Py2HWSW

A Python Framework for HW/SW Co-design

March 24, 2025

User Guide, 0.81 , Build 5f8f1097

©2025 IObundle Lda. All rights reserved

Document Version History

Version	Date	Person	Changes from previous version
0.81	March 24, 2025	JTS	Unrelased

©2025 IObundle Lda. All rights reserved

©2025 IObundle Lda. All rights reserved

Contents

©2025 IObundle Lda. All rights reserved

©2025 IObundle Lda. All rights reserved

List of Tables

©2025 IObundle Lda. All rights reserved

List of Figures

©2025 IObundle Lda. All rights reserved

1 Introduction

Open-source Python framework for managing files, automating project flows of embedded hardware/software codesign projects, and partially generating Verilog hardware components. The framework simplifies the project structure, addresses challenges in Hardware Design Languages like Verilog and VHDL, and automates emulation, simulation, FPGA, and ASIC flows. The proposed Verilog generator offers flexibility, user control, and ease of use, producing human-readable code compatible across FPGAs and ASICs.

1.1 What Is Py2HWSW?

In the rapidly evolving landscape of hardware design, the need for efficient and flexible tools is paramount. Enter py2hwsw, a powerful tool designed to streamline the process of generating Verilog cores from high-level descriptions provided in Python or JSON dictionaries. With py2hwsw, engineers can easily translate their design specifications into functional hardware components, significantly reducing development time and complexity.

1.2 What Is Py2HWSW For?

Py2HWSW is designed to do the following:

- **Core Generation**: Generates Verilog cores from descriptions in Python or JSON dictionaries.
- **Framework Compatibility**: Integrates seamlessly with existing Verilog cores and frameworks.
- **High-Level Configuration**: Allows configuration of cores via high-level Python parameters.
- **Automated Resources**: Produces scripts and Makefiles for deployment in various FPGAs, simulators, and synthesis tools, along with documentation.
- **Readable Code**: Generates legible Verilog code with comments for better understanding and maintenance.

1.3 What Problem Does Py2HWSW Solve?

Py2hwsw addresses several key challenges in the hardware design process:

- **Complexity of Verilog Coding**: Writing Verilog code can be intricate and error-prone, especially for those who may not be deeply familiar with hardware description languages. Py2hwsw simplifies this by allowing designers to specify their hardware requirements using high-level Python or JSON dictionaries, reducing the need for extensive Verilog knowledge.
- **Integration of Existing Designs**: Many projects involve legacy Verilog cores that need to be integrated with new designs. Py2hwsw facilitates this integration, enabling users to leverage existing components while still benefiting from the tool's advanced features.
- **Configuration Challenges**: Customizing hardware components often requires deep dives into low-level code. Py2hwsw allows for high-level configuration through Python parameters, making it easier for designers to adjust their designs without getting bogged down in the details of Verilog.

©2025 IObundle Lda. All rights reserved

- **Resource Generation**: The process of preparing scripts and Makefiles for various deployment environments can be tedious and time-consuming. Py2hwsw automates this process, providing users with the necessary resources to run their designs on different FPGAs, simulators, and synthesis tools.
- **Code Readability and Maintenance**: Maintaining and debugging hardware designs can be challenging, especially when the code is not well-documented. Py2hwsw generates legible Verilog code with comments, enhancing readability and making it easier for teams to collaborate and maintain their designs over time.

In summary, Py2hwsw streamlines the hardware design workflow, making it more accessible, efficient, and manageable for engineers and designers.

1.4 What Design Principles Underlie Py2HWSW?

Py2HWSW works by:

- **Standard Py2HWSW syntax**: Use a standard Py2HWSW syntax to describe each core.
- **Support Python dictionaries and JSON files**: Supports Python dictionaries to generate dynamic cores based on python parameters. And supports JSON files to describe fixed cores and for compatibily with cores generated by external tools.
- **Support custom verilog snippets**: Each core may include custom verilog snippets for any edge-case which cannot be described using Py2HWSW syntax.
- **Internal Object-Oriented structure**: Py2HWSW converts core descriptions into its internal object-oriented system, creating high-level abstractions of Verilog building blocks.

1.5 How Does Py2HWSW Accomplish Its Goals?

- **Two-Step Development Process**: The core development is divided into two distinct phases: the **setup** phase and the **build** phase. During the setup phase, Verilog source files are generated based on high-level descriptions provided in Python or JSON format. The build phase then utilizes these Verilog sources to produce the necessary FPGA bitstreams, netlists, and other deployment files.
- **Independent Setup Folders**: Each core is organized within its own independent setup folder, containing high-level description files and, if needed, low-level files as well.
- **Core Description Input**: The core's specifications are provided to Py2hwsw in the form of JSON or a Python dictionary, utilizing standard Py2hwsw attributes.
- **Flexible Attribute Handling**: When generating the cores dictionary via a Python script, users can include a set of standard Py2hwsw attributes alongside their own custom-defined attributes.

2 Getting Started

2.1 Setup Directory

The setup directory of a core may have the following structure:

©2025 IObundle Lda. All rights reserved

```
.
├── core_name.py
├── core_name.json
├── document
│   ├── doc_build.mk
│   ├── figures
│   └── tsrc
├── hardware
│   ├── src
│   ├── fpga
│   │   ├── fpga_build.mk
│   │   ├── src
│   │   ├── quartus
│   │   └── vivado
│   ├── modules
│   ├── simulation
│   │   ├── sim_build.mk
│   │   └── src
│   └── syn
│       ├── src
│       └── genus
├── software
│   ├── sw_build.mk
│   └── src
├── scripts
├── submodules
├── Makefile
├── README.md
├── LICENSE
├── CITATION.cff
└── default.nix
```

Only the `core_name.py` or `core_name.json` file is needed to pass the core's description to Py2HWSW. The remaining directories and files are optional.

If the `document`, `hardware`, and `software` directories exist, they will be copied to the `build` directory, overriding any files already present there, such as standard ones or files from other cores.

The `*_build.mk` files allow the user to include core specific Makefile targets and variables from the build process. These will be copied to the `build` directory and included in the standard build process Makefiles.

The `src` directories contain manually written Verilog/C/TeX sources for the core, should they be needed.

The following directories and files do not follow a mandatory structure, but are typically used for the following purposes:

The `hardware/modules` and `submodules` directories typically contain setup directories of other cores.

The `scripts` directory contains scripts specific to the core, and may be called by the user or from the `core_name.py` script.

A simple example of a core's setup directory is available for the iob_and core.

A more complex example of a core's setup directory is available for the iob_soc core.

©2025 IObundle Lda. All rights reserved

2.2 Create An AND Gate Core: iob_and

The simplest core description for Py2HWSW is as follows:

```
# SPDX-FileCopyrightText: 2025 IObundle
#
# SPDX-License-Identifier: MIT


def setup(py_params_dict):
    attributes_dict = {
        "generate_hw": True,
        "confs": [
            {
                "name": "general",
                "descr": "General group of confs",
                "confs": [
                    {
                        "name": "W",
                        "type": "P",
                        "val": "21",
                        "min": "1",
                        "max": "32",
                        "descr": "IO width",
                    },
                ],
            },
        ],
        "ports": [
            {
                "name": "a_i",
                "descr": "Input port",
                "signals": [
                    {"name": "a_i", "width": "W"},
                ],
            },
            {
                "name": "b_i",
                "descr": "Input port",
                "signals": [
                    {"name": "b_i", "width": "W"},
                ],
            },
            {
                "name": "y_o",
                "descr": "Output port",
                "signals": [
                    {"name": "y_o", "width": "W"},
                ],
            },
        ],
        "snippets": [{"verilog_code": "   assign y_o = a_i & b_i;"}],
    }

    return attributes_dict
```


©2025 IObundle Lda. All rights reserved

View Source

A set of basic cores to showcase the various Py2HWSW features can be found in the basic_tests directory.

2.3 Setup And Build

To checkout the source and setup the example iob_and core:

```
$ git clone --recursive git@github.com:IObundle/py2hwsw.git
$ cd py2hwsw/
$ nix-shell py2hwsw/lib/ # Optional step to install environment with
    necessary dependencies
$ py2hwsw iob_and setup --no_verilog_lint
```

To do a clean setup:

```
$ py2hwsw iob_and clean
$ py2hwsw iob_and setup --no_verilog_lint
```

The setup process will generate a build directory containing the core's verilog sources and build files. By default, the build directory is '../[core_name]_V[core_version]'.

To build and run the core in simulation:

```
$ make -C ../iob_and_V* sim-run
```

2.4 Installation

Py2HWSW uses a Nix-shell environment to handle dependencies. The full list of dependencies is available as Nix packages in the `default.nix` file, which can be found at https://github.com/IObundle/py2hwsw/blob/main/py2hwsw/lib/default.nix.

The recommended way to install Py2HWSW is by using Nix-shell. Most Makefiles in IObundle projects call Nix-shell by default, so it is expected that a user will install Py2HWSW via Nix-shell. To do this, simply install Nix by following the instructions at https://nixos.org/download.html#nix-install-linux. Then, navigate to a directory that contains the Py2HWSW `default.nix` file and run `nix-shell`. Py2HWSW will self-install, and all dependencies will be installed automatically.

Alternatively, it is possible to install Py2HWSW manually by removing the Nix-shell commands from the Makefiles and installing the dependencies manually. After doing so, the user can call Py2HWSW by adding the `py2hwsw` file from the `bin/` folder to the `PATH` environment variable. The `py2hwsw` file can be found at https://github.com/IObundle/py2hwsw/blob/main/bin/py2hwsw.

Another option is to install Py2HWSW using pip with the following command:

```
pip install -e path/to/py2hwsw_directory
```

However, please note that this method is not officially supported, and dependencies will still need to be handled manually or by using Nix.

©2025 IObundle Lda. All rights reserved

Py2HWSW is primarily maintained and tested on Linux, but it should also work on macOS and Windows Subsystem for Linux (WSL) since Nix is supported on these platforms.

2.5 Basic Usage

To use Py2HWSW, you can run the following command:

```
nix-shell --run "py2hwsw $(CORE) setup --build_dir '$(BUILD_DIR)' --py_params 'param1=param1_val:param2
```

This command sets up a core using Py2HWSW, where (CORE) is the name of the core, (BUILD DIR) is the directory where the build files will be generated, and (param1=param1_val:param2=param2_val) are optional Python parameters that can be used to customize the core.

The –build_dir option allows you to specify the location of the generated build directory. If not specified, the build directory will be generated in the parent directory of where the Py2HWSW command is called.

You can also use the –help option to list all available options and a brief description of each:

```
py2hwsw --help
```

To create a new core, you will need to create a setup directory with the same name as the core. This directory should contain at least one file with the same name as the core, either with a .py or .json extension, that describes the core using attributes of the core dictionary. The setup directory may also contain other files and folders following a standard hierarchy, which is described in more detail in other sections of this document.

For examples of simple cores, you can refer to the basic_tests folder in the Py2HWSW library: https://github.com/IObundle/py2hwsw/tree/main/py2hwsw/lib/hardware/basic_tests. For creating System On Chips, you can use the iob-soc repository as a template: https://github.com/IObundle/iob-soc/tree/main.

Some key concepts to understand when using Py2HWSW include:

- Setup directory: The folder that contains the core description and base files, templates, scripts, and sources.
- Build directory: The folder generated by the Py2HWSW setup process, which contains a standard file hierarchy and all the necessary makefiles to build and run the core on various simulators, FPGA, ASIC tools, and linters.
- Core: An IP core that contains Verilog sources, documentation, scripts, high-level attributes, and possibly software.
- Module: Sometimes used as an alternative to core, but it is recommended to use the term "core" instead. May also refer to Verilog modules and Python modules.

Py2HWSW can be used to create a wide variety of cores, from simple to complex. One of the main advantages of using Py2HWSW is that it generates readable Verilog code and all the necessary makefiles to run the core on various flows, making it a powerful tool for hardware design and development.

 ©2025 IObundle Lda. All rights reserved

2.6 Universal Testbench

Py2HWSW supports *Universal Test Bench*.

Create iob_vlt_tb.h verilator file and define the dut_t type as the top module for simulation of the core. For example, for the iob_uart core, we use the iob_uart_sim.v as the top module for simulation, so we define the Viob_uart_sim as the dut_t type.

```
/*
 * SPDX-FileCopyrightText: 2025 IObundle
 *
 * SPDX-License-Identifier: MIT
 */

#include "Viob_uart_sim.h"
typedef Viob_uart_sim dut_t;
```

View Source

Create the iob_core_tb.c source to drive the verification instruments (usually instantiated in the simulation wrapper). For example, the iob_uart core is also used as a verification instrument to test itself. It is instantiated in the iob_uart_sim.v file, and its RS232 ports are connected in loopback. The tesbench then drives this core, writing data to it, and reading back the data received from the loopback.

```
/*
 * SPDX-FileCopyrightText: 2025 IObundle
 *
 * SPDX-License-Identifier: MIT
 */

#include "iob_uart_csrs.h"
#include <stdio.h>

#define FREQ 100000000
#define BAUD 3000000

int iob_core_tb() {

  int failed = 0;

  // print welcome message
  printf("IOB UART testbench\n");

  // print the reset message
  printf("Reset complete\n");

  // hold soft reset low
  iob_uart_set_softreset(0);

  // print the soft reset message
  printf("Soft reset set to 0\n");

  // disable TX and RX

  iob_uart_set_txen(0);
```

©2025 IObundle Lda. All rights reserved

```
  iob_uart_set_rxen(0);

  // set the divisor

  int div = FREQ / BAUD;
  iob_uart_set_div(div);

  // print the baud rate
  printf("Baud rate set to %d\n", BAUD);

  // assert tx and rx not ready
  uint8_t tx_ready = iob_uart_get_txready();
  if (tx_ready != 0) {
    printf("Error: TX ready initially\n");
    failed = 1;
  }

  uint8_t rx_ready = iob_uart_get_rxready();
  if (rx_ready != 0) {
    printf("Error: RX ready initially");
    failed = 1;
  }

  printf("TX and RX ready deasserted\n");

  // pulse soft reset
  iob_uart_set_softreset(1);
  iob_uart_set_softreset(0);

  // enable RX
  iob_uart_set_rxen(1);

  unsigned int version;

  // read version 20 times to burn time
  for (int i = 0; i < 20; i++) {
    version = iob_uart_get_version();
  }
  printf("Version is %d\n", version);

  // enable TX
  iob_uart_set_txen(1);

  printf("TX and RX enabled\n");

  // data send/receive loop
  for (int i = 0; i < 256; i++) {
    // wait for tx ready
    while (!iob_uart_get_txready())
      ;

    // write word to send
    iob_uart_set_txdata(i);
    // wait for rx ready
    while (!iob_uart_get_rxready())
```


©2025 IObundle Lda. All rights reserved

```
87        ;
88
89      // read received word
90      uint8_t rx_data = iob_uart_get_rxdata();
91
92      // check if received word is the same as sent word
93      if (rx_data != i) {
94        // signal error printing expected and received word
95        printf("Error: expected %d, received %d\n", i, rx_data);
96        failed += 1;
97      }
98    }
99
100   return failed;
101 }
```

View Source

3 How It Works

This section gives a detailed description of the Py2HWSW framework.

3.1 Overview

The Py2HWSW framework is organized into a repository with several key folders and scripts. The repository contains the main Py2HWSW module, as well as a library of cores and peripherals. The framework uses a combination of Python scripts and Makefiles to automate the generation of build directories for hardware components.

The setup process in Py2HWSW begins with the user providing a description of the core, which can be in the form of a Python script or a JSON file, in a setup directory. This description is then used to trigger the setup process, which involves gathering all dependency cores and generating the necessary Verilog code. The setup process creates a build directory, where all the generated Verilog modules are stored, correctly connected and structured based on the user's description. The build directory is independent of Py2HWSW and can be used on any machine with the necessary toolchain.

The build process is a separate step that takes the generated build directory as input and uses the Makefiles to run the toolchain for a specific flow, such as simulation or FPGA synthesis. For example, in the case of FPGA synthesis, the build process takes the generated Verilog sources as input, generates a bitstream, uploads it to the FPGA, and runs the design. In the case of simulation, the build process takes the Verilog sources and generates a simulator executable (for Verilator) or runs the simulation directly. The build process can be run on any machine with the necessary toolchain, without requiring Py2HWSW to be installed.

The main launch script, py2hwsw.py, serves as the entry point for the framework, and is responsible for orchestrating the setup process. The script takes care of setting up the build environment, generating Verilog code, and creating the build directory. Once the build directory is generated, the user can run the build process independently of Py2HWSW, using the Makefiles to automate the simulation, synthesis, and compilation of the hardware components.

©2025 IObundle Lda. All rights reserved

3.2 Technical Details

From the user's perspective, interacting with Py2HWSW is straightforward and intuitive. Users describe cores using dictionaries, lists, and strings, which are then converted internally into object representations of the correct class. The main attributes of Py2HWSW, such as ports, wires, and configuration, each have their own class, organizing the properties of each attribute. These attributes are described by a dictionary, where each key is a property, and are converted to the corresponding property of the class for the internal object representation when calling the Py2HWSW process.

As described in the "Standard Interfaces" section, users only need to interact with Py2HWSW using standard interfaces based on dictionaries, lists, and strings. Internally, Py2HWSW converts these inputs into object representations, but these are usually only modified by developers. A typical user does not need to understand the inner workings of Py2HWSW, making it easy to use and focus on designing and developing hardware components.

The iob_core.py class is the central component of Py2HWSW, aggregating all the properties of an IP core. Its constructor is responsible for the setup process of the core, which involves converting and initializing the attributes of the core, setting up submodules (each one a new iob_core instance), setting up superblocks (only if the current core is the top module or another superblock), and generating the sources for the current core in the build directory. If the current core is the top module, the setup process terminates by formatting and linting the code, as well as cleaning up temporary files from the build directory.

In terms of dependencies, Py2HWSW itself has a minimal set of requirements, including Python and certain Python libraries, as well as optional dependencies such as formatters and linters like Black, Verible, and Verilator. These formatters can be skipped if the user chooses not to use formatting and linting during setup. The generated build directory, on the other hand, may have additional dependencies specific to the build process, such as Makefiles, Verilog compilers and simulators. However, these dependencies are independent of Py2HWSW and are only required for the build process. Makefiles are not a required dependency of Py2HWSW itself, but can be useful for automating the setup process and integrating Py2HWSW into a larger project workflow.

3.3 Setup Flow Chart

Figure 1 presents a high-level flow chart of the Py2HWSW setup procedure.

©2025 IObundle Lda. All rights reserved

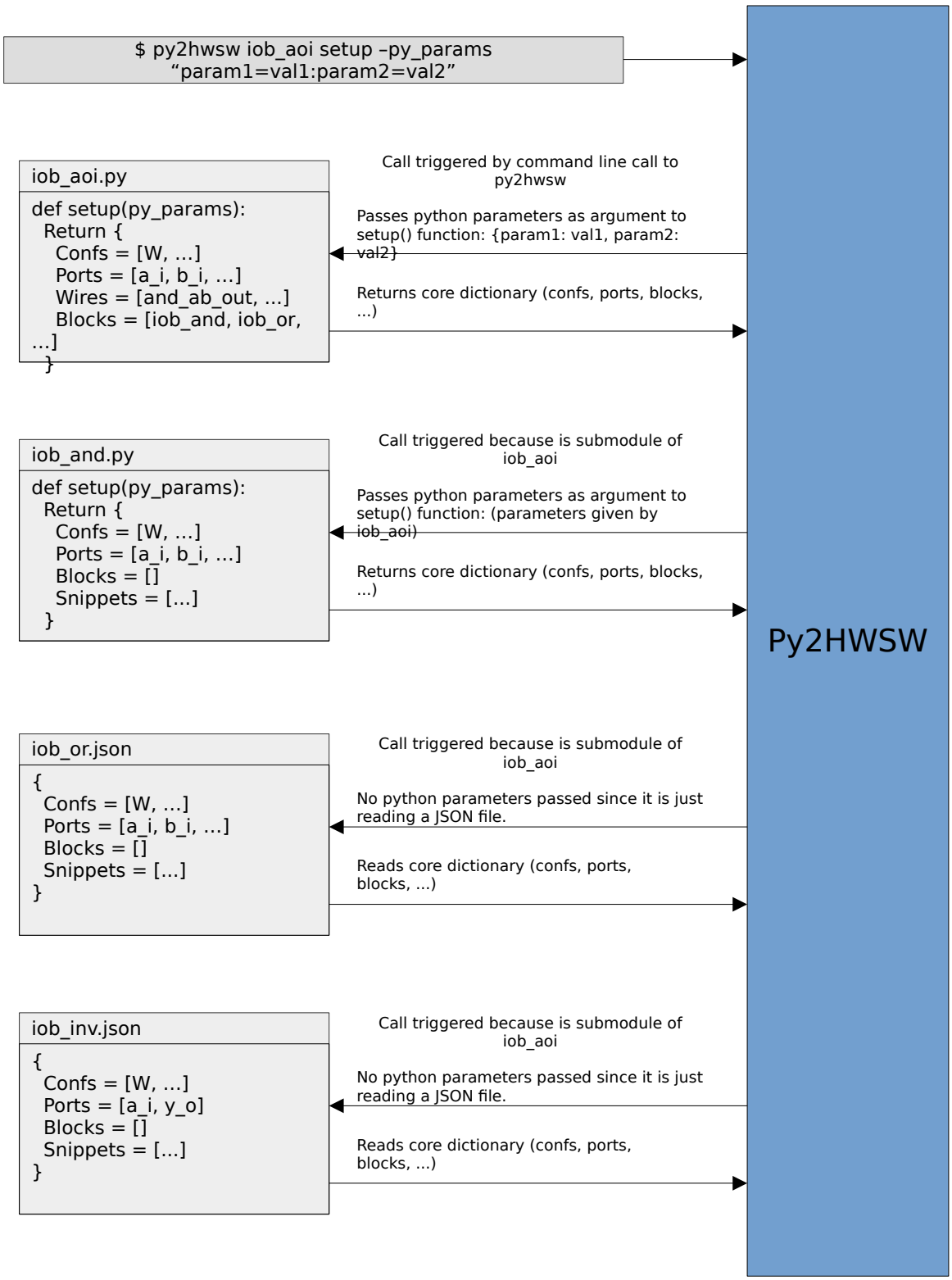

Figure 1: High-Level Flow Chart of Py2HWSW Setup Procedure

©2025 IObundle Lda. All rights reserved

3.4 Standard Interfaces

The Py2HWSW framework provides the following two standard interfaces:

1. **Python Parameters**: Core ”setup” function receives information from Py2HWSW via a dictionary in its first argument, referred to as *Python Parameters*.
2. **Core Dictionary**: Core ”setup” function returns a core description dictionary to Py2HWSW, referred to as *Core Dictionary*.

The core’s ”setup” function is the python function defined by the user in the ¡core_name¿.py file.

If the core is described by a JSON file, then the *Python Parameters* interface is not available. The JSON file gives a dictionary to Py2HWSW, similar to the python dictionary of the ”setup” function. This allows the user to use external tools to generate cores in JSON format.

The *Python Parameters* received by the core’s ”setup” function is a dictionary containing both parameters passed by its instantiator and standard parameters passed by Py2HWSW. Each key, value pair in the dictionary is a *Python Parameter*. The value of the python parameter may be of any data type.

Name	Data Type	Description
core_name	<class ’str’>	Name of current core (determined by the core’s file name).
build_dir	<class ’str’>	Build directory of this core. Usually defined by ‘–build-dir‘ flag or instantiator.
py2hwsw_target	<class ’str’>	The reason why py2hwsw is invoked. Usually ‘setup‘ meaning the Py2HWSW is calling the core’s script to obtain information on how to generate the core. May also be other targets like ‘clean‘, ‘print_attributes‘, or ‘deliver‘. These are usually to obtain information about the core for various purposes, but not to generate the build directory.
instantiator	<class ’dict’>	Core dictionary with attributes of the instantiator core (if any). Allows subblocks to obtain information about their instantiator core.
py2hwsw_version	<class ’str’>	Version of Py2HWSW.

Table 1: Standard *Python Parameters* passed by Py2HWSW to every core’s ”setup” function.

The standard python parameters passed by Py2HWSW are listed in Table 1.

The python parameters supported by each core is available in the respective core’s user guide, as long as they have the *Python Parameters* attribute defined. Instructions on how to build a core’s user guide can be found in Section 4.7.

Name	Data Type	Description
original_name	<class ’str’>	Original name of the module. (The module name commonly used in the files of the setup dir.)
name	<class ’str’>	Name of the generated module.
description	<class ’str’>	Description of the module
confs	<class ’list’>	List of module macros and Verilog (false-)parameters.
ports	<class ’list’>	List of module ports.
wires	<class ’list’>	List of module wires.

©2025 IObundle Lda. All rights reserved

Name	Data Type	Description
snippets	<class 'list'>	List of core Verilog snippets.
comb	<class 'iob_comb.iob_comb'>	Verilog combinatory circuit.
fsm	<class 'iob_fsm.iob_fsm'>	Verilog finite state machine.
subblocks	<class 'list'>	List of instances of other cores inside this core.
superblocks	<class 'list'>	List of wrappers for this core. Will only be setup if this core is a top module, or a wrapper of the top module.
sw_modules	<class 'list'>	List of software modules required by this core.
instance_name	<class 'str'>	Name of the instance
instance_description	<class 'str'>	Description of the instance
parameters	typing.Dict	Verilog parameter values
if_defined	<class 'str'>	Only use this instance in Verilog if this Verilog macro is defined
if_not_defined	<class 'str'>	Only use this instance in Verilog if this Verilog macro is not defined
instantiate	<class 'bool'>	Select if should intantiate the module inside another Verilog module.
build_dir	<class 'str'>	Path to folder of build directory to be generated for this project.
version	<class 'str'>	Core version. By default is the same as Py2HWSW version.
previous_version	<class 'str'>	Core previous version.
setup_dir	<class 'str'>	Path to root setup folder of the core.
use_netlist	<class 'bool'>	Copy '<SETUP_DIR>/CORE.v' netlist instead of '<SETUP_DIR>/hardware/src/*'
is_system	<class 'bool'>	Sets 'IS_FPGA=1' in config_build.mk
board_list	<class 'list'>	List of FPGAs supported by this core. A standard folder will be created for each board in this list.
dest_dir	<class 'str'>	Relative path inside build directory to copy sources of this core. Will only sources from 'hardware/src/*'
ignore_snippets	<class 'list'>	List of '.vs' file includes in verilog to ignore.
generate_hw	<class 'bool'>	Select if should try to generate '<corename>.v' from py2hwsw dictionary. Otherwise, only generate '.vs' files.
parent	<class 'dict'>	Select parent of this core (if any). If parent is set, that core will be used as a base for the current one. Any attributes of the current core will override/add to those of the parent.
is_top_module	<class 'bool'>	Selects if core is top module. Auto-filled. DO NOT CHANGE.
is_superblock	<class 'bool'>	Selects if core is superblock of another. Auto-filled. DO NOT CHANGE.
is_tester	<class 'bool'>	Generates makefiles and depedencies to run this core as if it was the top module. Used for testers (superblocks of top moudle).
python_parameters	<class 'list'>	List of core Python Parameters. Used for documentation.
license	<class 'iob_license.iob_license'>	License for the core.

©2025 IObundle Lda. All rights reserved

Name	Data Type	Description

Table 2: Table of supported Py2HWSW attributes in the **Core Dictionary**. The *Data Type* column specifies the type of internal object that the Py2HWSW will convert the attribute's value to (usually the user inputs a string, list, or dictionary value and then py2 converts it to an internal object).

The list of attributes supported by the Py2HWSW framework is given in Table 2. If a core provides a dictionary with keys not listed in Table 2, then the Py2HWSW framework will raise an error. Each key, value pair in the dictionary is a *Core Attribute*. The data type of the core attribute may be of any data type, but are usually a string, list, or dictionary. If the data type is a string, it may also represent an object using Py2HWSW's *Short Notation*.

3.5 Block hierarchy

Figure 2 presents an example block hierarchy for a Py2HWSW project. Superblocks are only used if they are superblocks of the project's top module or of one of its wrappers.

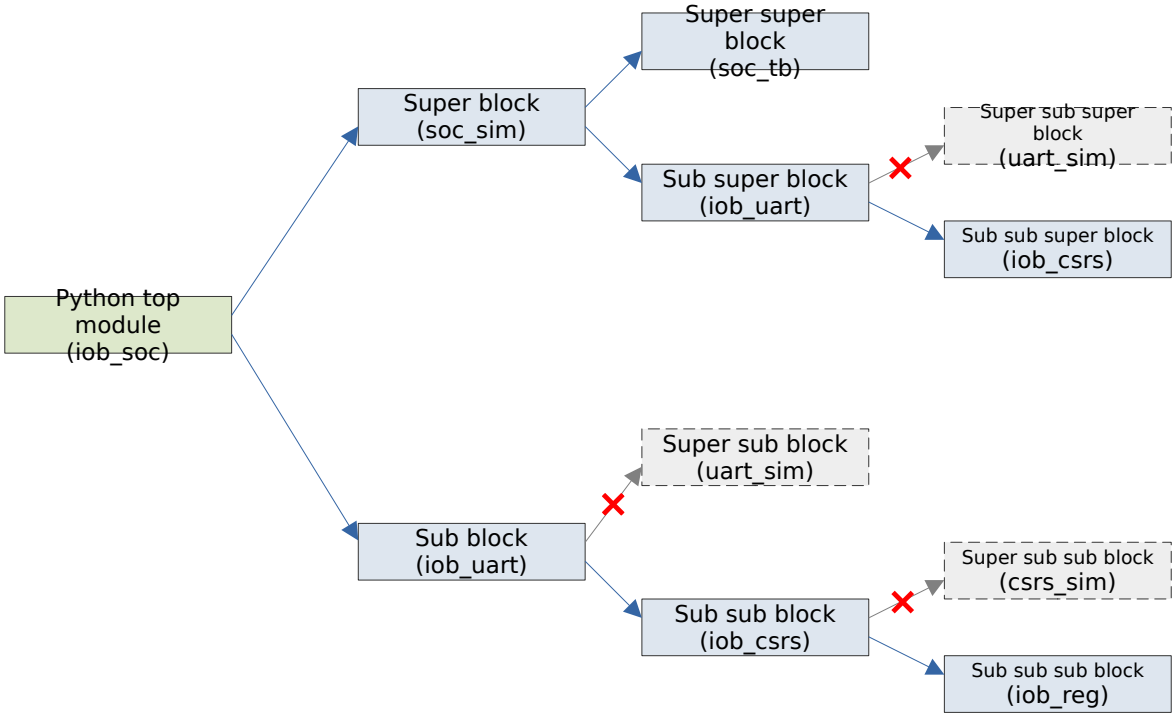

Figure 2: Block Hierarchy of a Py2HWSW Project

©2025 IObundle Lda. All rights reserved

3.6 Main launch script: py2hwsw.py

The main launch script for the Py2HWSW progam is the ‘py2hwsw.py‘ script.

The following code snippet from that script processes the command line arguments and launches the program for the specified ”target”.

```
        action="store_true",
        help="Print supported Py2HWSW core dictionary attributes",
    )
    args = parser.parse_args()

    # print(f"Args: {args}", file=sys.stderr)  # DEBUG

    iob_core.global_build_dir = args.build_dir
    iob_core.global_project_root = args.project_root
    iob_core.global_project_vformat = args.verilog_format
    iob_core.global_project_vlint = args.verilog_lint
    iob_core.global_clang_format_rules_filepath = args.clang_rules
    iob_base.debug_level = args.debug_level

    if args.py2hwsw_docs:
        iob_core.setup_py2_docs(PY2HWSW_VERSION)
        exit(0)

    if args.print_py2hwsw_attributes:
        iob_core.print_py2hwsw_attributes()
        exit(0)

    if not args.core_name:
        parser.print_usage(sys.stderr)
        exit(1)

    py_params = {}
    if args.py_params:
        for param in args.py_params.split(":"):
            k, v = param.split("=")
            py_params[k] = v

    if args.target == "setup":
        iob_core.get_core_obj(args.core_name, **py_params)
    elif args.target == "clean":
        iob_core.clean_build_dir(args.core_name)
    elif args.target == "print_build_dir":
        iob_core.print_build_dir(args.core_name, **py_params)
    elif args.target == "print_core_name":
        iob_core.print_core_name(args.core_name, **py_params)
    elif args.target == "print_core_version":
        iob_core.print_core_version(args.core_name, **py_params)
    elif args.target == "print_core_dict":
        iob_core.print_core_dict(args.core_name, **py_params)
    elif args.target == "deliver":
        iob_core.deliver_core(args.core_name, **py_params)
```

©2025 IObundle Lda. All rights reserved

View Source

3.7 Simulate with Verilator

With mandatory structured IOs, the testbench behaves like a processor reading and writing to its CSR. A universal Verilator testbench has been developed for an IP with a structured IOb native interface (bridges to standard AXI-Lite, APB or Wishbone are supplied). The testbench is a C++ program provides hardware reset and CSR read and write functions.

3.7.1 IP core simulation

The IP cores using this testbench must provide a C function called `iob_core_tb()`, the IP core's specific test. They also must provide a C header called `iob_vlt_tb.h` that defines the Device Under Test (DUT) as a Verilator type called `dut_t`. With knowledge of the DUT and its test, the universal Verilator testbench will exercise any IP core. Interestingly, `iob_core_tb()` also runs, without modifications, on a RISC-V processor with the IP as a submodule, for example, for FPGA testing or emulation.

The `iob_uart` core is used as an example, located in the `py2hwsw/lib/peripherals/iob_uartiob-uart` directory.

```
$ git clone --recursive git@github.com:IObundle/py2hwsw.git
$ cd py2hwsw/lib
$ make sim-run CORE=iob\_uart SIMULATOR=verilator
```

The `make sim-run` command will run core setup, creating the build directory at `../../../iob_uart_V0.1`. The Verilator simulator will be run in the build directory. The testbench will be compiled and run, and the output will be displayed on the console.

3.7.2 Subsystem simulation

To illustrate system test capabilities with the universal Verilator testbench, the `iob_system` subsystem core is used as an example, located in the `py2hwsw/lib/iob_system` directory.

```
$ git clone --recursive git@github.com:IObundle/py2hwsw.git
$ cd py2hwsw/lib
$ make sim-run CORE=iob\_uart SIMULATOR=verilator
```

In this case the `iob_core_tb()` function is running on the desktop, emualting a system tester. The console output comes from teh system itself runnig its embedded test, a more elaborated form of a hello world program.

3.8 Deliver an IP core

From the build directory, we select the essential files to create a tarball, all containing a Makefile-driven environment for the user who, in this way, will not need any ancillary tools beyond the standard EDA tools.

```
$ git clone --recursive git@github.com:IObundle/py2hwsw.git
$ cd py2hwsw/
```


©2025 IObundle Lda. All rights reserved

```
$ nix-shell py2hwsw/lib/ # Optional step to install environment with
    necessary dependencies
$ py2hwsw iob_uart setup --no_verilog_lint
$ py2hwsw iob_uart deliver
```

The tarball wil be created in the `../iob_uart_V0.1` directory, which is also the home of the default build directory.

4 Py2HWSW Classes

4.1 Main class for core representation: iob_core.py

The iob_core class is a central component of the Py2HWSW framework, responsible for representing and managing IP cores. The class provides a structured way to describe and generate IP cores, making it easier to create and integrate complex digital designs. At the heart of the iob_core class is its constructor, which plays a crucial role in setting up and initializing the core.

The constructor of the iob_core class is responsible for converting and initializing the attributes of the core, setting up subblocks, and generating the sources for the current core in the build directory. When the constructor is called, it distinguishes between two situations: when it is the top module, it creates a build directory for users with all the dependencies and project flows, and when it is a subblock, it provides all the information necessary for instantiation and integration. The constructor takes care of setting up the build environment, generating Verilog code, and creating the build directory, making it a key component of the Py2HWSW framework.

The iob_core constructor also handles the setup process for subblocks and superblocks. When a subblock is encountered, the constructor is called recursively to set up the subblock's attributes and generate its sources. Similarly, when a superblock is encountered, the constructor sets up the superblock's attributes and generates its sources, ensuring that the entire hierarchy of modules and subblocks is properly initialized and configured, as detailed in Section 3.5

Overall, the iob_core class and its constructor provide a powerful and flexible way to represent and manage IP cores, making it a fundamental component of the Py2HWSW framework.

It inherits attributes from its parent classes iob_module and iob_instance.

View Source

The get_core_obj function is used to generate an instance of a core based on a given core name and python parameters. This method will search for the corresponding Python or JSON file of the core, and generate a python object based on info stored in that file, and info passed via python parameters.

```
        __class__.global_special_target = "print_core_dict"
        # Build a new module instance, to obtain its attributes
        module = __class__.get_core_obj(core_name, **kwargs)
        print(json.dumps(module.attributes_dict, indent=4))

    @staticmethod
    def print_py2hwsw_attributes():
        """Print the supported attributes of the py2hwsw interface.
```

©2025 IObundle Lda. All rights reserved

```
        The attributes listed can be used in the 'attributes' dictionary of
            cores.
        """
        # Set project wide special target (will prevent normal setup)
        __class__.global_special_target = "print_attributes"
        # Build a new dummy module instance, to obtain its attributes
        module = __class__()
        print("Attributes supported by the 'py2hwsw' core dictionary
            interface:")
        for name in module.ATTRIBUTE_PROPERTIES.keys():
            datatype = module.ATTRIBUTE_PROPERTIES[name].datatype
            descr = module.ATTRIBUTE_PROPERTIES[name].descr
            align_spaces = " " * (20 - len(name))
            align_spaces2 = " " * (18 - len(str(datatype)))
            print(f"- {name}:{align_spaces}{datatype}{align_spaces2}{descr}"
                )

    @staticmethod
    def get_core_obj(core_name, **kwargs):
        """Generate an instance of a core based on given core_name and
            python parameters
        This method will search for the .py and .json files of the core, and
             generate a
        python object based on info stored in those files, and info passed
            via python
        parameters.
        Calling this method may also begin the setup process of the core,
            depending on
        the value of the `global_special_target` attribute.
        """
        core_dir, file_ext = find_module_setup_dir(core_name)

        if file_ext == ".py":
            import_python_module(
                os.path.join(core_dir, f"{core_name}.py"),
            )
            core_module = sys.modules[core_name]
            instantiator = kwargs.pop("instantiator", None)
            # Call `setup(<py_params_dict>)` function of `<core_name>.py` to
            # obtain the core's py2hwsw dictionary.
            # Give it a dictionary with all arguments of this function,
                since the user
            # may want to use any of them to manipulate the core attributes.
            core_dict = core_module.setup(
                {
                    # "core_name": core_name,
                    "build_dir": __class__.global_build_dir,
                    "py2hwsw_target": __class__.global_special_target or "
                        setup",
                    "instantiator": (
```

View Source

©2025 IObundle Lda. All rights reserved

4.2 Configuration class: iob_conf.py

The iob_conf class is used to represent a configuration option of the core. This class contains a set of attributes, each preceded by a comment describing the purpose of the attribute.

```python
class iob_conf:
    """Class to represent a configuration option."""

    # Identifier name for the configuration option.
    name: str = ""
    # Type of configuration option, either M (Verilog macro), P (Verilog
       parameter) or F (Verilog false-parameter).
    # False-parameters are the same as verilog parameters except that the
       its value must not be overriden.
    type: str = ""
    # Value of the configuration option.
    val: str | int | bool = ""
    # Minimum value supported by the configuration option (NA if not
       applicable).
    min: str | int = "NA"
    # Maximum value supported by the configuration option (NA if not
       applicable).
    max: str | int = "NA"
    # Description of the configuration option.
    descr: str = "Default description"
    # Only applicable to Verilog macros: Conditionally enable this
       configuration if the specified Verilog macro is defined/undefined.
    if_defined: str = ""
    if_not_defined: str = ""
    # If enabled, configuration option will only appear in documentation.
       Not in the verilog code.
    doc_only: bool = False
```

View Source

The iob_conf_group class is used to represent a group of configuration options. This class contains a set of attributes, each preceded by a comment describing the purpose of the attribute.

```python
class iob_conf_group:
    """Class to represent a group of configurations."""

    # Identifier name for the group of configurations.
    name: str = ""
    # Description of the configuration group.
    descr: str = "Default description"
    # List of configuration objects.
    confs: list = field(default_factory=list)
    # If enabled, configuration group will only appear in documentation. Not
        in the verilog code.
    doc_only: bool = False
    # If enabled, the documentation table for this group will be terminated
       by a TeX '\clearpage' command.
    doc_clearpage: bool = False
```

View Source

©2025 IObundle Lda. All rights reserved

The py2hwsw tool uses methods from the config_gen.py script to generate the '*_conf.vh' file, which contains all the Verilog macros that must be held for every design instance of the core.

Each generated Verilog macro is based on the attributes from the corresponding instance of the 'iob_conf' class.

```
    for group in macros:
        # If group has 'doc_only' attribute set to True, skip it
        if group.doc_only:
            continue
        for macro in group.confs:
            # If macro has 'doc_only' attribute set to True, skip it
            if macro.doc_only:
                continue
            if macro.if_defined:
                file2create.write(f"`ifdef {macro.if_defined}\n")
            if macro.if_not_defined:
                file2create.write(f"`ifndef {macro.if_not_defined}\n")
            # Only insert macro if its is not a bool define, and if so only
               insert it if it is true
            if type(macro.val) is not bool:
                m_name = macro.name.upper()
                m_default_val = macro.val
                file2create.write(f"`define {core_prefix}{m_name} {
                   m_default_val}\n")
            elif macro.val:
                m_name = macro.name.upper()
                file2create.write(f"`define {core_prefix}{m_name} 1\n")
            if macro.if_defined or macro.if_not_defined:
                file2create.write("`endif\n")
```

View Source

The py2hwsw tool uses methods from the param_gen.py script to generate the Verilog parameters code that is automatically inserted in the core's Verilog module and instances.

Each generated Verilog parameter is based on the attributes from the corresponding instance of the 'iob_conf' class.

```
    lines = []
    core_prefix = f"{core.name}_".upper()
    for idx, parameter in enumerate(core_parameters):
        # If parameter has 'doc_only' attribute set to True, skip it
        if parameter.doc_only:
            continue

        p_name = parameter.name.upper()
        p_comment = ""
        if parameter.type == "F":
            p_comment = "  // Don't change this parameter value!"
        lines.append(f"    parameter {p_name} = `{core_prefix}{p_name},{
           p_comment}\n")

    # Remove comma from last line
    if lines:
        lines[-1] = lines[-1].replace(",", "", 1)
```


©2025 IObundle Lda. All rights reserved

View Source

4.3 Signal class: iob_signal.py

The iob_signal class is used to represent a signal for a hardware wire or port. This class contains a set of attributes, each preceded by a comment describing the purpose of the attribute.

```python
class iob_signal:
    """Class that represents a wire/port signal"""

    # Identifier name for the signal.
    name: str = ""
    # Number of bits in the signal.
    width: str or int = 1
    # Description of the signal.
    descr: str = "Default description"
    # If enabled, signal will be generated with type ‘reg‘ in Verilog.
    isvar: bool = False

    # Used for ‘iob_comb‘: If enabled, iob_comb will infer a register for
       this signal.
    isreg: bool = False
    # Used for ‘iob_comb‘: List of signals associated to the infered
       register.
    reg_signals: list[str] = field(default_factory=list)

    # Logic value for future simulation effort using global signals list.
    # See 'TODO' in iob_core.py for more info: https://github.com/IObundle/
       py2hwsw/blob/a1e2e2ee12ca6e6ad81cc2f8f0f1c1d585aaee73/py2hwsw/scripts
       /iob_core.py#L251-L259
    value: str or int = 0
```

View Source

The py2hwsw tool uses the ‘get_verilog_wire‘/‘get_verilog_port‘ methods from the ‘iob_signal‘ class to generate the Verilog code for the hardware wire/port based on the attributes from the corresponding instance of the ‘iob_signal‘ class.

```python
    def get_verilog_wire(self):
        """Generate a verilog wire string from this signal"""
        wire_type = "reg" if self.isvar or self.isreg else "wire"
        width_str = "" if self.get_width_int() == 1 else f"[{self.width
           }-1:0] "
        return f"{wire_type} {width_str}{self.name};\n"
```

```python
    def get_verilog_port(self, comma=True):
        """Generate a verilog port string from this signal"""
        self.assert_direction()
        comma_char = "," if comma else ""
        port_type = " reg" if self.isvar or self.isreg else ""
        width_str = "" if self.get_width_int() == 1 else f"[{self.width
           }-1:0] "
        return f"{self.direction}{port_type} {width_str}{self.name}{
           comma_char}\n"
```

©2025 IObundle Lda. All rights reserved

View Source

4.4 Wire class: iob_wire.py

The iob_wire class is used to represent a group of hardware wires (signals) used to interconnect components automatically generated. This class contains a set of attributes, each preceded by a comment describing the purpose of the attribute.

```
class iob_wire:
    """Class to represent a wire in an iob module"""

    # Identifier name for the wire.
    name: str = ""
    # Name of the standard interface to auto-generate with `if_gen.py`
       script.
    interface: if_gen.interface = None
    # Description of the wire.
    descr: str = "Default description"
    # Conditionally define this wire if the specified Verilog macro is
       defined/undefined.
    if_defined: str = ""
    if_not_defined: str = ""
    # List of signals belonging to this wire
    # (each signal represents a hardware Verilog wire).
    signals: List = field(default_factory=list)
```

View Source

The ‘signals‘ attribute stores a list of signal objects, represented by the ‘iob_signal‘ class (Section 4.3).

The py2hwsw tool uses the ‘generate_wires‘ method from the ‘wire_gen.py‘ script to generate the Verilog code for the wire based on the attributes from the corresponding instance of the ‘iob_wire‘ class.

```
    for wire in core.wires:
        # Open ifdef if conditional interface
        if wire.if_defined:
            code += f"`ifdef {wire.if_defined}\n"
        if wire.if_not_defined:
            code += f"`ifndef {wire.if_not_defined}\n"

        signals_code = ""
        for signal in wire.signals:
            if isinstance(signal, iob_signal):
                signals_code += "    " + signal.get_verilog_wire()
        if signals_code:
            code += f"    // {wire.name}\n"
            code += signals_code

        # Close ifdef if conditional interface
        if wire.if_defined or wire.if_not_defined:
            code += "`endif\n"
```

View Source

©2025 IObundle Lda. All rights reserved

4.5 Port class: iob_port.py

The iob_port class is used to represent an interface for the core. An interface is a group of hardware ports (signals) that may be generic or follow a standard. Due to the similarities between a port and a wire, this class inherits the attributes from the ‘iob_wire‘ class (Section 4.4). Besides the inherited attributes, the class contains a set of new port specific attributes, each preceded by a comment describing the purpose of the attribute.

```
class iob_port(iob_wire):
    """Describes an IO port."""

    # External wire that connects this port
    e_connect: iob_wire | None = None
    # Dictionary of bit slices for external connections. Name: signal name; Value: bit slice
    e_connect_bit_slices: list = field(default_factory=list)
    # If enabled, port will only appear in documentation. Not in the verilog code.
    doc_only: bool = False
    # If enabled, the documentation table for this port will be terminated by a TeX '\clearpage' command.
    doc_clearpage: bool = False
```

View Source

Similar to the ‘iob_wire‘ class, the ‘signals‘ attribute stores a list of signal objects, represented by the ‘iob_signal‘ class (Section 4.3).

The py2hwsw tool uses the ‘generate_ports‘ method from the ‘io_gen.py‘ script to generate the Verilog code for the port based on the attributes from the corresponding instance of the ‘iob_port‘ class.

```
    lines = []
    for port_idx, port in enumerate(core.ports):
        # If port has 'doc_only' attribute set to True, skip it
        if port.doc_only:
            continue

        # Open ifdef if conditional interface
        if port.if_defined:
            lines.append(f"`ifdef {port.if_defined}\n")
        if port.if_not_defined:
            lines.append(f"`ifndef {port.if_not_defined}\n")

        lines.append(f"    // {port.name}\n")

        for signal_idx, signal in enumerate(port.signals):
            if isinstance(signal, iob_signal):
                lines.append("    " + signal.get_verilog_port())

        # Close ifdef if conditional interface
        if port.if_defined or port.if_not_defined:
            lines.append("`endif\n")

    # Remove comma from last port line
    if lines:
        i = -1
```

©2025 IObundle Lda. All rights reserved

```
26          while lines[i].startswith("`endif") or lines[i].startswith("     // "
                ):
27              i -= 1
28          lines[i] = lines[i].replace(",", "", 1)
```

View Source

4.6 Special cases

Most of the cores provived by the py2hwsw's library are built using the standard interfaces mentioned in section 3.4.

However, there are some cores that due to limitations of the standard interfaces, rely instead on internal py2hwsw methods for extra features. The following list describes the cores don't rely solely on the standard interfaces.

- **iob_system**: This core uses the 'is_system' attribute to enable an internal py2hwsw method that automatically fixes the address widths of the cbus interfaces of the system's peripherals.
- **iob_csrs**: The py2hwsw tool contains an internal method to automatically search for the "iob_csrs" subblock and insert a "<prefix>_cbus_s" port on the instantiator core of this subblock. It then connects this newly created "<prefix>_cbus_s" port of the instantiator core to the iob_csrs "control_if_s" port. The '<prefix>' is replaced by instance name of iob_csrs subblock.

4.7 Core library

The Py2HWSW framework includes a library of cores ready for use.

Name	Directory
iob_alt_iobuf	hardware/altera/iob_alt_iobuf
iob_altddio_in	hardware/altera/iob_altddio_in
iob_altddio_out	hardware/altera/iob_altddio_out
iob_altera_alt_ddr3	hardware/altera/iob_altera_alt_ddr3
iob_altera_clk_buf_altclkctrl	hardware/altera/iob_altera_clk_buf_altclkctrl
iob_altera_ddio_out_clkbuf	hardware/altera/iob_altera_ddio_out_clkbuf
iob_xilinx_axi_interconnect	hardware/amd/iob_xilinx_axi_interconnect
iob_xilinx_clock_wizard	hardware/amd/iob_xilinx_clock_wizard
iob_xilinx_ddr4_ctrl	hardware/amd/iob_xilinx_ddr4_ctrl
iob_xilinx_ibufg	hardware/amd/iob_xilinx_ibufg
iob_xilinx_oddre1	hardware/amd/iob_xilinx_oddre1
iob_acc	hardware/arith_logic/accumulators/iob_acc
iob_acc_ld	hardware/arith_logic/accumulators/iob_acc_ld
iob_counter	hardware/arith_logic/counter/iob_counter
iob_counter_ld	hardware/arith_logic/counter/iob_counter_ld
iob_modcnt	hardware/arith_logic/counter/iob_modcnt
iob_div_pipe	hardware/arith_logic/div/iob_div_pipe
iob_div_slice	hardware/arith_logic/div/iob_div_pipe/hardware/modules/iob_div_slice
iob_div_subshift	hardware/arith_logic/div/iob_div_subshift

©2025 IObundle Lda. All rights reserved

Name	Directory
iob_div_subshift_frac	hardware/arith_logic/div/iob_div_subshift_frac
iob_div_subshift_signed	hardware/arith_logic/div/iob_div_subshift_signed
iob_add	hardware/arith_logic/iob_add
iob_add2	hardware/arith_logic/iob_add2
iob_ctls	hardware/arith_logic/iob_ctls
iob_diff	hardware/arith_logic/iob_diff
iob_edge_detect	hardware/arith_logic/iob_edge_detect
iob_fp_add	hardware/arith_logic/iob_fp/iob_fp_add
iob_fp_clz	hardware/arith_logic/iob_fp/iob_fp_clz
iob_fp_cmp	hardware/arith_logic/iob_fp/iob_fp_cmp
iob_fp_div	hardware/arith_logic/iob_fp/iob_fp_div
iob_fp_dq	hardware/arith_logic/iob_fp/iob_fp_dq
iob_fp_float2int	hardware/arith_logic/iob_fp/iob_fp_float2int
iob_fp_float2uint	hardware/arith_logic/iob_fp/iob_fp_float2uint
iob_fp_fpu	hardware/arith_logic/iob_fp/iob_fp_fpu
iob_fp_int2float	hardware/arith_logic/iob_fp/iob_fp_int2float
iob_fp_minmax	hardware/arith_logic/iob_fp/iob_fp_minmax
iob_fp_mul	hardware/arith_logic/iob_fp/iob_fp_mul
iob_fp_round	hardware/arith_logic/iob_fp/iob_fp_round
iob_fp_special	hardware/arith_logic/iob_fp/iob_fp_special
iob_fp_sqrt	hardware/arith_logic/iob_fp/iob_fp_sqrt
iob_fp_uint2float	hardware/arith_logic/iob_fp/iob_fp_uint2float
iob_functions	hardware/arith_logic/iob_functions
iob_int_sqrt	hardware/arith_logic/iob_int_sqrt
iob_prio_enc	hardware/arith_logic/iob_prio_enc
iob_xor	hardware/arith_logic/iob_xor
iob_2to1mux	hardware/basic_tests/iob_2to1mux
iob_and	hardware/basic_tests/iob_and
iob_aoi	hardware/basic_tests/iob_aoi
iob_aoi_tester	hardware/basic_tests/iob_aoi/iob_aoi_tester
iob_csrs_demo	hardware/basic_tests/iob_csrs_demo
iob_fsm3	hardware/basic_tests/iob_fsm3
iob_fsm_defaults	hardware/basic_tests/iob_fsm_defaults
iob_inv	hardware/basic_tests/iob_inv
iob_or	hardware/basic_tests/iob_or
iob_rom_acc	hardware/basic_tests/iob_rom_acc
iob_address_translator	hardware/buses/iob_address_translator
iob_apb2iob	hardware/buses/iob_apb2iob
iob_arbiter	hardware/buses/iob_arbiter
iob_asym_converter	hardware/buses/iob_asym_converter
iob_axi2axil	hardware/buses/iob_axi2axil
iob_axi2iob	hardware/buses/iob_axi2iob
iob_axi_crossbar	hardware/buses/iob_axi_crossbar
iob_axi_full_xbar	hardware/buses/iob_axi_full_xbar
iob_axi_interconnect	hardware/buses/iob_axi_interconnect
iob_axi_interconnect_wrapper	hardware/buses/iob_axi_interconnect_wrapper
iob_axi_merge	hardware/buses/iob_axi_merge
iob_axi_split	hardware/buses/iob_axi_split
iob_axil2iob	hardware/buses/iob_axil2iob
iob_axil_split	hardware/buses/iob_axil_split
iob_axis2axi	hardware/buses/iob_axis2axi
iob_axis2axi_in	hardware/buses/iob_axis2axi/submodules/iob_axis2axi_in

©2025 IObundle Lda. All rights reserved

Name	Directory
iob_axis2axi_out	hardware/buses/iob_axis2axi/submodules/iob_axis2axi_out
iob_axis2fifo	hardware/buses/iob_axis2fifo
iob_axis_tasks	hardware/buses/iob_axis_tasks
iob_bus_demux	hardware/buses/iob_bus_demux
iob_bus_width_converter	hardware/buses/iob_bus_width_converter
iob_demux	hardware/buses/iob_demux
iob_fifo2axis	hardware/buses/iob_fifo2axis
iob_iob2apb	hardware/buses/iob_iob2apb
iob_iob2axi	hardware/buses/iob_iob2axi
iob_iob2axi_rd	hardware/buses/iob_iob2axi/submodules/iob_iob2axi_rd
iob_iob2axi_wr	hardware/buses/iob_iob2axi/submodules/iob_iob2axi_wr
iob_iob2axil	hardware/buses/iob_iob2axil
iob_iob2wishbone	hardware/buses/iob_iob2wishbone
iob_merge	hardware/buses/iob_merge
iob_mux	hardware/buses/iob_mux
iob_reverse	hardware/buses/iob_reverse
iob_split	hardware/buses/iob_split
iob_tasks	hardware/buses/iob_tasks
iob_wishbone2iob	hardware/buses/iob_wishbone2iob
iob_clkbuf	hardware/clocks_resets/iob_clkbuf
iob_clkmux	hardware/clocks_resets/iob_clkmux
iob_clock	hardware/clocks_resets/iob_clock
iob_pulse_gen	hardware/clocks_resets/iob_pulse_gen
iob_pulse_gen_tester	hardware/clocks_resets/iob_pulse_gen/iob_pulse_gen_tester
iob_reset	hardware/clocks_resets/iob_reset
iob_bfifo	hardware/fifo/iob_bfifo
iob_fifo_async	hardware/fifo/iob_fifo_async
iob_fifo_sync	hardware/fifo/iob_fifo_sync
iob_gray2bin	hardware/fifo/iob_gray2bin
iob_gray_counter	hardware/fifo/iob_gray_counter
iob_csrs	hardware/iob_csrs
iob_iobuf	hardware/iob_iobuf
iob_axi_ram	hardware/memories/iob_axi_ram
iob_memwrapper	hardware/memories/iob_memwrapper
iob_ram_2p	hardware/memories/ram/iob_ram_2p
iob_ram_at2p	hardware/memories/ram/iob_ram_at2p
iob_ram_atdp	hardware/memories/ram/iob_ram_atdp
iob_ram_atdp_be	hardware/memories/ram/iob_ram_atdp_be
iob_ram_sp	hardware/memories/ram/iob_ram_sp
iob_ram_sp_be	hardware/memories/ram/iob_ram_sp_be
iob_ram_sp_se	hardware/memories/ram/iob_ram_sp_se
iob_ram_t2p	hardware/memories/ram/iob_ram_t2p
iob_ram_t2p_be	hardware/memories/ram/iob_ram_t2p_be
iob_ram_t2p_tiled	hardware/memories/ram/iob_ram_t2p_tiled
iob_ram_tdp	hardware/memories/ram/iob_ram_tdp
iob_ram_tdp_be	hardware/memories/ram/iob_ram_tdp_be
iob_ram_tdp_be_xil	hardware/memories/ram/iob_ram_tdp_be_xil
iob_regfile_2p	hardware/memories/regfile/iob_regfile_2p
iob_regfile_at2p	hardware/memories/regfile/iob_regfile_at2p
iob_regfile_sp	hardware/memories/regfile/iob_regfile_sp
iob_rom_2p	hardware/memories/rom/iob_rom_2p
iob_rom_atdp	hardware/memories/rom/iob_rom_atdp

©2025 IObundle Lda. All rights reserved

Name	Directory
iob_rom_sp	hardware/memories/rom/iob_rom_sp
iob_rom_tdp	hardware/memories/rom/iob_rom_tdp
iob_reg	hardware/registers/iob_reg
iob_pack	hardware/shifters/iob_pack
iob_piso_reg	hardware/shifters/iob_piso_reg
iob_shift_reg	hardware/shifters/iob_shift_reg
iob_sipo_reg	hardware/shifters/iob_sipo_reg
iob_unpack	hardware/shifters/iob_unpack
iob_f2s_1bit_sync	hardware/synchronizers/iob_f2s_1bit_sync
iob_neg2posedge_sync	hardware/synchronizers/iob_neg2posedge_sync
iob_reset_sync	hardware/synchronizers/iob_reset_sync
iob_sync	hardware/synchronizers/iob_sync
iob_system	iob_system
iob_system_iob_cyclonev_gt_dk	iob_system/hardware/fpga/quartus/iob_cyclonev_gt_dk/iob_system_iob_cyclonev_gt_dk
iob_system_iob_aes_ku040_db_g	iob_system/hardware/fpga/vivado/iob_aes_ku040_db_g/iob_system_iob_aes_ku040_db_g
iob_system_iob_basys3	iob_system/hardware/fpga/vivado/iob_basys3/iob_system_iob_basys3
iob_system_iob_zybo_z7	iob_system/hardware/fpga/vivado/iob_zybo_z7/iob_system_iob_zybo_z7
iob_system_sim	iob_system/hardware/simulation/iob_system_sim
iob_system_syn	iob_system/hardware/syn/iob_system_syn
iob_system_tester	iob_system/iob_system_tester
iob_vexriscv	iob_system/submodules/iob_vexriscv
iob_bootrom	iob_system/submodules/iob_bootrom
iob_axistream_in	peripherals/iob_axistream_in
iob_axistream_out	peripherals/iob_axistream_out
iob_gpio	peripherals/iob_gpio
iob_nco	peripherals/iob_nco
iob_nco_sync	peripherals/iob_nco/hardware/modules/iob_nco_sync
iob_regfileif	peripherals/iob_regfileif
iob_nativebridgeif_setup	peripherals/iob_regfileif/iob_nativebridgeif_wrappper
iobnativebridge	peripherals/iob_regfileif/software/python
mkregsregfileif	peripherals/iob_regfileif/software/python
iob_timer	peripherals/iob_timer
iob_timer_core	peripherals/iob_timer/hardware/iob_timer_core
iob_uart	peripherals/iob_uart
iob_uart_core	peripherals/iob_uart/hardware/iob_uart_core
iob_uart_tester	peripherals/iob_uart/iob_uart_tester
iob_printf	software/iob_printf
iob_str	software/iob_str

Table 3: Table of cores available in the library of the Py2HWSW framework. The *Directory* column is the path to the core's setup directory, relative to the Py2HWSW lib directory `py2hwsw/lib/`.

Table 3 lists the cores available in the Py2HWSW framework's core library.

Each core contains its own user guide, which can be built using the following commands:

```
py2hwsw <core_name> setup
make -C ../<core_name>_V<core_version>/ doc-build
xdg-open ../<core_name>_V<core_version>/document/ug.pdf
```

©2025 IObundle Lda. All rights reserved

5 How To Use

5.1 Setup

To set up a core with Py2HWSW, you'll need to have Nix installed on your system. You can download and install Nix from the official Nix website. Once Nix is installed, you can clone the Py2HWSW repository using the command `git clone --recursive git@github.com:IObundle/py2hwsw.git`.

Next, navigate to the Py2HWSW directory and run the command `nix-shell` to enter the Nix-shell environment. This will ensure that all dependencies required by Py2HWSW are installed and available.

To set up a core, you can use the command:

```
nix-shell --run "py2hwsw $(CORE) setup --build_dir '$(BUILD_DIR)' --py_params 'param1=param1_val:param2
```

This command will generate the necessary files and directories for your core in the specified build directory.

You can customize the setup process by passing additional options to the `py2hwsw` command. For example, you can disable format and linting checks by adding the options `--no_verilog_lint` and `--no_verilog_format` to the command.

Here's an example of a setup directory structure:

```
mycore/
  mycore.py
  hardware/
    src/
      mycore.v
```

In this example, the `mycore.py` file contains the core description, and the `hardware/src` directory contains the Verilog source files for the core.

In some cases, the Verilog source file (`mycore.v`) may not be necessary, as the `mycore.py` file can describe the entire core, including its ports, wires, components, and even custom Verilog code. This allows for a high degree of flexibility and customization, as users can define their core's architecture and behavior entirely within the Python description file.

The Python description is particularly useful because it enables the creation of higher-level abstractions, making it easier to design and work with complex hardware components. Additionally, the use of Python parameters allows for dynamic modification of cores, enabling users to easily customize and adapt their designs to different use cases and applications.

To set up this core, you would run the command:

```
nix-shell --run "py2hwsw mycore setup --build_dir './build' --py_params 'param1=param1_val:param2=param
```

This would generate the necessary files and directories for the core in the `./build` directory.

Note that you can customize the setup process to fit your specific needs by modifying the core description, Verilog source files, and setup command options.

©2025 IObundle Lda. All rights reserved

5.2 Simulation

To simulate a core using Py2HWSW, you can use the `make sim-run` command inside the generated build directory. This command will run the simulation using the default simulator (Icarus Verilog). You can specify the simulator to be used using the `SIMULATOR` variable.

For example, to simulate the core using Verilator, you can run the command `make sim-run SIMULATOR=verilator` inside the core's build directory. This will compile the testbench and run the simulation, displaying the output on the console.

Py2HWSW also provides a universal Verilator testbench that can be used to simulate IP cores. The testbench behaves like a processor reading and writing to the core's control and status registers (CSRs), allowing for easy testing and verification of the core's functionality.

You can customize the simulation process by modifying the testbench and simulation parameters, such as the simulation time, input stimuli, and output signals to be monitored. Additionally, you can use other simulators, such as VCS or QuestaSim, by specifying the corresponding simulator variable.

Some cores in the Py2HWSW library also include a tester that can be used to verify their functionality. Examples of such cores include `iob_aoi`, `iob_pulse_gen`, and `iob_system`. These testers can be run along with the core to test its behavior and ensure that it is working as expected.

To run the tester, simply navigate to the tester's build directory, usually located inside the core's build directory in a folder named `<core_name>_tester/`, and run the command `make sim-run`. This will compile and run the tester, allowing you to verify the core's functionality and debug any issues that may arise. By providing these testers, Py2HWSW makes it easier to develop and test complex hardware components, and ensures that the cores in the library are reliable and functional.

5.3 End to End Examples

This section provides three end-to-end examples of using Py2HWSW to generate and verify digital hardware cores. The examples cover the iob_aoi, iob_pulse_gen, and iob_soc cores, showcasing the automation of Verilog module generation from attributes.

5.3.1 iob_aoi Example

The iob_aoi core is a simple example that combines an AND, OR, and invert logic gates. The core's attributes are defined in the iob_aoi.py file, available at https://github.com/IObundle/py2hwsw/tree/main/py2hwsw/lib/hardware/basic_tests/iob_aoi. To generate the iob_aoi core, follow these steps:

1. Create or modify the iob_aoi.py file to set the attributes of the core, describing how it should be generated using the Py2HWSW standard core dictionary interface.
2. Optionally, add more files to the setup directory as needed, such as manual Verilog sources or templates, scripts, or software.
3. Call the Py2HWSW setup process using the command:

```
nix-shell --run "py2hwsw iob_aoi setup --build_dir '$(BUILD_DIR)'"
```

©2025 IObundle Lda. All rights reserved

4. The generated build directory contains all the Verilog sources, Makefile, and configurations to run the core in various flows (simulation, FPGA).
5. To run the core in simulation, call the command: `make sim-run` from the build directory.

5.3.2 iob_pulse_gen Example

The iob_pulse_gen core is used to generate signal pulses with configurable start and duration. The core's attributes are defined in the iob_pulse_gen.py file, available at https://github.com/IObundle/py2hwsw/blob/main/py2hwsw/lib/hardware/clocks_resets/iob_pulse_gen/iob_pulse_gen.py. To generate the iob_pulse_gen core, follow these steps:

1. Create or modify the iob_pulse_gen.py file to set the attributes of the core, describing how it should be generated using the Py2HWSW standard core dictionary interface.
2. Optionally, add more files to the setup directory as needed, such as manual Verilog sources or templates, scripts, or software.
3. Call the Py2HWSW setup process using the command:

   ```
   nix-shell --run "py2hwsw iob_pulse_gen setup --build_dir '$(BUILD_DIR)'"
   ```

4. The generated build directory contains all the Verilog sources, Makefile, and configurations to run the core in various flows (simulation, FPGA).
5. To run the core in simulation, call the command: `make sim-run` from the build directory.

5.3.3 iob_soc Example

The iob_soc core is a more complex example used to create a system on chip. The core's attributes are defined in the iob_soc.py file, available at https://github.com/IObundle/iob-soc. The iob_soc.py file supports high-level Python parameters that allow configuring main SoC components like the CPU, memories, and peripherals. To generate the iob_soc core, follow these steps:

1. Create or modify the iob_soc.py file to set the attributes of the core, describing how it should be generated using the Py2HWSW standard core dictionary interface.
2. Optionally, add more files to the setup directory as needed, such as manual Verilog sources or templates, scripts, or software.
3. Call the Py2HWSW setup process using the command:

   ```
   nix-shell --run "py2hwsw iob_soc setup --build_dir '$(BUILD_DIR)'"
   ```

4. The generated build directory contains all the Verilog sources, Makefile, and configurations to run the core in various flows (simulation, FPGA).
5. To run the core in simulation, call the command: `make sim-run` from the build directory.

These examples demonstrate the automation of Verilog module generation from attributes using Py2HWSW, showcasing the flexibility and ease of use of the framework.

©2025 IObundle Lda. All rights reserved

5.4 Customizing Py2HWSW

Py2HWSW allows users to customize its behavior and core generation process. When running cores directly from the cloned Py2HWSW repository, users can modify the cores or Py2HWSW scripts for debugging purposes.

The Py2HWSW repository contains several main folders, including lib, scripts, and generic folders.

```
.
└── py2hwsw
    ├── <py2 generic folders>
    ├── lib
    └── scripts
```

The py2hwsw folder contains generic folders that are copied to every core build directory set up via Py2HWSW. These folders include standard Makefiles, FPGA board constraints, simulator dependencies, and other essential files.

To override the default files, users can create a file with the same name in their core's setup directory. For example, to override the default py2hwsw/document/tsrc/sim_desc.tex, create a new document/tsrc/sim_desc.tex in the core's setup directory. Py2HWSW will first copy the generic default file to the build directory and then copy the core files from the setup directory, overriding the default file.

The lib directory contains a library of cores provided by Py2HWSW. These cores are intended to be bug-free and do not typically require modifications. However, if users need to modify a core for their project, they can copy the corresponding core's setup directory to a subfolder in their project directory. Py2HWSW will use the first core it finds with the required name, so users can create custom modifications to the core specific to their project.

For example, to modify the iob_and core, copy its folder from the Py2HWSW repository and place it in the user's project directory. When calling Py2HWSW from the project directory, it will find the copied iob_and folder first and use it to generate the iob_and core.

The scripts folder contains the Python scripts that make up the Py2HWSW tool. These scripts are typically only modified by developers, as they directly change the Py2HWSW program's behavior. However, users can modify these scripts for quick bug fixes or to add custom functionality.

By customizing Py2HWSW, users can tailor the tool to their specific needs and create custom cores and workflows. This flexibility allows users to adapt Py2HWSW to their project's requirements and create complex digital designs with ease.

5.5 Troubleshooting

When encountering errors during the setup process with Py2HWSW, there are several steps you can take to diagnose and resolve the issue.

©2025 IObundle Lda. All rights reserved

5.5.1 Error Messages

The main error message is usually printed in red color and provides information on where the issue originates, often due to a misconfiguration in the provided core dictionary. The traceback that follows is more useful for Py2HWSW developers, as it contains information on which Py2HWSW function has thrown the error.

5.5.2 Debugging Options

If more information is required to troubleshoot the issue, you can use the following options:

- The –debug_level flag: When calling py2hwsw with the –debug_level flag, you can print debug messages during the setup process. The higher the debug level, the more messages are printed.
- Adding print statements: You can add print statements in your own core's .py file to understand when the script is being called and what contents it contains.
- Modifying Py2HWSW scripts: Adding print statements to the Py2HWSW main scripts can also be useful, but this requires understanding the inner workings of Py2HWSW and is usually reserved for developers.

5.5.3 Build Process Errors

If the Py2HWSW setup process completes successfully, but the build process for a flow from the build directory gives errors (e.g., calling the Makefile from the build directory for simulation), follow these steps:

1. Check the generated Verilog sources: Verify that the contents of the generated Verilog sources are as intended.
2. Check tool-specific files: If the error message is simulator/tool-specific and does not seem related to the Verilog sources, check the constraints files, Makefiles, and other tool-specific files to determine where the issue originates.

5.5.4 Overriding Py2HWSW Generated Files

If you need to modify a Py2HWSW generated file, you can override it by creating a new file with the same name in your core's setup directory. This allows you to customize the generated files to suit your specific needs.

By following these troubleshooting steps, you should be able to identify and resolve issues that arise during the setup and build process with Py2HWSW.

©2025 IObundle Lda. All rights reserved