18 Appendix: Configuring Standard Extensions

This appendix provides detailed information about configuring the model to implement standard extensions using parameters.

18.1 Ratified Extension Support

The following ratified extensions are supported in the model using appropriate parameterization, with the exception of the `Sscofpmf` extension which is not currently supported. See subsections on following pages for information about how these extensions can be configured in the OVP RISC-V model. Ratified extensions are listed at Ratified Extensions - Home - RISC-V International (riscv.org).

RISC-V Quality-of-Service (QoS) Identifiers	`Ssqosid`
Obviating Memory-Management Instructions after Marking PTEs Valid	`Svvptc`
Resumable Non-Maskable Interrupts	`Smrnmi`
Shadow Stacks and Landing Pads	`Zicfiss, Zicfilp`
BF16 Extensions	`Zfbfmin, Zvfbfmin, Zvfbfwma`
Zaamo and Zalrsc Extensions	`Zaamo, Zalrsc`
B Standard Extension for Bit Manipulation Instructions	`B`
Byte and Halfword Atomic Memory Operations (Zabha)	`Zabha`
RISC-V Supervisor Counter Delegation	`Smcdeleg, Ssccfg`
May-Be-Operations	`Zimop, Zcmop`
RISC-V Indirect CSR Access (Smcsrind/Sscsrind)	`Smcsrind, Sscsrind`
RISC-V Pointer Masking Extensions	`Ssnpm, Smnpm, Smmpm`
RISC-V Integer Conditional (Zicond) operations extension	`Zicond`
Hardware Updating of PTE A/D Bits (Svadu)	`Svadu`
RISC-V Cycle and Instret Privilege Mode Filtering (Smcntrpmf)	`Smcntrpmf`
Atomic Compare-and-Swap (CAS) Instructions (Zacas)	`Zacas`
RISC-V Cryptography Extensions Volume II: Vector Instructions	`Zvbb, Zvbc, Zvkb, Zvkg, Zvkn, Zvknc, Zvkned, Zvkng, Zvknha, Zvknhb, Zvks, Zvksc, Zvksed, Zvksg, Zvksh, Zvkt`
"Zfa" Standard Extension for Additional Floating-Point Instructions	`Zfa`
RISC-V Advanced Interrupt Architecture	`Smaia, Ssaia`
"Zvfh/Zvfhmin:" Vector Extension for Half-Precision Floating-Point Arithmetic/Vector Extension for Minimal Half-Precision Floating-Point Arithmetic	`Zvfh, Zvfhmin`
"Zihintntl" Non-Temporal Locality Hints	`Zihintntl`
RISC-V Code Size Reduction	`Zca, Zcb, Zcd, Zce, Zcf, Zcmp, Zcmt`
RISC-V Profiles	`RVA20, RVI20, RVA22`
"Zicntr" and "Zihpm" Counters	`Zicntr, Zihpm`
RV32E and RV64E Base Integer Instruction Sets	`RV32E/RV64E`

"Ztso" Standard Extension for Total Store Ordering	`Ztso`
RISC-V Wait-on-Reservation-Set (Zawrs) extension	`Zawrs`
Zmmul Extension	`Zmmul`
PMP Enhancements for memory access and execution prevention on Machine mode (Smepmp)	`Smepmp`
RISC-V Base Cache Management Operation ISA Extensions	`Zicbom, Zicbop, Zicboz`
RISC-V Bit-Manipulation ISA-extensions	`Zba, Zbb, Zbc, Zbs`
RISC-V Count Overflow and Mode-Based Filtering Extension	`Sscofpmf`
RISC-V Cryptography Extensions Volume I: Scalar & Entropy Source Instructions	`Zbkb, Zbkc, Zbkx, Zknd, Zkne, Zknh, Zksed, Zksh, Zkn, Zks, Zkt, Zk, Zkr`
RISC-V State Enable Extension	`Smstateen`
RISC-V "stimecmp / vstimecmp" Extension	`Sstc`
RISC-V Vector Extension	`Zve32x, Zve32f, Zve64x, Zve64f, Zve64d, Zve, Zvl32b, Zvl64b, Zvl128b, Zvl256b, Zvl512b, Zvl1024b, Zvl, Zv`
The RISC-V Instruction Set Manual Volume II: Privileged Architecture	`Sm1p12, Ss1p12, Sv57, Hypervisor, Svinval, Svnapot, Svpbmt`
"Zfh" and "Zfhmin" Standard Extensions for Half-Precision Floating-Point	`Zfh, Zfhmin`
"Zfinx", "Zdinx", "Zhinx", "Zhinxmin": Standard Extensions for Floating-Point in Integer Registers	`Zfinx, Zdinx, Zhinx, Zhinxmin`
"Zihintpause" Pause Hint	`Zihintpause`
"Zicsr", Control and Status Register (CSR) Instructions	`Zicsr`
"Zifencei" Instruction-Fetch Fence	`Zifencei`

18.1.1 *RISC-V Quality-of-Service (QoS) Identifiers*

Enabled using parameter `Ssqosid`.

18.1.2 *Obviating Memory-Management Instructions after Marking PTEs Valid*

The model always behaves as if this extension is enabled.

18.1.3 *Resumable Non-Maskable Interrupts*

Enabled using parameter `rnmi_version` with version 0.4.

18.1.4 *Shadow Stacks and Landing Pads*

The following table shows how to configure the model for each sub-extension listed here.

`Zicfiss`	Set parameter `Zicfiss` to `T`.
`Zicfilp`	Set parameter `Zicfilp` to `T`.

18.1.5 *BF16 Extensions*

The following table shows how to configure the model for each sub-extension listed here.

Zfbfmin	Set parameter Zfbfmin to T on a variant with the F extension.
Zvfbfmin	Set parameter Zvfbfmin to T on a variant with the F and V extensions.
Zvfbfwma	Set parameter Zvfbfwma to T on a variant with the F and V extensions.

18.1.6 *Zaamo and Zalrsc Extensions*

The following table shows how to configure the model for each sub-extension listed here when the A extension is configured.

Zaamo	Set parameter Zaamo to T.
Zalrsc	Set parameter Zalrsc to T.

18.1.7 *B Standard Extension for Bit Manipulation Instructions*

Enabled using parameter misa_B_Zba_Zbb_Zbs when the B extension is configured.

18.1.8 *Byte and Halfword Atomic memory Operations (Zabha)*

Enabled using parameter Zabha when the A extension is configured.

18.1.9 *RISC-V Supervisor Counter Delegation (Smcdeleg/Ssccfg)*

Enabled using parameter Smcdeleg.

18.1.10 *May-Be-Operations*

The following table shows how to configure the model for each sub-extension listed here.

Zimop	Set parameter Zimop to T.
Zcmop	Set parameter Zcmop to T.

18.1.11 *RISC-V Indirect CSR Access (Smcsrind/Sscsrind)*

Enabled using parameter Smcsrind. (Sscsrind is enabled if this parameter is set and Supervisor mode is present).

18.1.12 *RISC-V Pointer Masking Extensions*

The following table shows how to configure the model for each sub-extension listed here.

Ssnpm	Set parameter Ssnpm to T.
Smnpm	Set parameter Smnpm to T.
Smmpm	Set parameter Smmpm to T.

18.1.13 *RISC-V Integer Conditional (Zicond) operations extension*

Enabled using parameter Zicond.

18.1.14 *Hardware Updating of PTE A/D Bits (Svadu)*

Enabled using parameter `Svadu`.

18.1.15 *RISC-V Cycle and Instret Privilege Mode Filtering (Smcntrpmf)*

Enabled using parameter `Smcntrpmf`.

18.1.16 *Atomic Compare-and-Swap (CAS) Instructions (Zacas)*

Enabled using parameter `Zacas`.

18.1.17 *RISC-V Cryptography Extensions Volume II: Vector Instructions*

For all listed extensions, the V and K extensions must be enabled by setting parameter `add_Extensions` to include `V` and `K`. The following table shows how to further configure the model for each sub-extension listed here.

`Zvbb`	Set parameter `Zvbb` to `T`.
`Zvbc`	Set parameter `Zvbc` to `T`.
`Zvkb`	Set parameter `Zvkb` to `T`.
`Zvkg`	Set parameter `Zvkg` to `T`.
`Zvkn`	Set parameter `Zvkn` to `T`.
`Zvknc`	Set parameters `Zvkn` and `Zvbc` to `T`.
`Zvkned`	Set parameter `Zvkned` to `T`.
`Zvkng`	Set parameters `Zvkn` and `Zvkg` to `T`.
`Zvknha`	Set parameter `Zvknha` to `T`.
`Zvknhb`	Set parameter `Zvknhb` to `T`.
`Zvks`	Set parameter `Zvks` to `T`.
`Zvksc`	Set parameters `Zvks` and `Zvbc` to `T`.
`Zvksg`	Set parameters `Zvks` and `Zvkg` to `T`.
`Zvksh`	Set parameter `Zvksh` to `T`.
`Zvksed`	Set parameter `Zvksed` to `T`.
`Zvkt`	*Data-independent execution latency, not relevant for this model*

18.1.18 *"Zfa" Standard Extension for Additional Floating-Point Instructions*

Enabled using parameter `Zfa` on variant with floating point configured.

18.1.19 *RISC-V Advanced Interrupt Architecture Extension*

The following table shows how to configure the model for each sub-extension listed here. See section 5.7 for details of further parameters to refine the Advanced Interrupt Architecture configuration.

`Smaia`	Set parameter `Smaia` to `T`.
`Ssaia`	Set parameter `Smaia` to `T`, use variant with S extension

18.1.20 *"Zvfh/Zvfhmin:" Vector Extension for Half-Precision Floating-Point Arithmetic/Vector Extension for Minimal Half-Precision Floating-Point Arithmetic*

The following table shows how to configure the model for each sub-extension listed here.

`Zvfh`	Set parameter `Zvfh` to `T`, use variant with V extension and floating point configured
`Zvfhmin`	Set parameter `Zvfhmin` to `T`, use variant with V extension and floating point configured

18.1.21 *"Zihintntl" Non-Temporal Locality Hints*

Enabled using parameter `Zihintntl`. The hint instructions are executed but otherwise have no effect on the model.

18.1.22 *RISC-V Code Size Reduction*

For all listed extensions, the C extension must be enabled by either setting parameter `add_Extensions` to include `C`, or instantiating a base variant with C extension (e.g. `RV32GC`). The following table shows how to further configure the model for each sub-extension listed here.

`Zca`	Set parameter `Zca` to `T`
`Zcb`	Set parameter `Zcb` to `T`
`Zcd`	Implied if C and D extensions are enabled and all `Zcm*` extensions are absent.
`Zce`	(Collective name for `Zca`, `Zcb`, `Zcmp`, `Zcmt` and optionally `Zcf` extensions)
`Zcf`	Set parameter `Zcf` to `T`
`Zcmp`	Set parameter `Zcmp` to `T`
`Zcmt`	Set parameter `Zcmt` to `T`

18.1.23 *RISC-V Profiles*

Enabled using more fundamental parameters documented here.

18.1.24 "Zicntr" and "Zihpm" Counters

These extensions are both enabled by default. The CSRs that they affect can be selectively disabled as follows:

`cycle`	Set parameter `cycle_undefined` to `T`
`mcycle`	Set parameter `mcycle_undefined` to `T`
`time`	Set parameter `time_undefined` to `T`
`instret`	Set parameter `instret_undefined` to `T`
`minstret`	Set parameter `minstret_undefined` to `T`
`hpmcounter`	Set parameter `hpmcounter_undefined` to `T`. Note that these registers are implementation-defined and always read as zero in the OVP RISC-

	V base model.
mhpmcounter	Set parameter mhpmcounter_undefined to T. Note that these registers are implementation-defined and always read as zero in the OVP RISC-V base model.

18.1.25 RV32E and RV64E Base Integer Instruction Sets

Set parameter add_Extensions to include E, or instantiate base variant with E extension (e.g. RV32EC).

18.1.26 *"Ztso" Standard Extension for Total Store Ordering*

The OVP RISC-V model behaves as if this extension is always enabled.

18.1.27 *RISC-V Wait-on-Reservation-Set (Zawrs) extension*

Enabled using parameter Zawrs.

18.1.28 *Zmmul Extension*

Enabled using parameter Zmmul.

18.1.29 *RISC-V PMP Enhancements for memory access and execution prevention on Machine mode (Smepmp)*

Enabled using parameter Smepmp.

18.1.30 *RISC-V Base Cache Management Operation ISA Extensions*

The RISC-V model does not implement caches, but supports the instructions specified in this extension. The following table shows how to configure the model for each sub-extension listed here. See section 5.8 for information about other parameters that affect these extensions.

Zicbom	Set parameter Zicbom to T
Zicbop	Set parameter Zicbop to T
Zicboz	Set parameter Zicboz to T

18.1.31 *RISC-V Bit-Manipulation ISA-extensions*

For all listed extensions, the B extension must be enabled by either setting parameter add_Extensions to include B, or instantiating a base variant with B extension (e.g. RV32GCB). The following table shows how to further configure the model for each sub-extension listed here.

Zba	Set parameter Zba to T
Zbb	Set parameter Zbb to T
Zbc	Set parameter Zbc to T
Zbs	Set parameter Zbs to T

18.1.32 *RISC-V Count Overflow and Mode-Based Filtering Extension*

Enabled using parameter `Sscofpmf`. Note that performance monitors are implementation-defined, so this model implements only the CSR state defined by the `Sscofpmf` extension and not the counters themselves (all counters are hard-wired to zero).

18.1.33 *RISC-V Cryptography Extensions Volume I: Scalar & Entropy Source Instructions*

For all listed extensions, the K extension must be enabled by either setting parameter `add_Extensions` to include `K`, or instantiating a base variant with K extension (e.g. `RV32GCK`). The following table shows how to further configure the model for each sub-extension listed here.

`Zbkb`	Set parameter `Zbkb` to `T`
`Zbkc`	Set parameter `Zbkc` to `T`
`Zbkx`	Set parameter `Zbkx` to `T`
`Zknd`	Set parameter `Zknd` to `T`
`Zkne`	Set parameter `Zkne` to `T`
`Zknh`	Set parameter `Zknh` to `T`
`Zksed`	Set parameter `Zksed` to `T`
`Zksh`	Set parameter `Zksh` to `T`
`Zkn`	(Collective name for `Zbk`-prefixed and `Zkn`-prefixed extensions above)
`Zks`	(Collective name for `Zbk`-prefixed and `Zks`-prefixed extensions above)
`Zkt`	*Data-independent execution latency, not relevant for this model*
`Zkr`	Set parameter `Zkr` to `T`
`Zk`	(Collective name for `Zkn`, `Zkr`, and `Zkt` extensions above)

18.1.34 *RISC-V State Enable Extension*

Enabled using parameter `Smstateen`.

18.1.35 *RISC-V "stimecmp / vstimecmp" Extension*

Enabled using parameter `Sstc`.

18.1.36 *RISC-V Vector Extension*

For all listed extensions, the V extension must be enabled by either setting parameter `add_Extensions` to include `V`, or instantiating a base variant with V extension (e.g. `RV32GCV`). The following table shows how to further configure the model for each sub-extension listed here.

`Zve32x`	Set parameter `Zve32x` to `T`
`Zve32f`	Set parameter `Zve32f` to `T`
`Zve64x`	Set parameter `Zve64x` to `T`
`Zve64f`	Set parameter `Zve64f` to `T`
`Zve64d`	Set parameter `Zve64d` to `T`

Zve	(Family name for Zve-prefixed extensions above)
Zvl32b	Set parameter VLEN=32.
Zvl64b	Set parameter VLEN=64.
Zvl128b	Set parameter VLEN=128.
Zvl256b	Set parameter VLEN=256.
Zvl512b	Set parameter VLEN=512.
Zvl1024b	Set parameter VLEN=1024.
Zvl	(Family name for Zvl-prefixed extensions above)
Zv	Instantiate either RV32GCV or RV64GCV variants

18.1.37 *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*

The following table shows how to configure the model for each sub-extension listed here.

Sm1p12	Set parameter priv_version to 1.12
Ss1p12	Set parameter priv_version to 1.12
Sv57	Set parameter Sv_modes to include bit 10 (0x400). Other bits in this mask can be set to specify Sv32, Sv39, Sv48 and Sv64.
Hypervisor	Set parameter add_Extensions to include H, or instantiate base variant with H extension (e.g. RV64GCH).
Svinval	Set parameter Svinval to T
Svnapot	Set parameter Svnapot_page_mask to indicate implemented intermediate page sizes (e.g. 1<<16 means 64KiB contiguous regions are supported)
Svpbmt	Set parameter Svpbmt to T

18.1.38 *"Zfh" and "Zfhmin" Standard Extensions for Half-Precision Floating-Point*

The following table shows how to configure the model for each sub-extension listed here.

Zfh	Set parameter add_Extensions to include F and set parameter Zfh to T.
Zfhmin	Set parameter add_Extensions to include F and set parameter Zfhmin to T.

18.1.39 *"Zfinx", "Zdinx", "Zhinx", "Zhinxmin": Standard Extensions for Floating-Point in Integer Registers*

The following table shows how to configure the model for each sub-extension listed here.

Zfinx	Set parameter add_Extensions to include F and set parameter Zfinx_version to 1.0.
Zdinx	Set parameter add_Extensions to include F and D, and set parameter Zfinx_version to 1.0.
Zhinx	Set parameter add_Extensions to include F, set parameter

	Zfinx_version to 1.0 and set parameter Zfh to T.
Zhinxmin	Set parameter add_Extensions to include F, set parameter Zfinx_version to 1.0 and set parameter Zfhmin to T.

18.1.40 *"Zihintpause" Pause Hint*

The PAUSE hint is always treated as a NOP by the model, so there is no option to explicitly enable this extension.

18.1.41 *"Zicsr", Control and Status Register (CSR) Instructions*

Enabled using parameter Zicsr. Note that disabling Zicsr is not useful unless the model is extended to support a custom replacement for standard CSRs.

18.1.42 *"Zifencei" Instruction-Fetch Fence*

Enabled using parameter Zifencei.

18.2 RISC-V Profile Extensions

The RISC-V Profile definition (riscv-profiles/profiles.adoc at main · riscv/riscv-profiles · GitHub) gives new names for a number of existing RISC-V features. The following table shows how to configure the model for each extension listed there.

`Ziccif`	*Always enabled*
`Ziccrse`	*Always enabled*
`Ziccamoa`	*Always enabled*
`Zicclsm`	Set parameter `unaligned` to `T`
`Za64rs`	Set parameter `lr_sc_grain` to `64`
`Za128rs`	Set parameter `lr_sc_grain` to `128`
`Zic64b`	Set parameters `cmomp_bytes` and `cmoz_bytes` to `64`
`Svbare`	Set parameter `Sv_modes` to include bit 0 (`0x1`).
`Svade`	Set parameters `updatePTEA` and `updatePTED` to `F`
`Sccptr`	*Always enabled*
`Sscounterenw`	*Not applicable: base model does not implement HPM counters*
`Sstvecd`	*Always enabled*
`Sstvala`	Set parameter `tval_zero` to `F` and parameter `tval_ii_code` to `T`
`Ssu64xl`	Run variant with `XLEN=64`
`Ssstateen`	Set parameter `Smstateen` to `T`
`Shcounterenw`	*Not applicable: base model does not implement HPM counters*
`Shvstvala`	Set parameter `tval_zero` to `F` and parameter `tval_ii_code` to `T`
`Shtvala`	Set parameter `tval_zero` to `F` and parameter `tval_ii_code` to `T`
`Shvsatpa`	*Always enabled*
`SvNNx4`	*Always enabled*

18.3 Other Extensions

The following extensions are supported in the model using appropriate parameterization, but are not mentioned on the *Recently Ratified Extensions* page. Some are not yet ratified, in which case note that behavior may change prior to ratification.

"Sdext" ISA Extension	`Sdext`
"Sdtrig" ISA Extension	`Sdtrig`
Core-Local Interrupt Controller (CLIC) RISC-V Privileged Architecture Extensions	`smclic, ssclic, suclic, smclicshv, smclicconfig`

18.3.1 "Sdext" ISA Extension

This is enabled by setting parameter `debug_mode` to a value other than `none`. See section 5.17 for details of further parameters to refine Debug mode.

18.3.2 "Sdtrig" ISA Extension

This is enabled by setting parameter `trigger_num` to a non-zero value. See section 5.18 for details of further parameters to refine the trigger module.

18.3.3 Core-Local Interrupt Controller (CLIC) RISC-V Privileged Architecture Extensions

The CLIC extension can be configured to either use an internal model or an external CLIC model – see section 5.5.

`smclic`	Set parameter `CLICLEVELS` to a non-zero value
`ssclic`	Set parameter `CLICLEVELS` to a non-zero value on a variant with S mode.
`suclic`	Set parameter `CLICLEVELS` to a non-zero value on a variant with U mode and N extension.
`smclicshv`	Set parameter `CLICLEVELS` to a non-zero value and set parameter `CLICSELHVEC` to `T`.
`smclicconfig`	Set parameter `CLICLEVELS` to a non-zero value