RISC-V Configuration Validator Documentation

Release 2.6.3

InCore Semiconductors Pvt. Ltd.

Jan 19, 2021

Contents

CHAPTER 1

Introduction

RISCV-Config (RISCV Configuration Leagalizer) is a YAML based framework which can be used to validate the specifications of a RISC-V implementation against the RISC-V privileged and unprivileged ISA spec and generate standard specification yaml file.

Caution: This is still a work in progress and non-backward compatible changes are expected to happen.

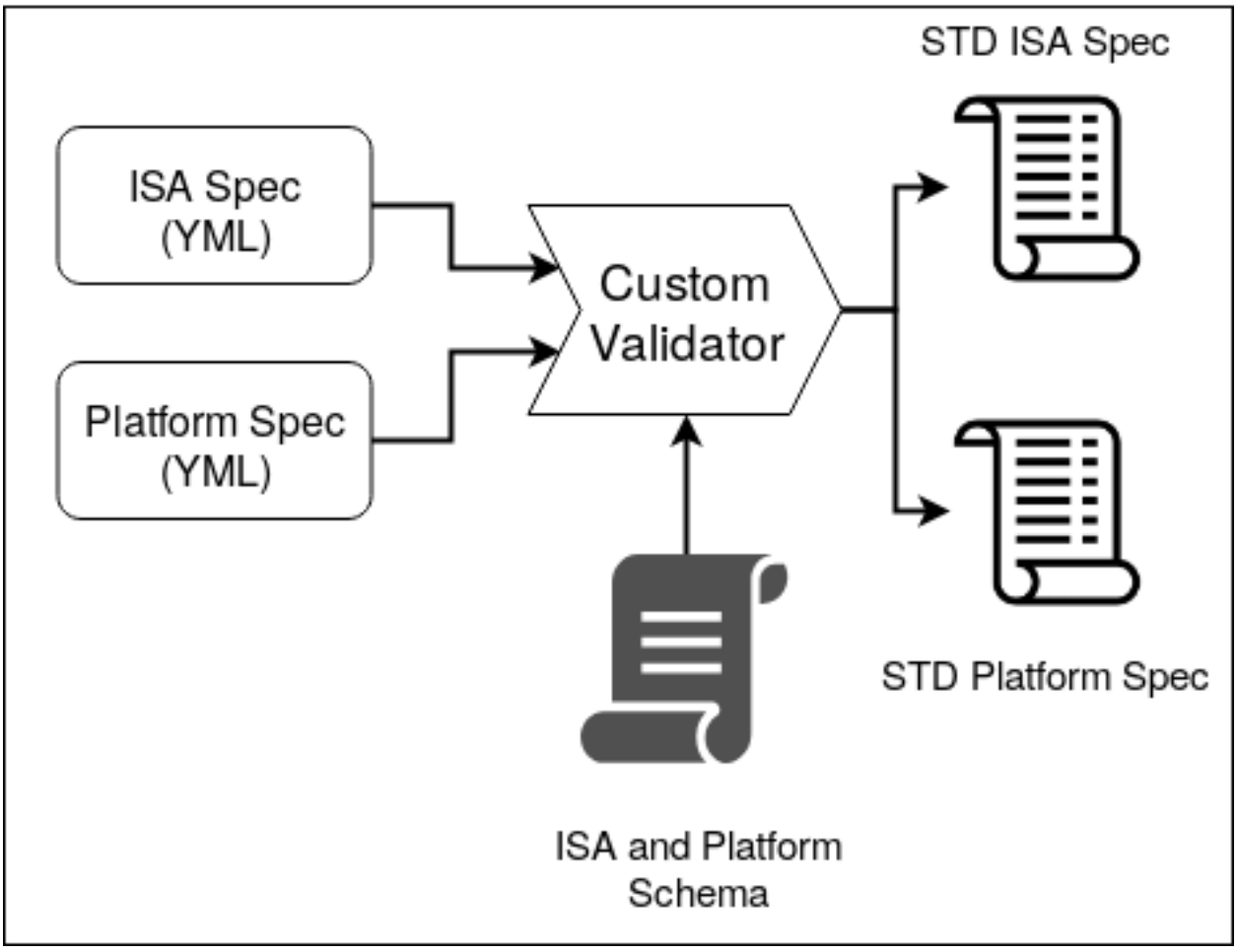
For more information on the official RISC-V spec please visit: RISC-V Specs

RISCV-Config [Repository]

CHAPTER 2

Overview

The following diagram captures the overall-flow of RISCV-Config.



The user is required to provide 2 YAML files as input:

1. **ISA Spec**: This YAML file is meant to capture the ISA related features implemented by the user. Details of this input file can be found here : *ISA YAML Spec*.
2. **Platform Spec**: This YAML file is meant to capture the platform specific features implemented by the user. Details of this input file can be found here : *Platform YAML Spec*.

2.1 Working

The ISA and Platform spec are first checked by the validator for any inconsistencies. Checks like 'F' to exist for 'D' are performed by the validator. The validator exits with an error if any illegal configuration for the spec is provided. Once the validator checks pass, two separate standard yaml files are generated, one for each input type. These standard yaml files contain all fields elaborated and additional info for each node. While the user need not specify all the fields in the input yaml files, the validator will assign defaults to those fields and generate a standard exhaustive yaml for both ISA and Platform spec.

CHAPTER 3

Quickstart

This doc is meant to serve as a quick-guide to setup RISCV-CONFIG and perform a sample validation of target specifications.

3.1 Install Python

RISCV-CONFIG requires *pip* and *python* (>=3.6) to be available on your system.

3.1.1 Ubuntu

Ubuntu 17.10 and 18.04 by default come with python-3.6.9 which is sufficient for using riscv-config.

If you are are Ubuntu 16.10 and 17.04 you can directly install python3.6 using the Universe repository:

```
$ sudo apt-get install python3.6
$ pip3 install --upgrade pip
```

If you are using Ubuntu 14.04 or 16.04 you need to get python3.6 from a Personal Package Archive (PPA):

```
$ sudo add-apt-repository ppa:deadsnakes/ppa
$ sudo apt-get update
$ sudo apt-get install python3.6 -y
$ pip3 install --upgrade pip
```

You should now have 2 binaries: `python3` and `pip3` available in your $PATH. You can check the versions as below:

```
$ python3 --version
Python 3.6.9
$ pip3 --version
pip 20.1 from <user-path>.local/lib/python3.6/site-packages/pip (python 3.6)
```

3.1.2 Centos:7

The CentOS 7 Linux distribution includes Python 2 by default. However, as of CentOS 7.7, Python 3 is available in the base package repository which can be installed using the following commands:

```
$ sudo yum update -y
$ sudo yum install -y python3
$ pip3 install --upgrade pip
```

For versions prior to 7.7 you can install python3.6 using third-party repositories, such as the IUS repository:

```
$ sudo yum update -y
$ sudo yum install yum-utils
$ sudo yum install https://centos7.iuscommunity.org/ius-release.rpm
$ sudo yum install python36u
$ pip3 install --upgrade pip
```

You can check the versions:

```
$ python3 --version
Python 3.6.8
$ pip --version
pip 20.1 from <user-path>.local/lib/python3.6/site-packages/pip (python 3.6)
```

3.1.3 Using Virtualenv for Python

Many a times folks face issues in installing and managing python versions, which is actually a major issue as many gui elements in Linux use the default python versions. In which case installing python3.6 using the above methods might break other software. We thus advise the use of **pyenv** to install python3.6.

For Ubuntu and CentosOS, please follow the steps here: https://github.com/pyenv/pyenv#basic-github-checkout

RHEL users can find more detailed guides for virtual-env here: https://developers.redhat.com/blog/2018/08/13/install-python3-rhel/#create-env

Once you have pyenv installed do the following to install python 3.6.0:

```
$ pyenv install 3.6.0
$ pip3 install --upgrade pip
$ pyenv shell 3.6.0
```

You can check the version in the **same shell**:

```
$ python --version
Python 3.6.0
$ pip --version
pip 20.1 from <user-path>.local/lib/python3.6/site-packages/pip (python 3.6)
```

3.2 Install RISCV-CONFIG

Note: If you are using a virtual environment make sure to enable that environment before performing the following steps.

```
$ pip3 install riscv_config
```

To update an already installed version of RISCV-CONFIG to the latest version:

```
$ pip3 install -U riscv_config
```

To checkout a specific version of riscv_config:

```
$ pip3 install riscv_config--1.x.x
```

Once you have RISCV_CONFIG installed, executing `riscv_config --help` should print the following output

```
riscv_config [-h] [--version] [--isa_spec YAML] [--platform_spec YAML]
                    [--work_dir DIR] [--verbose]

RISC-V Configuration Validator

optional arguments:
  --isa_spec YAML, -ispec YAML
                        The YAML which contains the ISA specs.
  --platform_spec YAML, -pspec YAML
                        The YAML which contains the Platfrorm specs.
  --verbose             debug | info | warning | error
  --version, -v         Print version of RISCV-CONFIG being used
  --work_dir DIR        The name of the work dir to dump the output files to.
  -h, --help            show this help message and exit
```

3.3 RISCV_CONFIG for Developers

Clone the repository from git and install required dependencies.

Note: you will still need python (>=3.6.0) and pip. If you are using *pyenv* as mentioned above, make sure to enable that environment before performing the following steps.

```
$ git clone https://github.com/riscv/riscv-config.git
$ cd riscv_config
$ pip3 install -r requirements.txt
```

Executing `python -m riscv_config.main --help` should display the same help message as above.

3.4 Usage Example

```
$ riscv-config -ispec examples/rv32i_isa.yaml -pspec examples/rv32i_platform.yaml
```

Executing the above command should display the following on the terminal:

```
[INFO]    : Input-ISA file
[INFO]    : Loading input file: /scratch/git-repo/github/riscv-config/examples/rv32i_
↪isa.yaml
[INFO]    : Load Schema /scratch/git-repo/github/riscv-config/riscv_config/schemas/
↪schema_isa.yaml
```

(continues on next page)

(continued from previous page)

```
[INFO]    : Initiating Validation
[INFO]    : No Syntax errors in Input ISA Yaml. :)
[INFO]    : Initiating post processing and reset value checks.
[INFO]    : Dumping out Normalized Checked YAML: /scratch/git-repo/github/riscv-
↪config/riscv_config_work/rv32i_isa_checked.yaml
[INFO]    : Input-Platform file
[INFO]    : Loading input file: /scratch/git-repo/github/riscv-config/examples/rv32i_
↪platform.yaml
[INFO]    : Load Schema /scratch/git-repo/github/riscv-config/riscv_config/schemas/
↪schema_platform.yaml
[INFO]    : Initiating Validation
[INFO]    : No Syntax errors in Input Platform Yaml. :)
[INFO]    : Dumping out Normalized Checked YAML: /scratch/git-repo/github/riscv-
↪config/riscv_config_work/rv32i_platform_checked.yaml
```

CHAPTER 4

YAML Specifications

This section provides details of the ISA and Platform spec YAML files that need to be provided by the user.

4.1 ISA YAML Spec

NOTE:

1. All integer fields accept values as integers or hexadecimals(can be used interchangeably) unless specified otherwise.
2. Different examples of the input yamls and the generated checked YAMLs can be found here : Examples

4.1.1 Vendor

Description: Vendor name.

Examples:

```
Vendor: Shakti
Vendor: Incoresemi
```

4.1.2 Device

Description: Device Name.

Examples:

```
Device: E-Class
Device: C-Class
```

Constraints:

- None

4.1.3 ISA

Description: Takes input a string representing the ISA supported by the implementation. All extension names (other than Zext) should be mentioned in upper-case. Z extensions should begin with an upper-case 'Z' followed by lower-case extension name (without Camel casing)

Examples:

```
ISA: RV32IMA
ISA: RV64IMAFDCZifencei
```

Constraints:

- Certain extensions are only valid in certain user-spec version. For, eg. Zifencei is available only in user-spec 2.3 and above.
- The ISA string must be specified as per the convention mentioned in the specifications(like subsequent Z extensions must be separated with an '_')

4.1.4 User_Spec_Version

Description: Version number of User/Non-priveleged ISA specification as string. Please enclose the version in "" to avoid type mismatches.

Examples:

```
User_Spec_Version: "2.2"
User_Spec_Version: "2.3"
```

Constraints:

- should be a valid version later than 2.2

4.1.5 Privilege_Spec_Version

Description: Version number of Priveleged ISA specification as string. Please enclose the version in "" to avoid type mismatches.

Examples:

```
Privilege_Spec_Version: "1.10"
Privilege_Spec_Version: "1.11"
```

Constraints:

- should be a valid version later than 1.10

4.1.6 hw_data_misaligned_support

Description: A boolean value indicating whether hardware support for misaligned load/store requests exists.

Examples:

```
hw_data_misaligned_support: True
hw_data_misaligned_support: False
```

Constraints:

- None

4.1.7 supported_xlen

Description: list of supported xlen on the target

Examples:

```
supported_xlen : [32]
supported_xlen : [64, 32]
supported_xlen : [64]
```

Constraints:

- None

4.1.8 pmp_granularity

Description: Granularity of pmps

Examples:

```
pmp_granularity : 2
pmp_granularity : 4
```

Constraints:

- None

4.1.9 physical_addr_sz

Description: size of the physical address

Examples:

```
physical_addr_sz : 32
```

Constraints:

- None

4.2 CSR Template

All csrs are defined using a common template. Two variants are available: csrs with subfields and those without

4.2.1 CSRs with sub-fields

```
<name>:                                  # name of the csr
  description: <text>                    # textual description of the csr
  address: <hex>                         # address of the CSR
  priv_mode: <D/M/H/S/U>                 # privilege mode that owns the register
  reset-val: <hex>                       # Reset value of the register. This an␣
↪accumulation
                                         # of the all reset values of the sub-fields
  rv32:                                  # this node and its subsequent fields can␣
↪exist
                                         # if [M/S/U]XL value can be 1
    accessible: <boolean>                # indicates if the csr is accessible in␣
↪rv32 mode or not.
                                         # When False, all fields below will be␣
↪trimmed off
                                         # in the checked yaml. False also indicates␣
↪that
                                         # access-exception should be generated.
    fields:                              # a quick summary of the list of all fields␣
↪of the
                                         # csr including a list of WPRI fields of␣
↪the csr.
      - <field_name1>
      - <field_name2>
      - - [23,30]                        # A list which contains a squashed pair
        - 6                              # (of form [lsb,msb]) of all WPRI bits␣
↪within the
                                         # csr. Does not exist if there are no WPRI␣
↪bits

    <field_name1>:                       # name of the field
      description: <text>                # textual description of the csr
      shadow: <csr-name>::<field>        # which this field shadows,'none' indicates␣
↪that
                                         # this field does not shadow anything.
      msb: <integer>                     # msb index of the field. max: 31, min:0
      lsb: <integer>                     # lsb index of the field. max: 31, min:0
      implemented: <boolean>             # indicates if the user has implemented␣
↪this field
                                         # or not. When False, all
                                         # fields below this will be trimmed.
      type:                              # type of field. Can be only one of the␣
↪following
        wlrl: [list of value-descriptors] # field is wlrl and the set of legal values.
        ro_constant: <hex>               # field is readonly and will return the␣
↪same value.
        ro_variable: True                # field is readonly but the value returned␣
↪depends
                                         # on other arch-states
        warl:                            # field is warl type. Refer to WARL section
          dependency_fields: [list]
          legal: [list of warl-string]
          wr_illegal: [list of warl-string]
  rv64:                                  # this node and its subsequent fields can␣
↪exist
                                         # if [M/S/U]XL value can be 2
    accessible: <boolean>                # indicates if this register exists in rv64␣
↪mode
```

(continues on next page)

(continued from previous page)

```
                                        # or not. Same definition as for rv32 node.
  rv128:                                # this node and its subsequent fields can␣
↪exist if
                                        # [M/S/U]XL value can be 3
    accessible: <boolean>               # indicates if this register exists in␣
↪rv128 mode
                                        # or not. Same definition as for rv32 node.
```

4.2.2 CSRs without sub-fields

```
<name>:                                 # name of the csr
  description: <text>                   # textual description of the csr
  address: <hex>                        # address of the CSR
  priv_mode: <D/M/H/S/U>                # privilege mode that owns the register
  reset-val: <hex>                      # Reset value of the register. This an␣
↪accumulation
                                        # of the all reset values of the sub-fields
  rv32:                                 # this node and its subsequent fields can␣
↪exist
                                        # if [M/S/U]XL value can be 1
    accessible: <boolean>               # indicates if the csr is accessible in rv32␣
↪mode or not.
                                        # When False, all fields below will be␣
↪trimmed off
                                        # in the checked yaml. False also indicates␣
↪that
                                        # access-exception should be generated
    fields: []                          # This should be empty always.
    shadow: <csr-name>::<register>      # which this register shadows,'none'␣
↪indicates that
                                        # this register does not shadow anything.
    msb: <int>                          # msb index of the csr. max: 31, min:0
    lsb: <int>                          # lsb index of the csr. max: 31, min:0
    type:                               # type of field. Can be only one of the␣
↪following
      wlrl: [list of value-descriptors] # field is wlrl and the set of legal values.
      ro_constant: <hex>                # field is readonly and will return the same␣
↪value.
      ro_variable: True                 # field is readonly but the value returned␣
↪depends
                                        # on other arch-states
      warl:                             # field is warl type. Refer to WARL section
        dependency_fields: [list]
        legal: [list of warl-string]
        wr_illegal: [list of warl-string]
  rv64:                                 # this node and its subsequent fields can␣
↪exist
                                        # if [M/S/U]XL value can be 2
    accessible: <boolean>               # indicates if this register exists in rv64␣
↪mode
                                        # or not. Same definition as for rv32 node.
  rv128:                                # this node and its subsequent fields can␣
↪exist if
                                        # [M/S/U]XL value can be 3
    accessible: <boolean>               # indicates if this register exists in rv128␣
↪mode
```

(continues on next page)

(continued from previous page)

4.2.3 Constraints

Each CSR undergoes the following checks:

1. All implemented fields at the csr-level, if set to True, are checked if they comply with the supported_xlen field of the ISA yaml.
2. The reset-val is checked against compliance with the type field specified by the user. All unimplemented fields are considered to be hardwired to 0.

For each of the above templates the following fields for all standard CSRs defined by the spec are frozen and **CANNOT** be modified by the user.

- description
- address
- priv_mode
- fields
- shadow
- msb
- lsb
- The type field for certain CSRs (like readonly) is also constrained.
- fields names also cannot be modified for standard CSRs

Only the following fields can be modified by the user:

- reset-value
- type
- implemented

4.2.4 Example

Following is an example of how a user can define the mtvec csr in the input ISA YAML for a 32-bit core:

```
mtvec:
reset-val: 0x80010000
rv32:
  accessible: true
  base:
    implemented: true
    type:
      warl:
        dependency_fields: [mtvec::mode]
        legal:
          - "mode[1:0] in [0] -> base[29:0] in [0x20000000, 0x20004000]"
↪# can take only 2 fixed values in direct mode.
          - "mode[1:0] in [1] -> base[29:6] in [0x000000:0xF00000] base[5:0] in [0x00]
↪" # 256 byte aligned values only in vectored mode.
        wr_illegal:
```

(continues on next page)

(continued from previous page)

```
          - "mode[1:0] in [0] -> Unchanged"
          - "mode[1:0] in [1] and wr_val in [0x2000000:0x4000000] -> 0x2000000"
          - "mode[1:0] in [1] and wr_val in [0x4000001:0x3FFFFFFF] -> Unchanged"
  mode:
    implemented: true
    type:
      warl:
        dependency_fields: []
        legal:
          - "mode[1:0] in [0x0:0x1] # Range of 0 to 1 (inclusive)"
        wr_illegal:
          - "Unchanged"
```

The following is what the riscv-config will output after performing relevant checks on the above user-input:

```
mtvec:
  description: MXLEN-bit read/write register that holds trap vector configuration.
  address: 773
  priv_mode: M
  reset-val: 0x80010000
  rv32:
    accessible: true
    base:
      implemented: true
      type:
        warl:
          dependency_fields: [mtvec::mode]
          legal:
          - 'mode[1:0] in [0] -> base[29:0] in [0x20000000, 0x20004000]'
↪ # can take only 2 fixed values in direct mode.
          - 'mode[1:0] in [1] -> base[29:6] in [0x000000:0xF00000] base[5:0] in [0x00]
↪'   # 256 byte aligned values only in vectored mode.
          wr_illegal:
          - 'mode[1:0] in [0] -> Unchanged'
          - 'mode[1:0] in [1] and wr_val in [0x2000000:0x4000000] -> 0x2000000'
          - 'mode[1:0] in [1] and wr_val in [0x4000001:0x3FFFFFFF] -> Unchanged'
      description: Vector base address.
      shadow: none
      msb: 31
      lsb: 2
    mode:
      implemented: true
      type:
        warl:
          dependency_fields: []
          legal:
          - 'mode[1:0] in [0x0:0x1] # Range of 0 to 1 (inclusive)'
          wr_illegal:
          - Unchanged

      description: Vector mode.
      shadow: none
      msb: 1
      lsb: 0
    fields:
    - mode
    - base
```

(continues on next page)

(continued from previous page)

```
  rv64:
    accessible: false
```

4.3 WARL field Definition

Since the RISC-V privilege spec indicates several CSRs and sub-fields of CSRs to be WARL (Write-Any-Read-Legal), it is necessary to provide a common scheme of representation which can precisely define the functionality of any such WARL field/register.

4.3.1 Value Descriptors

This is a list which describes a set of values. Each entry can either represent a single distinct value or a range of values.

- **distinct-values** - This specifies that only the particular value should be added to the set.

  ```
  val
  ```

- **range** - This specifies that all the values greater than or equal to lower and less than or equal to upper is to be included in the set.

  ```
  lower:upper
  ```

Example:

```
# To represent the set {0, 1, 2, 3, 4, 5}
  [0:5]

# To represent the set {5, 10, 31}
  [5, 10, 31]

# To represent the set {2, 3, 4, 5, 10}
  [2:5, 10]
```

4.3.2 WARL Node definition

A WARL csr/field has the following skeleton in the riscv-config:

```
WARL:
   dependency_fields: [list]
   legal: [list of warl-string]
   wr_illegal: [list of warl-string]
```

- **dependency_fields** : A list of other csrs/fields whose values define the set of legal values the csr/field under question can take. We use `::` as a hierarchy separator. This field can be empty as well indicating no other state affects this csr/field. The fields within csrs can be specified as follows:

  ```
  - dependency_fields: [mtvec::mode]
  - dependency_fields: [misa::mxl, mepc]
  ```

- **legal** : This field takes in a list of strings which define the WARL functions. Each string needs to adhere to the following syntax:

```
- [dependency-vals] -> field-name[index-hi:index-lo] in [legal-values]
- [dependency-vals] -> field-name[index-hi:index-lo] bitmask [mask, fixedval]

# if no dependency_fields exists then following is also allowed:

field-name[index-hi:index-lo] bitmask [mask, fixedval]
```

In short it means that under certain values of the dependency_fields the warl-field can take only the legal values defined by either *[legal-values]* or by the *bitmask* function.

- **dependency-vals** : A comma separated list of value-descriptors indicating the values the corresponding fields in the dependency_fields take.
- **->**: represents "imply".
- **field-name**: should be the same as the csr/field-name for which this WARL function is being described.
- **index-**: These are unsigned integers not exceeding the size of field. Sometimes it easier to define the WARL function by splitting the fields. Thus the following is also a legal form:

```
[dependency-vals] -> field-name[index-hi:index-lo] in [legal-values1] & field-
↪name[index-lo-1:0] in [legal-values2]
```

- **in**: key-word indicating that *field-name[index-hi:index-lo]* should takes values defined within *[legal_values]*.
- **bitmask**: keyword indicating that the legal values are defined using a mask and fixedval variables. The fixedval variable defines the default value of the masked bits.
- **legal-values**: a list of value-descriptors indicating the set of legal values *field-name[index-hi:index-lo]* can take.

  Restrictions:

  1. No legal value must exceed the maximum value which can be supported (based on the width of the field).
  2. Functions should be exhaustive with respect to every possible combination of the dependency values.
  3. within a string for *legal* all bits of the csr/field should be covered. No bits must be left undefined.
  4. A legal string should not be a combination ranges split into parts or a simple bitmask function for the entire field. mixing bitmask and ranges it allowed. The following example is an invalid spec:

```
[0] -> field[31:6] in [0x10000000: 0x3FFFFFF] & field[5:0] bitmask [0x30,␣
↪0x0F]
```

- **wr_illegal** : This field takes in a list of strings which define the next legal value of the field when an illegal value is written. Each string needs to adhere to the following syntax:

```
[dependency-vals] wr_val in [illegal-values] -> update_mode
                    OR
[dependency-vals] -> update_mode
```

In short this means that under certain values of the dependency_fields when an illegal write happens (either defined by the wr_val or for all illegal values) the next legal value is defined by the *update_mode*.

- **dependency-vals** : A comma separated list of value-descriptors indicating the values the corresponding fields in the dependency_fields take.
- **wr_val**: key-word indicating the illegal write-value

- **in**: same meaning as the before.
- **illegal-values**: a list of value-descriptors indicating the set of illegal values for the csr/field under question.
- **update_mode** : This field dictates what the next legal read value is when an illegal write happens:
  - **unchanged**: The value remains unchanged to the current legal value.
  - **<val>**: A single value can also be specified
  - **nextup**: ceiling(*wr_val*) i.e. the next larger or the largest element of the legal list
  - **nextdown**: floor(*wr_val*) i.e. the next smallest or the smallest element of the legal list
  - **nearup**: celing(*wr_val*) i.e. the closest element in the list, with the larger element being chosen in case of a tie.
  - **neardown**: floor(*wr_val*) i.e. the closes element in the list, with the smaller element being chosen in case of a tie
  - **max**: maximum of all legal values
  - **min**: minimum of all legal values
  - **addr**:

```
if ( val < base || val > bound)
    return Flip-MSB of field
```

Restrictions:

1. wr_illegal will not exists for a legal list defined as a bitmask.

Example:

```
# When base of mtvec depends on the mode field.
WARL:
  dependency_fields: [mtvec::mode]
  legal:
    - "[0] -> base[29:0] in [0x20000000, 0x20004000]"  # can take only 2 fixed values␣
↪when mode==0.
    - "[1] -> base[29:6] in [0x000000:0xF00000] base[5:0] in [0x00]" # 256 byte␣
↪aligned when mode==1
  wr_illegal:
    - "[0] -> unchanged"
    - "[1] wr_val in [0x2000000:0x4000000] -> 0x2000000" # predefined value if write␣
↪value is
    - "[1] wr_val in [0x4000001:0x3FFFFFF] -> unchanged"

# When base of mtvec depends on the mode field. Using bitmask instead of range
WARL:
  dependency_fields: [mtvec::mode]
  legal:
    - "[0] -> base[29:0] in [0x20000000, 0x20004000]"  # can take only 2 fixed values␣
↪when mode==0.
    - "[1] -> base[29:0] bitmask [0x3FFFFFC0, 0x00000000]" # 256 byte aligned when␣
↪mode==1
  wr_illegal:
    - "[0] -> unchanged" # no illegal for bitmask defined legal strings.


# no dependencies. Mode field of mtvec can take only 2 legal values using range-
↪descriptor
```

(continues on next page)

(continued from previous page)

```
WARL:
  dependency_fields:
  legal:
    - "mode[1:0] in [0x0:0x1] # Range of 0 to 1 (inclusive)"
  wr_illegal:
    - "0x00"

# no dependencies. using single-value-descriptors
WARL:
  dependency_fields:
  legal:
    - "mode[1:0] in [0x0,0x1] # Range of 0 to 1 (inclusive)"
  wr_illegal:
    - "0x00"
```

4.4 Platform YAML Spec

This section describes each node of the PLATFORM-YAML. For each node, we have identified the fields required from the user and also the various constraints involved.

4.4.1 reset

Description: Stores the value for the reset vector. It can either be a label or an address.

- label: A string field equal to the label in the assembly code
- address: A value equal to the absolute address where the vector is present

Examples:

```
reset:
  label: reset_vector
reset:
  label: 0x80000000
```

4.4.2 nmi

Description: Stores the value for the nmi vector. It can either be a label or an address.

- label: A string field equal to the label in the assembly code.
- address: A value equal to the absolute address where the vector is present.

Examples:

```
nmi:
   label: nmi_vector

nmi:
   address: 0x8000000
```

4.4.3 mtime

Description: Stores the fields for memory mapped *mtime* register.

- implemented: A boolean field indicating that the register has been implemented.
- address: A value equal to the physical address at which the register is present.

Examples:

```
mtime:
   implemented: True
   address: 0x458
```

Constraints:

- None

4.4.4 mtimecmp

Description: Stores the fields for memory mapped *mtimecmp* register.

- implemented: A boolean field indicating that the register has been implemented.
- address: A value equal to the physical address at which the register is present.

Examples:

```
mtimecmp:
   implemented: True
   address: 0x458
```

Constraints:

- None

4.4.5 mcause_non_standard

Description: Stores the fields for the *mcause* register.

- implemented: A boolean field indicating that the register has been implemented.
- values: The list of exception values greater than 16 as assumed by the platform as integers.

Examples:

```
mcause_non_standard:
   implemented: True
   value: [16,17,20]
```

Constraints:

- None

4.4.6 mtval_condition_writes

Description: Stores the fields for *mtval_condition_writes* register.

- implemented: A Boolean value indicating whether the register is implemented.

- behaviour: A dictionary type to specify which of the exceptions modify the mtval_condition_writes reg
  - e0: A string type describing the behaviour of exception 0.
  - e1: A string type describing the behaviour of exception 1.
  - e2: A string type describing the behaviour of exception 2.
  - e3: A string type describing the behaviour of exception 3.
  - e4: A string type describing the behaviour of exception 4.
  - e5: A string type describing the behaviour of exception 5.
  - e6: A string type describing the behaviour of exception 6.
  - e7: A string type describing the behaviour of exception 7.
  - e8: A string type describing the behaviour of exception 8.
  - e9: A string type describing the behaviour of exception 9.
  - e10: A string type describing the behaviour of exception 10.
  - e11: A string type describing the behaviour of exception 11.
  - e12: A string type describing the behaviour of exception 12.
  - e13: A string type describing the behaviour of exception 13.
  - e15: A string type describing the behaviour of exception 15.

Examples:

```
TBD: Provide a concrete use-case for the above.
```

Constraints:

- None

4.4.7 scause_non_standard

Description: Stores the fields for the *scause* register.

- implemented: A boolean field indicating that the register has been implemented.
- values: The list of exception values greater than 16 as assumed by the platform as integers.

Examples:

```
scause_non_standard:
   implemented: True
   value: [16,17,20]
```

Constraints:

- None

4.4.8 stval_condition_writes

Description: Stores the fields for *stval_condition_writes* register.

- implemented: A Boolean value indicating whether the field is implemented.

- behaviour: A dictionary type to specify which of the exceptions modify the stval_condition_writes reg
  - e0: A string type describing the behaviour of exception 0.
  - e1: A string type describing the behaviour of exception 1.
  - e2: A string type describing the behaviour of exception 2.
  - e3: A string type describing the behaviour of exception 3.
  - e4: A string type describing the behaviour of exception 4.
  - e5: A string type describing the behaviour of exception 5.
  - e6: A string type describing the behaviour of exception 6.
  - e7: A string type describing the behaviour of exception 7.
  - e8: A string type describing the behaviour of exception 8.
  - e9: A string type describing the behaviour of exception 9.
  - e10: A string type describing the behaviour of exception 10.
  - e11: A string type describing the behaviour of exception 11.
  - e12: A string type describing the behaviour of exception 12.
  - e13: A string type describing the behaviour of exception 13.
  - e15: A string type describing the behaviour of exception 15.

Examples:

```
TBD: Provide a concrete use-case for the above.
```

Constraints:

- None

CHAPTER 5

Code Documentation

riscv_config.checker.**add_def_setters**(*schema_yaml*)

Function to set the default setters for various fields in the schema

riscv_config.checker.**check_custom_specs**(*custom_spec*, *work_dir*, *logging=False*, *no_anchors=False*)

Function to perform ensure that the isa and platform specifications confirm to their schemas. The `Cerberus` module is used to validate that the specifications confirm to their respective schemas.

Parameters

- **isa_spec** (*str*) – The path to the DUT isa specification yaml file.
- **logging** (*bool*) – A boolean to indicate whether log is to be printed.

Raises **ValidationError** – It is raised when the specifications violate the schema rules. It also contains the specific errors in each of the fields.

Returns A tuple with the first entry being the absolute path to normalized isa file and the second being the absolute path to the platform spec file.

riscv_config.checker.**check_isa_specs**(*isa_spec*, *work_dir*, *logging=False*, *no_anchors=False*)

Function to perform ensure that the isa and platform specifications confirm to their schemas. The `Cerberus` module is used to validate that the specifications confirm to their respective schemas.

Parameters

- **isa_spec** (*str*) – The path to the DUT isa specification yaml file.
- **logging** (*bool*) – A boolean to indicate whether log is to be printed.

Raises **ValidationError** – It is raised when the specifications violate the schema rules. It also contains the specific errors in each of the fields.

Returns A tuple with the first entry being the absolute path to normalized isa file and the second being the absolute path to the platform spec file.

riscv_config.checker.**check_mhpm**(*spec*, *logging=False*)

Check if the mhpmcounters and corresponding mhpmevents are implemented and of the same size as the source

riscv_config.checker.**check_pmp**(*spec*, *logging=False*)
Check if the mhpmcounters and corresponding mhpmevents are implemented and of the same size as the source

riscv_config.checker.**check_reset_fill_fields**(*spec*, *logging=False*)
The check_reset_fill_fields function fills the field node with the names of the sub-fields of the register and then checks whether the reset-value of the register is a legal value. To do so, it iterates over all the subfields and extracts the corresponding field value from the reset-value. Then it checks the legality of the value according to the given field description. If the fields is implemented i.e accessible in both 64 bit and 32 bit modes, the 64 bit mode is given preference.

riscv_config.checker.**check_shadows**(*spec*, *logging=False*)
Check if the shadowed fields are implemented and of the same size as the source

riscv_config.checker.**delegset**()
Function to set "implemented" value for mideleg regisrer.

riscv_config.checker.**groupc**(*test_list*)
Generator function to squash consecutive numbers for wpri bits.

riscv_config.checker.**nregset**()
Function to set defaults based on presence of 'N' extension.

riscv_config.checker.**nuset**()
Function to check and set defaults for all fields which are dependent on the presence of 'U' extension and 'N' extension.

riscv_config.checker.**sregset**()
Function to set defaults based on presence of 'S' extension.

riscv_config.checker.**sregseth**()
Function to set defaults based on presence of 'S' extension.

riscv_config.checker.**sset**()
Function to set defaults based on presence of 'S' extension.

riscv_config.checker.**trim**(*foo*)
Function to trim the dictionary. Any node with implemented field set to false is trimmed of all the other nodes.

Parameters **foo** (*dict*) – The dictionary to be trimmed.

Returns The trimmed dictionary.

riscv_config.checker.**twset**()
Function to check and set value for tw field in misa.

riscv_config.checker.**uregset**()
Function to set defaults based on presence of 'U' extension.

riscv_config.checker.**uregseth**()
Function to set defaults based on presence of 'U' extension.

riscv_config.checker.**uset**()
Function to set defaults based on presence of 'U' extension.

class riscv_config.schemaValidator.**schemaValidator**(**args*, ***kwargs*)
Custom validator for schema having the custom rules necessary for implementation and checks.

__init__(**args*, ***kwargs*)
The arguments will be treated as with this signature:

__init__(self, schema=None, ignore_none_values=False, allow_unknown=False, require_all=False, purge_unknown=False, purge_readonly=False, error_handler=errors.BasicErrorHandler)

_check_with_cannot_be_false_rv32 (*field*, *value*)
Functions ensures that the field cannot be False in rv32 mode

_check_with_cannot_be_false_rv64 (*field*, *value*)
Functions ensures that the field cannot be False in rv64 mode

_check_with_capture_isa_specifics (*field*, *value*)
Function to extract and store ISA specific information(such as xlen,user spec version and extensions present) and check whether the dependencies in ISA extensions are satisfied.

_check_with_max_length (*field*, *value*)
Function to check whether the given value is less than the maximum value that can be stored(2^xlen-1).

_check_with_xcause_check (*field*, *value*)
Function to verify the inputs for mcause.

_check_with_xtveccheck (*field*, *value*)
Function to check whether the inputs in range type in mtvec are valid.

5.1 Utils

class riscv_config.utils.**ColoredFormatter**(**args*, ***kwargs*)
Class to create a log output which is colored based on level.

__init__(**args*, ***kwargs*)
Initialize the formatter with specified format strings.

Initialize the formatter either with the specified format string, or a default as described above. Allow for specialized date formatting with the optional datefmt argument. If datefmt is omitted, you get an ISO8601-like (or RFC 3339-like) format.

Use a style parameter of '%', '{' or '$' to specify that you want to use one of %-formatting, `str.format()` ({}) formatting or `string.Template` formatting in your format string.

Changed in version 3.2: Added the `style` parameter.

format(*record*)
Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

class riscv_config.utils.**SortingHelpFormatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

riscv_config.utils.**setup_logging**(*log_level*)
Setup logging

Verbosity decided on user input

Parameters **log_level** (*str*) – User defined log level

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index

Symbols

A

C

D

F

G

N

R

S

T

U