RISC-V Configuration Validator Documentation

Release [3.0.0]

InCore Semiconductors Pvt. Ltd.

Aug 18, 2022

CONTENTS

CHAPTER

ONE

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

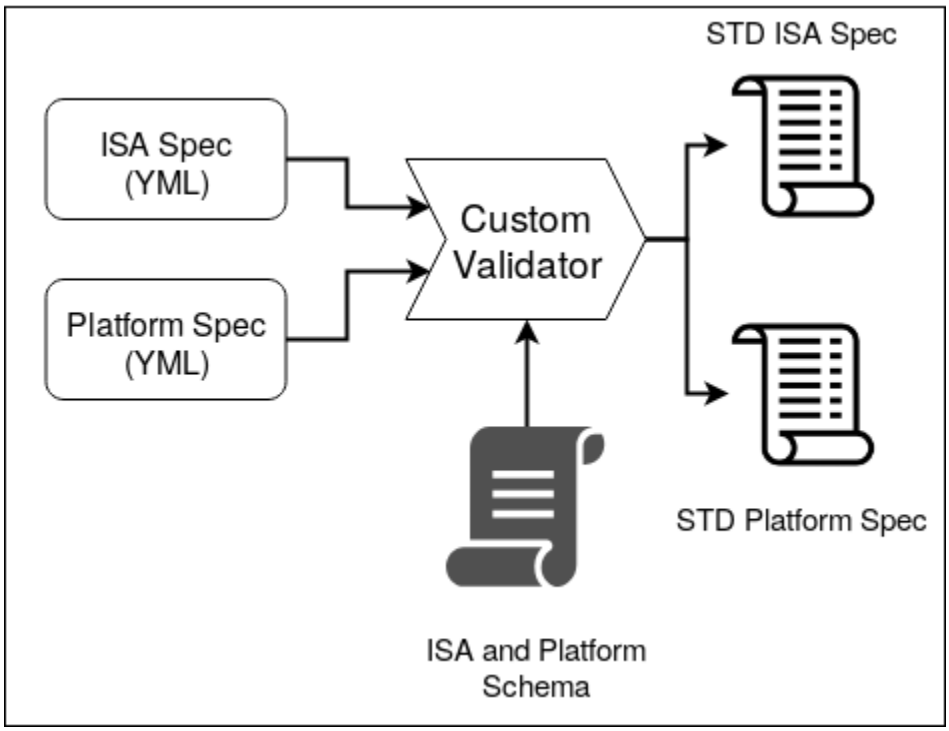
RISCV-Config [Repository]

CHAPTER

TWO

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

THREE

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
[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

FOUR

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

4.1.10 custom_exceptions

Description: list of custom exceptions implemented

Examples:

```
custom_exceptions:
  - cause_val: 25
    cause_name: mycustom
  - cause_val: 26
    cause_name: mycustom2
```

Constraints:

- None

4.1.11 custom_interrupts

Description: list of custom interrupts implemented

Examples:

```
custom_interrupts:
  - cause_val: 25
    cause_name: mycustom
  - cause_val: 26
    cause_name: mycustom2
```

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␣
```

(continues on next page)

(continued from previous page)

```
↪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
                                        # 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
```

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:
          - "mode[1:0] in [0] -> Unchanged"
          - "mode[1:0] in [1] && writeval in [0x2000000:0x4000000] -> 0x2000000"
```

(continues on next page)

(continued from previous page)

```
          - "mode[1:0] in [1] && writeval 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, writeval]
          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] && writeval in [0x2000000:0x4000000] -> 0x2000000'
          - 'mode[1:0] in [1] && writeval 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
```

(continues on next page)

(continued from previous page)

```
    fields:
    - mode
    - base
  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

Value descriptors are standard syntaxes that are used to define values in any part of the WARL string. The 2 basic descriptors are : distinct-values and range-values as described below:

- **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
```

For any variable in the WARL string, the values can an amalgamation of distinct-values and/or range-values. They are typically captured in a list as shown in the below examples:

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, 11, 12, 13, 50}
  [2:5, 10:13, 50]
```

4.3.2 WARL Node definition

A typical WARL node (used for a WARL csr or subfield) has the following skeleton in the riscv-config:

```
warl:
   dependency_fields: [list of csrs/subfields that legal values depend on]
   legal: [list of strings adhering to the warl-syntax for legal assignments]
   wr_illegal: [list of strings ahdering to the warl-syntax for illegal assignments]
```

- **dependency_fields** : This is a list of csrs/subfields whose values affect the legal values of the csr under question. `::` is used as a hierarchy separator to indicate subfields. This list can be empty to indicate that the csr under question is not affected by any other architectural state. The ordering of the csr/subfields has no consequence. Examples of the list are provided below:

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

  The following keywords are reserved and can be used accordingly in the dependency_fields list:

  - `writeval` : to represent dependency on the current value being written to the csr/subfield
  - `currval` : to represent dependency on the value of the csr/subfield before performing the write operation

  Restrictions imposed: The following restrictions are imposed on the elements of the list:

  1. The csrs/subfields mentioned in the list must have their accessible/implemented fields set to True in the isa yaml.

- **legal** : This field is a list of strings which define the warl functions of the csr/subfield. Each string needs to adhere to the following warl-syntax:

```
dependency_string -> legal_value_string
```

  The `dependency_string` substring is basically a string defining a boolean condition based on the dependent csrs (those listed in the `dependency_fields`). Only when the boolean condition is satisfied, the corresponding warl function defined in `legal_value_string` substring is evaluated. A write only occurs when the evaluation of the `legal_value_string` also is True. The symbol `->` is used to denote *implies* and is primarily used to split the string in to the above two substrings. If none of the entries in the list evaluate to True, then the current write value is considered illegal and the actions defined by the *wr_illegal* field is carried out.

  The substrings `dependency_string ->` is optional. If the `dependency_fields` list is empty, then the substring `dependency_string ->` must be omitted from the warl string.

  The `dependency_string` and the `legal_value_string` both follow the same legal syntax:

```
<variable-name>[<hi-index>:<lo-index>] <op> <value-descriptors>
```

  The `variable-name` field can be the name a csr or a subfield (without the hierarchical delimiter `::`). Within the `dependency_string` substring the variable names can only be those listed in the `dependency_fields` list. In the `legal_value_string` substring however, the `variable-name` should be either *writeval* or the name the csr or the subfield (without the hierarchical delimiter `::`) that the warl node belongs to.

  The indices fields `hi-index` and `lo-index` are used to indicate the bit range of the variable that being looked-up or modified. The basic constraints are that `hi-index` must be greater than the `lo-index`. If only a single-bit is being looked-up/assigned, then `:lo-index` can be skipped. This definition applies to both the `dependency_string` and the `legal_value_string`.

  The `op` field in the `dependency_string` substring can be one of `in` or `not in` to indicate that the variable takes the values defined in the `value-descriptors` field or does not take those values respectively. In addition to the above operators, the `legal_value_string` can include one more operator : `bitmask`. When using the `bitmask` operator the value-descriptors have to be a list of two distinct-values as follows:

```
csr_name[hi:lo] bitmask [mask, fixedval]
```

  Both the `mask` and `fixedval` fields are integers. All bits set in the mask indicates writable bits of the variable. All bits bits cleared in the mask indicate bits with a constant value which is derived from the corresponding bit in the fixedval field.

Since the dependency_string is supposed to represent a boolean condition, it also has the flexibility to use basic boolean operators like && and || around the above legal syntax. Examples are provided below:

```
(csrA[2:0] in [0, 1]) && (csrB[5:0] in [0:25] || csrB[5:0] in [31,30]) ->
```

Restrictions imposed: The following restrictions are imposed on the above substrings:

1. No element of the value-descriptors must exceed the maximum value which can be supported by the indices of the csr/subfield.
2. The csrs/subfields used in the dependency_string must be in those listed in the dependency_fields list.
3. Valid operators in the dependency_string substring are in and not in.
4. Valid operators in the legal_value_string substring are in, not in and bitmask
5. Within the legal_value_string substrings the legal values of all bits of the csr/subfield must be specified. No bits must be left undefined.
6. If the dependency_fields is empty, then only one legal string must be defined in this list.
7. The first combination of the dependency_string and legal_value_string to evaluate to True, starting from the top of the list is given highest priority to define the next legal value of the csr/subfield.
8. The reset-value of the csr/subfield must cause atleast one of the legal strings in the list to evaluate to True.

Assumptions

1. Since the list of all dependency_string substrings is not required to be exhaustively defined by the user, if none of the dependency_strings in the list evaluate to true, then the current write operation should be treated as an illegal write operation, and the action defined by the wr_illegal node must be carried out.
2. If one of the dependent csrs/subfield defined in the dependency_fields is not used in the dependency_strings, then it implictly assumed that, the variable does not affect the legal value for that string

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

```
dependency_string -> update_mode
```

The dependency_string follows the same rules, assumptions and restrictions described above. When the dependency_string evaluates to True the update_mode substring defines the next legal value of the csr/subfield. The supported values of the update_mode string are :

  - **Unchanged**: The value remains unchanged to the current legal value held in the csr/subfield.
  - **<val>**: A single value can also be specified
  - **Nextup**: ceiling(*writeval*) i.e. the next larger or the largest element of the legal list
  - **Nextdown**: floor(*writeval*) i.e. the next smallest or the smallest element of the legal list
  - **Nearup**: celing(*writeval*) i.e. the closest element in the list, with the larger element being chosen in case of a tie.
  - **Neardown**: floor(*writeval*) 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
```

4.3.3 Examples

```
# When base of mtvec depends on the mode field.
WARL:
  dependency_fields: [mtvec::mode]
  legal:
    - "mode[1:0] in [0] -> base[29:0] in [0x20000000, 0x20004000]"  # can take only 2␣
↪fixed values when mode==0.
    - "mode[1:0] in [1] -> base[29:6] in [0x000000:0xF00000] base[5:0] in [0x00]" # 256␣
↪byte aligned when mode==1
  wr_illegal:
    - "mode[1:0] in [0] -> unchanged"
    - "mode[1:0] in [1] && writeval in [0x2000000:0x4000000] -> 0x2000000" # predefined␣
↪value if write value is
    - "mode[1:0] in [1] && writeval in [0x4000001:0x3FFFFFFF] -> unchanged"

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


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

4.4.6 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

4.4.7 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

4.4.8 zicbo_cache_block_sz

Description: byte size of the cache block

Examples:

```
zicbo_cache_block_sz :
  implemented: true
  zicbom_sz: 64
  zicboz_sz: 64
```

Constraints:

- None

4.5 Debug YAML Spec

4.5.1 supported_xlen

Description: list of supported xlen on the target

Examples:

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

Constraints:

- None

4.5.2 Debug_Spec_Version

Description: Version number of Debug specification as string. Please enclose the version in “” to avoid type mismatches.

Examples:

```
Debug_Spec_Version: "1.0.0"
Debug_Spec_Version: "0.13.2"
```

Constraints:

- should be a valid version later than 1.0.0

4.5.3 debug_mode

Description: Boolean value indicating if the debug instructions are accessible.

Examples:

```
debug_mode: False
```

4.5.4 parking_loop

Description: Integer value indicating the address of the debug parking loop

Examples:

```
parking_loop: 0x800
```

CHAPTER

FIVE

ADDING SUPPORT FOR NEW EXTENSIONS

Adding support for a new ISA extension or an adjoining spec to RISCV-CONFIG could entail one or more of the following updates:

1. Updating the ISA string and its constraints to recognize valid configurations of the new extension
2. Updating the schema_isa.yaml with new CSRs defined by the new ISA extension
3. Adding new schemas and a new cli argument for supporting adjoining RISC-V specs like debug, trace, etc.

This chapter will descrive how one can go about RISC-V achieving the above tasks.

5.1 Updates to the ISA string

5.1.1 Modifications in Schema_isa.yaml

As shown in the example below, any new extensions and sub extensions have to be enabled by adding them in the regex expression of the ISA node. Following is an instance of the node for reference:

```
ISA: { type: string, required: true, check_with: capture_isa_specifics,
       regex: "^
↪RV(32|64|128)[IE]+[ABCDEFGIJKLMNPQSTUVX]*(Zicsr|Zifencei|Zihintpause|Zam|Ztso|Zkne|Zknd|Zknh|Zkse|Zksh
↪{,1}(_Zicsr){,1}(_Zifencei){,1}(_Zihintpause){,1}(_Zam){,1}(_Ztso){,1}(_Zkne){,1}(_
↪Zknd){,1}(_Zknh){,1}(_Zkse){,1}(_Zksh){,1}(_Zkg){,1}(_Zkb){,1}(_Zkr){,1}(_Zks){,1}(_
↪Zkn){,1}(_Zbc){,1}(_Zbb){,1}(_Zbp){,1}(_Zbm){,1}(_Zbe){,1}(_Zbf){,1}$" }
```

Note: If you are adding a new Z extension, note that it must be added in 2 places in the regex. The first immediately after the standard extension in the format *|Zgargle*. This is to support that fact that the new Z extension could start immediately after the standard extensions which an underscore. The second will be after the first set of Z extensions in the format *{,1}(_Zgargle)*.

5.1.2 Adding constraints in the SchemaValidator.py file

While adding a new extension, there can be certain legal and illegal combinations which cannot be easily expressed using the regex above. To facilitate defining illegal conditions, riscv-config allows user to define specific checks via custom python functions.

For the ISA field riscv-config uses the _check_with_capture_isa_specifics function to return an error if an illegal combination of the extesions (or subextension) is found.

Following is an example of the constraints imposed by the K extesion and its subset. Within the K (Crypto-Scalar extension), subextensions Zkn, Zks, K are supersets of other Zk* abbreviations. Thus, if the superset extension exists in the ISA, none of the corresponding subset ZK* should be present in the ISA at the same time.

Constraints used here :

1.If Zkn is present , its subset extensions Zkne, Zknh, Zknd, Zkg and Zkb cannot be present in the ISA string.

2.If Zks is present , its subset extensions Zkse, Zksh, Zkg and Zkb cannot be present in the ISA string.

3.If K extension is present , its subset extensions Zkn, Zkr, Zkne, Zknh, Zknd, Zkg and Zkb cannot be present in the ISA string.

4. If **B extension** Zbp is present , its subset extensions Zkb cannot be present in the ISA string. Cross-checking across two different extensions can also be done. Zkb contains instructions from other subextensions in B extension like Zbm, Zbe, Zbf and Zbb , but unlike Zbp is not a proper superset.

5. If **B extension** Zbc is present , its subset extensions Zkg cannot be present in the ISA string.

```
(...)
if 'Zkg' in extension_list and 'Zbc' in extension_list:
    self._error(field, "Zkg being a proper subset of Zbc (from B extension) should be␣
↪ommitted from the ISA string")
if 'Zkb' in extension_list and 'Zbp' in extension_list :
    self._error(field, "Zkb being a proper subset of Zbp (from B extension) should be␣
↪ommitted from the ISA string")
if 'Zks' in extension_list and ( set(['Zkse', 'Zksh','Zkg','Zkb']) & set(extension_list)␣
↪):
    self._error(field, "Zks is a superset of Zkse, Zksh, Zkg and Zkb. In presence of Zks␣
↪the subsets must be ignored in the ISA string.")
if 'Zkn' in extension_list and ( set(['Zkne','Zknd','Zknh','Zkg','Zkb']) & set(extension_
↪list) ):
    self._error(field, "Zkn is a superset of Zkne, Zknd, Zknh, Zkg and Zkb, In presence␣
↪of Zkn the subsets must be ignored in the ISA string")
if 'K' in extension_list and ( set(['Zkn','Zkr','Zkne','Zknd','Zknh','Zkg','Zkb']) &␣
↪set(extension_list) ) :
    self._error(field, "K is a superset of Zkn and Zkr , In presence of K the subsets␣
↪must be ignored in the ISA string")
(...)
```

5.2 Assing new CSR definitions

There are two parts to addition of a new csr definition to riscv-config

5.2.1 Addition of new csrs to schema

The first step is to add the schema of the new csr in the schema_isa.yaml file. Following is an example of how the *stval* csr of the "S" extension is a added to the schema.

Note: for each csr the user is free to define and re-use existing check_with functions to impose further legal conditions. In the example below, the stval should only be implemented if the "S" extension in the ISA field is set. This is checked using the *s_check* function. Any new check_with functions must be defined in the schemaValidator.py file

```
stval:
   type: dict
   schema:
     description:
       type: string
       default: The stval is a warl register that holds the address of the instruction
         which caused the exception.
     address: {type: integer, default: 0x143, allowed: [0x143]}
     priv_mode: {type: string, default: S, allowed: [S]}
     reset-val:
       type: integer
       default: 0
       check_with: max_length
     rv32:
       type: dict
       check_with: s_check
       schema:
         fields: {type: list, default: []}
         shadow: {type: string, default: , nullable: True}
         msb: {type: integer, default: 31, allowed: [31]}
         lsb: {type: integer, default: 0, allowed: [0]}
         type:
           type: dict
           check_with: wr_illegal
           schema: { warl: *ref_warl }
           default:
             warl:
               dependency_fields: []
               legal:
               - stval[31:0] in [0x00000000:0xFFFFFFFF]
               wr_illegal:
               - unchanged

         accessible:
           type: boolean
           default: true
           check_with: rv32_check
```

(continues on next page)

(continued from previous page)

```
      default: {accessible: false}
    rv64:
      type: dict
      check_with: s_check
      schema:
        fields: {type: list, default: []}
        shadow: {type: string, default: , nullable: True}
        msb: {type: integer, default: 63, allowed: [63]}
        lsb: {type: integer, default: 0, allowed: [0]}
        type:
          type: dict
          check_with: wr_illegal
          schema: { warl: *ref_warl }
          default:
            warl:
              dependency_fields: []
              legal:
              - stval[63:0] in [0x00000000:0xFFFFFFFFFFFFFFFF]
              wr_illegal:
              - unchanged

        accessible:
          default: true
          check_with: rv64_check
      default: {accessible: false}
```

5.2.2 Adding default setters in checker.py

The next step in adding a new csr definition if to add its default values. This is done in checker.py

Example of adding a default setter for *stval* is show below. This code basically makes the stval csr accessible by default when the "S" extension is enabled in the ISA string.

```
schema_yaml['stval']['default_setter'] = sregsetter
```

```
def sregset():
 '''Function to set defaults based on presence of 'S' extension.'''
 global inp_yaml
 temp = {'rv32': {'accessible': False}, 'rv64': {'accessible': False}}
 if 'S' in inp_yaml['ISA']:
   if 32 in inp_yaml['supported_xlen']:
     temp['rv32']['accessible'] = True
   if 64 in inp_yaml['supported_xlen']:
     temp['rv64']['accessible'] = True
 return temp
```

5.3 Adding support for Adjoining RISC-V specs

5.3.1 Adding new CLI

For supporting any new adjoining specs, they need to be supplied via a new cli (command line interface) argument. This new argument needs to be added in the to the parser module in *Utils.py <https://github.com/riscv/riscv-config/blob/d969b7dc5b2b308bb43b0aa65932fe2e7f8c756c/riscv_config/utils.py#L106>*.

The code below shows an example of how the debug spec is added as an argument to the cli parser module:

```
parser.add_argument('--debug_spec', '-dspec', type=str, metavar='YAML', default=None,␣
↪help='The YAML which contains the debug csr specs.')
```

5.3.2 Adding a new schema

Each new adjoining spec must have a YAML schema defined in the *schemas <https://github.com/riscv/riscv-config/tree/master/riscv_config/schemas>* director.

5.3.3 Adding checks through checker.py and SchemaValidator.py

The user might want to add more custom checks in checker.py and SchemaValidator.py for the adjoining spec.

For example the check_debug_specs() is a function that ensures the isa and debug specifications conform to their schemas. For details on check_debug_specs() check here : *riscv_config.checker*.

Details on the checks like s_debug_check() and u_debug_check, that can also be added to SchemaValidator.py are here: *riscv_config.schemaValidator*.

5.3.4 Modifications in Constants.py

The new schema must be added in the constants.py to detect its path globally across other files.

```
debug_schema = os.path.join(root, 'schemas/schema_debug.yaml')
```

5.3.5 Performing new spec checks

Finally, in the main.py file the user must call the relevant functions from checker.py for validating the inputs against the schema.

```
if args.debug_spec is not None:
    if args.isa_spec is None:
     logger.error(' Isa spec missing, Compulsory for debug')
    checker.check_debug_specs(os.path.abspath(args.debug_spec), isa_file, work_dir, True,
↪ args.no_anchors)
```

CHAPTER

SIX

CODE DOCUMENTATION

6.1 riscv_config.checker

riscv_config.checker.**add_debug_setters**(*schema_yaml*)

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

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

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

riscv_config.checker.**add_reset_setters**(*schema_yaml*)

Function to set the default setters for extension subfields in the misa

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_debug_specs**(*debug_spec*, *isa_spec*, *work_dir*, *logging=False*, *no_anchors=False*)

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

Parameters

- **debug_spec** – The path to the DUT debug specification yaml file.
- **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 pmp csrs are implemented correctly as per spec. The following checks are performed:

- the number of accessible pmpaddr csrs must be 0, 16 or 64
- the number of implemented pmpcfg csrs must be 0, 16 or 64
- the pmpaddr and pmpcfgs must be implemented implemented from the lowest numbered indices and be contiguous
- the number of accessible pmpaddr csrs and the implemented pmpcfg csrs must be the same
- for each accesible pmpaddr csr the corresponding pmpcfg csr must be implemented
- reset values of the accessible pmpaddr csrs must be coherent with the pmp_granularity field.

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.**fsset**()

Function to set defaults based on presence of 'F' extension.

riscv_config.checker.**groupc**(*test_list*)

Generator function to squash consecutive numbers for wpri bits.

riscv_config.checker.**hregset**()

Function to set defaults based on presence of 'H' extension.

riscv_config.checker.**hregseth**()

Function to set defaults based on presence of 'H' extension.

riscv_config.checker.**hset**()

Function to set defaults based on presence of 'U' extension.

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.**reset**()

Function to set defaults to reset val of misa based on presence of ISA extensions.

riscv_config.checker.**reset_vsstatus**()

Function to set defaults to reset val of mstatus based on the xlen and S, U extensions

riscv_config.checker.**resetsu**()

Function to set defaults to reset val of mstatus based on the xlen and S, U extensions

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.

6.2 riscv_config.schemaValidator

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_max_length32(*field*, *value*)

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

_check_with_s_debug_check(*field*, *value*)

Function ensures that the ro_constant is hardwired to zero when S is present in the ISA string Used mainly for debug schema

_check_with_u_debug_check(*field*, *value*)

Function ensures that the ro_constant is hardwired to zero when U is present in the ISA string Used mainly for debug schema

_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.

6.3 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

6.4 WARL

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

Symbols

A

C

D

F

G

H

M

N

R

S

T

U