# SORNgen

Marvin Henkel

26.09.2024

## Version

| | |
|---|---|
| **Project Version:** | v1.2.8 |
| **Documentation Version:** | v0.5 |

## Overview

**SORNgen** is a python-based tool to generate VHDL code for arithmetic algorithms processed in SORN arithmetic.
SORN (set of real numbers) is a datatype related to type-2 unum format. For more information see `http://www.johngustafson.net/pubs/RadicalApproach.pdf`
This documentation (in its current state) is just containing the sorngen GUI. For information regarding the base project contact Jochen.Rust@haw-hamburg.de

## Contact


| | |
|---|---|
| **Maintainer:** | Jochen Rust |
| **Email:** | Jochen.Rust@haw-hamburg.de |
| **GitHub:** | @JochenRust |
| | |
| **Developer (GUI):** | Marvin Henkel |
| **Email:** | Marvin.Henkel@haw-hamburg.de |
| **GitHub:** | @Maloxoc |


## Additional Resources

**Project Repository:** `https://github.com/Maloxoc/sorngen_2`

# Contents

# 1 Installation

In order to install SORNgen, download the project from `https://github.com/Maloxoc/sorngen_2`. If you have Git installed, you can also run the following command:

```
git clone https://github.com/Maloxoc/sorngen_2
```

This will download the project into your current working directory.

Additionally, an installation of Python 3.* is required. You can download it from `https://www.python.org/downloads/` (recommended version: 3.12.3)
Lastly, you have to install numpy. To do so, execute the following command in a command prompt:

```
pip3 install numpy
```

# 2 Getting Started

## 2.1 Starting the tool

The SORNgen-tool is available in the sorngen.py file. To launch the GUI, please refer to section 2.3. Since the project is organized as a Python module, you need to start it using the '-m' switch from a command line interface (CLI) or a similar environment. Additionally, you must provide the path to a specification file (see 2.2). Assuming your current working directory is set to the project root (which contains the playground and stable directories), you can execute the following command:

python -m stable.sorngen <specification file>

If the specification file is valid, the tool will generate VHDL code in the current working directory. To specify a different output directory, simply append the desired path:

python -m stable.sorngen <specification file> [output directory]

Regardless of where the output directory is located, a 'VHDL' folder will be created, and the generated code will be placed inside.

## 2.2 The specification file

The specification file is crucial, as it defines the type of module(s) the tool will generate. Understanding the structure and contents of this file is essential for effectively using SORNgen.
The specification file has a '.sorn' ending and contains all information about name, datatype, pipeline registers and the integrated equations of your Hardware design.

- Name: Set the name of the design. The toplevel VHDL file will be "[name].vhd".
- Datatype: ['lin'/'log'/'man', '[start,stop,step]', 'zero', 'negative', 'infinity']
  - 'lin'/'log': choose either a linear or logarithmic spacing of the lattice values
  - 'man': choose a fully manually defined datatype
    - define datatype as ['man','¡interval1¿;¡interval2¿;¡...¿'] with ¡interval¿ having open "(" and closed "[" interval bounds
    - see "MIMO_solver_N2" inside the "stable/templates" directory for an example
  - '[start,stop,step]': choose the start and end value for the lattice values and a stepsize for a linear scale
  - 'zero', 'negative', 'infinity': extend the datatype by the given options (any combination can be choosen)
- Pipeline registers: The amount of specified registers will be inserted in the design.
- Equations: Specify one or multiple equations with consistent variable names, round brackets and python-based arithmetic operators. Variables may appear in multiple equations.
- Examples: See the files "MIMO_solver_N2.sorn" and "MIMO_solver_N4.sorn" inside the "stable/templates" directory.

## 2.3 The GUI

To launch the graphical user interface (GUI) of the SORNgen-tool, use the 'App.py' file. Just like starting the tool via the command line, the GUI must be launched using the '-m' switch. Assuming your current working directory is the project root, run the following command:

python -m stable.App

This will open the GUI, where you can input the specification (see 2.2) in the main text area. If you have an existing specification file, you can load it using the "Open Specification" button at the top of the window.
Once the hardware behavior is defined in the specification, proceed to the next design step by pressing the "Next Step" button. If an error occurs during any design step, it will be highlighted in the upper-right section.
When a design step completes successfully, a confirmation will also appear in the top-right section, and the interface will transition to the next design step screen. As of now, only the specification phase is implemented, so subsequent steps will display a "Not Implemented" message. Future versions will include detailed information on the processing status of the input data during these stages.
To the right of the main panel, additional details and potential issues about the current design step will be displayed. You can also review the log of the entire process by clicking the "Open Log" button in the lower-right corner. This log contains all outputs generated by the script up to that point.
You can revisit previous design steps by selecting the buttons on the left side of the window. If you want to regenerate the output for a specific step—such as after modifying the specification—you can do so by pressing the "Regenerate" button in the lower-left corner. Keep in mind, if you make changes to the specification, you will need to regenerate the output for all subsequent steps.
If you prefer to execute all steps in sequence without reviewing each individually, click the "Run All Steps" button.
Additionally, you can save and load your project at any time using the "Save Project" and "Open Project" buttons located in the upper-left corner. Projects are saved as .sorngen files. Important: Never open untrusted .sorngen files, as they may contain malicious code.

# 3 Developer Guide

## 3.1 General Information

The base SORNgen-tool is based on the Paper "A Hardware Generator for SORN Arithmetic" (especially Section III. "The Hardware Generator"). While the paper contains some outdated information, it still outlines the general architecture and functionality of the tool.
The tool is written entirely in Python, so a basic understanding of Python is recommended. To ensure easier refactoring and scalability, the project follows the Python package structure.

## 3.2 The GUI Structure

### 3.2.1 General Architecture

The GUI was developed using python 3.12.3 and is implemented using the built-in tkinter package.
The interface layout currently uses the grid layout manager (see resource 1) to allow for flexible positioning of widgets. However, it may be beneficial to switch to a different layout manager, like stacking frames side by side, instead of directly positioning widgets within the root.
For details regarding the grid layout manager, refer to `https://www.pythonguis.com/tutorials/create-ui-with-tkinter-grid-layout-manager/`
The sections (as seen in resource 2) are named as follows:

1. main_frame: Placeholder for the content frames (see Section 3.2.2)
2. project_frame: Hosts buttons for project management.
3. windows_frame: Contains tabs for switching between different content areas in the main frame.
4. navigation_frame: Hosts navigation buttons for design flow steps.
5. status_frame: Displays the current status of each design step.
6. info_frame: Displays quick log messages or information generated during the design process.
7. This area has no specific name but contains the toggle button for the console.

### 3.2.2 Main Frames

Main frames display critical information for the current or previous design flow steps. Every step (except for the final code generation) has its own dedicated frame:

1. specification_frame: A text field for entering specifications.
2. parsing_frame: Displays parsing information (currently not implemented).
3. elaboration_frame: Displays elaboration information (currently not implemented).
4. architecture_frame: Displays architecture information (currently not implemented).
5. sorn_op_frame: Displays information about SORN operations (currently not implemented).
6. console_frame: Contains the log field.

The switch_to(tk.Frame) method can change the currently displayed main frame. Since it applies layout configurations to the new frame, any tk.Frame object can become a main frame. These frames are generated via the gui.components script (see Section 3.3)

### 3.2.3 Surrounding/ Meta Frames

The surrounding Frames around the main frame are mostly static tk.Frame objects containing different buttons and labels. They are already describe in 3.2.1 (from 2). They are generated using the gui.components script (see 3.3 for more information)

#### 3.2.4 STD Handler

To capture and manage all output from the script, the class gui.aux_lib.std_handler is used. You can start capturing output with the capture method, and stop with the stop method.

### 3.3 Adding new content

The gui.components script contains useful methods for adding new widgets.

#### 3.3.1 Versioning

For compatibility and error tracking between different project versions, a simple version control number is implemented (see control.version). This version must be manually updated. Although it has no automatic effect, it's recommended to update the version with each addition or modification.

#### 3.3.2 Frames

To create new tk.Frame objects, use the gui.components.get_frame function. The required input is a root widget, which can be any tkinter widget (e.g., tk.Frame). Optional parameters include:

- bg: str, background color.
- row, col: int: grid layout positions for the widget.
- rowspan, colspan: int, span of rows/columns.
- sticky: str, directions to stick (north, south, east, west).
- padx, pady: int/ tuple[int, int], padding around the widget.
- render: bool, whether to render the widget (default is true).
- border: bool, if set to true, the frame will have a border.

To add a title bar to a frame, use the add_title_frame method, specifying the frame and the title (as a tk.StringVar object). Optionally, you can add buttons to the title bar by passing a list of tuples (label, function).
This method will push all containing elements of a tk.Frame one row lower and add a title bar with your specified title and if given buttons.

#### 3.3.3 Buttons

Buttons are created using the get_button method, which essentially works the same as get_frame. In addition to the root widget, you must provide a name and a callable function for the button. Some optional arguments (e.g., render and border) are not available for buttons.

#### 3.3.4 Text Fields

For text fields, use the get_text_field method, specifying the root widget and optional background color and padding. The two types of text fields are:

- tk.Text objects, created using get_text_field.
- Log fields, a subclass of tk.Text, created using get_log_field. Log fields are used to record script outputs. To append to the log, use the log method. If you created the Log field via get_log_field, you can add a severity to the log method ("info", "warning", "error", "success")

#### 3.3.5 Canvas

Although a get_canvas method exists, it currently generates a placeholder canvas that displays "not implemented."

### 3.3.6 Labels

There are two types of labels:

- Regular labels, created with the get_label method.
- Status labels, created with get_status_label. Status labels are specialized for displaying statuses ("idle", "running", "success", "failure") and should be managed via their methods, not directly modified.

## 3.4 Localisation

### 3.4.1 How it works

The current localisation system, implemented in the gui.localisation.translator.Translator class, is essentially a lightweight, dictionary-based solution. During the setup process, the class generates a tk.StringVar object for each key in the default (initial) language.
At the moment, only English is supported. However, if additional languages are available, the translate(str) method can be used to switch the language. This method will update all tk.StringVar objects to the desired translation automatically. To retrieve a tk.StringVar object, call the get(str) method with the corresponding key. This will return a reference to the tk.StringVar object, so any future calls to translate(str) will update the object dynamically.
For cases where a key has multiple meanings (e.g., a toggle button that switches between "open log" and "close log"), the set_to(str, str) method can be used. This requires two arguments:

1. The key of the tk.StringVar object you want to modify.
2. An additional key that references the specific value within the first key.

Essentially, this acts as a nested dictionary structure, where a single key can have multiple sub-keys to handle context-specific translations.

### 3.4.2 How to add new languages

To add a new language, follow these steps:

1. Create a new language script in the gui/localisation directory. For simplicity, it is recommended to copy an existing language script (e.g., the English file) and rename it.
2. Manually translate each value in the file.

Once the new language script is ready, add it to the get_dict(str) function. A basic language script follows this structure (see gui.localisation.english for reference):

$$\text{translations} = \{ < key_1 >:< translation_1 >, ... < key_n >:< translation_n > \}$$

### 3.4.3 How to add new keys

The current localisation system requires each new key to be added manually to every language file. If a key has a single meaning, you can assign it directly to the desired translation. For keys that handle multiple meanings (such as toggle states), you can assign a nested dictionary with sub-keys and corresponding translations.
The structure for multiple meanings looks like this:

$$< key >: \{< subkey_1 >: subtranslation_1, ... < subkey_n >: subtranslation_n \}$$

During the initialisation process, the first sub-translation will be selected by default. For examples of this structure, refer to the gui.localisation.english file.

# 4 Roadmap

## 4.1 Design-flow-step Frames

As mentioned in section 2.3, not all design-flow-step frames have been implemented yet. The remaining frames to be added include:

1. The Parsing Frame
2. The Elaboration Frame
3. The Architecture Frame
4. The SORN operations Frame

These Frames should contain and give information about the current phase of the data processing. The intended contents of each frame are as follows:

| | |
|---|---|
| Parsing Frame: | Displays the Abstract Syntax Tree (AST) |
| Elaboration Frame: | Displays the SORN Syntax Tree (SST) |
| Architecture Frame: | Shows HDL Diagrams |
| SORN operations Frame: | Displays the relevant SORN Intervals |

### Implementation Notes:

The first three frames (Parsing, Elaboration, and Architecture) will likely use Canvas widgets to render trees and diagrams directly, potentially based on XML data. In contrast, the SORN Operations Frame will likely involve a table-based representation. Since Tkinter does not provide a native table widget, a custom table will need to be implemented.
Additionally, simply adding these frames to the GUI will not be sufficient; the underlying data structures need to be handled efficiently. This includes modifying the current saving and loading mechanisms to support storing and retrieving information specific to each frame.
While some of this information is currently available in the log as text, visualizing it more intuitively will be crucial for troubleshooting and improving usability.
The following resources may be helpful when implementing these features:

- `https://tkdocs.com/tutorial/canvas.html`
- `https://stackoverflow.com/questions/9348264/does-tkinter-have-a-table-widget`
- `https://www.geeksforgeeks.org/create-table-using-tkinter/`

## 4.2 Optimisation

While the current framework performs adequately, certain sections of the code may require refactoring and optimization, particularly when handling larger specifications. Although no major performance issues have been identified, minor slowdowns could occur with more complex inputs. Therefore, ongoing optimization is recommended to ensure scalability and efficiency.
Additionally, as stated in 3.2.1, it is recommended to switch from using the grid layout manager to a different layout manager, firstly because the grid layout manager may be prone to errors (on this scale of layout) and because it makes adding new content more difficult.

## 4.3 Localisation

As mentioned in Section3.4the current implementation of localisation is quite basic. Adding a new language requires creating a new language script that mirrors every key present in the existing language scripts, and then incorporating this script into the translator.
While this process is relatively straightforward, it becomes more error-prone when adding new keys (e.g., for new features). Each new key must be added manually to every language script, which increases the risk of missing or inconsistent translations.
To mitigate these issues, it is recommended to enhance the localisation system. Potential improvements include:

- Implementing a base language that the translator can fall back on if a key is missing in another language script.
- Centralizing all keys in a single, unified script to simplify management.
- Using .yml or .json formats for the language files instead of Python scripts, as these formats are more widely used for localisation and are easier to maintain.

By making these improvements, the localisation system would become more scalable, easier to manage, and less error-prone.

# 5 License

This project is licensed under the MIT License - see the LICENSE file for details.

# 6 Resources

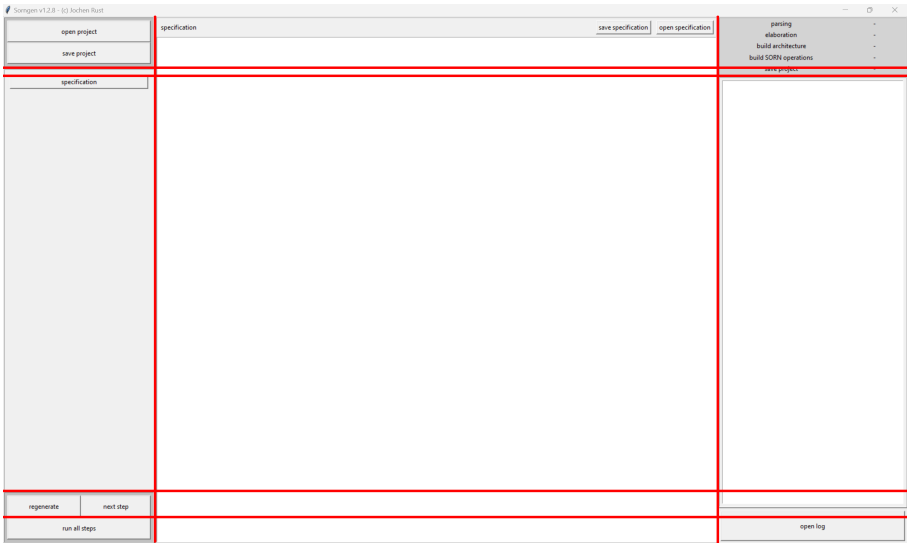


Figure 1: The root window layout. The red lines show columns and rows

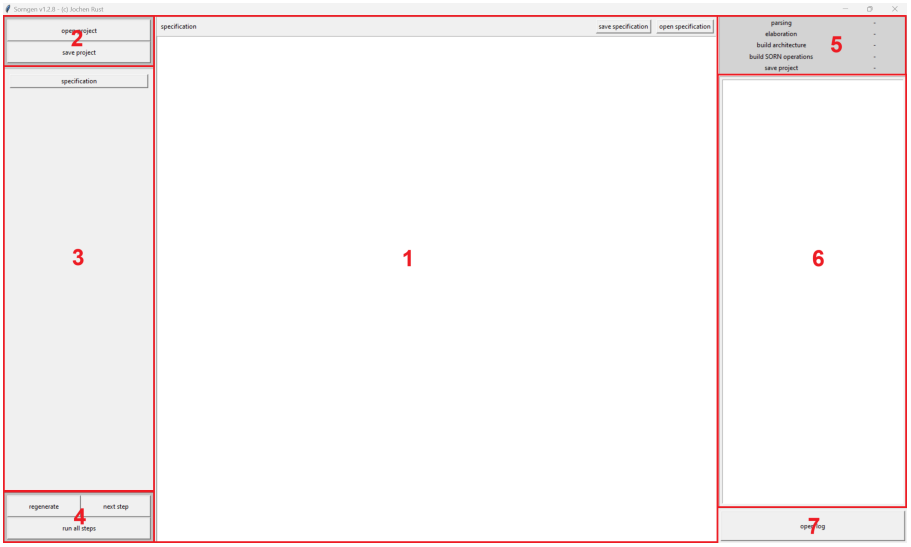


Figure 2: The root Map. See 3.2.1 for legend