Py2HWSW

Python framework for embedded hardware/software codesign

December 5, 2024

User Guide, 0.8 , Build 414ed91b

Document Version History

Version	Date	Person	Changes from previous version
May/30/2022	Document released with product version 0.8 .		

Contents

List of Tables

List of Figures

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.

- **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?

Py2HWSW does this by:

- **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.

Learn more about 3[[How It Works]] and **??**[[How To Use]].

2 Getting Started

2.1 Setup Directory

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

```
.
├── 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.

2.2 Create An AND Gate Core: iob_and

The simplest core description for Py2HWSW is as follows:

```
# SPDX-FileCopyrightText: 2024 IObundle
#
# SPDX-License-Identifier: MIT


def setup(py_params_dict):
    attributes_dict = {
        "version": "0.1",
        "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"},
                ],
            },
        ],
        "superblocks": [
            {
                "name": "simulation",
                "descr": "Blocks for simulation",
                "blocks": [
                    # Simulation wrapper
                    {
                        "core_name": "iob_sim",
                        "instance_name": "iob_sim",
                        "instantiate": False,
                        "dest_dir": "hardware/simulation/src",
                    },
                ],
            },
        ],
        "snippets": [{"verilog_code": "   assign y_o = a_i & b_i;"}],
    }

    return attributes_dict
```

View Source

More examples and information can be found in the **??**[[How To Use]] section.

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
```

3 How It Works

This section gives a detailed description of the Py2HWSW framework.

3.1 Setup Flow Chart

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

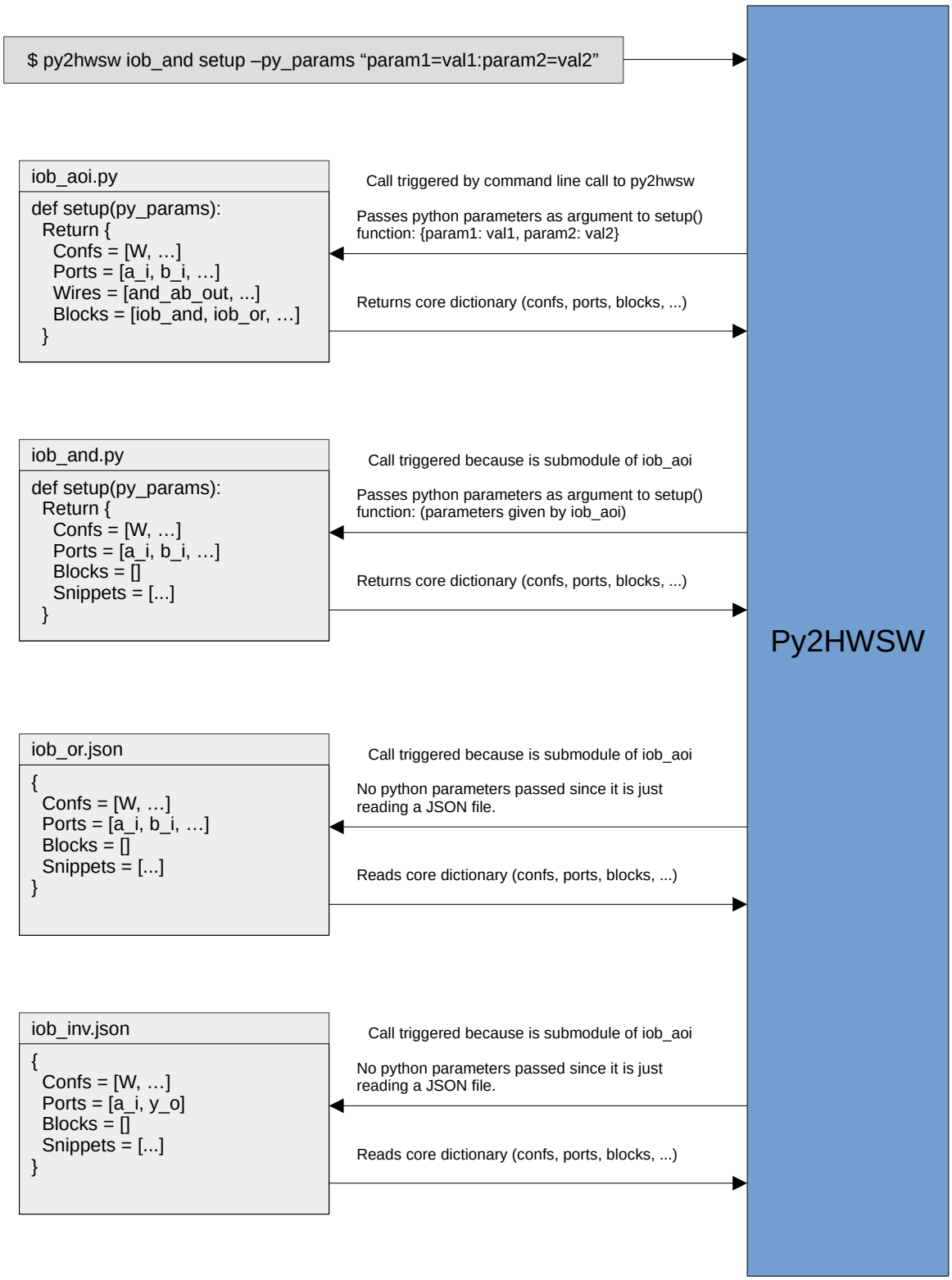

Figure 1: High-Level Flow Chart of Py2HWSW Setup Procedure

3.2 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.

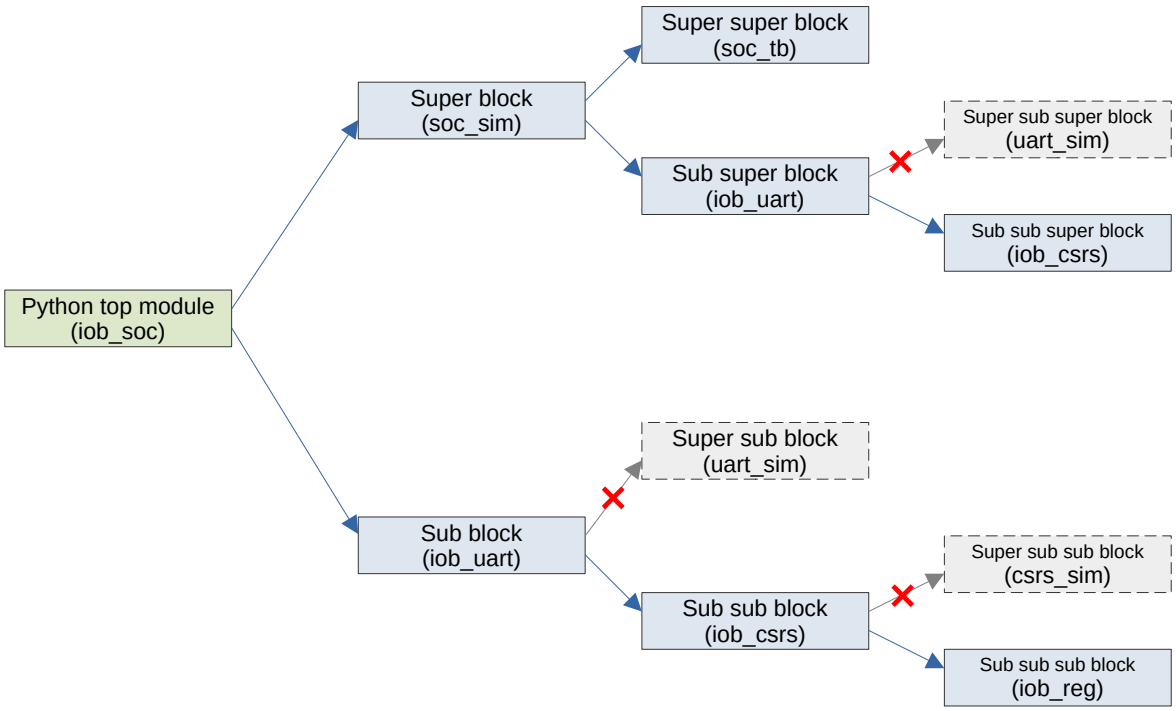

Figure 2: Block Hierarchy of a Py2HWSW Project

3.3 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".

```
    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 not args.core_name:
        parser.print_usage(sys.stderr)
        fail_with_msg("Core name is required.")

    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_dict":
        iob_core.print_core_dict(args.core_name, **py_params)
    elif args.target == "print_py2hwsw_attributes":
        iob_core.print_py2hwsw_attributes(args.core_name, **py_params)
    else:
        fail_with_msg(f"Unknown target: {args.target}")
```

View Source

3.4 Main class for core representation: iob_core.py

The iob_core class is the main class used to represent a core.

It inherits attributes from its parent classes iob_module and iob_instance.

```
class iob_core(iob_module, 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.

```
            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": (
                        instantiator.attributes_dict if instantiator else ""
                    ),
                    **kwargs,
                }
            )
            py2_core_dict = {"original_name": core_name, "name": core_name}
            py2_core_dict.update(core_dict)
            instance = __class__.py2hw(
                py2_core_dict,
                instantiator=instantiator,
                # Note, any of the arguments below can have their values overridden by
                # the py2_core_dict
                **kwargs,
            )
        elif file_ext == ".json":
            instance = __class__.read_py2hw_json(
                os.path.join(core_dir, f"{core_name}.json"),
                # Note, any of the arguments below can have their values overridden by
```

View Source