

Contents

1	Introduction	2
2	Usage	2
3	Format	2
3.1	Main layout	2
3.2	Structure	3
3.3	Range	3
3.4	Dependencies	4
3.5	Colors	5
4	Loading	6
4.1	JSON Format	6
4.2	YAML Format	6
4.3	Typst Format	6
4.4	XML Format	7
5	Config presets	8
6	Reference	9
6.1	config	9
6.1.1	config	9
6.1.2	dark	13
6.1.3	blueprint	13
6.2	schema	14
6.2.1	load	14
6.2.2	render	14

1 Introduction


This package provides a way to make beautiful register diagrams using the CeTZ package. It can be used to document Assembly instructions or binary registers

This is a port of the [homonymous Python script](#) for Typst.

2 Usage

Simply import the `schema` module and call `schema.load` to parse a schema description. Then use `schema.render` to render it, et voilà !

```
1 #import "@preview/rivet:0.3.0": schema
2 #let doc = schema.load(yaml("path/to/schema.yaml"))
3 #schema.render(doc)
```

 Typst

Please read the [Loading](#) chapter for more detailed explanations on how to load schema descriptions.

3 Format

This section describes the structure of a schema definition. The examples given use the JSON syntax. For examples in different formats, see [test.yaml](#), [test.json](#) and [test.xml](#). You can also directly define a schema using Typst dictionaries and arrays.


Since the XML format is quite different from the other, you might find it helpful to look at the examples in the [Gitea repo](#) to get familiar with it.

3.1 Main layout

A schema contains a dictionary of structures. There must be at least one defined structure named “main”.

It can also optionnaly contain a “colors” dictionary. More details about this in [Colors](#)

```
1 {
2   "structures": {
3     "main": {
4       ...
5     },
6     "struct1": {
7       ...
8     },
9     "struct2": {
10      ...
11    },
12    ...
13  }
14 }
```

 JSON

3.2 Structure

A structure has a given number of bits and one or multiple ranges. Each range of bits can have a name, a description and / or values with special meaning (see [Range](#)). A range's structure can also depend on another range's value (see [Dependencies](#)).

The range name (or key) defines the left- and rightmost bits (e.g. 7-4 goes from bit 7 down to bit 4). The order in which you write the range is not important, meaning 7-4 is equivalent to 4-7. Bits are displayed in big-endian, i.e. the leftmost bit has the highest value, except if you enable the `ltr-bits` [config\(\)](#) option.

```

1  "main": {
2    "bits": 8,
3    "ranges": {
4      "7-4": {
5        ...
6      },
7      "3-2": {
8        ...
9      },
10   "1": {
11     ...
12   },
13   "0": {
14     ...
15   }
16 }
17 }
```

3.3 Range

A range represents a group of consecutive bits. It can have a name (displayed in the bit cells), a description (displayed under the structure) and / or values.

For values depending on other ranges, see [Dependencies](#).

Note

In YAML, make sure to wrap values in quotes because some values can be interpreted as octal notation (e.g. 010 → 8)

```

1  "3-2": {
2    "name": "op",
3    "description": "Logical operation",
4    "values": {
5      "00": "AND",
6      "01": "OR",
7      "10": "XOR",
8      "11": "NAND"
9    }
10 }
```

3.4 Dependencies

The structure of one range may depend on the value of another. To represent this situation, first indicate on the child range the range on which it depends.

Then, in its values, indicate which structure to use. A description can also be added (displayed below the horizontal dependency arrow)

```

1  "7-4": {
2    ...
3    "depends-on": "0",
4    "values": {
5      "0": {
6        "description": "immediate value",
7        "structure": "immediateValue"
8      },
9      "1": {
10       "description": "value in register",
11       "structure": "registerValue"
12     }
13   }
14 }
```

Finally, add the sub-structures to the structure dictionary:

```

1  {
2    "structures": {
3      "main": {
4        ...
5      },
6      "immediateValue": {
7        "bits": 4,
8        ...
9      },
10     "registerValue": {
11       "bits": 4,
12       ...
13     },
14     ...
15   }
16 }
```

3.5 Colors

You may want to highlight some ranges to make your diagram more readable. For this, you can use colors. Colors may be defined in a separate dictionary, at the same level as the “structures” dictionary:

```

1 {
2   "structures": {
3     ...
4   },
5   "colors": {
6     ...
7   }
8 }
```

It can contain color definitions for any number of ranges. For each range, you may then define a dictionary mapping bit ranges to a particular color:

```

1 "colors": {
2   "main": {
3     "31-28": "#ABCDEF",
4     "27-20": "12,34,56"
5   },
6   "registerValue": {
7     "19-10": [12, 34, 56]
8   }
9 }
```

Valid color formats are:

- hex string starting with #, e.g. "#23fa78"
- array of three integers (only JSON, YAML and Typst), e.g. [35, 250, 120]
- string of three comma-separated integers (useful for XML), e.g. "35,250,120"
- a Typst color (only Typst), e.g. colors.green or rgb(35, 250, 120)

Note

The XML format implements colors a bit differently. Instead of having a “colors” dictionary, color definitions are directly put on the same level as structure definitions. For this, you can use a color node with the attributes “structure”, “color”, “start” and “end”, like so:

```

1 <schema>
2   <structure id="main" bits="8">
3     ...
4   </structure>
5   ...
6   <color structure="main" color="#FF0000" start="4" end="7" />
7   <color structure="main" color="255,0,0" start="0" end="3" />
8 </schema>
```

4 Loading

Due to current limitations of the Typst compiler, the package can only access its own files, unless directly included in your project. For this reason, rivet cannot load a schema from a path, and you will need to read the files yourself to pass their contents to the package.

Here are a number of ways you can load your schemas:

4.1 JSON Format

```
1 // From file (ONLY IF PACKAGE INSTALLED IN PROJECT)
2 #let s = schema.load("schema.json")
3 // From file
4 #let s = schema.load(json("schema.json"))
5 // Raw block
6 #let s = schema.load(``json
7 {
8   "structures": {
9     "main": {
10       ...
11     }
12   }
13 }
14 ```)
```

4.2 YAML Format

```
1 // From file (ONLY IF PACKAGE INSTALLED IN PROJECT)
2 #let s = schema.load("schema.yaml")
3 // From file
4 #let s = schema.load(yaml("schema.yaml"))
5 // Raw block
6 #let s = schema.load(``yaml
7 structures:
8   main:
9     ...
10 ```)
```

4.3 Typst Format

```
1 #let s = schema.load((
2   structures: (
3     main: (
4       ...
5     )
6   )
7 ))
```

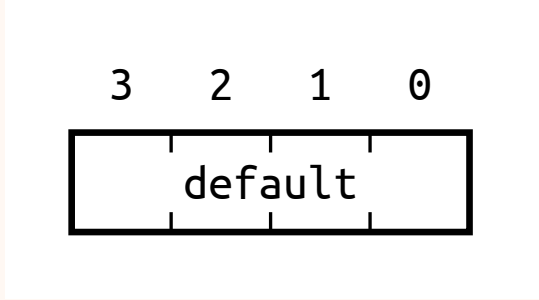
4.4 XML Format

```
1 // From file (ONLY IF PACKAGE INSTALLED IN PROJECT)
2 #let x = schema.xml-loader.load("schema.xml")
3 #let s = schema.load(x)
4 // From file
5 #let x = schema.xml-loader.parse(yaml("schema.yaml").first())
6 #let s = schema.load(x)
7 // Raw block
8 #let s = schema.load(```xml
9 <schema>
10   <structure id="main" bits="32">
11     ...
12   </structure>
13 </schema>
14 ```)
```

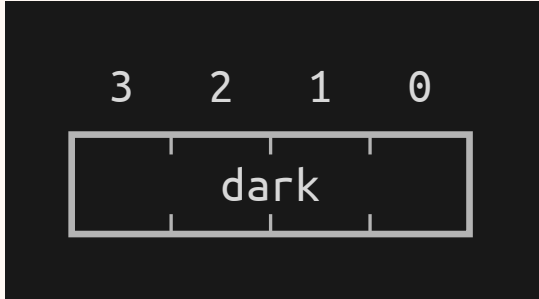
5 Config presets

Aside from the default config, some example presets are also provided:

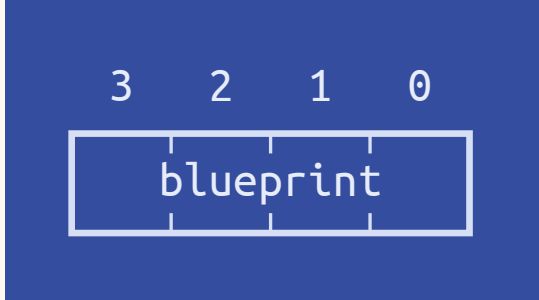
- `config.config()`: the default theme, black on white

	<pre>let ex = schema.load(`yaml structures: main: bits: 4 ranges: 3-0: name: default `) schema.render(ex, config: config.config())</pre>
---	--

- `config.dark()`: a dark theme, with white text and lines on a black background

	<pre>let ex = schema.load(`yaml structures: main: bits: 4 ranges: 3-0: name: dark `) schema.render(ex, config: config.dark())</pre>
--	---

- `config.blueprint()`: a blueprint theme, with white text and lines on a blue background

	<pre>let ex = schema.load(`yaml structures: main: bits: 4 ranges: 3-0: name: blueprint `) schema.render(ex, config: config.blueprint())</pre>
---	---

6 Reference

6.1 config

- `config()`
- `dark()`
- `blueprint()`

6.1.1 config

Creates a dictionary of all configuration parameters

Parameters

```
config(
    default-font-family: str,
    default-font-size: length,
    italic-font-family: str,
    italic-font-size: length,
    background: color,
    text-color: color,
    link-color: color,
    bit-i-color: color,
    border-color: color,
    bit-width: float,
    bit-height: float,
    description-margin: float,
    dash-length: float,
    dash-space: float,
    arrow-size: float,
    margins: tuple,
    arrow-margin: float,
    values-gap: float,
    arrow-label-distance: float,
    force-descs-on-side: bool,
    left-labels: bool,
    width: float,
    height: float,
    full-page: bool,
    all-bit-i: bool,
    ltr-bits: bool
) -> dictionary
```

default-font-family `str`

The default font family

Default: "Ubuntu Mono"

default-font-size `length`

The absolute default font size

Default: 15pt

italic-font-family str

The italic font family (for value descriptions)

Default: "Ubuntu Mono"

italic-font-size length

The absolute italic font size

Default: 12pt

background color

The diagram background color

Default: white

text-color color

The default color used to display text

Default: black

link-color color

The color used to display links and arrows

Default: black

bit-i-color color

The color used to display bit indices

Default: black

border-color color

The color used to display borders

Default: black

bit-width float

The width of a bit

Default: 30

bit-height float

The height of a bit

Default: 30

description-margin float

The margin between descriptions

Default: 10

dash-length float

The length of individual dashes (for dashed lines)

Default: 6

dash-space float

The space between two dashes (for dashed lines)

Default: 4

arrow-size float

The size of arrow heads

Default: 10

margins tuple

TODO -> remove

Default: (20, 20, 20, 20)

arrow-margin float

The margin between arrows and the structures they link

Default: 4

values-gap float

The gap between individual values

Default: 5

arrow-label-distance float

The distance between arrows and their labels

Default: 5

force-descs-on-side bool

If true, descriptions are placed on the side of the structure, otherwise, they are placed as close as possible to the bit

Default: false

left-labels bool

If true, descriptions are put on the left, otherwise, they default to the right hand side

Default: false

width float

TODO -> remove

Default: 1200

height float

TODO -> remove

Default: 800

full-page bool

If true, the page will be resized to fit the diagram and take the background color

Default: false

all-bit-i bool

If true, all bit indices will be rendered, otherwise, only the ends of each range will be displayed

Default: true

ltr-bits bool

If true, bits are placed with the LSB on the left instead of the right

Default: false

6.1.2 dark

Dark theme config

Parameters

`dark(..args: any)`

`..args` `any`

see `config()`

6.1.3 blueprint

Blueprint theme config

Parameters

`blueprint(..args: any)`

`..args` `any`

see `config()`

6.2 schema

- [load\(\)](#)
- [render\(\)](#)

6.2.1 load

Loads a schema from a file or a raw block. This function returns a dictionary of structures
See the [Loading](#) chapter for examples of schema loading for each supported format

Supported formats: .yaml, .json, .xml

Parameters

`load(path-or-schema: str raw dictionary) -> dictionary`

path-or-schema `str` or `raw` or `dictionary`

- If it is a string, defines the path to load.
⚠ Warning: this will only work if this package is part of your project, as packages installed in the `@local` or `@preview` namespace cannot access project files
- If it is a raw block, its content is directly parsed (the block's language will define the format to use)
- If it is a dictionary, it directly defines the schema structure

6.2.2 render

Renders the given schema

Parameters

```
render(
  schema: dictionary,
  config: auto dictionary,
  width: ratio length
)
```

schema `dictionary`

A schema dictionary, as returned by [load\(\)](#)

config `auto` or `dictionary`

The configuration parameters, as returned by [config\(\)](#)

Default: `auto`

width `ratio` or `length`

The width of the generated figure

Default: `100%`